future internet

MDPI

Article

DGDroid: Android Malware Detection Using Dual-Graph Optimization

Xiangjun Li [1,2], Hanwen Zhang [1,2,*], Jingyao Wei [3], Chunfang Chen [1,2] and Fanyu Wen [4]

[1] School of Mathematics and Computer Science, Nanchang University, Nanchang 330019, China; lixiangjun@ncu.edu.cn (X.L.); chenchunfang@ncu.edu.cn (C.C.)
[2] Jiangxi Provincial Key Laboratory of Data Security Technology, Nanchang 330047, China
[3] School of Public Policy and Administration, Nanchang University, Nanchang 330047, China; 402700260086@email.ncu.edu.cn
[4] School of Industrial Software, Jiangxi University of Science and Technology, Nanchang 330013, China; 1120245663@mail.jxust.edu.cn
* Correspondence: 359100230010@email.ncu.edu.cn; Tel.: +86-13330115032

Abstract

Android malware continues to threaten user privacy and data security. Static API-based detection uses code-level evidence, but a single relation granularity cannot retain local API connections, class combinations and their repetition equally well. We present DGDroid, a static Android malware detector that constructs within-method API adjacency and class co-occurrence graphs. Binary features preserve relation identity, while saturated support features encode repetition across methods. Shared method sampling and cross-view alignment support joint pretraining, after which the encoders are frozen and their standardized outputs are fused with the explicit relation features. DGDroid achieves accuracies of 97.57%, 97.82% and 99.26% on Drebin, CIC2017 and CIC2020, respectively. Under the tested independent-method insertions, the complete model achieves higher F1 than either single-view model and retains strong detection performance in most random relation-deletion settings.

Keywords: Android malware detection; API relations; dual graphs; relation support; contrastive learning

1. Introduction

Android applications can conceal malicious operations through legitimate-looking functionality and code obfuscation, threatening user privacy and device security. Existing detectors differ in how they extract features, organize evidence and classify applications [1,2]. Some connect predictions to specific code operations to explain the behavior underlying a decision [3]. Changes in malware alter detector inputs, while sample quality and evaluation design affect measured performance [4,5]. Extracting discriminative code evidence and organizing it into representations that retain relevant relationships therefore remain central to static Android malware detection.

Static detectors use feature selection to reduce redundant inputs [6], bytecode images to learn patterns from code content [7,8], and API sequences to preserve call order [9]. Graph and multi-view methods further organize relations and combine structural and content features [10,11]. Their information coverage depends on how the input relations are defined. Graph pooling compresses nodes and edges into a fixed-length vector,

Academic Editor: Xueyi Li

Received: 21 October 2026
Revised: 21 November 2026
Accepted: 25 November 2026
Published: 26 November 2026

Copyright: © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

whereas explicit coordinates can retain the identity of individual relations and the number of methods supporting them. When two graph views derive from the same method records, their augmentations can also be paired through this shared code content. Multimodal pretraining has aligned static and dynamic observations [12], providing a basis for studying correspondence between relation views from one static source. We develop DGDroid around these two requirements: preserving explicit relation evidence and pairing graph augmentations through their methods of origin. Its two contributions are as follows.

1. We construct directed API adjacency and undirected class co-occurrence relations within methods, and separately encode relation presence and repeated support across methods. The resulting representation retains local API order, class combinations and repetition as complementary evidence for malware detection.

2. We combine explicit relations with neighborhood context learned by graph encoders. We use shared method sampling to pair the two views during pretraining, then fuse the frozen graph representations with explicit relation-presence and support features for classification. The framework achieves 97.57%, 97.82% and 99.26% accuracy on Drebin, CIC2017 and CIC2020. Component ablations and structural perturbations examine how the different information sources contribute.

2. Related Work

2.1. Static Feature-Based Detection

Static detectors analyze permissions, code and resources in APKs. PermGuard maps permissions to exploitation techniques to describe risk-related associations [13]. FSSDroid uses binarization and subset selection to reduce redundant inputs [6], while DroidRL treats feature selection as a sequential decision problem and searches feature combinations through reinforcement learning [14]. These approaches improve the use of extracted information. Permissions, however, primarily describe requested access, and similar permissions can serve different functions. The order and relationships of code operations provide additional evidence for distinguishing those functions.

Wang et al. extract API sequences through multigraph simplification and pruned search to preserve code organization while reducing redundant traversal [9]. MOSDroid represents method-level opcode sequences as multisets [15]. To accommodate API changes, AMDASE combines semantic clustering with call-graph optimization [16], while MDADroid constructs functionality–application–API relations to maintain functionality mappings [17]. Sequence and method representations preserve local organization, whereas functional abstraction groups APIs with related roles. Because abstraction merges some distinctions between concrete calls, API-level relations and class-level combinations offer complementary granularities.

Feature fusion expands the information available to a detector. MSerNetDroid combines permissions, APIs and hardware features through multi-head squeeze-and-excitation residual modules [18]. Li et al. combine sensitive-API code snippets with binary images and learn fine-grained interactions through multi-head attention [19]. FARO-Droid adjusts feature weights and models interactions to account for changes introduced by obfuscation [20]. These methods show how input reliability and feature interactions shape the value of fusion.

2.2. Graph-Based Detection

Graph-based detectors organize code entities and their relations so that models can use neighborhood structure. SeGDroid prunes function-call graphs around sensitive APIs and retains semantic and structural attributes [21]. SNDGCN combines subgraph

reconstruction with denoising graph convolution [22], and SIGDroid transforms heterogeneous function-call graphs into attribute-enriched homogeneous graphs before multistage knowledge distillation [23]. These approaches refine the input structure or its learning process. Within-method class co-occurrence supplies a further relation granularity alongside concrete API adjacency.

GDroid models information shared across applications using an App–API and API–API heterogeneous graph [10]. HAWK combines meta-structures with incremental learning [24], Z2F aggregates permissions, APIs, packages and classes through meta-structures [25], and SimDroid couples adaptive aggregation with incremental similarity search [26]. Applications exchange information through shared entities and higher-order relations, and new applications must be linked to the existing graph. Preserving method boundaries within each application additionally identifies the local code scope from which a relation originates.

Multi-view approaches combine different organizations of code evidence. DeepCatra derives opcode sequences and flow graphs from static call traces [27], while SAC fuses DEX images with hierarchical graph representations [11]. The correspondence between such views depends on how each is constructed. For API and class graphs derived from the same methods, method membership supplies a direct pairing criterion, while explicit coordinates preserve relations alongside their pooled graph representations.

Pretraining learns representations before supervised classification. GraphCL forms graph augmentations through node deletion, edge perturbation, attribute masking and subgraph sampling [28]. Yang et al. learn unlabeled application representations using a BYOL-based mechanism [29], and MaskDroid combines masked reconstruction with contrastive learning [30]. These methods tie the learning task to changes in the graph input. MPDroid further aligns dynamic API graphs with static function-call graphs during multimodal pretraining, then uses static features downstream [12].

GDroid emphasizes relations shared across applications [10], SAC combines structure and content [11], and MPDroid aligns static and dynamic observations [12]. MaskDroid learns masked function-call-graph representations [30]. DGDroid instead derives API adjacency and class co-occurrence from the same static method sequences. It adapts graph contrastive augmentation [28] by sampling common method content and retains repeated relation support outside graph-level pooling.

3. Method

The discriminative value of static API evidence depends on its relation granularity and organization. DGDroid (Dual-Graph Android) constructs two relation views from one static code source and combines explicit evidence with learned graph context. Representation pretraining is followed by fusion-based classification. The classifier minimizes

$$\mathcal{L}_{cls} = -\frac{1}{\sum_{i=1}^{N} \omega_{y_i}} \sum_{i=1}^{N} \omega_{y_i} [y_i \, log \, p_i + (1 - y_i) \, log(1 - p_i)] + \frac{\lambda_{reg}}{2} \|w\|_2^2 \quad (1)$$

where N is the number of classifier-fitting applications, y_i is the label (0 for benign and 1 for malware), $p_i = \sigma(w^T h_i + b)$ is the predicted malware probability, and h_i is the fused feature vector. The parameters w and b are the classifier weights and intercept. The class weight ω_{y_i} is determined by the fitting class counts, and λ_{reg} controls regularization. This objective fits the classifier. The graph encoders are learned through the pretraining objective defined below.

Figure 1 summarizes the framework. We extract API sequences while preserving method boundaries, construct API adjacency and class co-occurrence graphs, and encode relation presence and repeated support. Shared method sampling is used to learn graph

representations, which are then frozen and combined with the explicit features for classification.

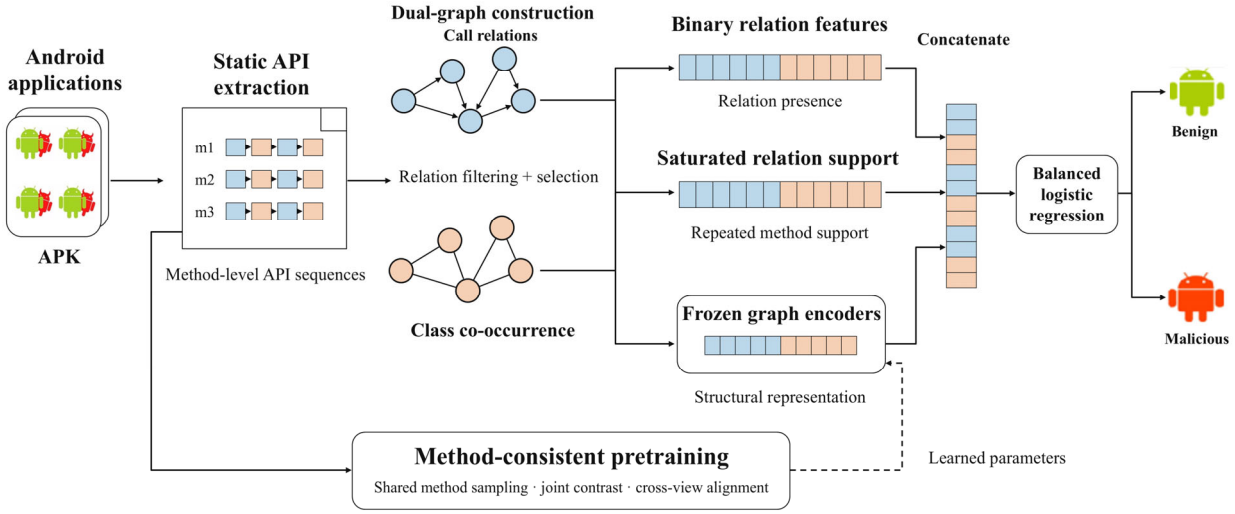


Figure 1. Overview of DGDroid. The C view represents static API adjacency within methods.

3.1. API Representation

Methods define the local scope of the relations. Let $A = \{a_1, a_2, \ldots, a_n\}$ be the APK collection, F(a) the files parsed from application a, and M(f) the methods in file f. The method records M(a) are collected across those files. For method m, the API sequence is $S_m = (u_{m,1}, u_{m,2}, \ldots, u_{m,l})$. Each identifier contains the API owner class and method name, and $\pi(u)$ gives the owner class of API u. This representation preserves API positions and method membership.

The simplified smali method in Figure 2 contains Context.getContentResolver, Settings$Secure.getString and TextUtils.isEmpty. The extractor reads the class and method targeted by each invocation, while register assignments and constants provide context in the example. API positions follow static code order. Context and getContentResolver jointly identify one API, while the class mapping groups APIs with the same owner. A single method record therefore supplies both concrete API identifiers and the coarser class information needed for the two views.

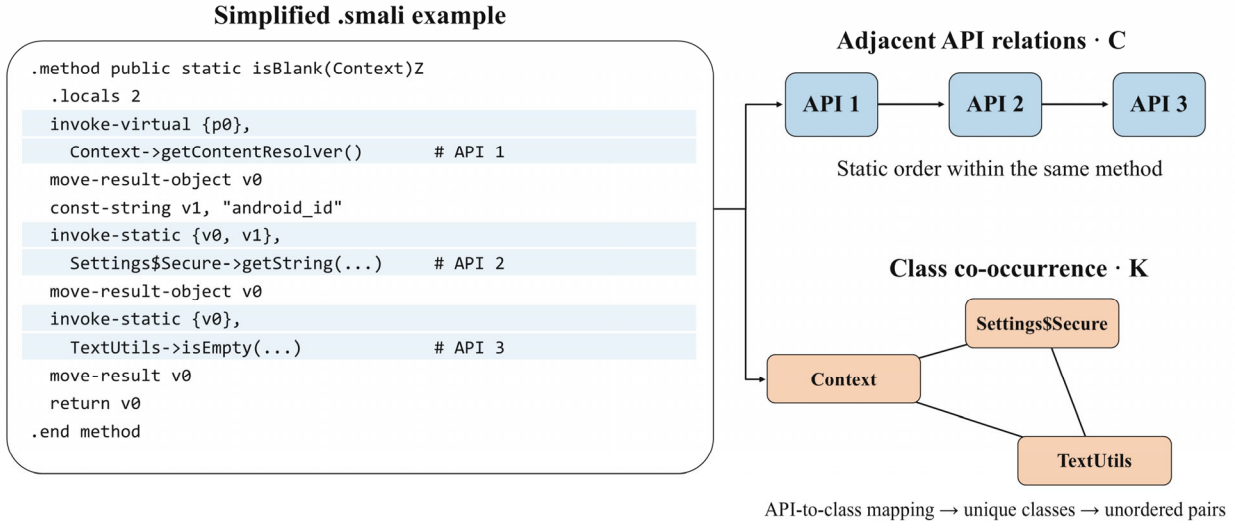


Figure 2. A simplified smali method and its API adjacency and class co-occurrence relations.

We build a shared identifier space from the fitting subset. API document frequencies are counted per application, filtered, and ranked by the difference between benign and malware occurrence proportions. Fixed owner-class and method-name rules add sensitive APIs, after which the class vocabulary is constructed. Repeated occurrences within an application contribute once to document frequency and remain in the method records for later support encoding. All applications use the same indices, with original positions and vocabulary membership retained. Configuration details are described in Section 4.3 and the released implementation.

3.2. Relation Construction

3.2.1. Class Co-Occurrence Relations

The K graph describes classes appearing in the same method. We map the API sequence to the class vocabulary, deduplicate classes in order of first occurrence, and retain the first H as U_m. Distinct classes are paired as

$$E_m^K = \{\{v, w\} : v, w \in U_m,\ v \prec w\}, \qquad E_a^K = \bigcup_{m \in M(a)} E_m^K. \tag{2}$$

The fixed order $\prec$ gives each unordered class pair a canonical identifier. The edge set E_a^K and its endpoints define the application graph G_a^K. APIs with the same owner share a class node, so the view describes combinations of API classes. Limiting the retained classes bounds pair enumeration within a method. The chosen bound and its observed effect are reported in the experimental settings.

Class deduplication makes each class participate once in a pair within a method. The example in Figure 2 yields Context–Settings$Secure, Context–TextUtils and Settings$Secure–TextUtils. Identical pairs from different methods form one application-level edge, while their supporting method occurrences are counted separately. Candidate relations are selected on the training side by per-application document frequency and class-conditional occurrence differences. Chi-square scores then select the binary co-occurrence coordinates T^K. These ordered coordinates define both the selected edges and the explicit feature indices.

Selection establishes coordinates for graph encoding and explicit classification. Let T_{enc}^v denote the encoder relations and T_{cls}^v the final explicit relations, with v ∈ {C,K}. Both are established on the fitting subset during development. At final fitting, the explicit branch repeats chi-square selection on the combined fitting and validation subsets. The branches retain the same relation definitions, with separate selected coordinates.

3.2.2. API Adjacency Relations

The C graph records directed adjacency in the static API sequence of each method. We scan consecutive original positions and connect distinct identifiers when both belong to the API vocabulary:

$$E_m^C = \{(u_{m,t}, u_{m,t+1}) : 1 \le t < l_m,\ u_{m,t}, u_{m,t+1} \in V_{\text{API}},\ u_{m,t} \ne u_{m,t+1}\}. \tag{3}$$

Edge direction follows the order of the two calls. A.a → B.b → A.c produces two edges when all three APIs are retained. If B.b is excluded, its position breaks adjacency. We construct edges within each method and take their union to form G_a^c. The API branch follows the same selection procedure as K, yielding the ordered relation set T^c. Forward and reverse edges occupy separate coordinates.

In Figure 2, the C view links getContentResolver, getString and isEmpty in their static sequence. Out-of-vocabulary positions act as separators, preserving adjacency in the original method. Shared API nodes unite the local edges into an application graph. An

https://doi.org/10.3390/fi18120655

excluded API can still contribute its owner class to K if that class remains in the vocabulary. Thus, C represents within-method static adjacency rather than inter-method calls or runtime control flow. For either view, let $n_{a,e}^{v}$ count method records supporting selected relation e in application a, with one count per relation per method and v ∈ {C,K}. Binary presence x and saturated support s are encoded as

$$x_{a,e}^{v} = \mathbf{1}[n_{a,e}^{v} > 0], \qquad s_{a,e}^{v} = \begin{cases} 0, & n_{a,e}^{v} = 0, \\ 1 - 1/n_{a,e}^{v}, & n_{a,e}^{v} \geq 1. \end{cases} \tag{4}$$

The binary term retains relation presence, while saturated support records repetition across methods with diminishing increments for frequent relations. They share the selected coordinates and enter classification as explicit features alongside graph context. Support is counted over the extracted method records, including application and third-party-library code.

A relation supported by one method has a binary value of 1 and a support value of 0. With additional supporting methods, the binary value stays fixed and support increases. This keeps presence and repetition in separate coordinates. Counts use the original method records, whereas pretraining samples deduplicated sequences to form structural augmentations.

3.3. Dual-Graph Fusion

3.3.1. Dual-Graph Contrastive Pretraining

Both views derive from method-level API sequences. We therefore use method sequences as the common sampling unit, keeping graph augmentations linked to their code of origin. Building on graph contrastive learning [28], we combine a joint-representation objective with application-level cross-view alignment (Figure 3).

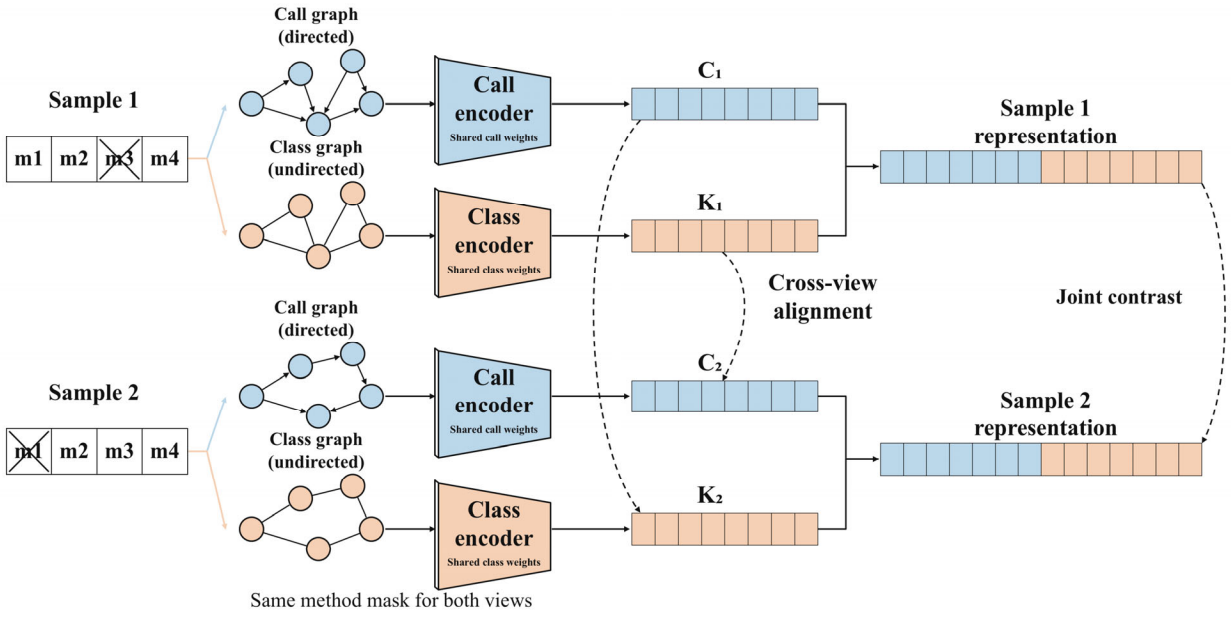


Figure 3. Method-consistent dual-graph pretraining. Samples 1 and 2 are two augmentations of the same APK. The C view labeled call graph represents static API adjacency within methods.

We first deduplicate identical API method sequences within an application and record the selected C and K relations supported by each unique sequence. Relevant sequences are retained independently with probability ϱ, using the same retention mask

for both views. Two sampling passes produce the two paired graph augmentations in Equation (5).

If sampling removes every sequence from a nonempty relevant set, one original sequence is retained at random. Each view is then formed from the relations supported by the retained sequences. Applications with an originally empty selected graph pair enter supervised classification with zero graph features, while contrastive pretraining uses applications containing selected relations.

$$E_{a,r}^{v} = \bigcup_{q \in Q_a:\, b_{a,q}^{(r)}=1} (E_q^v \cap T_{\text{enc}}^v), \qquad v \in \{C, K\},\ r \in \{1,2\} \tag{5}$$

Here Q_a is the set of relevant unique method sequences, b is the retention indicator, and Tenc^v is the selected encoder relation set. The same retained methods determine the two views. A relation supported by several sequences has more opportunities to survive sampling.

Samples 1 and 2 in Figure 3 are two augmentations of the same APK, with C_1/K_1 and C_2/K_2 derived from the corresponding method samples. Retaining a method retains its supported edges in both views. Removing it leaves relations supported by other retained methods available. Sampling therefore changes graph structure through method content while preserving the node definitions and edge directions of each view.

The encoders use neighborhood message aggregation [31]. Separate parameters embed API and class nodes. Each encoder concatenates incoming and outgoing neighbor messages with the node representation, applies two nonlinear mappings, and pools active nodes into a graph vector. Messages are normalized by square-root degree factors, and pooling uses a fixed scale estimated from the mean active-node count on the fitting subset. The C encoder aggregates incoming and outgoing edges separately. K expands each undirected relation in both directions and uses the same aggregation form. Parameters are shared between the two augmentations of a view, while the API and class branches learn independently.

The joint objective pairs concatenated C/K vectors from the two augmentations of an application. Cross-view alignment pairs the C vector from one augmentation with the K vector from the other. Both use bidirectional InfoNCE terms [32], giving

$$\begin{aligned} \mathcal{L}_{\text{pre}} &= \mathcal{L}_{\text{joint}} + \lambda \mathcal{L}_{\text{cross}}, \\ \mathcal{L}_{\text{cross}} &= 1/2\,[\mathcal{L}_{\text{NCE}}(Z_C^{(1)}, Z_K^{(2)}) + \mathcal{L}_{\text{NCE}}(Z_K^{(1)}, Z_C^{(2)})]. \end{aligned} \tag{6}$$

Z denotes the batch representations, and λ weights cross-view alignment. Before sampling, applications with identical joint binary graphs are assigned to the same group. Other instances in that group are excluded from the negative set. The joint term keeps the combined application representation consistent across augmentations, while the cross-view term relates the two granularities at graph level. Explicit coordinates retain the identities of the underlying relations.

Cross-view alignment is evaluated when all four C/K subgraphs from the two augmentations are nonempty. Batches containing a single original graph group are skipped. The designated positive and eligible negatives enter the normalized similarity objective. Together, shared sampling defines augmentation content, the two objectives define the learned correspondence, and graph grouping defines the comparison set.

3.3.2. Feature Fusion and Classification

After pretraining, the encoders are frozen and applied to the selected application graphs before sampling (Figure 4). Graph vectors summarize neighborhood context, while explicit features preserve relation identity and repeated support. The graph vectors are standardized and scaled using fitting-subset statistics before concatenation with the relation features. The fused classifier input is

$$h_a = [x_a^C; x_a^K; s_a^C; s_a^K; \tilde{z}_a] \quad (7)$$

The standardized graph vector is $\tilde{z_a}$ = Standardize($[z_a^C; z_a^K]$)/4, and the binary and support components use T_{cls}^v. Logistic regression optimizes Equation (1) and outputs malware probability. During final fitting, the encoder coordinates T_{enc}^v, encoder weights, pooling scale and standardization statistics remain fixed. The explicit branch repeats per-view chi-square selection on fitting plus validation samples to obtain T_{cls}^v, updates its binary and support coordinates, and fits the classifier with the frozen graph vectors. The factor 1/4 sets their input scale in the regularized classifier.

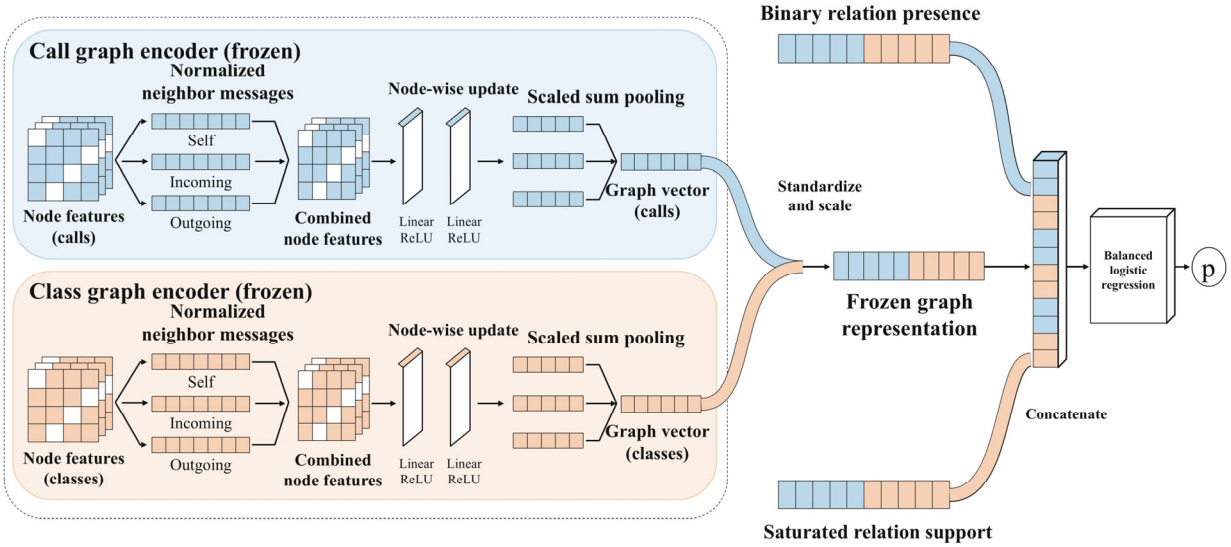


Figure 4. Frozen graph encoding and feature fusion. The C encoder aggregates static API adjacency, and the K encoder aggregates API-owner-class co-occurrence within methods.

4. Experiments and Evaluation

4.1. Datasets

We use Drebin, CICAndMal2017 (CIC2017) and CICMalDroid2020 (CIC2020). Drebin malware is paired with benign APKs obtained from AndroZoo. Static extraction preserves method boundaries and framework-API positions for the relations in Section 3. Table 1 gives the eligible application counts. A stratified 20% test set is held out, then 15% of the remaining applications are assigned to validation. Fitting, validation and testing therefore account for approximately 68%, 12% and 20% of each dataset. The frozen test sets contain 2102, 413 and 3235 applications. All methods use the same test indices, and the main experiments use seed 42.

Drebin benign and malware applications were not systematically matched by collection date or Android version. All 4038 retained CIC2020 benign identifiers match filenames or aliases in a local benign APK archive containing 4042 files. This collection uses raw APKs and follows a different selection scope from the public processed-log subset. The literature comparison retains all 16,174 CIC2020 applications. Reinspection of 35 previously abnormal extractions recovered complete parsing for three APKs and retained fully decoded methods for the other 32. One CADroid input lacks a manifest, and its corresponding features are encoded as unknown missing values.

Frozen subsets have distinct sample identifiers. Supplementary five-fold stratified group cross-validation groups applications by identical extracted method-sequence sets, with preprocessing, training, and validation selection repeated within each fold. Family-disjoint evaluation on Drebin and CIC2017 assigns malware families to separate fitting,

validation and test sets, while benign applications follow the method-set grouping rule. Evaluation studies show that time and sample distributions affect malware-classification estimates [33,34]. We accordingly report these grouped evaluations separately from the frozen random partition.

Table 1. Sample counts in the three datasets.

Class	Drebin	CIC2017	CIC2020
Benign	5395	1637	4038
Malware	5111	426	12,136

4.2. Evaluation Metrics

Malware is the positive class, and benign applications the negative class. TP and TN count correctly classified malware and benign applications. FP counts benign applications classified as malware, and FN counts missed malware.

We report accuracy, precision, recall and F1 to describe overall classification and error behavior.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

$$F1 = \frac{1}{2}\left(\frac{2TP}{2TP + FP + FN} + \frac{2TN}{2TN + FP + FN}\right)$$

Accuracy measures the fraction of correct decisions. Precision and recall describe malware alerts and malware detection, respectively. F1 is calculated for each class and averaged with equal weights. We also report FPR = FP/(FP + TN), FNR = FN/(FN + TP), and malware-positive PR-AUC computed by trapezoidal integration.

4.3. Experimental Settings

Experiments run on Windows 11 Pro 24H2 with an Intel Core i5-14600KF CPU, 64 GB RAM and an NVIDIA GeForce RTX 5070 GPU. Adam trains the graph encoders, which are then frozen. Their standardized and scaled vectors are concatenated with binary relations and saturated support features for regularized, class-balanced logistic regression (Table 2). After validation selection, the final classifier is fitted on the combined fitting and validation subsets. The explicit branch repeats chi-square selection within its established candidate space, while the graph coordinates, encoders and standardization statistics stay fixed.

The classifier uses L2-regularized liblinear logistic regression with balanced class weights and at most 500 iterations. Validation selects regularization C and classification thresholds of 0.25/0.55 for Drebin, 0.25/0.50 for CIC2017 and 4.00/0.425 for CIC2020. These settings are held fixed during testing and perturbation evaluation.

Pretraining uses seed 42, Adam with zero weight decay, gradient clipping at 10, and the final encoder after 10 epochs. Node and graph representations have eight dimensions per view, with mappings of 24 → 8 → 8 and a 16-dimensional concatenated output. Standardization and pooling statistics are estimated from the fitting subset, and standardized vectors are divided by four. The class bound H = 40 limits pair enumeration and was not reached in the audited records under the three fitting vocabularies. Empty selected graph pairs receive zero relation and graph vectors for classification. Extraction

failures follow the complete-method retention procedure. The released code specifies frequency thresholds, candidate budgets, sensitive-API rules and exception handling.

The CPU benchmark uses the first 128 applications in each frozen test inventory, with identical application order across the five methods. Batch size is eight, with two computation threads and one inter-operation thread. Each method runs sequentially in a fresh process. After one warm-up pass, three timed passes provide median per-application inference latency and peak process memory. Replayed probabilities are checked against saved predictions. A separate 12-APK DGDroid pilot records extraction, feature construction and scoring time from original inputs.

Table 2. Main parameter settings.

Parameter	Setting
Maximum distinct classes per method, H	40
Selected relations per view	Up to 4000
Method retention probability	0.8
Contrastive temperature/alignment weight	0.2/0.25
Representation dimension per view	8
Learning rate/batch size/epochs	0.001/32/10
LR regularization candidates	0.25, 1, 4, 16

4.4. Results

4.4.1. Comparison with Existing Methods

We adapt four literature methods to the same frozen test partitions. MalDozer, TextCNN and CADroid are primarily reconstructed from their papers. GHGDroid reuses the authors' GraphSAGE implementation, with feature and graph construction rebuilt from the paper. MalDozer and CADroid use their final fixed-epoch models, while TextCNN and GHGDroid select models on validation data. The baselines train on the fitting subset. DGDroid is refitted on fitting plus validation after model selection, and a fit-only control examines the effect of this difference.

MalDozer embeds statically extracted API sequences and uses one-dimensional convolutions, global max pooling and fully connected layers to learn local call patterns [35].

GHGDroid constructs a global heterogeneous graph of applications and sensitive APIs. Call frequency, class sensitivity and API-context similarity determine relation weights, and neighborhood aggregation learns application representations [36].

The reconstructed-API TextCNN tokenizes API names, weights their words by class-discriminative information, and orders the resulting representation before applying convolution kernels at multiple scales [37].

CADroid combines APIs, permissions, hardware and content-provider features through cross-combination and multi-head attention, learning interactions among the static feature groups [38].

DGDroid achieved the highest accuracy and F1 among these adapted implementations on all three datasets (Table 3). Compared with CADroid, accuracy increased by 0.81, 1.21 and 0.28 percentage points. GHGDroid uses sensitive APIs and context similarity shared across applications, whereas DGDroid retains concrete adjacency and class combinations within each application. MalDozer reached 97.38% accuracy on Drebin, 0.19 percentage points below DGDroid, with higher malware recall. On CIC2017, validation-only threshold calibration of the frozen MalDozer model increased recall from 67.06% to 77.65% and accuracy from 92.74% to 93.95%, showing that threshold selection accounts for part of the comparison.

Reconstructed-API TextCNN and MalDozer both use convolution, but the former organizes API-name words through weighting and sorting. DGDroid instead uses original within-method positions and owner classes, combining directed adjacency with nonadjacent class co-occurrence. Its fit-only accuracies were 97.62%, 98.31% and 99.04%, so the effect of final fitting scope also varied by dataset. The comparison reflects the stated adaptation, fitting and model-selection procedures. Numerical reproduction of MaskDroid, MPDroid, MaMaDroid, SeGDroid and the original Drebin detector requires their respective call graphs, dynamic traces or additional static inputs and remains outside this comparison.

Table 3. Adapted literature methods and DGDroid on the frozen test partitions (%).

Dataset	Method	Acc. (%)	P (%)	R (%)	F1 (%)
Drebin	MalDozer [35]	97.38	97.83	96.77	97.38
	GHGDroid [36]	83.87	87.17	78.40	83.78
	TextCNN [37]	94.05	97.06	90.52	94.03
	CADroid [38]	96.77	96.68	96.68	96.76
	DGDroid	97.57	98.99	95.99	97.57
CIC2017	MalDozer [35]	92.74	96.61	67.06	87.38
	GHGDroid [36]	93.46	95.31	71.76	88.95
	TextCNN [37]	94.19	94.20	76.47	90.42
	CADroid [38]	96.61	97.33	85.88	94.57
	DGDroid	97.82	94.19	95.29	96.68
CIC2020	MalDozer [35]	96.91	97.39	98.52	95.83
	GHGDroid [36]	92.92	94.93	95.67	90.48
	TextCNN [37]	98.52	98.81	99.22	98.01
	CADroid [38]	98.98	99.14	99.51	98.63
	DGDroid	99.26	99.34	99.67	99.01

Precision and recall use malware as the positive class. F1 is the unweighted mean of benign-class and malware-class F1.

Table 4 gives the full confusion counts. FNR exceeded FPR on Drebin and CIC2017, while CIC2020 showed the reverse. Malware-positive PR-AUC was 0.9962, 0.9796 and 0.9993, providing a threshold-independent view of probability ranking.

Table 4. Classification errors and supplementary metrics of DGDroid.

Dataset	TN	FP	FN	TP	FPR/%	FNR/%	PR-AUC
Drebin	1069	10	41	982	0.93	4.01	0.9962
CIC2017	323	5	4	81	1.52	4.71	0.9796
CIC2020	792	16	8	2419	1.98	0.33	0.9993

FPR = FP/(TN + FP); FNR = FN/(TP + FN). PR-AUC uses trapezoidal integration with malware as the positive class.

Five-fold grouped results are reported in Table 5. Mean DGDroid accuracy was 95.71%, 95.54% and 98.25% on Drebin, CIC2017 and CIC2020. DGDroid's mean accuracy exceeded binary fusion on Drebin and CIC2017. On CIC2020, the corresponding values were 98.25% and 98.74%, respectively. Family-disjoint tests contained 2101 Drebin and 412 CIC2017 applications, covering 33 and eight unseen malware families. DGDroid accuracy was 89.15% and 93.93%, compared with 88.20% and 90.05% for binary fusion. A chronological pilot fitted the models on 2016 applications, selected settings on 2017 applications and evaluated 77 successfully decoded 2018 APKs. DGDroid and binary dual-view fusion achieved accuracies of 88.31% and 90.91%, respectively.

Table 5. Five-fold grouped cross-validation (Accuracy/F1, %).

Protocol	Dataset	Baseline Acc.	Baseline F1	DGDroid Acc.	DGDroid F1
Five-fold grouped CV	Drebin	95.45 ± 1.42	95.44 ± 1.43	95.71 ± 1.44	95.70 ± 1.45
Five-fold grouped CV	CIC2017	95.44 ± 0.71	93.06 ± 1.12	95.54 ± 0.88	93.18 ± 1.40
Five-fold grouped CV	CIC2020	98.74 ± 0.36	98.33 ± 0.48	98.25 ± 0.64	97.69 ± 0.84

Values are mean ± standard deviation across five folds. Baseline denotes binary dual-view fusion. F1 averages the two class-specific F1 scores.

DGDroid had the lowest median prepared-input CPU latency on all three datasets (Figure 5), at 0.619, 0.670 and 0.488 ms per application. It also used the least peak process memory on CIC2017, while TextCNN used slightly less on Drebin and CIC2020. Latency and memory therefore give different rankings of model-stage cost.

The encoders contain approximately 10,000 parameters, and historical logs record 46.7–230.9 s of pretraining. For a method with L API positions and k retained classes, C adjacency scanning costs O(L) and K pair enumeration costs $O(\min(k,H)^2)$. In the separate 12-APK pilot, median total processing time was 78.95 s, dominated by extraction, while feature construction and scoring took 44.99 ms. The logs and pilot describe different stages of the processing cost.

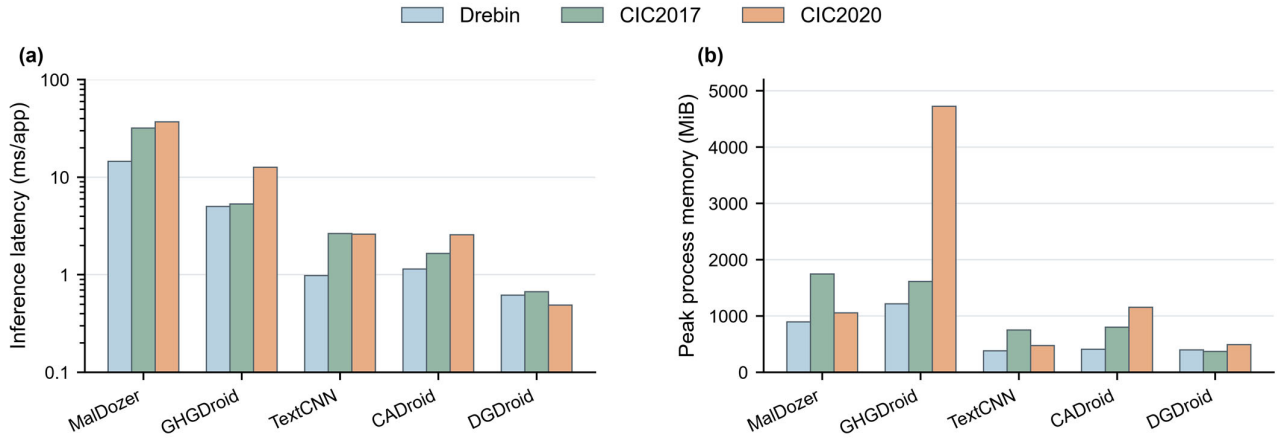


Figure 5. Common CPU benchmark. (**a**) Median prepared-input inference latency over three timed passes after warm-up (logarithmic axis). (**b**) Peak process memory (linear axis). Each dataset uses the same 128 test applications across methods. Batch size is 8, with 2 computation threads and 1 inter-operation thread. Timings exclude APK extraction, input preprocessing, model loading and disk I/O. Memory includes preparation, loaded inputs, models and runtime overhead.

4.4.2. Sensitivity Analysis

We vary one factor at a time on the fixed fitting and validation partition. Relation dimension, method retention probability and alignment weight take values {500,1000,2000,4000,8000}, {0.4,0.6,0.8,1.0} and {0,0.1,0.25,0.5,1.0}, respectively. Defaults are 4000, 0.8 and 0.25. The 4000-relation budget keeps dimensions consistent across datasets and components. Vocabularies and candidate pools remain fixed. Changing dimension updates binary relations, support features and graph inputs together, with the same training budget, seed and validation-selection procedure.

Validation accuracy generally increased with relation dimension (Figure 6a,d). Drebin peaked at 4000, while CIC2017 and CIC2020 peaked at 8000, with different recall trends across datasets. Retaining enough relations supplies discriminative adjacency and class combinations, while the benefit of further expansion depends on their distribution.

CIC2017 and CIC2020 were unchanged across the tested retention probabilities, and Drebin achieved high accuracy near the default (Figure 6b,e). Classification metrics also varied little over the tested alignment weights (Figure 6c,f). Dividing standardized graph vectors by 1, 2, 4 or 8 gave accuracy ranges of 97.34–97.57%, 97.58–97.82% and 99.23–99.29%. The submitted model retains its fixed scale setting.

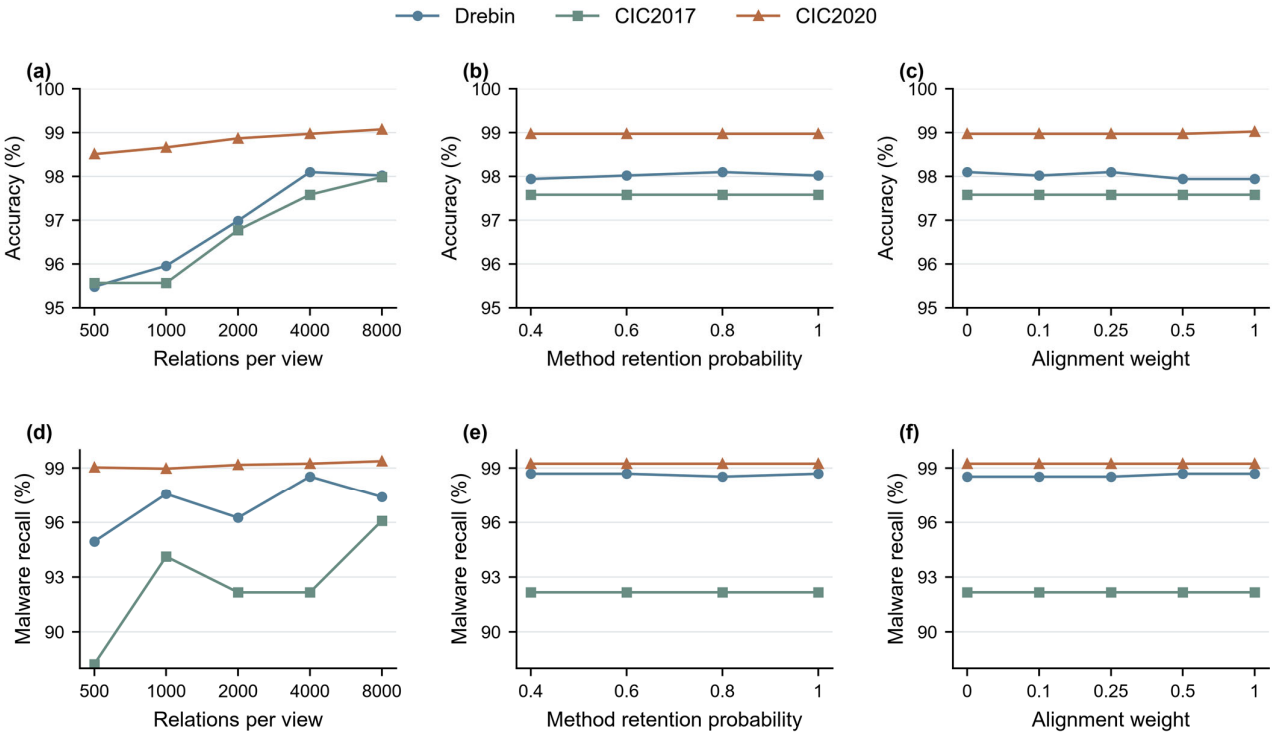


Figure 6. Validation sensitivity. Accuracy is shown in (**a**–**c**), and malware recall in (**d**–**f**), for relation dimension, method retention probability and alignment weight.

4.4.3. Ablation Study

We compare C-only adjacency features, K-only co-occurrence features and complete DGDroid to assess view complementarity (Figure 7). The complete model combines both views, relation support and shared-method pretraining with cross-view alignment.

DGDroid exceeded both single views in accuracy and F1, with larger differences on Drebin and CIC2017 (Figure 7). Table 6 separates binary fusion from five enhancement configurations. On Drebin, support alone matched complete-model accuracy and F1 at two-decimal precision. On CIC2017, alignment increased support-plus-pretraining accuracy by 0.48 percentage points. On CIC2020, aligned pretraining without support reached 99.29%, slightly above the complete model at 99.26%. Repetition and learned context therefore contribute differently across datasets.

DGDroid correctly classified three Drebin applications missed by both single views. For malware f8e1c68f05b65e3b…, C-only and K-only probabilities were 0.18 and 0.21, while DGDroid produced 0.59 against its 0.55 threshold. Binary fusion also missed this application. Controls using random encoders with explicit features achieved 97.53%, 97.58% and 99.20% accuracy. Graph-vector-only classification achieved 72.41%, 89.10% and 94.40%, and independent method sampling achieved 97.53%, 97.82% and 99.23%. Shared sampling added one, zero and one correct decision each in this comparison. Explicit coordinates carried most of the detection signal.

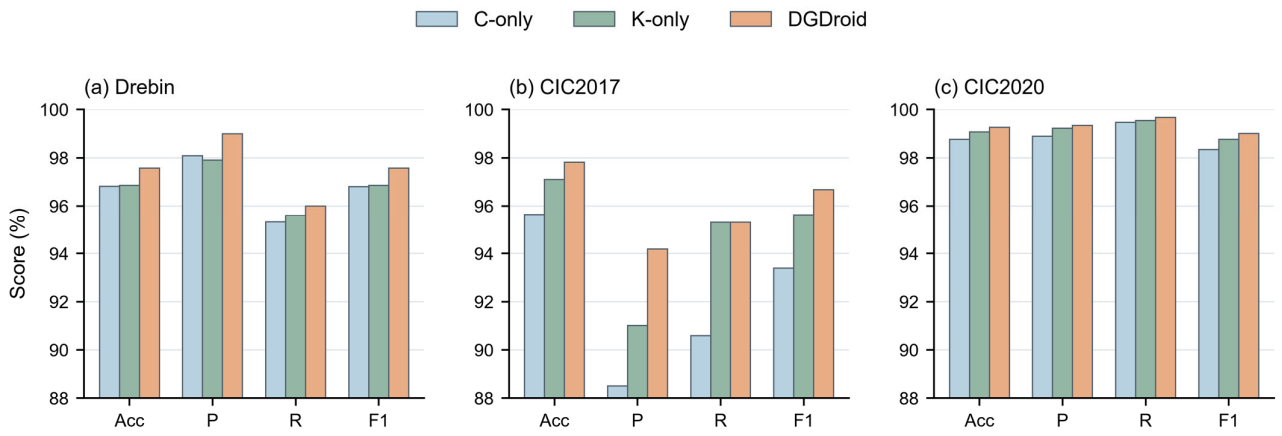


Figure 7. Accuracy, malware precision, malware recall and F1 for the single-view models and DGDroid. All panels use an 88–100% vertical range.

With the partition and classifier settings fixed, encoder seeds 17, 42 and 73 gave accuracy standard deviations of 0.05, 0.00 and 0.03 percentage points. Complete DGDroid added two, three and one correct decision each over binary fusion. The alignment increments on Drebin and CIC2017 corresponded to one and two applications. Exact paired McNemar *p*-values were 0.6875, 0.2500 and 1.0000. Together, the ablation results and paired tests describe how relation support and graph pretraining contribute under each dataset's evaluation setting.

Table 6. Component ablation of relation support, joint pretraining and cross-view alignment (Accuracy/F1, %).

Configuration	Drebin	CIC2017	CIC2020
Binary dual-view baseline	97.48/97.47	97.09/95.63	99.23/98.97
Dual view + support	97.57/97.57	97.34/95.98	99.23/98.97
Dual view + joint pretraining	97.43/97.43	97.34/95.98	99.23/98.97
Support + joint pretraining (no alignment)	97.53/97.52	97.34/95.98	99.26/99.01
Aligned pretraining (no support)	97.43/97.43	97.34/95.98	99.29/99.05
Full DGDroid	97.57/97.57	97.82/96.68	99.26/99.01

4.5. Robustness Analysis

We test relation deletion, independent-method insertion and changes within methods on the frozen test partitions, comparing C-only, K-only and complete DGDroid. Feature spaces, model parameters and thresholds use the clean-input settings. F1 follows Section 4.2. The models share each perturbation instance, then construct their corresponding inputs. Figure 8 shows the original fixed realization. Both-view 20% deletion and all insertion levels are additionally repeated with seeds 17, 42 and 73. Segmentation is deterministic.

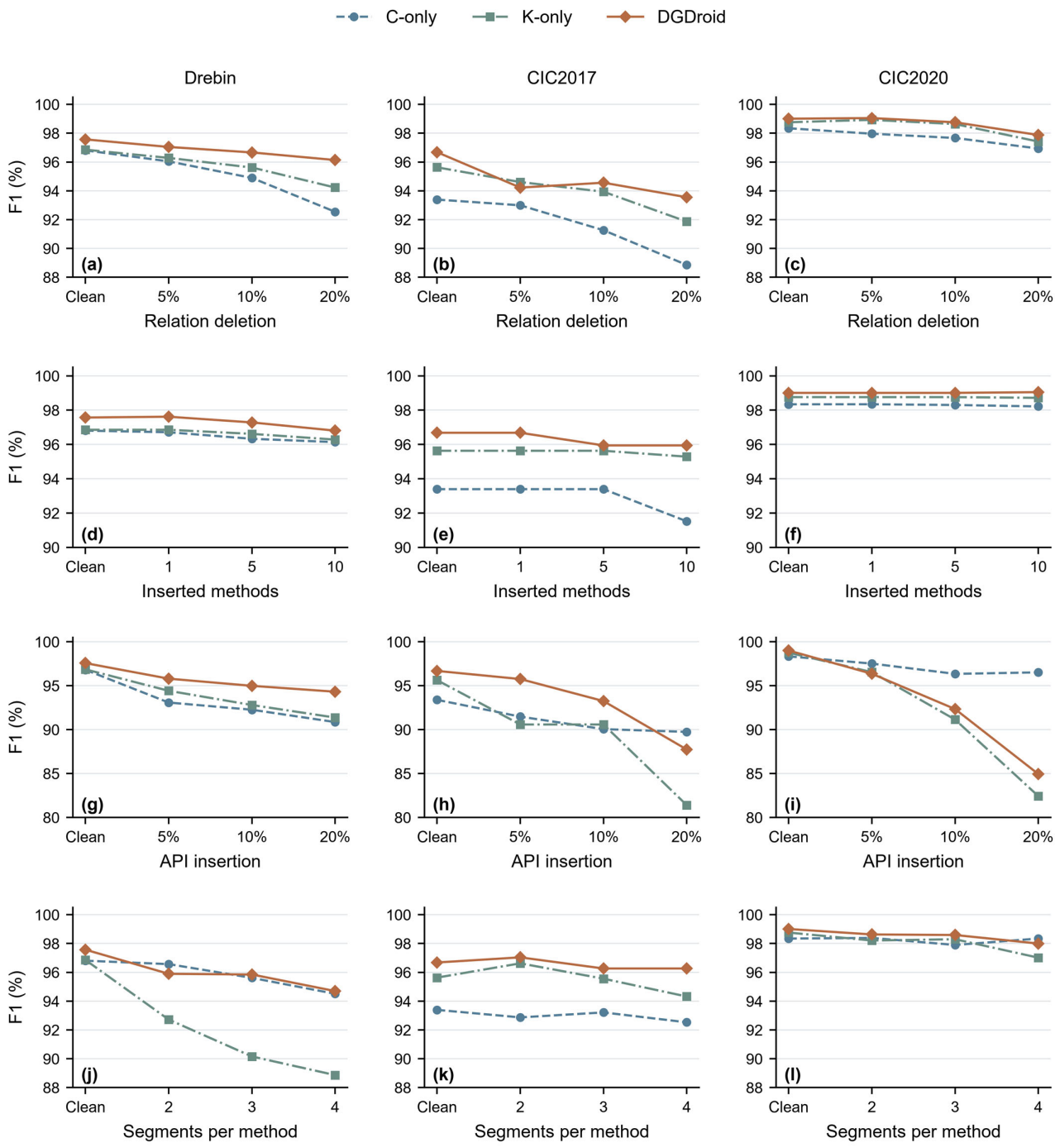


Figure 8. F1 under simultaneous relation deletion (**a–c**), independent-method insertion (**d–f**), within-method API insertion (**g–i**) and segmentation (**j–l**). Columns show Drebin, CIC2017 and CIC2020. Row-wise vertical ranges are 88–100%, 90–100%, 80–100% and 88–100%. Points show the original fixed realization; repeated perturbation summaries are reported in the text.

Random deletion removes C relations, K relations or both at 5%, 10% and 20%. The unaffected single view provides a reference when only one view is perturbed. Under simultaneous deletion, all models had lower F1 at 20% than on clean inputs, with fluctuations at intermediate levels (Figure 8a–c). DGDroid led across tested rates on Drebin and CIC2020. On CIC2017, K-only was slightly higher at 5%, while DGDroid led

at larger rates. Repeated both-view 20% deletion gave DGDroid F1 of 95.82 ± 0.43%, 94.60 ± 1.53% and 97.73 ± 0.12%, above the corresponding single-view means.

Independent-method insertion adds small method representations from benign fitting applications, retains the original methods, and updates both graphs and support features. DGDroid had higher F1 than either single view at every tested level (Figure 8d–f). Performance decreased on Drebin, C-only declined more on CIC2017, and changes were small on CIC2020. Relations are constructed within methods, so independent additions preserve the original local evidence while introducing new API edges and class combinations.

Problem-space evasion studies modify executable APK code and check preserved functionality [39,40]. Here, we first insert in-vocabulary APIs from benign fitting applications into extracted method representations at nominal rates of 5%, 10% and 20%, rounding counts upward before rebuilding relations and support. DGDroid retained higher F1 on Drebin, while C-only led at the largest rate on CIC2017 and was more stable on CIC2020 (Figure 8g–i). Within-method insertion alters existing adjacency and class combinations. A supplementary directed pilot places out-of-vocabulary markers at gaps ranked by C-only edge weights. These markers remove crossing C adjacency while leaving K unchanged, directly examining the effect of adjacency changes on fused decisions.

We also split methods with at least eight APIs into two, three or four consecutive segments, then rebuild each view and its support. K-only decreased more as segment count grew on Drebin, with DGDroid leading at larger counts (Figure 8j–l). DGDroid led at all tested counts on CIC2017, while C-only led at the largest count on CIC2020. Segmentation removes adjacency at new boundaries and co-occurrence across segments, giving different information losses in the two views.

These perturbations alter the scope of the relation evidence in distinct ways. Independent additions preserve local relations, within-method insertion changes adjacency and introduces class combinations, and segmentation removes cross-segment links. The directed pilot uses C-only weights to select gaps in extracted representations. APK reconstruction, runtime behavior and functionality preservation remain to be validated.

5. Discussion

DGDroid retains two granularities of static information through within-method API adjacency and class co-occurrence. Binary coordinates preserve concrete relations, support records their repetition, and graph encoders add neighborhood context. Recovered single-view errors show that fusion changes some application decisions. Freezing the encoders allows this context to supplement explicit relation evidence during classifier fitting.

Explicit relation features supplied most of the measured detection signal, while support and pretrained graph context contributed differently across datasets. Component effects depend on the dataset. Support alone matches complete-model accuracy on Drebin, the complete combination performs better on CIC2017, and removing support from aligned pretraining slightly improves CIC2020 accuracy. Shared method sampling gives the views corresponding code content, with small classification differences from independent sampling in the present control. Prepared-input inference is fast in the common CPU benchmark, while extraction dominates original-APK processing, making front-end analysis an important deployment cost.

The study has several limitations. The main results use a fixed stratified random partition. Grouped and family-disjoint evaluations extend the assessment beyond the original random partition. The chronological pilot offers a first view of later-year

detection, with binary fusion obtaining the higher score in that cohort. Packing, reflection and dynamic loading limit static extraction, and support counts also include library relations. The adapted baselines differ in fitting scope and tuning budgets. Perturbations operate on extracted representations, and executable APK changes require functionality checks. Future evaluation will use successive time cohorts, matched training conditions and verified APK transformations [33,34].

6. Conclusions

DGDroid combines fine-grained API adjacency and class co-occurrence with saturated support and method-consistent cross-view pretraining. It achieved 97.57%, 97.82% and 99.26% accuracy on Drebin, CIC2017 and CIC2020, respectively. Under the reported evaluation protocols, it achieved the highest accuracy and F1 among the evaluated single-view models and adapted literature implementations on the frozen test partitions. The experiments show that explicit relations provide the main detection signal, with support and learned graph context contributing according to the dataset. DGDroid also achieved higher F1 than either single-view model under the tested independent-method insertions. Future work will assess the approach across successive time cohorts and executable, functionality-preserving APK transformations.

Author Contributions: Conceptualization, X.L.; methodology, H.Z. and X.L.; software, H.Z.; validation, H.Z. and F.W.; formal analysis, H.Z. and J.W.; investigation, H.Z., J.W. and X.L.; resources, X.L. and C.C.; data curation, H.Z. and F.W.; writing—original draft preparation, H.Z., J.W. and F.W.; writing—review and editing, X.L. and C.C.; visualization, H.Z. and J.W.; supervision, X.L. and C.C.; project administration, X.L.; funding acquisition, X.L. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported in part by the National Natural Science Foundation of China (Grant No. 62562047), the Key Research and Development Project of Jiangxi Province (Grant Nos. 20243BBG71035 and 20252BCE310020), the Jiangxi Provincial Key Laboratory of Data Security Technology (Grant No. 20242BCC32026), the Key Project of the Joint Fund for Natural Science Research of Jiangxi Province (Grant No. 20253BAC280081), the General Project of Jiangxi Province Natural Science Foundation (Grant Nos. 20242BAB25079 and 20252BAC240082), and the Finance Science and Technology Special "Contract System" Project of Jiangxi Province (Grant Nos. ZBG20230418001 and ZBG20230418014).

Data Availability Statement: The study uses Drebin, AndroZoo, CICAndMal2017 and CICMalDroid2020 under the providers' access conditions. English code, instructions, frozen partitions and sample identifiers are available at https://github.com/Zhanghanwen-1/DGDroid(accessed on 24 November 2026). Some identifiers retain source filenames or aliases. The repository includes a tool for checking APK hashes after the original files have been obtained.

Acknowledgments: The authors acknowledge their institutions for support. OpenAI's Codex assisted with experimental scripts, result organization, drafting, translation, editing, and formatting. The authors reviewed the material and take responsibility for the manuscript.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Mehrabi Koushki, M.; AbuAlhaol, I.; Raju, A.D.; Zhou, Y.; Giagone, R.S.; Shengqiang, H. On building machine learning pipelines for Android malware detection: A procedural survey of practices, challenges and opportunities. *Cybersecurity* **2022**, *5*, 16. https://doi.org/10.1186/s42400-022-00119-8.
2. Dhalaria, M.; Gandotra, E. MalDetect: A classifier fusion approach for detection of android malware. *Expert Syst. Appl.* **2024**, *235*, 121155. https://doi.org/10.1016/j.eswa.2023.121155.

3. Soi, D.; Sanna, A.; Maiorca, D.; Giacinto, G. Enhancing android malware detection explainability through function call graph APIs. *J. Inf. Secur. Appl.* **2024**, *80*, 103691. https://doi.org/10.1016/j.jisa.2023.103691.
4. Molina-Coronado, B.; Mori, U.; Mendiburu, A.; Miguel-Alonso, J. Towards a fair comparison and realistic evaluation framework of android malware detectors based on static analysis and machine learning. *Comput. Secur.* **2023**, *124*, 102996. https://doi.org/10.1016/j.cose.2022.102996.
5. Gao, C.; Huang, G.; Li, H.; Wu, B.; Wu, Y.; Yuan, W. A Comprehensive Study of Learning-based Android Malware Detectors under Challenging Environments. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, Lisbon, Portugal, 14–20 April 2024; pp. 1–13. https://doi.org/10.1145/3597503.3623320.
6. Polatidis, N.; Kapetanakis, S.; Trovati, M.; Korkontzelos, I.; Manolopoulos, Y. FSSDroid: Feature subset selection for Android malware detection. *World Wide Web* **2024**, *27*, 50. https://doi.org/10.1007/s11280-024-01287-y.
7. Ding, Y.; Zhang, X.; Hu, J.; Xu, W. Android malware detection method based on bytecode image. *J. Ambient Intell. Humaniz. Comput.* **2023**, *14*, 6401–6410. https://doi.org/10.1007/s12652-020-02196-4.
8. Yadav, P.; Menon, N.; Ravi, V.; Vishvanathan, S.; Pham, T.D. EfficientNet convolutional neural networks-based Android malware detection. *Comput. Secur.* **2022**, *115*, 102622. https://doi.org/10.1016/j.cose.2022.102622.
9. Wang, T.; Xu, Y.; Zhao, X.; Jiang, Z.; Li, R. Android malware detection via efficient application programming interface call sequences extraction and machine learning classifiers. *IET Softw.* **2023**, *17*, 348–361. https://doi.org/10.1049/sfw2.12083.
10. Gao, H.; Cheng, S.; Zhang, W. GDroid: Android malware detection and classification with graph convolutional network. *Comput. Secur.* **2021**, *106*, 102264. https://doi.org/10.1016/j.cose.2021.102264.
11. Yang, J.; Liang, H.; Ren, H.; Jia, D.; Wang, X. SAC: Collaborative learning of structure and content features for Android malware detection framework. *Neurocomputing* **2025**, *637*, 130053. https://doi.org/10.1016/j.neucom.2025.130053.
12. Zhang, S.; Su, H.; Liu, H.; Yang, W. MPDroid: A multimodal pre-training Android malware detection method with static and dynamic features. *Comput. Secur.* **2025**, *150*, 104262. https://doi.org/10.1016/j.cose.2024.104262.
13. Prasad, A.; Chandra, S.; Uddin, M.; Al-Shehari, T.; Alsadhan, N.A.; Ullah, S.S. PermGuard: A Scalable Framework for Android Malware Detection Using Permission-to-Exploitation Mapping. *IEEE Access* **2025**, *13*, 507–528. https://doi.org/10.1109/access.2024.3523629.
14. Wu, Y.; Li, M.; Zeng, Q.; Yang, T.; Wang, J.; Fang, Z.; Cheng, L. DroidRL: Feature selection for android malware detection with reinforcement learning. *Comput. Secur.* **2023**, *128*, 103126. https://doi.org/10.1016/j.cose.2023.103126.
15. Sharma, Y.K.; Tomar, D.S.; Pateriya, R.K.; Bhandari, S. MOSDroid: Obfuscation-resilient android malware detection using multisets of encoded opcode sequences. *Comput. Secur.* **2025**, *152*, 104379. https://doi.org/10.1016/j.cose.2025.104379.
16. Yang, H.; Wang, Y.; Zhang, L.; Cheng, X.; Hu, Z. A novel Android malware detection method with API semantics extraction. *Comput. Secur.* **2024**, *137*, 103651. https://doi.org/10.1016/j.cose.2023.103651.
17. Yang, J.; Li, H.; He, L.; Xiang, T.; Jin, Y. MDADroid: A novel malware detection method by constructing functionality-API mapping. *Comput. Secur.* **2024**, *146*, 104061. https://doi.org/10.1016/j.cose.2024.104061.
18. Zhu, H.J.; Gu, W.; Wang, L.M.; Xu, Z.-C.; Sheng, V.S. Android malware detection based on multi-head squeeze-and-excitation residual network. *Expert Syst. Appl.* **2023**, *212*, 118705. https://doi.org/10.1016/j.eswa.2022.118705.
19. Li, X.; Liu, L.; Liu, Y.; Liu, H. Detecting Android malware: A multimodal fusion method with fine-grained feature. *Inf. Fusion* **2025**, *114*, 102662. https://doi.org/10.1016/j.inffus.2024.102662.
20. Yang, J.; Cai, X.; Wan, Z.; Mei, F.; Xiang, T. FARO-Droid: Reliability-aware fusion for obfuscation-resilient Android malware detection. *J. Inf. Secur. Appl.* **2026**, *100*, 104503. https://doi.org/10.1016/j.jisa.2026.104503.
21. Liu, Z.; Wang, R.; Japkowicz, N.; Gomes, H.M.; Peng, B.; Zhang, W. SeGDroid: An Android malware detection method based on sensitive function call graph learning. *Expert Syst. Appl.* **2024**, *235*, 121125. https://doi.org/10.1016/j.eswa.2023.121125.
22. Lu, X.; Zhao, J.; Zhu, S.; Lio, P. SNDGCN: Robust Android malware detection based on subgraph network and denoising GCN network. *Expert Syst. Appl.* **2024**, *250*, 123922. https://doi.org/10.1016/j.eswa.2024.123922.
23. Wang, Z.; Zhang, K.; Yang, S.; He, D.; Chan, S. Enhancing Android malware detection via knowledge distillation on homogenized function call graphs. *Knowl.-Based Syst.* **2025**, *323*, 113687. https://doi.org/10.1016/j.knosys.2025.113687.
24. Hei, Y.; Yang, R.; Peng, H.; Wang, L.; Xu, X.; Liu, J.; Liu, H.; Xu, J.; Sun, L. Hawk: Rapid Android Malware Detection Through Heterogeneous Graph Attention Networks. *IEEE Trans. Neural Netw. Learn. Syst.* **2024**, *35*, 4703–4717. https://doi.org/10.1109/tnnls.2021.3105617.
25. Ma, Z.; Luktarhan, N. Z2F: Heterogeneous graph-based Android malware detection. *PLoS ONE* **2024**, *19*, e0300975. https://doi.org/10.1371/journal.pone.0300975.

26. Li, T.; Shou, P.; Wan, X.; Li, Q.; Wang, R.; Jia, C.; Xiao, Y. A fast malware detection model based on heterogeneous graph similarity search. *Comput. Netw.* **2024**, *254*, 110799. https://doi.org/10.1016/j.comnet.2024.110799.
27. Wu, Y.; Shi, J.; Wang, P.; Zeng, D.; Sun, C. DeepCatra: Learning flow- and graph-based behaviours for Android malware detection. *IET Inf. Secur.* **2023**, *17*, 118–130. https://doi.org/10.1049/ise2.12082.
28. You, Y.; Chen, T.; Sui, Y.; Chen, T.; Wang, Z.; Shen, Y. Graph Contrastive Learning with Augmentations. *Adv. Neural Inf. Process. Syst.* **2020**, *33*, 5812–5823.
29. Yang, S.; Wang, Y.; Xu, H.; Xu, F.; Chen, M. An Android Malware Detection and Classification Approach Based on Contrastive Learning. *Comput. Secur.* **2022**, *123*, 102915. https://doi.org/10.1016/j.cose.2022.102915.
30. Zheng, J.; Liu, J.; Zhang, A.; Zeng, J.; Yang, Z.; Liang, Z.; Chua, T.-S. MaskDroid: Robust Android Malware Detection with Masked Graph Representations. In Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, Sacramento, CA, USA, 27 October–1 November 2024; pp. 331–343. https://doi.org/10.1145/3691620.3695008.
31. Gilmer, J.; Schoenholz, S.S.; Riley, P.F.; Vinyals, O.; Dahl, G.E. Neural Message Passing for Quantum Chemistry. In Proceedings of the 34th International Conference on Machine Learning, Sydney, Australia, 6–11 August 2017; Volume 70, pp. 1263–1272.
32. van den Oord, A.; Li, Y.; Vinyals, O. Representation Learning with Contrastive Predictive Coding. *arXiv* **2018**, arXiv:1807.03748.
33. Pendlebury, F.; Pierazzi, F.; Jordaney, R.; Kinder, J.; Cavallaro, L. TESSERACT: Eliminating Experimental Bias in Malware Classification across Space and Time. In Proceedings of the 28th USENIX Security Symposium, Santa Clara, CA, USA, 14–16 August 2019; pp. 729–746. Available online: https://www.usenix.org/conference/usenixsecurity19/presentation/pendlebury (accessed on 5 October 2026).
34. Arp, D.; Quiring, E.; Pendlebury, F.; Warnecke, A.; Pierazzi, F.; Wressnegger, C.; Cavallaro, L.; Rieck, K. Dos and Don'ts of Machine Learning in Computer Security. In Proceedings of the 31st USENIX Security Symposium, Boston, MA, USA, 10–12 August 2022; pp. 3971–3988. Available online: https://www.usenix.org/conference/usenixsecurity22/presentation/arp (accessed on 5 October 2026).
35. Karbab, E.B.; Debbabi, M.; Derhab, A.; Mouheb, D. MalDozer: Automatic framework for android malware detection using deep learning. *Digit. Investig.* **2018**, *24*, S48–S59. https://doi.org/10.1016/j.diin.2018.01.007.
36. Shen, L.; Fang, M.; Xu, J. GHGDroid: Global heterogeneous graph-based android malware detection. *Comput. Secur.* **2024**, *141*, 103846. https://doi.org/10.1016/j.cose.2024.103846.
37. Jiang, J.; Yang, J.; Wang, J.; Fan, J.; Xiang, T. An Android malware detection based on reconstructed API with TextCNN. *Int. J. Inf. Secur.* **2025**, *24*, 143. https://doi.org/10.1007/s10207-025-01058-4.
38. Ma, K.; Lu, B.; Yin, S.; Zheng, C.; Zhu, H. CADroid: A cross-combination attention based framework for android malware detection. *Expert Syst. Appl.* **2026**, *297*, 129446. https://doi.org/10.1016/j.eswa.2025.129446.
39. Li, H.; Cheng, Z.; Wu, B.; Yuan, L.; Gao, C.; Yuan, W.; Luo, X. Black-box Adversarial Example Attack towards FCG Based Android Malware Detection under Incomplete Feature Information. In Proceedings of the 32nd USENIX Security Symposium, Anaheim, CA, USA, 9–11 August 2023; pp. 1181–1198.
40. Bostani, H.; Moonsamy, V. EvadeDroid: A practical evasion attack on machine learning for black-box Android malware detection. *Comput. Secur.* **2024**, *139*, 103676. https://doi.org/10.1016/j.cose.2023.103676.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.