If the paper title is too long for the running head, you can set an abbreviated paper title here

DWFS-Obfuscation: Dynamic Weighted Feature Selection for Robust Malware Familial Classification under Obfuscation

Xingyuan Wei[1,2][0009−0001−6595−4222], Zijun Cheng[4], Ning Li[1], Qiujian Lv[1], Ziyang Yu ∗[1,2], and Degang Sun[2,3]

[1] Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
`talentedyuan@gmail.com`
[2] University of Chinese Academy of Sciences
[3] Computer Network Information Center Chinese Academy of Sciences Beijing, China
[4] Space Engineering University

Abstract. Due to its open-source nature, the Android operating system has consistently been a primary target for attackers. Learning-based methods have made significant progress in the field of Android malware detection. However, traditional detection methods based on static features struggle to identify obfuscated malicious code, while methods relying on dynamic analysis suffer from low efficiency. To address this, we propose a dynamic weighted feature selection method that analyzes the importance and stability of features, calculates scores to filter out the most robust features, and combines these selected features with the program's structural information. We then utilize graph neural networks for classification, thereby improving the robustness and accuracy of the detection system. We analyzed 8,664 malware samples from eight malware families and tested a total of 44,940 malware variants generated using seven obfuscation strategies. Experiments demonstrate that our proposed method achieves an F1-score of 95.56% on the unobfuscated dataset and 92.28% on the obfuscated dataset, indicating that the model can effectively detect obfuscated malware.

Keywords: Dynamic Weighted Feature Selection · Code Obfuscation · Android Malware Family Classification · Robustness.

1 Introduction

Malware, particularly on the Android mobile platform, poses an increasingly severe threat. According to a report by AVTEST, as of January 2025, the number of Android malware instances reached 35,641,466[1], resulting in substantial losses. Due to the Android platform's ease of cloning software, many vendors and malware authors employ obfuscation strategies to protect software intellectual property. However, Malware creators also frequently use obfuscation techniques to conceal malicious code, making malware detection significantly more challenging.

To detect Android malware, existing research has utilized machine learning methods[2],[20] and deep learning approaches[3],[4],[5] to analyze code. Current static analysis methods, such as those extracting opcodes, API calls, and permission requests or relying on single feature types, offer high execution efficiency and accuracy. However, their performance degrades significantly under the interference of code obfuscation strategies. Dynamic analysis methods[6],[7], which require running the software to obtain features, can enhance detection robustness but suffer from low efficiency. The feature subsets of Android applications are virtually inexhaustible, and inputting all features into a model results in low efficiency and excessive redundant data. Therefore, identifying a subset of features from a vast feature pool that can resist the effects of obfuscation techniques is particularly critical.

Although obfuscation techniques can, to some extent, alter the code structure of an application and cause changes at the code level, such as in information flow, the core operational logic of the application should still exhibit significant similarity. We filter out features that are minimally affected by obfuscation, which we refer to as Anti-Obfuscation features. To this end, we propose the Dynamic Weighted Feature Selection (DWFS) algorithm, integrating the selected features with Graph Neural Networks (GNNs). Specifically: 1) Extract a rich candidate feature set from a large number of Android APK files. 2) Filter out features from this vast candidate set that are both significant for malware detection and resistant to obfuscation. 3) Construct a method-level Sensitive Behavior Subgraph (SBS) by parsing APK files to capture the program's behavioral information. Assign the features selected in step 1) to each node in the SBS, generating feature vectors. Finally, leverage GNNs to learn a joint representation of the SBS's graph structure and node features, enabling malware family classification.

Our main contributions are as follows:

(1) Malware Familial Classification. We design a novel framework for malware familial classification that integrates obfuscation-resistant features and the program's structural information, utilizing GNNs for classification.

(2) Anti-Obfuscation. We innovatively propose a dynamic weighted feature selection method that analyzes the importance and stability of features to automatically select the most robust ones. Additionally, we make these features publicly available, providing empirical insights for future research on the robustness of malware detection. We also open-source our code, which can be accessed from GitHub [5].

(3) Effective Detection. We optimize the sensitive behavior subgraph derivation algorithm, combining the selected obfuscation-resistant factors with GNNs to efficiently capture the malicious behaviors of programs, thereby enhancing the accuracy and robustness of the malware detection system.

[5] https://github.com/XingYuanWei/DWFS-Obfuscation

2 Related Work

2.1 Android Malware Familial Classification Based On Static Analysis

Previous research works[8],[9],[10],[11] have proposed dividing malware samples into their respective families. Malware family classification based on static analysis has achieved promising results. For example, FalDroid[9] conducts frequent subgraph analysis to extract common subgraphs for each family and uses them to perform familial classification. Apaposcopy[10] considers both data flow and control flow information from malware samples, classifying them by performing weighted program analysis. These methods leverage different types of program information to achieve accurate malware classification; however, none of them account for the impact of obfuscation on family classification.

2.2 Anti-obfuscation Android Malware Familial Classification

AndrODet[12] is one of the Android obfuscation detectors. This system detects obfuscated applications using online machine learning algorithms, with features statically extracted from bytecode. It achieves an accuracy of 92.02% in string encryption detection, 81.41% in identifying string encryption, and 68.32% in control flow obfuscation detection. Orlis[13] is a library detector for Android applications. It extracts features from function calls and Android API calls, making them resilient to the most common obfuscation techniques. RevealDroid[14] is an obfuscation-resilient malware classifier. The features used to train the model are derived from Android API usage, reflection, and application permissions. While it achieves strong results (98% accuracy in detecting malware and 95% accuracy in identifying its family), the obfuscation techniques considered in its experiments are relatively simple.

2.3 Dynamic Weighted Feature Selection

Feature selection contributes to improving machine learning performance by selecting a subset of relevant features for the learning algorithm. The DFWS method proposed in this paper resembles an adversarial defense approach, aiming to enhance model performance by selecting robust features, particularly when confronting code obfuscation in Android malware families. Our method evaluates the stability and importance of features under malware code obfuscation, selecting those that remain reliable even after obfuscation. The earliest similar idea was proposed by Sun et al.[15], who introduced a dynamic weighted feature selection method based on feature interactions, demonstrating its effectiveness on four gene microarray datasets. Meanwhile, we draw inspiration from the robustness of feature selection in adversarial machine learning[16], combining dynamic weighting with adversarial machine learning principles. A related work, DroidRL[17], leverages a Reinforcement Learning algorithm to obtain a

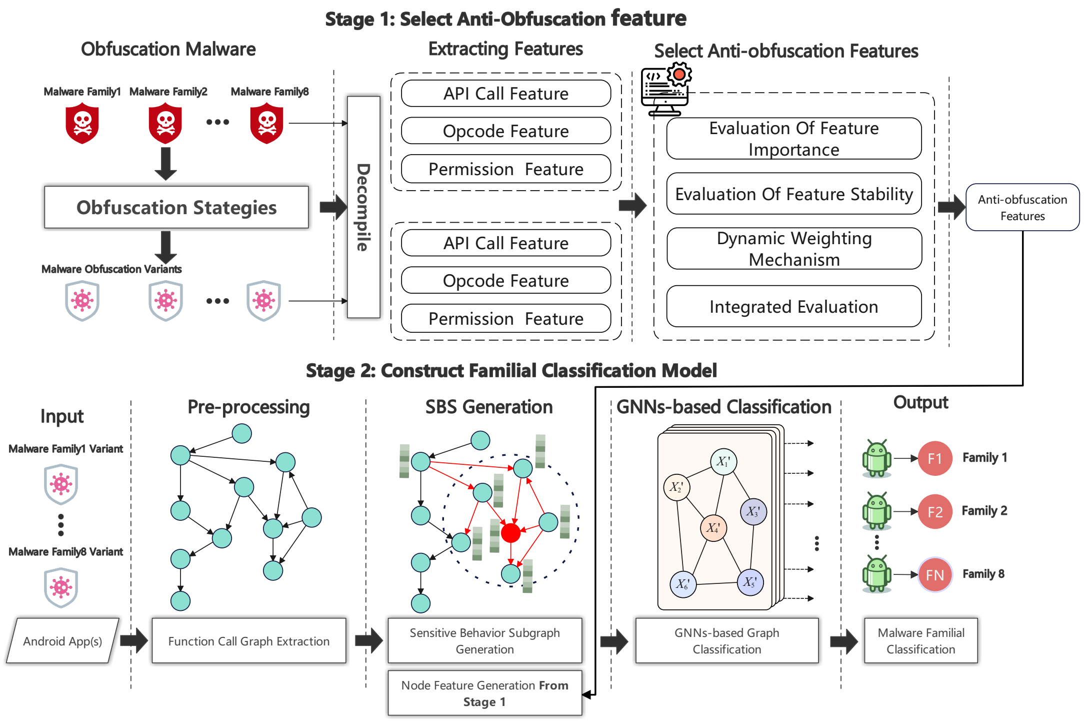
Stage 1: Select Anti-Obfuscation feature
Obfuscation Malware
Malware Family1
Malware Family2
Malware Family8
Obfuscation Stategies
Malware Obfuscation Variants
Decompile
Extracting Features
API Call Feature
Opcode Feature
Permission Feature
Select Anti-obfuscation Features
Evaluation Of Feature Importance
Evaluation Of Feature Stability
Dynamic Weighting Mechanism
Integrated Evaluation
Anti-obfuscation Features
Stage 2: Construct Familial Classification Model
Input
Malware Family1 Variant
Malware Family8 Variant
Android App(s)
Pre-processing
Function Call Graph Extraction
SBS Generation
Sensitive Behavior Subgraph Generation
Node Feature Generation From Stage 1
GNNs-based Classification
GNNs-based Graph Classification
Output
Family 1
Family 2
Family 8
Malware Familial Classification

Fig. 1. DWFS-Obfuscation Overview.

feature subset effective for malware classification. This framework's feature selection exhibits high performance without any manual intervention, but its feature selection process remains static.

3 Methodology

This paper proposes a malware detection method for Android APK files that combines a dynamic weighted feature selection algorithm with graph neural networks. The overall architecture of the system is illustrated in Figure 1, consisting of two main stages: Stage 1, including 1) obfuscating malware, 2) feature extraction, and 3) dynamic weighted feature selection; Stage 2, including 1) preprocessing to extract the Function Call Graph (FCG), 2) constructing the Sensitive Behavior Subgraph and generating node features, and 3) GNNs training and classification.

3.1 Stage 1

Feature Extraction Feature extraction is a foundational step in malware detection. This paper extracts the following three types of static features from Android APK files, forming a high-dimensional candidate feature set:

Opcodes, By analyzing the Dalvik bytecode of the APK, we extracted all Dalvik opcodes from the official Android website[18]. The frequency of opcodes can reflect the behavior of a program and its execution logic. For example, opcodes such as move and invoke-virtual have a total of 235.

API Calls, We utilized APIChecker[19] to statistic the usage of Android APIs. We focus on analyzing sensitive APIs, including sendTextMessage, getDeviceId, and others have a total of 426, which reflect the functional behavior of the program.

Permissions, We record the permissions requested by the APK, represented as binary features, such as READ_SMS and INTERNET, which indicate the program's access requirements to system resources. We include a total of 86 commonly used Android permission features in our overall set, sourced from NATICUSdroid[20].

Our goal is to select Anti-Obfuscation features from the 747 feature dimensions mentioned above, reducing the dimensionality of the features. In this process, we achieve a refinement of the information, thereby increasing its density.

Dynamic Weighted Feature Selection To filter out features that are both significant for malware detection and resistant to obfuscation from a high-dimensional pool of candidate features, this study designs the Dynamic Weighted Feature Selection (DWFS) algorithm. DWFS comprehensively evaluates the importance of features on unobfuscated samples and their stability on obfuscated samples, balancing these two aspects through a dynamic weighting mechanism.

Evaluation Of Feature Importance On the unobfuscated sample set, a random forest classifier is used to compute the importance of each feature. A classifier is trained using the unobfuscated training data $X_{unobf}$and its corresponding labels y_{unobf}. in this paper, the random forest algorithm is employed. Feature importance scores $I(f_i)$ are extracted from the trained model, where f_i represents the i-th feature. Subsequently, the model's accuracy on the unobfuscated data, denoted as acc_{unobf}, is calculated.

Evaluation Of Feature Stability To evaluate the stability of features under different obfuscation techniques, for each obfuscation strategy j (as shown in Table 4.4), a classifier is trained using the obfuscated training data X_{obf_j} and its corresponding labels y_{obj_j}. The model's accuracy under this obfuscation strategy denoted as acc_{obf_j}, is calculated. Feature importance scores $I_{obf_j(f_i)}$ are extracted from the model. Subsequently, the obfuscation impact factor α_j, which represents the degree of impact of obfuscation on model performance, is computed. The calculation of the obfuscation impact factor is shown in Equation 1.

$$\alpha_j = \frac{acc_{unobf} - acc_{obf_j}}{acc_{unobf}} \tag{1}$$

The obfuscation factors α_j for all obfuscation strategies are normalized to ensure $\Sigma\alpha_j = 1$. Finally, the weighted sum of the importance changes for each feature across all obfuscation strategies is calculated, with the formula shown in Equation 2.

$$\varDelta I(f_i) = \Sigma a_j \cdot \left| I(f_i) - I_{obf_j}(f_i) \right| \tag{2}$$

Dynamic Weighting Mechanism Dynamically adjust the weights of importance and stability based on the obfuscation impact factor. A hyperparameter β is set to balance importance and stability. The importance weight is calculated as $\omega_1 = 1 - \beta \cdot \bar{a}$, where $\bar{a}$ is computed as shown in Equation 3, with m representing the number of obfuscation strategies.

$$\bar{a} = \frac{1}{m} \sum_{j=1}^{m} \alpha_j \tag{3}$$

Integrated Evaluation A composite score is calculated for each feature through a formula that balances its importance and stability, with the calculation method shown in Equation 4. Finally, robust features are θ selected based on the composite score, and we set a hyperparameter as the threshold for feature selection. The entire algorithm process is detailed in Algorithm 3.1.

$$S(f_i) = \omega_1 \cdot I(f_i) - \omega_2 \cdot \varDelta I f(_i) \tag{4}$$

By comprehensively evaluating the importance and stability of features, and leveraging a dynamic weighting mechanism to adaptively adjust weights, it is possible to filter out a set of features that are both significant and resistant to obfuscation from a vast pool of features. The output features of DWFS can provide high-quality input for subsequent detection models, such as GNNs. thereby enhancing the detection accuracy and robustness of the detection system.

3.2 Stage 2

3.3 Pre-processing

As shown in Table 4, for 8,664 malware samples, the FCG extracted from each APK has, on average, 201,699 nodes and 603,901 edges. If a graph neural network classification method is applied directly to the FCG, the model would need to process a massive amount of node information, resulting in extremely low efficiency and negatively impacting detection accuracy. Therefore, to efficiently capture the malicious behaviors of malware, this paper constructs a method-level Sensitive Behavior Subgraph (SBS). The SBS is a directed graph, which we define as $G = (V, E)$. Here, Vrepresents the set of nodes, where each node

Algorithm 1 Dynamic Weighted Feature Selection (DWFS)

```
Input  : Unobfuscated data $X_{\text{unobf}}$, labels $y_{\text{unobf}}$, obfuscated data $\{(X_{\text{obf}_j}, y_{\text{obf}_j})\}_{j=1}^{m}$, hyperparameter $\beta$, threshold $\theta$
Output: Selected feature subset $F_{\text{selected}}$
1  Train Random Forest on $X_{\text{unobf}}$, $y_{\text{unobf}}$ to obtain importances $I$ and accuracy $\text{acc}_{\text{unobf}}$ ;  // $I$: importances, $\text{acc}_{\text{unobf}}$: baseline accuracy
2  for each obfuscation type $j$ from 1 to $m$ do
3      Train Random Forest on $X_{\text{obf}_j}$, $y_{\text{obf}_j}$ to obtain importances $I_{\text{obf}_j}$ and accuracy $\text{acc}_{\text{obf}_j}$ ;  // $I_{\text{obf}_j}$: importances, $\text{acc}_{\text{obf}_j}$: accuracy
4      Compute impact $\alpha_j$ as relative accuracy drop from $\text{acc}_{\text{unobf}}$ to $\text{acc}_{\text{obf}_j}$ ; // $\alpha_j$, obfuscation impact factor
5  end
6  Compute total impact $\text{sum}_\alpha$ as sum of all $\alpha_j$
7  Normalize each $\alpha_j$ by dividing by $\text{sum}_\alpha$ ;  // Normalized weights
8  Initialize $\Delta I$ as zero vector matching feature count ;  // $\Delta I$: stability change
9  for each $j$ from 1 to $m$ do
10     Update $\Delta I$ by adding $\alpha_j$ times absolute difference between $I$ and $I_{\text{obf}_j}$ ;  // Weighted stability change
11 end
12 Compute average impact $\overline{\alpha}$ by averaging all $\alpha_j$ over $m$ ;  // $\overline{\alpha}$: mean impact
13 Set importance weight $w_1$ by reducing 1 by $\beta$ times $\overline{\alpha}$ ;  // $w_1$: importance weight
14 Set stability weight $w_2$ as $\beta$ times $\overline{\alpha}$ ;  // $w_2$: stability weight
15 Compute scores $S$ by combining $I$ scaled by $w_1$ and $\Delta I$ scaled by $w_2$, subtracting latter ;  // $S$: feature scores
16 Initialize $F_{\text{selected}}$ as empty set ;  // $F_{\text{selected}}$: selected features
17 for each feature index $i$ do
18     if $S_i$ exceeds $\theta$ then
19         Add feature $f_i$ to $F_{\text{selected}}$ ;  // $f_i$: feature at index $i$
20     end
21 end
22 return $F_{selected}$
```

$v \in V$ denotes a method, and E represents the set of edges, where each directed edge $e = (u, v) \in E$ indicates that method u calls method v. It is a subgraph of the FCG.

We first extract the FCG from APK files. We use apktool[23] and androguard[24] tools for decompilation to obtain the FCG. The SBS is then extracted from the FCG, with the extraction process detailed in Algorithm 3.3.

SBS Generation Previous work relied on sensitive APIs to simplify the FCG[22],[25]. Some of these approaches used BFS or DFS algorithms to traverse all nodes, directly extracting the N-hop neighborhood around sensitive nodes. Through a not particularly rigorous empirical analysis, we found that the aforementioned approaches lead to two issues: one is the inclusion of excessive redundant nodes, which still fails to efficiently capture malicious behaviors; the other is the disconnection of nodes in the graph, preventing the complete preservation of relationships between multiple malicious behaviors. Our method first retains the direct predecessors and successors of sensitive APIs and then optionally extends to an N-hop neighborhood, addressing the above issues. In this paper, sensitive nodes are sourced from the PScout[21]

Specifically, the **direct caller** nodes and **callee** nodes of sensitive nodes have, on average, K predecessors and L successors. Here, we retain both these predecessor and successor nodes, resulting in a simplified graph with approximately M + M*K + M*L nodes and about M*K + M*L edges, where M is the number of sensitive nodes. When we set N=3, i.e., extending to the 3-hop adjacent nodes of sensitive nodes, calculations show an increase of 16.5% in the number of nodes and edges, which is still significantly less than the original node and edge counts. The features selected by 3.1 are stored in the node as node features.

Algorithm 2 Get Sensitive Behavior Subgraph

```
Input : Original function call graph $G = (V, E)$, List of sensitive API nodes $S$, Extension hop
        count $N$.
Output: Simplified function call graph $G' = (V', E')$.
Initialize: Set $V' \leftarrow \emptyset$ (set of retained nodes), $E' \leftarrow \emptyset$ (set of retained edges).
 for each sensitive API node $s \in S$ do
    Add $s$ to $V'$.

    for each predecessor $p$ of $s$ do
        Add $p$ to $V'$ and add edge $(p, s)$ to $E'$.
    end
    for each successor $c$ of $s$ do
        Add $c$ to $V'$ and add edge $(s, c)$ to $E'$.
    end
    if $N > 0$ then
        Initialize visited set visited $\leftarrow \{s\}$ and queue queue $\leftarrow$ deque([$(s, 0)$]).
        while queue is not empty do
            Dequeue $(u, h)$ from queue.
            if $h \geq N$ then
                continue.
            end
            for each predecessor $p$ of $u$ do
                if $p \notin$ visited then
                    Add $p$ to $V'$ and add edge $(p, u)$ to $E'$. Add $p$ to visited and enqueue $(p, h+1)$.
                end
            end
            for each successor $c$ of $u$ do
                if $c \notin$ visited then
                    Add $c$ to $V'$ and add edge $(u, c)$ to $E'$. Add $c$ to visited and enqueue $(c, h+1)$.
                end
            end
        end
    end
end
Generate subgraph $G' \leftarrow G$.subgraph($V'$) based on retained nodes and edges.
return $G'$
```

4 Experimental Evaluation

4.1 Datasets and Metrics

The dataset for this study consists of malicious executable files obtained from AndroZoo[26]. It includes a total of 8,664 malware files (80.558 GB) across eight malware families. Additionally, we utilized the powerful obfuscation tool Obfuscapk[28] to obfuscate the malware, employing obfuscation techniques encompassing all the contents listed in Table 4.4, resulting in 44,940 malware variants.

These files were disassembled, and their corresponding features were extracted to obtain the respective SBS.

To comprehensively evaluate the proposed method and assess the effectiveness of the GMC framework, we adopted a series of widely used and representative performance metrics. These metrics include Accuracy, Precision, Recall, and F1 Score.

4.2 Simplification Effect

For graph scale simplification, the experimental results are presented in Table 1 and Table 2, which provide statistical information on the FCG and SBS for eight malware categories, respectively. Table 1 shows that the FCG is large in scale, with an average node count ranging from 3505.65 to 60254.29, an edge count from 7740.72 to 170902.83, and a total storage space of 203.3 GB. In contrast, Table 2 indicates that the SBS is significantly reduced, with the average node count dropping to 536 to 6350, the edge count decreasing to 853 to 15408, and the total storage space reduced to only 34.63 GB, a reduction of approximately 83%. The method I propose, by extracting sensitive behavior subgraphs, not only simplifies the graph structure (reducing the average number of nodes and edges by about 90%), facilitating analysis, but also significantly improves computational efficiency while lowering storage requirements, making it an efficient and practical solution for large-scale malware detection.

Table 1. Function Call Graph Statistic Infomation

Class	Apps	Function Call Graph					
		# Graph	Avg # Nodes	Avg # Edges	Median # Nodes	Median # Edges	Storage Space (GB)
adPush	1500	1409	5809.85	16797.31	16797.31	10386.5	8.51
artemis	1032	1032	3505.65	7740.72	7740.72	2135	3.50
openconnection	1494	1488	60254.29	170902.83	170902.83	187081	87.50
kuguo	1500	1500	10414.22	34736.42	34736.42	24961	15.35
spyagent	528	528	16322.40	42001.09	42001.09	45138	8.37
dzhtny	560	552	53210.26	167443.68	167443.68	175712	29.18
igexin	1500	1500	24293.67	82755.62	82755.62	72486	35.83
leadbolt	554	554	27888.68	81523.72	81523.72	77009	15.08
Total Statistic	8664	8563	201699.02	603901.39	198975.50	594908.50	203.3

Table 2. Sensitive Behavior Subgraph Statistic Infomation

Class	Apps	Function Call Graph					
		# Graph	Avg # Nodes	Avg # Edges	Median # Nodes	Median # Edges	Storage Space (GB)
adPush	1500	1409	536	853	327	385	1.14
artemis	1032	1032	701	1240	136	160	1.09
openconnection	1494	1488	6142	13308	6705	14763	13.75
kuguo	1500	1500	1379	2853	957	1673	3.12
spyagent	528	528	1491	3090	1731	3622	1.19
dzhtny	560	552	6350	15408	6516	15462	5.29
igexin	1500	1500	2912	6949	2707	6495	6.59
leadbolt	554	554	2958	5077	3023	5587	2.46
Total Statistic	8664	8563	22468	48779	22102	48147	34.63

4.3 Effectiveness on Classifying General Malware

The classification results for unobfuscated malware are presented in Table 4.3. In terms of overall performance, the GAT model achieves the best results, with an accuracy of 95.56% and an F1 score of 95.52%, surpassing GraphSAGE (94.73% accuracy), GCN (94.47%), and TAGCN (94.35%). Across specific families, GAT exhibits outstanding performance on most families, particularly on artemis and kuguo, where its accuracy exceeds 99%. However, for adpush and spyagent, the F1 scores are slightly lower (91-93%), suggesting a marginally higher classification difficulty. Notably, on the igexin family, GraphSAGE achieves a higher F1 score (96.21%) compared to GAT (94.77%), highlighting differences in model performance on specific tasks. Overall, the features we selected and the methods we proposed demonstrate robust capabilities in the classification of unobfuscated malware.

Table 3. Unobfuscation Malware Family Classification Result(%)

		GAT				GraphSAGE				GCN				TAGCN			
ID	**Family**	Acc	Pre	Rec	F1	Acc	Pre	Rec	F1	Acc	Pre	Rec	F1	Acc	Pre	Rec	F1
1	adpush	97.81	93.81	92.98	93.39	97.46	95.27	89.05	92.05	97.14	92.54	89.92	91.21	97.15	91.68	91.28	91.48
2	artemis	99.75	98.32	99.63	98.97	99.53	96.44	99.75	98.07	99.72	98.44	99.27	98.86	99.46	95.68	100.00	97.79
3	openconnection	99.53	94.69	98.17	96.40	99.30	92.12	97.80	94.87	99.28	93.41	95.36	94.37	99.24	92.54	95.91	94.20
4	kuguo	99.52	98.42	98.83	98.62	99.30	97.46	98.59	98.02	99.17	96.55	98.74	97.63	99.40	97.75	98.82	98.28
5	spyagent	97.80	94.57	92.76	93.66	97.33	92.04	92.28	92.16	97.68	93.81	92.96	93.38	97.24	93.42	90.52	91.95
6	dzhtny	99.36	91.99	98.38	95.08	98.93	87.45	97.25	92.09	98.92	87.45	97.30	92.11	99.01	89.42	96.21	92.69
7	igexin	98.18	94.65	94.89	94.77	98.67	97.55	94.91	96.21	98.23	94.56	95.42	94.99	98.39	96.13	94.50	95.31
8	leadbolt	99.18	96.98	89.77	93.24	98.93	92.91	89.62	91.24	98.79	96.94	82.86	89.35	98.80	94.50	86.30	90.21
9	Overall	95.56	95.43	95.68	95.52	94.73	93.91	94.91	94.34	94.47	94.21	93.98	93.99	94.35	93.89	94.19	93.99

4.4 Effectiveness on Classifying Obfuscated Malware

Common obfuscation strategies, as shown in Table 4.4, are primarily categorized into Trivial Obfuscation and Non-trivial Obfuscation[27]. We tested the classification capability of our method under malware obfuscation scenarios. The results are presented in Table 5 and Table 6. Experimental results validate the effectiveness and robustness of combining DWFS with GNNs in classifying obfuscated malware families, achieving an overall performance above 92%, with near-perfect classification on families such as artemis and igexin. GAT exhibits excellent performance on unobfuscated data (95.56% accuracy) and maintains a high level on obfuscated data (92.48% accuracy), confirming the obfuscation resistance of features selected by DWFS. However, obfuscation has a more pronounced impact on certain families (e.g., openconnection).

We also found that the performance gaps between different obfuscation strategies are not significant. This can be attributed to two reasons. First, unobfuscated and obfuscated samples are independently sampled—specifically, a certain number of unobfuscated samples are randomly drawn from each family, followed by independently drawing a certain number of samples from the obfuscated sample pool for each obfuscation strategy, without requiring correspondence to specific unobfuscated samples. This approach prevents direct comparison of feature

Table 4. Descriptions of Obfuscators Used in Our Experiments

Obfuscators		Descriptions
Trivial	Repackaging	Unzipping the APK file and re-signing it with a different signing certificate.
	Disassembling and Reassembling	Disassembling the app using a reverse-engineering tool, By disassembling and reassembling the app.
	Manifest Change	This transformation changes the manifest by adding permissions or adding components'capabilities.
	Alignment	This transformation changes the cryptographic hash of an APK file.
Non-trivial	Junk code insertion	Adds code that does not affect the execution of an app.
	Control-flow manipulation	Changes the methods'control flow graph by adding conditions and iterative constructs.
	Members reordering	Changes the order of instance variables or methods in a classes.*dex* file.
	String encryption	Encrypts the strings in *classes.dex* andadds afunction that decrypts the encrypted strings at runtime.
	Identifier Renaming	Renames the instance variables and/or the method names in each Java class with randomly generatednames.
	Class renaming	Renames the classes and/or the packages in an app with randomly generated names.
	Reflection	Transformations convert direct method invocations into reflective calls using the Java reflection API.

changes for the same unobfuscated sample across different obfuscation strategies. Second, the features selected by DWFS exhibit such strong robustness against these obfuscation techniques that the variations in feature vectors across different obfuscation strategies are negligible. We plan to address these issues in future improvements.

Table 5. Obfuscation Malware Family Classification Result In GraphSAGE Model(%)

		Encryption				Rename				Reflection				Trivial			
ID	**Family**	Acc	Pre	Rec	F1	Acc	Pre	Rec	F1	Acc	Pre	Rec	F1	Acc	Pre	Rec	F1
1	adpush	90.02	94.79	90.07	92.37	90.03	94.79	90.07	92.37	90.11	94.79	90.07	92.37	90.11	94.79	90.07	92.37
2	artemis	99.61	96.43	99.61	98.00	99.61	96.43	99.61	98.00	99.61	96.43	99.61	98.00	99.61	96.43	99.61	98.00
3	openconnection	73.23	84.56	73.33	78.55	73.23	84.56	73.33	78.55	73.23	84.56	73.33	78.55	73.23	84.56	73.33	78.55
4	kuguo	91.11	93.52	91.11	92.30	91.11	93.52	91.11	92.30	91.11	93.52	91.11	92.30	91.11	93.52	91.11	92.30
5	spyagent	90.80	93.49	90.80	92.13	90.80	93.49	90.80	92.13	90.80	93.49	90.80	92.13	90.80	93.49	90.80	92.13
6	dzhtny	97.47	91.41	97.47	94.34	97.47	91.41	97.47	94.34	97.47	91.41	97.47	94.34	97.47	91.41	97.47	94.34
7	igexin	98.80	96.64	98.80	97.71	98.80	96.64	98.80	97.71	98.80	96.64	98.80	97.71	98.80	96.64	98.80	97.71
8	leadbolt	96.60	79.01	96.60	86.93	96.60	79.01	96.60	86.93	96.60	79.01	96.60	86.93	96.60	79.01	96.60	86.93
9	Overall	92.26	91.23	92.22	91.54	92.26	91.23	92.22	91.54	92.26	91.23	92.22	91.54	92.26	91.23	92.22	91.54

5 Conclusion and Future Work

This paper integrates DWFS with GNNs to construct an efficient and robust malware family detection system. DWFS filters out obfuscation-resistant features from a vast feature pool, while GNNs leverage these features and the program's function call graph for deep learning. This approach not only improves detection accuracy but also significantly enhances the system's ability to counter code obfuscation techniques. In the future, we will try to adjust the DWFS strategy, incorporate more sensitive characteristics, combine the feature

Table 6. Obfuscation Malware Family Classification Result In GAT Model(%)

		Encryption				Rename				Reflection				Trivial			
ID	**Family**	Acc	Pre	Rec	F1	Acc	Pre	Rec	F1	Acc	Pre	Rec	F1	Acc	Pre	Rec	F1
1	adpush	94.59	93.64	94.49	94.07	94.51	93.64	94.49	94.07	94.44	93.64	94.49	94.07	94.44	93.64	94.49	94.07
2	artemis	99.61	97.35	99.61	98.47	99.61	97.35	99.61	98.47	99.61	97.35	99.61	98.47	99.61	97.35	99.61	98.47
3	openconnection	67.72	76.53	67.72	71.85	67.72	76.53	67.72	71.85	67.72	76.53	67.72	71.85	67.72	76.53	67.72	71.85
4	kuguo	92.74	95.08	92.74	93.90	92.74	95.08	92.74	93.90	92.74	95.08	92.74	93.90	92.74	95.08	92.74	93.90
5	spyagent	89.85	96.30	89.85	92.96	89.85	96.30	89.85	92.96	89.85	96.30	89.85	92.96	89.85	96.30	89.85	92.96
6	dzhtny	99.54	92.01	99.54	95.63	99.54	92.01	99.54	95.63	99.54	92.01	99.54	95.63	99.54	92.01	99.54	95.63
7	igexin	99.20	98.96	99.20	99.08	99.20	98.96	99.20	99.08	99.20	98.96	99.20	99.08	99.20	98.96	99.20	99.08
8	leadbolt	96.60	88.28	96.60	92.25	96.60	96.60	88.28	96.60	96.60	88.28	96.60	92.25	96.60	88.28	96.60	92.25
9	Overall	92.48	92.27	92.47	92.28	92.48	92.27	92.47	92.28	92.48	92.27	92.47	92.28	92.48	92.27	92.47	92.28

selection algorithm with different depth learning methods, and further distinguish the confusion method.

References

1. AV-TEST - The Independent IT-Security Institute, "AV-ATLAS - Malware & PUA — portal.av-atlas.org," https://portal.av-atlas.org/malware/statistics, [Accessed 22-01-2025].
2. D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck, "DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket," in *Symposium on Network and Distributed System Security (NDSS)*, Feb. 2014, doi: 10.14722/ndss.2014.23247.
3. Y. He, Y. Liu, L. Wu, Z. Yang, K. Ren, and Z. Qin, "MsDroid: Identifying Malicious Snippets for Android Malware Detection," *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 3, pp. 2025–2039, 2023, doi: 10.1109/TDSC.2022.3168285.
4. Y. Wu, X. Li, D. Zou, W. Yang, X. Zhang, and H. Jin, "MalScan: Fast Market-Wide Mobile Malware Scanning by Social-Network Centrality Analysis," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 139–150, doi: 10.1109/ASE.2019.00023.
5. J. Gong, W. Niu, S. Li, M. Zhang, and X. Zhang, "Sensitive Behavioral Chain-Focused Android Malware Detection Fused With AST Semantics," *IEEE Transactions on Information Forensics and Security*, vol. 19, pp. 9216–9229, 2024, doi: 10.1109/TIFS.2024.3468891.
6. C. Li, Q. Lv, N. Li, Y. Wang, D. Sun, and Y. Qiao, "A novel deep framework for dynamic malware detection based on API sequence intrinsic features," *Computers & Security*, vol. 116, p. 102686, 2022, doi: 10.1016/j.cose.2022.102686. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404822000840.
7. O. Or-Meir, N. Nissim, Y. Elovici, and L. Rokach, "Dynamic Malware Analysis in the Modern Era—A State of the Art Survey," *ACM Comput. Surv.*, vol. 52, no. 5, art. no. 88, Sep. 2020, doi: 10.1145/3329786. [Online]. Available: https://doi.org/10.1145/3329786.
8. Y. Wu, S. Dou, D. Zou, W. Yang, W. Qiang, and H. Jin, "Contrastive Learning for Robust Android Malware Familial Classification," *IEEE Transactions on Dependable and Secure Computing*, pp. 1–14, 2022, doi: 10.1109/TDSC.2022.3219082.
9. M. Fan, J. Liu, X. Luo, K. Chen, Z. Tian, Q. Zheng, and T. Liu, "Android Malware Familial Classification and Representative Sample Selection via Frequent Subgraph Analysis," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 8, pp. 1890–1905, 2018, doi: 10.1109/TIFS.2018.2806891.

10. Y. Feng, S. Anand, I. Dillig, and A. Aiken, “Apposcopy: semantics-based detection of Android malware through static analysis,” in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*, New York, NY, USA: Association for Computing Machinery, 2014, pp. 576–587, doi: 10.1145/2635868.2635869. [Online]. Available: https://doi.org/10.1145/2635868.2635869.
11. V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden, “Mining Apps for Abnormal Usage of Sensitive Data,” in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015, pp. 426–436, doi: 10.1109/ICSE.2015.61.
12. O. Mirzaei, J. M. de Fuentes, J. Tapiador, and L. Gonzalez-Manzano, “AndrODet: An adaptive Android obfuscation detector,” *Future Generation Computer Systems*, vol. 90, pp. 240–261, 2019, doi: 10.1016/j.future.2018.07.066. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X18309312.
13. Y. Wang, H. Wu, H. Zhang, and A. Rountev, “Orlis: Obfuscation-Resilient Library Detection for Android,” in *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, 2018, pp. 13–23.
14. J. Garcia, M. Hammad, and S. Malek, “Lightweight, obfuscation-resilient detection and family identification of Android malware,” in *Proceedings of the 40th International Conference on Software Engineering (ICSE ’18)*, New York, NY, USA: Association for Computing Machinery, 2018, p. 497, doi: 10.1145/3180155.3182551. [Online]. Available: https://doi.org/10.1145/3180155.3182551.
15. X. Sun, Y. Liu, M. Xu, H. Chen, J. Han, and K. Wang, “Feature selection using dynamic weights for classification,” *Knowledge-Based Systems*, vol. 37, pp. 541–549, 2013, doi: 10.1016/j.knosys.2012.10.001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950705112002699.
16. J. Vitorino, M. Silva, E. Maia, and I. Praça, “Reliable Feature Selection for Adversarially Robust Cyber-Attack Detection,” *CoRR*, vol. abs/2404.04188, 2024, doi: 10.48550/arXiv.2404.04188. [Online]. Available: https://doi.org/10.48550/arXiv.2404.04188.
17. Y. Wu, M. Li, Q. Zeng, T. Yang, J. Wang, and Z. Fang, “DroidRL: Feature selection for android malware detection with reinforcement learning,” *Computers & Security*, vol. 128, p. 103126, 2023, doi: 10.1016/j.cose.2023.103126. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404823000366.
18. “Dalvik executable instruction formats | Android Open Source Project — source.android.google.cn,” https://source.android.google.cn/docs/core/runtime/instruction-formats?hl=en, [Accessed 07-04-2025].
19. L. Gong, H. Lin, Z. Li, F. Qian, Y. Li, X. Ma, and Y. Liu, “Systematically Landing Machine Learning onto Market-Scale Mobile Malware Detection,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 7, pp. 1615–1628, 2021, doi: 10.1109/TPDS.2020.3046092.
20. A. Mathur, L. M. Podila, K. Kulkarni, Q. Niyaz, and A. Y. Javaid, “NATICUSdroid: A malware detection framework for Android using native and custom permissions,” *Journal of Information Security and Applications*, vol. 58, p. 102696, 2021, doi: 10.1016/j.jisa.2020.102696. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2214212620308437.
21. K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, “PScout: analyzing the Android permission specification,” in *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS ’12)*, New York, NY, USA: Association for

Computing Machinery, 2012, pp. 217–228, doi: 10.1145/2382196.2382222. [Online]. Available: https://doi.org/10.1145/2382196.2382222.

22. V. K. V and J. C. D, “Android Malware Detection using Function Call Graph with Graph Convolutional Networks,” in *2021 2nd International Conference on Secure Cyber Computing and Communications (ICSCCC)*, 2021, pp. 279–287, doi: 10.1109/ICSCCC51823.2021.9478141.
23. “Apktool | Apktool — apktool.org,” https://apktool.org/, [Accessed 07-04-2025].
24. “GitHub - androguard/androguard: Reverse engineering and pentesting for Android applications — github.com,” https://github.com/androguard/androguard, [Accessed 07-06-2024].
25. X. Lu, J. Zhao, and P. Lio, “Robust android malware detection based on subgraph network and denoising GCN network,” in *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services (MobiSys ’22)*, New York, NY, USA: Association for Computing Machinery, 2022, pp. 549–550, doi: 10.1145/3498361.3538778. [Online]. Available: https://doi.org/10.1145/3498361.3538778.
26. K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, “AndroZoo: Collecting Millions of Android Apps for the Research Community,” in *Proceedings of the 13th International Conference on Mining Software Repositories (MSR ’16)*, New York, NY, USA: ACM, 2016, pp. 468–471, doi: 10.1145/2901739.2903508. [Online]. Available: http://doi.acm.org/10.1145/2901739.2903508.
27. M. Hammad, J. Garcia, and S. Malek, “A large-scale empirical study on the effects of code obfuscations on Android apps and anti-malware products,” in *Proceedings of the 40th International Conference on Software Engineering (ICSE ’18)*, New York, NY, USA: Association for Computing Machinery, 2018, pp. 421–431, doi: 10.1145/3180155.3180228. [Online]. Available: https://doi.org/10.1145/3180155.3180228.
28. S. Aonzo, G. C. Georgiu, L. Verderame, and A. Merlo, “Obfuscapk: An open-source black-box obfuscation tool for Android apps,” *SoftwareX*, vol. 11, p. 100403, 2020, doi: 10.1016/j.softx.2020.100403. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352711019302791.