Enhancing Malware Detection for Android Apps: Detecting Fine-granularity Malicious Components

1st Zhijie Liu
School of Information Science and Technology
ShanghaiTech University
Shanghai, China
liuzhj2022@shanghaitech.edu.cn

2nd Liang Feng Zhang
School of Information Science and Technology
ShanghaiTech University
Shanghai, China
zhanglf@shanghaitech.edu.cn

3rd Yutian Tang*
School of Computing Science
University of Glasgow
Scotland, United Kingdom
Yutian.Tang@glasgow.ac.uk
Y. Tang is the corresponding author

2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 979-8-3503-2996-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/ASE56229.2023.00074

***Abstract*—Existing Android malware detection systems primarily concentrate on detecting malware apps, leaving a gap in the research concerning the detection of malicious components in apps. In this work, we propose a novel approach to detect fine-granularity malicious components for Android apps and build a prototype called *AMCDroid*. For a given app, *AMCDroid* first models app behavior to a homogenous graph based on the call graph and code statements of the app. Then, the graph is converted to a statement tree sequence for malware detection through the AST-based Neural Network with Feature Mapping (ASTNNF) model. Finally, if the app is detected as malware, *AMCDroid* applies fine-granularity malicious component detection (*MCD*) algorithm which is based on many-objective genetic algorithm to the homogenous graph for detecting malicious component in the app adaptively. We evaluate *AMCDroid* on 95,134 samples. Compared with the other two state-of-the-art methods in malware detection, *AMCDroid* gets the highest performance on the test set with 0.9699 F1-Score, and shows better robustness in facing obfuscation. Moreover, *AMCDroid* is capable of detecting fine-granularity malicious components of (obfuscated) malware apps. Especially, its average F1-Score exceeds another state-of-the-art method by 50%.**

***Index Terms*—Android, malware app, deep learning, many-objective genetic algorithm, malicious component**

I. Introduction

Android is the most popular mobile operating system, however, its open source also makes it a vulnerable target for malware. Therefore, A lot of android malware detection (AMD) systems have been proposed for detecting malware apps. Most existing AMDs [1], [2], [3], [4], [5], [6] mainly focus on malware detection (e.g., binary classification to apps, malicious or benign). They only output the classified result of a given app (application level), but cannot provide the location of malicious component for further analysis if the app is classified as malware. Malicious components are the malicious code at method granularity in malware, which are helpful to security analysts and can be the bases for many downstream tasks like more accurate malicious behavior interpretation [7], [8], repackaged malware detection [9], [10] and so forth.

Motivation. Existing AMDs now mainly utilizing machine learning [7], [3], [11], [4], [5] (i.e., ML-based methods). These methods first extract app features (e.g., Dalvik bytecode, graph, Android APIs, and so on) for behavioral modeling. Then, they leverage a large number of samples to train detection models and utilize the trained models to detect malware apps. Although these approaches perform well on relevant datasets and show a certain generalization, they still face the following three **challenges**:

- **C1: The Diversity of Malware:** To evade existing AMDs, malware developers often modify malware source code constantly to create multiple variants. However, most existing AMDs only extract single and simple information for behavioral modeling [12], [4], [5], [13] (e.g., API sequence, Android APIs, graph), which makes malware able to evade them [14], [15], [16];
- **C2: Time Dependency:** Time dependency represents the execution logic of code and contains semantic and structural information (i.e., control flow, data flow, and invocation relationship) [17], [3], [18]. Capturing time dependency is necessary since malicious components may be some small parts of malware apps (e.g., 2%) [19] and distributed separately in them [17]. Most existing AMDs [20], [7], [21], [22], [4], [5] neglect it or consider parts of it, allowing malware to evade detection with specific modifications [17], [16]; and
- **C3: Fine-granularity Malicious Component Detection:** Existing few malicious code fragment detection systems [23], [24], [12] can locate malicious code at class or method level. However, their location results are determined by the local information inside single units (i.e., basic blocks, methods, and classes) but without contextual information (e.g., neighboring methods), making them locate benign parts as malicious easily. Moreover, these methods locate fixed number malicious classes or methods, which are not adaptive.

Our Solution. To address the above three challenges, we propose three solutions and develop a prototype system called *AMCDroid*:

- **S1:** To address **C1**, we extract multi-dimensional features of apps through static analysis for modeling app behaviors comprehensively. We map all features into a homogeneous graph and eliminate redundant information for extracting core semantics (Sec. III-B);
- **S2:** To address **C2**, we develop a DL-based component detection model, called AST-based Neural Network with Feature Mapping (ASTNNF) (Sec. III-C). ASTNNF is constructed

2643-1572/23/$31.00 ©2023 IEEE
DOI 10.1109/ASE56229.2023.00074

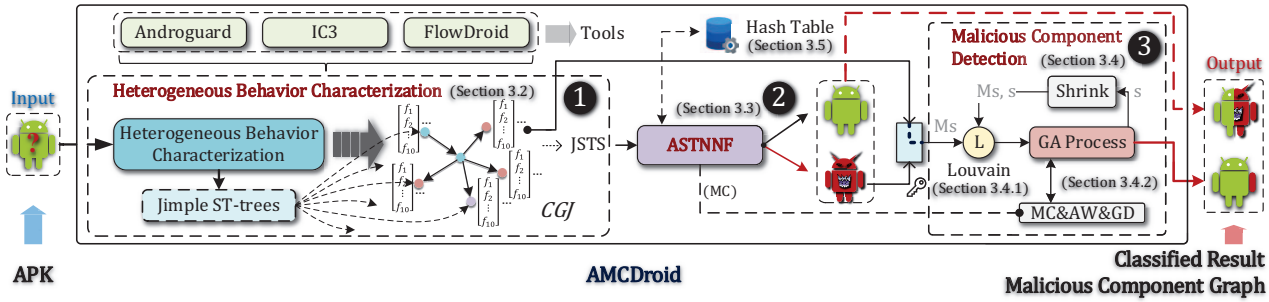

Fig. 1: Overview of *AMCDroid*'s architecture.

on Tree-based Neural Network and bidirectional Recurrent Neural Network (RNN), and takes Jimple abstract syntax statement tree (JST) sequence as input. Tree-based Neural Network captures JST's lexical and syntactic information, and bidirectional RNN captures time dependency information among JSTs from two directions; and

• **S3:** To address **C3**, we propose a fine-granularity malicious component detection (*MCD*) algorithm (Sec. III-D). *MCD* fits into the homogeneous graph, and leverages community detection and many-objective genetic algorithm to detect malicious components adaptively. The former selects initial individuals (i.e., modules in graph). The latter utilizes two-dimensional crossover and mutation to combine these modules randomly, considering invocation relationships. Moreover, *MCD* uses three fitness functions from three different respective aspects to guide the evolution direction.

Contribution. This paper makes three major contributions:

• We propose a novel fine-granularity malicious component detection approach for Android apps based on Tree-based Neural Network and many-objective genetic algorithm. The former captures time dependency information from modeled homogeneous graph of apps. The latter with three fitness functions searches for optimal solutions on the same homogeneous graph adaptively;

• We develop a prototype system (*AMCDroid*), which can not only detect malware apps, but also detect fine-granularity malicious components; and

• We evaluate *AMCDroid* on 95,134 samples, one of the largest-scale datasets. Compared with two state-of-the-art methods in malware detection, *AMCDroid* gets the highest performance on the test set with 1.5% to 9% advantage in F1-Score, and shows robustness in facing obfuscation with up to 20% higher TPR. Moreover, *AMCDroid* can also detect fine-granularity malicious components of (obfuscated) malware apps. In particular, its average F1-Score exceeds another state-of-the-art method by 50%, and its average Recall is above 0.54 in facing obfuscated malware apps.

II. Preliminary

In this section, we introduce two bases for the component detection model and malicious component detection. For the former, we utilize Tree-based Neural Network to capture time dependency information (Sec. II-A). As for the latter, we leverage many-objective genetic algorithm to detect fine-granularity malicious components (Sec. II-B).

A. Tree-based Neural Network

Syntactic knowledge contributes more to modeling code and can obtain better representation for classification [18]. Thus, in this research, we build our component detection model, AST-based Neural Network with Feature Mapping (ASTNNF) (Sec. III-C) based on ASTNN [18], a Tree-based Neural Network learning such information sufficiently. ASTNN takes statement tree (ST-tree) sequence of code fragment as input, and outputs representation for downstream tasks. ST-trees are split from the AST of code fragment. They are multi-way trees consisting of statement nodes as roots and corresponding AST nodes of the statements [18]. The ST-tree sequence is in the same order as the original statements in code fragment. First, ASTNN encodes ST-trees based on Word2Vec [25] and Recursive Neural Network (RvNN) [26]. Word2Vec is used to convert symbols in ST-trees into embedding vectors, and RvNN is a bottom-up model to capture both lexical and syntactic information of each ST-tree. Then, bidirectional GRU (Gated Recurrent Unit), one of RNNs, is used to further capture time dependency information among ST-trees from two directions. Where GRU [27] is used to tackle the vanishing gradient problem in the standard RNN. Finally, ASTNN utilizes one-dimensional max pooling to obtain the final code representation. More details of ASTNN can be found here [18].

B. Many-objective Genetic Algorithm

Malicious component is presented in the call graph (CG) of malware app with a small number of malicious methods. Thus, the search for malicious components can be seen as a search for malicious subgraphs, which is suitable for heuristic algorithms. Many-objective genetic algorithm (GA) provides an ability to solve problems adaptively based on multiple fitness functions [28], [29], [30]. Moreover, GA's crossover and mutation are also applicable in graph (Sec. III-D2). In this research, we leverage *MOSA* [28] to consider three kinds of information for fine-granularity malicious component detection (Sec. III-D). *MOSA* is a variant of *NSGA-II* [29]. It has multiple fitness functions and maintains a fitness vector for each individual, where each element in vector is an objective score of a corresponding fitness function. Through Pareto

TABLE I: Features for Heterogeneous Behavior Characterization

Dimension	Feature	Extraction Tool
Permission	*Normal Permission*	Androguard
	Dangerous Permission	Androguard
APP Component	*Activity*	Androguard
	Service	Androguard
	Broadcast Receiver	Androguard
	Content Provider	Androguard
Resource	*UI* and *UI's Callbacks*	Androguard
Code	*Android Official API*	Androguard
	ICC	IC3
	CG	FlowDroid

dominance [29], *MOSA* sorts offspring by preference and selects better solutions as next round population based on fitness vectors. More details of *MOSA* can be found here [28].

III. AMCDroid

A. Overview

As shown in Fig. 1, *AMCDroid* includes three steps for detecting a given app: ① heterogeneous behavior characterization (Sec. III-B) extracts static features of the app and maps them into a homogeneous graph; ② component detection model ASTNNF (Sec. III-C) takes Jimple ST-tree sequence generated from the homogeneous graph as input for malware detection, outputting detected result; ③ if the app is detected as malware, malicious component detection (Sec. III-D) does further detection on the homogeneous graph and outputs the malware app's malicious component graph.

B. Heterogeneous Behavior Characterization

The first step of *AMCDroid* is to model app behaviors, preparing for subsequent detection.

Input. The input is an Android APK file.

Heterogeneous Behavior Characterization. To capture sufficient behavioral characteristics of apps, we propose a multi-feature modeling approach extracting 10 features statically from 4 dimensions. These features are listed in Tab. I. Moreover, we integrate all features into a homogeneous graph to retain sufficient semantic information, represented as a graph $G = \{V, E\}$ called *CGJ*. *CGJ* contains the call graph (*CG*) of the given app, and each method in *CG* stores its Jimple [31] ST-trees (JSTs) (Sec. II-A). V in G represents the method vertexes in *CGJ* and E in G represents the invocation relationships. For each Jimple statement s_j (include method declaration) in V_i, its representation is JST. Moreover, a 10-dimensional feature vector $F = [f_1, f_2, f_3, ..., f_{10}]$ is used to characterize s_j and assigned to the JST. Each element f_i in the feature vector corresponds to one feature in Tab. I and $\in \{0, 1\}$, where the value of 0 indicates that s_j does not contain corresponding features.

① *Permission*. After Android 6.0, permission is divided into two categories, ❶ *Normal Permission* (f_1) and ❷ *Dangerous Permission* (f_2). The value of 1 for f_1 and f_2 indicates that s_j utilizes the corresponding permissions. We utilize Androguard [32] to assign permissions to each s_j.

② *App Component*. Four types of app components includes ❸ *Activity* (f_3), ❹ *Service* (f_4), ❺ *Broadcast Receiver* (f_5) and ❻ *Content Provider* (f_6). The value of 1 for f_3 to f_6 indicates that s_j belongs to a corresponding app component's method. We utilize Androguard [32] to extract app component classes for setting f_3 to f_6 of each s_j.

③ *Resource*. *UI* is a bridge between users and apps. We consider ❼ *UI* and *UI*'s *callbacks* (f_7). The value of 1 for f_7 indicates that s_j contains *UI*'s method (e.g., `TextView.setText`) or belongs to an *UI*'s *callback* (e.g., `View$OnClickListener`). We utilize Androguard [32] to extract UI classes and callbacks for setting f_7 of each s_j.

④ *Code*. App behaviors execute through *Code* ultimately. We utilize ❽ *Android Official API* (f_8), ❾ *inter-component communication* (*ICC*, f_9), and ❿ *call graph* (*CG*, f_{10}) to represent *Code*. *Android Official APIs* enable apps to access OS functions and system resources. The value of 1 for f_8 indicates that s_j invokes an Android API. *ICC* leads to additional control and data flows across app components by using intents [33], [34], [35]. The value of 1 for f_9 indicates that s_j has *ICC* behavior. Moreover, *ICC* can supplement *CG* [4], [5]. The value of 1 for f_{10} indicates that s_j invokes a method. For each s_j, We utilize Androguard [32] to extract Android APIs for setting its f_8, IC3 [36] to extract *ICC* for setting its f_9, and FlowDroid [37] with the *ICC* result (`config.getIccConfig().setIccModel(ICC_Result)`) to extract *CG* for setting its f_{10}.

To capture time dependency information in *CGJ*, we convert it into a JST sequence (JSTS). To make *CGJ*'s JSTS contain certain information of method execution sequence, we utilize spark pointer analysis to generate *CG* and use simultaneously generated invocation relationship sequence [38] to generate method sequence result (MSR). We generate MSR by traversing invocation relationships sequentially. Caller is traversed before callee to one invocation relationship, and one method is traversed only once. JSTS in one method are arranged in their

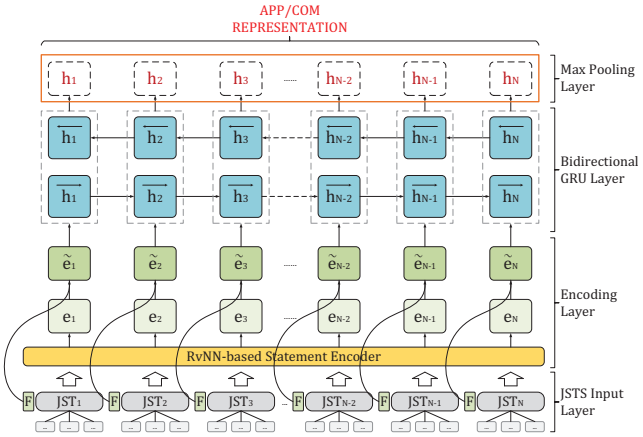

Fig. 2: The architecture of ASTNNF. (COM represents component)

statement order, and all JSTSs are concatenated to generate the *CGJ*'s JSTS according to MSR.

Output. The output includes *CGJ* and its JSTS.

C. Component Detection Model

Component detection model, AST-based Neural Network with Feature Mapping (ASTNNF) based on ASTNN (Sec. II-A) [18], predicts malicious confidence to the input. It is used for malware detection and malicious component detection.

Input. The input is *CGJ* or its subgraph (component)'s JSTS (Sec. III-D), used in the two detections, respectively.

Model Architecture. ASTNNF is constructed on ASTNN model [18] and can capture the information among the feature vectors of JSTs. The reason for using ASTNN as the base is three-fold: 1) it achieves state-of-the-art classification results in other related fields [18], [39]; 2) its encoding layer provides efficient computation for multi-way trees; and 3) RNN has a strong capability capturing time dependency [3], [40]. Fig. 2 shows the architecture of ASTNNF. ASTNNF contains four layers similar to ASTNN. They are 1) JSTS input layer, 2) encoding layer, 3) bidirectional GRU layer, and 4) max pooling layer, where the encoding layer is different from ASTNN. JST_i are first encoded into encoded vectors e_i by RvNN-based statement encoder (Sec. II-A). Moreover, the encoding layer of ASTNNF has one more concatenation between JST_i's feature vectors and corresponding e_i than ASTNN. The final encoded vectors $\widetilde{e}_i$ contain the lexical and syntactic information of JST_i themselves, and their feature vectors. $\widetilde{e}_i$ are then fed into bidirectional GRU to capture time dependency information (semantic and structural information, i.e., control flow, data flow, and invocation relationship) [17], [3], [18] among JSTs from two directions (i.e., forward and backward). Finally, ASTNNF stacks hidden states h_i generated from the previous phase, utilizes max pooling to capture their core features, and generates the representation of the input JSTS.

The function of ASTNNF is to distinguish malicious characteristics from benign ones. Thus, we utilize one linear layer and *Softmax* [12] as our classifier combined with ASTNNF, and cross-entropy as our loss function.

Output. The output is malicious confidence of the input JSTS. It is the probability of malicious, and $\in [0, 1]$. In malware detection, malicious confidence exceeding a certain threshold (e.g., 0.5) represents malicious to the input JSTS; otherwise, the JSTS is benign.

D. Malicious Component Detection

Malware detection detects malware apps, and the detected malware apps are the key to activating further detection since benign apps have no malicious behavior [17]. Existing approaches [24], [12] detect malicious components by choosing top n classes or methods with the highest malicious confidence as the results. However, no one can know n to real malware apps in advance. An inappropriate n to a specific malware app may generate a large number of false positives or negatives. To detect malicious components adaptively, we propose a fine-granularity malicious component detection algorithm called *MCD* based on many-objective genetic algorithm (GA) [28], [29] on *CGJ*. In addition, to search for solutions accurately, we introduce community detection [41] to select initial population to reduce the loss of initial evolutionary information.

Algorithm 1: *Malicious Component Detection*

Input : CGJ – malware app's *CGJ*;
Output: M_s – malicious subgraph;

Initialize parameters PC, PI, PM, M, T, A;
$M_s \leftarrow CGJ$;
$s \leftarrow \phi$, $f \leftarrow$ `True`;
while f **do**
  $s, f \leftarrow$ `GAPROCESS(`M_s, s`)`;
  $M_s \leftarrow$ merge all subgraphs in s;
end
return M_s;
Function `GAPROCESS(`M_s, s`)`**:**
  $pop \leftarrow$ `LOUVAIN(`M_s`)`;
  Set subgraphs' `ages` to 0 and `reverses` to `False` in pop;
  Append s to pop and set their `ages` to A and `reverses` to `True`;
  $s \leftarrow \phi$, $f \leftarrow$ `False`, $i \leftarrow 0$;
  while $i < T$ **do**
    $Q \leftarrow$ `GENERATE-OFFSPRING(`pop`)`;
    $R \leftarrow pop \cup Q$;
    $\mathbb{F} \leftarrow$ `PREFERENCE-SORTING-U(`R`)`;
    Get new pop according to $\mathbb{F}$ and M;
    Subtract 1 from the `ages` of the first-front subgraphs with `True reverses` in pop. If one's `age` equal to 0, then break out of the `while` loop;
    Add 1 to the `ages` of the first-front subgraphs with `False reverses` in pop. If one's `age` equal to A, then set f to `True` and break out of the `while` loop;
    $i++$;
  end
  $s \leftarrow$ all first-front subgraphs in pop;
  return s, f;

Input. The input includes the malware detection result (Sec. III-C) and the *CGJ* (Sec. III-B). The former activates further detection if the detection result is malware. The latter is the initial search space.

Malicious Component Detection. *MCD* includes two main parts: 1) select initial subgraphs from *CGJ* as the initial population (Sec. III-D1), and 2) utilize GA to detect malicious components (Sec. III-D2).

1) Selection of Initial Subgraphs: The selection of initial subgraphs (individuals) has an impact on malicious component detection. Small initial individuals (e.g., one method) lack contextual information [22], which can affect the evolution direction of GA since malicious behavior often consists of multiple methods (i.e., call chain). Huge initial individuals (e.g., *CGJ*) contain sufficient contextual information, but it is difficult to refine them. To resolve it, we utilize modules (communities) [41] in graph as initial subgraphs. From the software engineering perspective, each module should have high modularity [42], [43], [44]. A module in a program can be regarded as one function (feature) for users [33], [45], [46], [47], and malicious behaviors are distributed in these functions [19]. Therefore, it reduces the influence on the evolution direction and is easier to refine solutions. Moreover, many-objective GA needs modules to avoid being approximated as single-objective and losing initial evolutionary information (e.g., small initial

individuals may lack graph structure information).

Specifically, we find modules by using community detection [41]. We first transform *CGJ* into an undirected graph G [48]. To make G contain more original invocation relationships, we leverage 0, 0.5 and 1 as three different weights of edges to model Android API call, unidirectional call and bidirectional call, respectively. We set 0 weight to exclude API calls since many methods can call the same API, which messes with G during community detection. Then, we utilize *Louvain* algorithm to find modules in G [48], [33]. *Louvain* can be applied to a large graph with limited computing resources, which is suitable to G. Finally, the found modules mapped to *CGJ* are taken as initial subgraphs for subsequent GA process.

2) Detecting Malicious Component: Fine-granularity malicious component can be distributed in one module or multiple modules. We leverage *MOSA* [28], [29] (Sec. II-B) for heuristic detection on *CGJ* due to its preference on potentially better solutions. GA's crossover can simulate the reorganization between modules, and its mutation can introduce new methods into one subgraph or remove old methods from it. However, crossover and mutation are difficult to refine solutions containing too many methods. To resolve it, we introduce a strategy of shrinking search space to GA. Thus, *MCD* includes three steps: 1) find modules in search space; 2) detect malicious subgraph through crossover and mutation evolutionarily; and 3) shrink search space and redo the above two steps until obtaining fine-granularity malicious components. *MCD* is presented in Alg. 1, and its explanation and theoretics are as follows.

• **Input**: The input of *MCD* is malware app's *CGJ* (Sec III-B).

• **Line 1-3**: *MCD* first initializes parameters and variables used in GA process, and sets *CGJ* as the initial search space M_s. Where PC is the probability of crossover, PI is the probability of crossover selection, PM is the probability of mutation, M is the offspring number, T is the max generation number, and A is the maximal `age` for shrinking or stopping.

• **Line 4-8**: *MCD* invokes `GAPROCESS` method to search for malicious subgraphs in the current M_s, where `GAPROCESS`'s argument s is the malicious subgraphs searched in the last round. Then, *MCD* merges all solutions (current detected malicious subgraphs) in s output from `GAPROCESS` into a new subgraph assigned to M_s. The new M_s is the new shrunk search space for the next round GA process. The next round GA is necessary since the new M_s may still contain some benign methods. However, if there is no solution found dominating s in the current round GA, f is set to `False` to terminate the search process, and the current round new M_s is taken as the detected fine-granularity malicious component.

The `GAPROCESS` method is designed for searching for malicious subgraphs in the given search space.

• **Input**: The input of `GAPROCESS` is a search space M_s and the last round searched malicious subgraphs s.

• **Line 10-13**: *MCD* first utilizes *Louvain* to select initial subgraphs and initializes population *pop* (line 11 and 12) and other variables. Note that current round generated subgraphs have the same initialization in line 11.

• **Line 14-22**: *MCD* iteratively searches for solutions based on *MOSA* [28]. However, there is one updated `PREFERENCE-SORTING-U` method (presented in **Fitness Function**), and two additional stopping criteria in `GAPROCESS`: ① one first-front individual's `age` is equal to 0 when its `reverse` is `True` (line 19, stop shrinking search space) and ② one first-front individual's `age` is equal to A[1] when its `reverse` is `False` (line 20, continue shrinking search space). `age` is used to judge whether GA process should be continued, and it is also a guarantee for fine granularity. The individual with `True reverse` is the malicious subgraph generated by the last round GA. The individual with `False reverse` is the malicious subgraph generated by the current round. For ①, there is no newly generated malicious subgraph that can dominate the last round ones in A iterations. Thus, the malicious component can be considered found with high probability. As for ②, it indicates that there exists one malicious subgraph that dominates the last round ones and survives in the first front $\mathbb{F}_0$ in A iterations. Thus, the search space should be shrunk and a new GA process is started. In addition, the number of iterations exceeding T also indicates a malicious component is found.

• **Line 23-24**: *MCD* assigns all solutions in the $\mathbb{F}_0$ of *pop* to s and returns s and f. We treat the solutions in $\mathbb{F}_0$ as equally important to reduce the loss of potentially malicious methods.

Alg. 1 is presented above and its theoretics is described as follows.

Encoding. The search space (*CGJ* or its subgraph) is encoded into a two-dimensional graph structure. The structure contains invocation relationships providing rich information for GA.

Fitness Function. We design three fitness functions (FFs). These three FFs are malicious confidence (MC), Android API weight (AW), and graph density (GD). MC represents the probability of malice through ASTNNF. However, since ASTNNF may not be able to learn specific malicious behaviors meaningful to humans directly [49], we introduce AW to point out meaningful and sensitive behaviors through sensitive APIs [7], [4], [5], [22]. Moreover, malicious behaviors often appear in the form of call chains, thus GD ensures that the localized methods in components are as connected as possible. Therefore, the larger the values of all three FFs, the more likely it is a malicious component.

Malicious confidence. ASTNNF is used to evaluate MC. The fitness function is defined as follows:

$$MC(J) = 1 - A_b(J) \tag{1}$$

Where, J represents the input JSTS of a subgraph and A_b is a function evaluating input's benign confidence $\in [0, 1]$. The order of the corresponding methods in the *CGJ*'s MSR is taken as the traversal result of the subgraph for generating its JSTS.

Android API weight. Each Android API in *CGJ* is assigned a weight which represents its sensitiveness. We only set weights to the sensitive APIs (SAs) used in APIGRAPH [50], [51]. To obtain the weights, *AMCDroid* counts numbers C_n

[1] A is experimentally obtained.

and C_d of *Normal Permission* and *Dangerous Permission* used by SAs; if their numbers are both greater than 0, then the weight of *Normal Permission* (NPW) is set to C_d/C_n; otherwise, it is 0.5; the weight of *Dangerous Permission* (DPW) is always set to 1. Such a definition ensures the balance between NPW and DPW. Thus, the weights of SAs are the sum of their permission weights. For other APIs, their weights are set to 0. Finally, AW's fitness function is defined as follows:

$$AW(s) = \frac{W(s)}{E(s)} \tag{2}$$

Where, $W(s)$ evaluates subgraph s's summation of API-related edges' weights. Each edge's weight is equal to the callee sensitive API's weight. $E(s)$ evaluates the number of edges in s.

Graph density. GD represents a graph's density in terms of edge connectivity. It is defined as follows:

$$GD(s) = \frac{m}{n(n-1)} \tag{3}$$

Where, n and m are the numbers of nodes and edges in s.

Moreover, in line 17 of Alg. 1, if a subgraph has one highest objective score for one FF but two lower objective scores for the other two than other subgraphs', it should not be placed to $\mathbb{F}_0$ by preference [28]. For example, one benign subgraph can have the highest AW and the lowest MC and GD. Thus, we reverse the `PREFERENCE-SORTING` method in *MOSA* [28] to store such subgraphs in a special $\mathbb{F}_{0.5}$ and store the remaining subgraphs starting from $\mathbb{F}_0$. The new `PREFERENCE-SORTING-U` method prefers subgraphs with more balanced three objective scores, likely to contain more complete malicious behaviors.

Crossover and Mutation. Since encoded individuals are in graph structure, the two operations need updating for generating diverse offspring and considering invocation relationships.

Crossover. The new crossover takes in two candidates and generates two or one new offspring, corresponding to two versions. For version 1, ❶ *MCD* combines two candidates together to a bigger graph and computes its maximal cliques. ❷ It chooses two maximal cliques randomly as two bases. ❸ It shuffles the two bases randomly and preferentially selects a neighboring clique (share common vertexes) randomly from the remaining maximal cliques. ❹ This selected clique is combined into the current traversed base following the shuffled order to generate a new base. ❺ It repeats the above process (❸ - ❹) until all cliques are combined and the two final bases are the offspring. Version 1 considers randomness and invocation relationships in the graph. For version 2, *MCD* only combines two candidates together, which provides a chance to reconstruct a bigger subgraph and generate one offspring. The selection of two versions is controlled by PI.

Mutation. The new mutation contains adding methods and removing methods. For adding methods, *MCD* chooses one method randomly in candidate, and merges all of its neighboring (i.e., connected) methods. As for removing methods, *MCD* repeats the above, but removes one method or removes it and its neighboring methods controlled by a fixed 0.5 probability.

```
    public final void run() {
...     ...
 1      HttpURLConnection httpURLConnection2 = this.b.openConnection();
...     ...
 2      inputStream = httpURLConnection2.getInputStream();
...     ...
 3      int read = inputStream.read(bArr, 0, 102400);
 4      fileOutputStream2.write(bArr, 0, read);}
```

Fig. 3: Java code of `com.zuse.co.b.b.run`.

This new mutation guarantees the correlation of the methods added or removed to the subgraph. It is noticeable that *MCD* does not choose Android APIs, since their connections.

Output. The output is detected malicious component graph.

E. Video Memory Optimization

[18] proposes *dynamic batching* for computation efficiency of encoding multi-way trees (i.e., ST-trees) in batch. However, used in ASTNNF, it consumes a large amount of video memory for apps (e.g., 20GB+ video memory for one app). JSTS can be very long and have many nodes (e.g., 200K+ length and 1M+ number of nodes with 10K vocabulary size. K - thousand; M - million). To resolve the problem, we propose a *computational graph-based approach* for optimizing video memory with batch. JSTs always contain a variety of special symbols like "`local`", "`type`", "`name`", and so on. These symbols' embedding vectors v_n are equivalent to coefficients in computational graph [52]. In addition, parameters do not change during forward propagation, so for all the same symbols' v_n, their encoded vectors $W^\top v_n$ (i.e., e_n) are also equal. Thus, considering the gradient accumulation, there is no need to allocate video memory for all the same symbols' $W^\top v_n$. The $W^\top v_n$ of each symbol in vocabulary is allocated only once and stored in a **hash table** for retrieving before batching.

F. Running Example

Finally, we utilize a piggybacked malware app sharq2[2] as an example to illustrate the process of *AMCDroid* in general. *AMCDroid* performs the following steps:

► **Step 1 (Heterogeneous Behavior Characterization):** In the first step, *AMCDroid* extracts sharq2's 10-dimensional features for behavior characterization. The behavior characterization is represented in the form of *CGJ*. *CGJ* is also transformed into JSTS as input of ASTNNF.

► **Step 2 (Malware Detection):** In step 2, *AMCDroid* utilizes ASTNNF to classify the input JSTS. The result is malware, thus the *CGJ* of sharq2 is passed to step 3 for further detection.

► **Step 3 (Malicious Component Detection):** In step 3, *MCD* first utilizes *Louvain* to get initial subgraphs. Then, *MCD* leverages many-objective GA to search malicious subgraphs. To detect fine-granularity malicious component, *MCD* shrinks search space three times (i.e., four

[2]MD5: 7e4a3180db2fcef82e82357ecafa4c36

times for GA). Finally, *MCD* locates 29 methods including two connected malicious ones `com.zuse.co.b.b.run` and `com.zuse.co.b.c.a`, and one invoked sensitive Android API `java.net.URL.openConnection`. From the code (see Fig. 3), we can see that the method downloads something. Tracing the upstream call chain from the method, we also find that it downloads an unwanted and unverified file (e.g., APK file).

► **Step 4 (Output):** In the final step, *AMCDroid* outputs classified result of sharq2 with its malicious component graph.

IV. Evaluation

A. Dataset Description

We collect 40,354 malware apps from VirusShare [53], and 51,780 benign samples (reported as benign by all antivirus scanners from VirusTotal [54]) through Androzoo [55], which contains a total of 92,134 samples. Where VirusShare is a massive repository of live malware samples, and Androzoo is a growing collection of Android apps from multiple sources (e.g., Google Play). Both of them are used in plenty of research [4], [3], [5]. We also utilize MYST dataset [24] for evaluating *AMCDroid*'s ability to detect malicious components. The dataset is the only one containing 3,000 malicious samples with truth labels of malicious classes, and is introduced in Sec. IV-C2. Finally, we collect a total of 95,134 samples, which is one of the largest-scale datasets for evaluation.

B. Experiment Setting

► **Parameters of ASTNNF.** The dimension size of word embedding vector is set to 128. The dimension size of final encoded vector is set to 110 (100 – encoded vector dimension size; 10 – feature vector) in the encoding layer. Thus, the dimension size of hidden state, in the bidirectional GRU layer, is set to 110 and subsequently, representation's dimension size is 220. We set 10 epochs for training ASTNNF and set mini-batch size to 2. We also use the Adam [56] optimizer with 0.001 learning rate.

► **Experiment Environment.** Our server contains an Intel(R) Core(TM) i9-10900X CPU (10 cores), 128 GB RAM, and an RTX 3080 GPU (10G).

► **Data Splitting.** We randomly divide our dataset (92,134 samples) into three parts 60%, 20%, and 20% which are corresponded to the training, validation, and test set, respectively.

C. Evaluation of AMCDroid

We evaluate *AMCDroid*'s malware detection and malicious component detection based on the following research questions (RQs):

RQ1: How does *AMCDroid* perform on detecting malware apps on the large-scale dataset?

RQ2: How does *AMCDroid* perform on detecting malware components?

RQ3: Is *AMCDroid* effective for detecting obfuscated malware apps?

RQ4: Is *AMCDroid* effective for detecting obfuscated malware components?

TABLE II: Detection Performance of *AMCDroid*, *MaMaDroid*, and *MalScan*

Method	Accuracy	Precision	Recall	F1-Score
AMCDroid	**0.9739**	**0.9781**	**0.9618**	**0.9699**
MaMaDroid	0.8855	0.8426	0.9119	0.8759
[5]*Degree*	0.9526	0.9443	0.9478	0.9461
[5]*Katz*	0.9568	0.9469	0.9550	0.9510
[5]*Closeness*	0.9584	0.9539	0.9512	0.9525
[5]*Harmonic*	0.9603	0.9606	0.9485	0.9545

1) RQ1: How does AMCDroid perform on detecting malware apps on the large-scale dataset?:

Motivation. We first evaluate the performance of malware detection of *AMCDroid*. It is the base for activating further malicious component detection.

Approach. We evaluate the performance of *AMCDroid* from ① detection performance on the test set, and ② comparing with two state-of-the-art methods *MaMaDroid* [4] and *MalScan* [5]. *MaMaDroid* is a CG-based AMD. It abstracts method names and extracts invocation relationships to build a Markov chain for modeling app behaviors. The Markov chain then is transformed into a feature vector for classification. *MalScan* is also a CG-based method. It leverages centralities of sensitive android APIs in CG as a feature vector for classification. We set family mode for *MaMaDroid*, and four centralities (Tab. II) for *MalScan*. They all have other options, but out of memory in our experiment. All methods are evaluated on Accuracy, Precision, Recall, and F1-Score. The classification threshold of *AMCDroid* is set to 0.5 based on the validation set result [57].

Result ①. The detection performance on the test set is shown in Tab. II. It shows that all four metrics are above 0.96. For each of the two misclassified ❶ benign and ❷ malicious samples, we choose 20 randomly and inspect them manually. For the ❶, 6 of them do not use any dangerous permission. Therefore, the misclassified reason can be that similar code snippets occur more often in malicious samples than benign ones in the training set. For the rest 14, they all display sensitive behaviors. In particular, one app provides a platform for users to present up-to-date social information. In addition to using Network, it also uses services like Crypto, SMS, IPC and so on. These mixed behaviors are similar to some malware apps, which leads *AMCDroid* to misclassify it and the other 13. For the ❷, half of them do not use any dangerous permission, but the rest half have some sensitive behaviors. We further utilize VirusTotal [54] to scan all misclassified malware apps. 70% of them are flagged by 65 antivirus scanners from VirusTotal as malware less than 10 times. These non-obvious malicious behaviors make *AMCDroid* fail to identify them.

Result ②. As shown in Tab. II, *AMCDroid* achieves better performance than *MaMaDroid* and *MalScan* in terms of all four metrics. *MaMaDroid* performs the least well. It only achieves 0.8855 Accuracy, and its Precision is 10% lower than other methods. *AMCDroid* surpasses *MaMaDroid* by 5% in all metrics, which indicates that our approach, with complete modeling, performs better than the approach abstracting graphs and methods. There are two possible reasons why

MaMaDroid's result is worse. 1) *MaMaDroid* abstracts the original full method into the corresponding family name. In particular, `java.lang.Throwable.getMessage` is abstracted to `java`. This abstraction makes most of the semantic information lost; 2) *MaMaDroid* abstracts developer-defined methods to `self-defined` label. This abstraction makes the constructed Markov Chain contains only one developer-defined method. Most APIs are connected to this node, which destroys the original invocation relationships.

Compared to *MalScan*, although, *AMCDroid* still outperforms it with different four centralities, all four metrics are quite similar. The result indicates that CG-based methods, without modifying CG, are able to capture the differences in structural information from benign and malicious graphs. *AMCDroid*'s four metrics are slightly higher than *MalScan*'s. The reason is that *AMCDroid* considers not only the structural information of graph, but also the semantic information of code in graph and the features of each JST.

Answer to RQ1. In the case of malware detection on a large-scale dataset, *AMCDroid* achieves 0.9699 F1-Score. Compared with the other two state-of-the-art methods, the four metrics of *AMCDroid* are all higher than theirs with 1% to 13% advantages.

2) RQ2: How does AMCDroid perform on detecting malware components?:

Motivation. We evaluate *AMCDroid*'s ability to detect fine-granularity malicious components in malware apps.

Approach. We utilize MYST malware dataset [24], [58]. The dataset contains 3K samples and is the only one annotated with locations of malicious classes. We compare *MCD* with one state-of-the-art method *MKLDroid* [24]. *MKLDroid* assigns malicious scores for every basic block in multi-view context-aware ICFG, and chooses the top n classes with the highest malicious scores as the final located results. Under the case of MYST, they set n to 10. Since we do not have their source code and benign samples used in training set, we can only compare ours with their results listed in the paper. [24] chooses 1K malware apps randomly from the whole dataset as the test set. We utilize average Precision, Recall, and F1-Score as the metrics. Precision is the percentage of malicious classes in the located classes, and Recall is the percentage of located malicious classes in the truth malicious classes, for one detection result. The average F1-Score is the harmonic mean calculated by average Precision and Recall.

Configuration. We set PC, PI, PM, M, and T to 0.5, 0.5, 0.5, 30, and 50 through our test on a few samples (from the training set of 92,134 samples). In our test for A of *MCD* on the same samples, we find that when $A < 3$, the GA process tends to end prematurely, and when $A > 7$, it is prone to perform useless calculations. Thus, A is set to 5 balanced between premature results and useless calculations. We apply *MCD* once for each sample obeying the law of large numbers.

Result. The experiment result is shown in Tab. III. *AMCDroid* achieves 0.8469 Precision and 0.6764 Recall under the whole dataset (3K), and 0.8462 Precision and 0.6788 Recall under the test set (1K). From this result, we can conclude that the malicious component detection of *AMCDroid* is stable. Compared with *MKLDroid*, *AMCDroid* achieves better Precision than *MKLDroid*'s 0.1434, but Recall lags behind *MKLDroid*'s 0.9433. *AMCDroid*'s F1-Score is much higher than *MKLDroid*'s. For Precision, although *MKLDroid* is able to locate malicious classes more completely, it also introduces many benign classes. The number of top n of *MKLDroid* is fixed for all apps. Only when the number of n is greater than the real malicious classes, all the malicious components can be located as much as possible. *MKLDroid*'s $n = 10$ (obtained through advanced statistics) is much greater than 2.48, the number of malicious classes per sample on average. However, *AMCDroid* utilizes GA to detect malicious components adaptively, choosing the most suspicious component based on three FFs. Thus, *AMCDroid* is less likely to locate benign components. From the method granularity, the detected benign components of *AMCDroid* only account for 10% of the completely located components. As for Recall, although it seems that *AMCDroid* does not perform well on this metric, from the perspective of semantics on sensitive APIs [5], [4], the **average sensitive API ratio** (defined as the mean ratio to the test set of *the invoked sensitive API numbers by the true malicious components located* to *the same ones by the true malicious components of corresponding malware apps*) is 91%. The result means that *AMCDroid* is able to locate malicious semantics completely in fine granularity. From the perspective of coarse-granularity hit (i.e., for one sample, as long as any malicious classes are located, it is considered a successful hit), its **Hit Rate** (number of hits / number of samples) is 0.9869. Moreover, each sample has 70.38 classes with 2.48 malicious ones on average. The result indicates that the three FFs guide the evolutionary direction of GA correctly.

TABLE III: Malicious Component Detection of *AMCDroid* and *MKLDroid*

Method	Avg. Precision	Avg. Recall	Avg. F1-Score
AMCDroid - 3K	0.8469	0.6764	0.7521
AMCDroid - 1K	**0.8462**	0.6788	**0.7533**
MKLDroid - 1K	0.1434	**0.9433**	0.2490

Answer to RQ2. *AMCDroid* achieves 0.7533 average F1-Score in malicious component detection on MYST dataset, which exceeds *MKLDroid* by 50%. Moreover, *MCD* can provide correct evolutionary direction for GA in detection.

3) RQ3: Is AMCDroid effective for detecting obfuscated malware apps?:

Motivation. Obfuscation allows malware developers to modify malware apps with the same semantics, harder to be identified.

Approach. We select 3.5K identified malware apps randomly from the test set for evaluating the robustness of *AMCDroid*. Ten obfuscations are used to modify them through *Obfuscapk* [14]. These obfuscations' prefixes are shown in Tab. IV

TABLE IV: Detection Performance of *AMCDroid*, *MaMaDroid* and *MalScan* against Ten Obfuscations

Obfuscation	*AMCDroid*	*MaMaDroid*	*MalScan*
Const	**0.9986**	0.8579	0.9539
Arithmetic	0.9996	0.9939	**1.0**
Reorder	0.9995	0.9927	**1.0**
Call	**0.9982**	0.8054	0.9328
Debug	0.9991	0.9943	**1.0**
Goto	**1.0**	0.9944	**1.0**
Method	**1.0**	0.9943	**1.0**
Nop	**1.0**	0.9941	**1.0**
Reflection	**0.9921**	0.8037	0.9918
Advanced	**0.9996**	0.9752	0.9942
Overall	**0.9987**	0.9415	0.9876

and their specific meanings can be found in [14]. 2.2K are generated for each obfuscation, and there are a total of 22,757 obfuscated malware apps. We evaluate the robustness of *AMCDroid* compared with the other two methods. For *MalScan*, we choose *Katz* based on its best result on obfuscation. We evaluate them through true positive rate (TPR).

Result. The experiment result is shown in Tab. IV. *AMCDroid* achieves 0.9987 TPR, the best performance, on the entire obfuscated dataset. *MalScan* and *MaMaDroid* achieve 0.9876 and 0.9415 TPRs, respectively. Refining to each obfuscation, *MalScan* can detect all obfuscated malware apps under the obfuscations of `Arithmetic`, `Reorder`, `Debug`, `Goto`, `Method`, and `Nop`. It is not affected since these obfuscations do not modify CG. *AMCDroid*, however, is slightly affected by 3 of them. Where `Arithmetic` and `Reorder` add and modify control flows. `Debug` can remove useful information like variable names. These three obfuscations are able to complicate the original semantics in method, which affects malware detection of *AMCDroid*. For the rest three obfuscations, `Goto`, `Method`, and `Nop`, *AMCDroid* is immune to them. `Method` only creates unreachable methods in CG. `Nop` inserts `nop` instructions into code which are removed at the reverse engineering phase. These junk codes cannot appear in JSTS. To `Goto`, it inserts two additional `goto` instructions into each method, which does not affect the original control flows. As for *MaMaDroid*, under these obfuscations, its performance is slightly reduced. *MaMaDroid* generates Markov chain by abstracting CG, which is resistant to obfuscating intra-procedural information.

However, *MaMaDroid* and *MalScan*'s TPRs have a drop under obfuscations of `Const` and `Call`. *AMCDroid* is still able to maintain high TPRs of 0.9986 and 0.9982, respectively. These two obfuscations can change the invocation relationships. `Const` only encrypts string constants and generally does not change the original CG. However, when string constants are related to reflection, the corresponding methods can be unreachable in CG. This makes Markov chains generated by *MaMaDroid* and centrality values evaluated by *MalScan* change, failing to identify obfuscated malware. `Call` inserts an intermediate method between caller and callee, which changes the structure of CG greatly. *MalScan* performs more robust than *MaMaDroid*. The reason is that *MalScan* takes local and global information into account, which alleviates the structural impact, however *MaMaDroid* only focuses on local information. In contrast, based on the semantic and structural knowledge learned from the large-scale dataset, *AMCDroid* is barely affected by `Const` and `Call`, indicating that it can resist graph structural obfuscation.

TABLE V: Malicious Component Detection of *AMCDroid* against Three Obfuscations

Obfuscation	Avg. Precision	Avg. Recall	Hit Rate
Arithmetic	0.8485	0.6783	0.9879
Call	0.8703	0.5430	0.9487
Reflection	0.6397	0.6695	0.9879

Under `Reflection` and `Advanced`, *AMCDroid* and *MalScan* are slightly affected, but *MaMaDroid*'s TPRs drop 20% and 2%, respectively. These two obfuscations also change CG but less than `Call`'s. `Reflection` redirects suitable invocations to a custom method that invokes original callees using reflection [14]. *MaMaDroid*'s abstraction process cannot resist such changes. To `Advanced`, it is `Reflection`, but its target is dangerous APIs, which alleviates the impact on *MaMaDroid* instead. For *MalScan*, these two obfuscations have a lower impact than `Call`, since local and global information of centrality is slightly affected. As for *AMCDroid*, most obfuscated malware apps' original semantics still exists. Semantics are stored in JSTs, though the tree structures may be different.

Answer to RQ3. In malware detection, *AMCDroid* is resistant to obfuscation and achieves 0.9987 TPR in total. In addition, *AMCDroid* has better robustness than the other two state-of-the-art methods, with up to 20% higher TPR.

4) RQ4: Is AMCDroid effective for detecting obfuscated malware components?:

Motivation. We answer the robustness of malicious component detection of *AMCDroid* against obfuscation.

Approach. We choose the 1K malware apps (i.e., test set) from MYST and utilize `Arithmetic`, `Call` and `Reflection` as representatives to obfuscate them. Finally, we get 2,981 obfuscated malware apps for evaluation. *AMCDroid* is evaluated on average Precision, average Recall, and Hit Rate (Sec. IV-C2).

Configuration. We use the same configuration as Sec. IV-C2.

Result. We denote the result of "*AMCDroid* - 1K" in Tab. III as `None`. The experiment result is listed in Tab. V. For `Arithmetic`, we can see that its result is similar to `None`. Since it does not change code structural information and semantics, *AMCDroid* is not affected by it. However, to `Call`, though *AMCDroid* achieves 0.8703 Precision, its Recall and Hit Rate are 14% and 4% lower than `None`, respectively. The large number intermediate methods inserted by `Call` change the overall CG, which affects the values of AW and GD and makes it more difficult for *AMCDroid* to detect malicious components completely. Fortunately, *AMCDroid* still achieves a high Hit Rate of 0.9487. As for `Reflection`, *AMCDroid*'s Recall and Hit Rate are similar to `None`, but Precision is lower than `None` by 20%. Because of `Reflection`'s changes

TABLE VI: Runtime Overheads of *AMCDroid*

Heterogeneous Behavior Characterization	Malware Detection	Malicious Component Detection
111.67 sec	7.29 sec	109.93 sec

to the code structure (Sec. IV-C3), it is easier for *AMCDroid* to introduce some benign methods, but the performance of locating malicious methods is not affected. In addition, under the three obfuscations, the **average sensitive API ratios** (Sec. IV-C2) are all greater than 91%, which indicates that *AMCDroid*'s ability to detect malicious semantics is also not affected.

Answer to RQ4. *AMCDroid*'s malicious component detection has the ability against obfuscation. Its three average Recall values and three Hit Rates are all above 0.54 and 0.94, respectively.

V. Discussion

A. Properties of AMCDroid

AMCDroid achieves high detection performance, and shows the capability of resisting obfuscation and capturing time dependency. Since *AMCDroid* learns on a large-scale dataset, it can also detect latent malicious components (e.g., native library).

Resisting Obfuscation. We evaluate the robustness of malware detection and malicious component detection of *AMCDroid*. For malware detection, *AMCDroid* is only slightly affected (Sec. IV-C3). As for malicious component detection, *AMCDroid* is not affected by intra-method obfuscations. However, for inter-method obfuscations, *AMCDroid* is affected to a certain extent (Sec. IV-C4).

Capturing Time Dependency. Time dependency represents semantic and structural information. To capture such rich information for detection, ASTNNF is constructed on tree-based neural network, and bidirectional RNN (Sec. III-C). The capture of time dependency is necessary since malicious components may be small parts of malware apps (e.g., 2%) [19] and distributed separately in JSTS.

B. Runtime Overhead

AMCDroid comprises three steps for detecting a given app: heterogeneous behavior characterization, malware detection, and malicious component detection[3], with the latter being specific to an identified malware app. During the training phase, the runtime overhead is mainly due to the training of ASTNNF. Training on our large-scale dataset requires 3 days for 1 epoch and 10 epochs for the entire training process. In the testing phase, the average runtime overheads for each of the three steps on one app are shown in Tab. VI. For one given app, *AMCDroid* takes 111.67 seconds, 7.29 seconds, and 109.93 seconds on the three steps of heterogeneous behavior characterization, malware detection, and malicious component detection, respectively. The first and third steps occupy the main time cost. In general, the total average runtime overhead for one app, including malicious component detection, is 228.89 seconds, less than 4 minutes.

[3]We select 3,000 identified malicious samples randomly from the test set to evaluate the runtime overhead of malicious component detection, aiming to better reflect the runtime overhead under the diversity of malware apps.

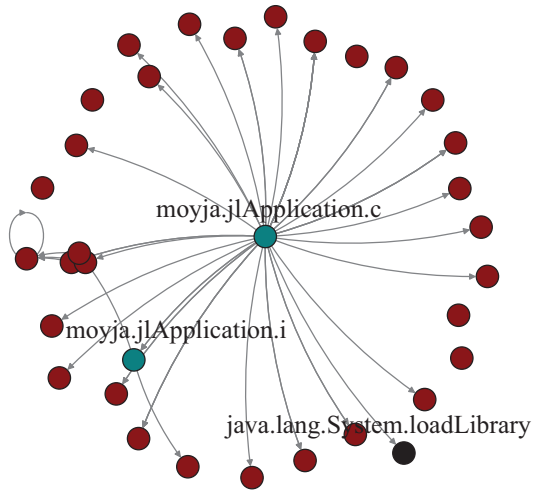

Fig. 4: Detected malicious subgraph of Roaming Mantis, where red represents Android API, teal represents developer-defined method and black represents `java.lang.System.loadLibrary`.

```
   private void c() {
1      System.loadLibrary("gv");
...    ...
2      int read = open.read();
3      ByteArrayOutputStream byteArrayOutputStream2 = new
   ByteArrayOutputStream();
4      byte[] bArr2 = new byte[4096];
5      while (true) {
6          int read2 = open.read(bArr2, 0, Math.min(i2, 4096));
7          if (read2 == -1 || read2 == 0) {break;}
8          i2 -= read2;
9          for (int i3 = 0; i3 < read2; i3++) {bArr2[i3] = (byte) (bArr2[i3] ^ read);}
10         byteArrayOutputStream2.write(bArr2, 0, read2); }
11     b(byteArrayOutputStream, bArr2, g(byteArrayOutputStream2.toByteArray(), 2,
   this));
...    ...      | mvp is the class composed of JNI |
12     i(this, mvp.hd(absolutePath, getFilesDir().getAbsolutePath() + "/qgyx"));}
```

Fig. 5: Java code of `moyja.jlApplication.c`.

C. Case Study: Detecting Malicious Native Library

If a malware app invokes malicious methods in a native library through JNI (Java Native Interface), *AMCDroid* can detect where the malicious native library is loaded and its JNI invocations. *AMCDroid* learns on a large-scale dataset. Thus, it can utilize knowledge of malicious characteristics learned and three fitness functions to search for the most likely malicious components. In addition, since malware apps in a malware family have similar semantics [2], even subtle

malicious characteristics related to JNI can also be captured by *AMCDroid*. However, the code in native library cannot be detected, since it is not stored in *CGJ*. For example, Roaming Mantis [59] is a kind of malware that targets Android devices recently. The malicious component of it is stored as payload decrypted by its native library. We apply *AMCDroid*'s malicious component detection to one[4] of them. *AMCDroid* locates 34 of 2,823 methods, including 2 developer-defined methods and 32 APIs. The located malicious subgraph is shown in Fig. 4. As we can see, the 2 developer-defined methods invoke quite a few APIs, which present suspicious behavior. One located method `moyja.jlApplication.c` is shown in Fig. 5. It first loads malicious native library in line 1, reads the payload during the `while` loop in line 6, and utilizes `XOR` to decrypt the read content in line 9. Then, in line 11, method `b` is invoked to decompress the read payload to a DEX file through JNI. Finally, in line 12, it invokes another located method `moyja.jlApplication.i` to get the malicious `Class` object of the DEX file.

D. Threats to Validity

Our study explores the fine-granularity malicious component detection on malware apps. Regarding external validity, MYST [24], [58] is a crafted malware dataset with two characteristics: 1) these apps are piggybacked malware, and 2) the dataset contains fine-granularity truth labels of malicious classes. Thus, it is suitable for evaluation. Nevertheless, considering the size of the dataset, we do not intend to generalize our results to all malware apps. Regarding internal validity, GA has the property of randomness. We further choose 100 samples in MYST dataset and apply *MCD* 30 times for each of them. From class level, the Hit Rate (Sec. IV-C2) is 0.9883, and 94 samples have up to 4 results, of which 42 samples have 1 result. The standard deviation (STD) of F1-Score (calculated by Precision and Recall, Sec. IV-C2) values of 3,000 results is 0.15. The maximum, minimum, and mean values of STD of the respective F1-Score values for all samples are 0.19, 0.0, and 0.06, respectively. Thus, we can conclude that *MCD* is stable at class level.

VI. Related Work

ML-based Methods. There are many existing methods of AMDs based on machine learning, utilizing features like permission, Android API, Android component, Dalvik code, Call Graph and so on [8], [3], [60], [7], [11], [4], [5], [34], [61], [6], [40], [62], [19]. *DREBIN* [7] extracts features from AndroidManifest.xml files and Dalvik bytecode by static analysis for behavioral modeling to detect malware. The features extracted include hardware information, requested permissions, app components, and so on. *StormDroid* [8] takes permissions, sensitive APIs, and other information (e.g., dynamic behaviors) for training detection model. *DeepRefiner* [3] has two malware detection layers and its second layer takes apps' smali code directly as input into LSTM-based model for detecting malware apps and achieves high accuracy with strong robustness. *ICCDetector* [34] leverages ICC-related features to model app behaviors. *DroidCat* [61] trains malware detection model through dynamic features based on method calls and ICC Intents. *MaMaDroid* [4] abstracts method names and extracts invocation relationships to build a Markov chain for modeling app behaviors. The Markov chain then is transformed into a feature vector to conduct classification. *MalScan* [5] leverages centralities of sensitive android APIs in CG as a feature vector for conducting classification. Similar related works also include [62] and [19].

Malicious Code Fragment Detection (MCFD). MCFDs can locate malicious code fragments [23], [63], [24], [12], [22]. Zheng et al. propose *DroidAnalytics* [23] which generates apps' signatures (e.g., MD5) at class level and application level for identifying malicious code fragments and malware apps, respectively. *HookRanker* [63] is designed to provide ranked lists of potentially malicious packages for piggybacked malware apps. [24] proposes a multi-view context-aware ICFG (CICFG) framework *MKLDroid*. Benefiting from its multi-view, *MKLDroid* outperforms in malware detection and is able to locate suspicious code fragments from five different views by assigning malicious scores to basic blocks in CICFG. *Droidetec* [12] leverages API sequence to detect malware apps through LSTM network and locates malicious code fragments by calculating the summation of API attention values. *MsDroid* [22] divides CG into subgraphs and passes them to GNN for malware detection. *MsDroid* also uses malicious snippets stored in database to match malicious components. Different from these MCFDs, *AMCDroid* is able to detect fine-granularity malicious components adaptively.

VII. Conclusion

We propose a novel fine-granularity malicious component detection approach for Android apps and develop *AMCDroid* to support malware detection and malicious component detection. We utilize 92,134 samples with three experiments to evaluate *AMCDroid*'s malware detection. Compared with two state-of-the-art methods, *AMCDroid* achieves the highest performance on the test set and robustness against obfuscation with 0.97 F1-Score and 0.99 TPR, respectively. We also set up two experiments with another 3,000 samples to evaluate *AMCDroid*'s malicious component detection. The result shows that it has the ability to detect malicious components in malware apps. In particular, its average F1-Score exceeds another state-of-the-art method by 50%, and its average Recall is above 0.54 in facing obfuscated malware apps.

VIII. Data Availability

The experiment data and the source code of *AMCDroid* can be found at: [57].

Acknowledgment

We would thank the anonymous reviewers for their insightful comments to improve the quality of the paper. This work is sponsored by Shanghai Pujiang Program (No. 21PJ1410700), National Natural Science Foundation of China (No. 62202306).

[4]MD5: ddd131d7f0918ece86cc7a68cbacb37d

REFERENCES

[1] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware android malware classification using weighted contextual api dependency graphs," in *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, 2014, pp. 1105–1116.

[2] 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*, 2014, pp. 576–587.

[3] K. Xu, Y. Li, R. H. Deng, and K. Chen, "Deeprefiner: Multi-layer android malware detection system applying deep neural networks," in *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2018, pp. 473–487.

[4] L. Onwuzurike, E. Mariconti, P. Andriotis, E. D. Cristofaro, G. Ross, and G. Stringhini, "Mamadroid: Detecting android malware by building markov chains of behavioral models (extended version)," *ACM Transactions on Privacy and Security (TOPS)*, vol. 22, no. 2, pp. 1–34, 2019.

[5] 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)*. IEEE, 2019, pp. 139–150.

[6] H. Gao, S. Cheng, and W. Zhang, "Gdroid: Android malware detection and classification with graph convolutional network," *Computers & Security*, vol. 106, p. 102264, 2021.

[7] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of android malware in your pocket." in *Ndss*, vol. 14, 2014, pp. 23–26.

[8] S. Chen, M. Xue, Z. Tang, L. Xu, and H. Zhu, "Stormdroid: A streaminglized machine learning-based system for detecting android malware," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, 2016, pp. 377–388.

[9] Y. Shao, X. Luo, C. Qian, P. Zhu, and L. Zhang, "Towards a scalable resource-driven approach for detecting repackaged android applications," in *Proceedings of the 30th Annual Computer Security Applications Conference*, 2014, pp. 56–65.

[10] X. Zhan, T. Zhang, and Y. Tang, "A comparative study of android repackaged apps detection techniques," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 321–331.

[11] S. Xi, S. Yang, X. Xiao, Y. Yao, Y. Xiong, F. Xu, H. Wang, P. Gao, Z. Liu, F. Xu *et al.*, "Deepintent: Deep icon-behavior learning for detecting intention-behavior discrepancy in mobile apps," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 2421–2436.

[12] Z. Ma, H. Ge, Z. Wang, Y. Liu, and X. Liu, "Droidetec: Android malware detection and malicious code localization through deep learning," *arXiv preprint arXiv:2002.03594*, 2020.

[13] N. McLaughlin, J. Martinez del Rincon, B. Kang, S. Yerima, P. Miller, S. Sezer, Y. Safaei, E. Trickel, Z. Zhao, A. Doupé *et al.*, "Deep android malware detection," in *Proceedings of the seventh ACM on conference on data and application security and privacy*, 2017, pp. 301–308.

[14] 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.

[15] V. Rastogi, Y. Chen, and X. Jiang, "Droidchameleon: evaluating android anti-malware against transformation attacks," in *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, 2013, pp. 329–334.

[16] K. Zhao, H. Zhou, Y. Zhu, X. Zhan, K. Zhou, J. Li, L. Yu, W. Yuan, and X. Luo, "Structural attack against graph based android malware detection," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 3218–3235.

[17] J. Qiu, J. Zhang, W. Luo, L. Pan, S. Nepal, and Y. Xiang, "A survey of android malware detection with deep neural models," *ACM Computing Surveys (CSUR)*, vol. 53, no. 6, pp. 1–36, 2020.

[18] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 783–794.

[19] Y. Wu, D. Zou, W. Yang, X. Li, and H. Jin, "Homdroid: detecting android covert malware by social-network homophily analysis," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 216–229.

[20] Y. Aafer, W. Du, and H. Yin, "Droidapiminer: Mining api-level features for robust malware detection in android," in *International conference on security and privacy in communication systems*. Springer, 2013, pp. 86–103.

[21] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel, "Adversarial examples for malware detection," in *European symposium on research in computer security*. Springer, 2017, pp. 62–79.

[22] 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*, 2022.

[23] M. Zheng, M. Sun, and J. C. Lui, "Droid analytics: a signature based analytic system to collect, extract, analyze and associate android malware," in *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, 2013, pp. 163–171.

[24] A. Narayanan, M. Chandramohan, L. Chen, and Y. Liu, "A multi-view context-aware approach to android malware detection and malicious code localization," *Empirical Software Engineering*, vol. 23, no. 3, pp. 1222–1274, 2018.

[25] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *Advances in neural information processing systems*, vol. 26, 2013.

[26] R. Socher, C. C. Lin, C. Manning, and A. Y. Ng, "Parsing natural scenes and natural language with recursive neural networks," in *Proceedings of the 28th international conference on machine learning (ICML-11)*, 2011, pp. 129–136.

[27] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[28] A. Panichella, F. M. Kifetew, and P. Tonella, "Reformulating branch coverage as a many-objective optimization problem," in *2015 IEEE 8th international conference on software testing, verification and validation (ICST)*. IEEE, 2015, pp. 1–10.

[29] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE transactions on evolutionary computation*, vol. 6, no. 2, pp. 182–197, 2002.

[30] Z. Zhou, Y. Zhou, C. Fang, Z. Chen, and Y. Tang, "Selectively combining multiple coverage goals in search-based unit test generation," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

[31] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," in *CASCON First Decade High Impact Papers*, 2010, pp. 214–224.

[32] androguard, "Androguard," 2022, https://github.com/androguard/androguard.

[33] Y. Tang, H. Zhou, X. Luo, T. Chen, H. Wang, Z. Xu, and Y. Cai, "Xdebloat: Towards automated feature-oriented app debloating," *IEEE Transactions on Software Engineering*, 2021.

[34] K. Xu, Y. Li, and R. H. Deng, "Iccdetector: Icc-based malware detection on android," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 6, pp. 1252–1264, 2016.

[35] A. Feizollah, N. B. Anuar, R. Salleh, G. Suarez-Tangil, and S. Furnell, "Androdialysis: Analysis of android intent effectiveness in malware detection," *computers & security*, vol. 65, pp. 121–134, 2017.

[36] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel, "Composite constant propagation: Application to android inter-component communication analysis," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 77–88.

[37] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[38] O. Lhoták, "Spark: A flexible points-to analysis framework for java," 2003.

[39] G. Partenza, T. Amburgey, L. Deng, J. Dehlinger, and S. Chakraborty, "Automatic identification of vulnerable code: Investigations with an ast-based neural network," in *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 2021, pp. 1475–1482.

[40] T. Lu, Y. Du, L. Ouyang, Q. Chen, and X. Wang, "Android malware detection based on a hybrid deep learning model," *Security and Communication Networks*, vol. 2020, 2020.

[41] S. Fortunato, "Community detection in graphs," *Physics reports*, vol. 486, no. 3-5, pp. 75–174, 2010.
[42] G. Nudelman, *Android design patterns: interaction design solutions for developers*. John Wiley & Sons, 2013.
[43] B. Hardy and B. Phillips, *Android programming: the big nerd ranch guide*. Addison-Wesley Professional, 2013.
[44] E. Giger and H. Gall, "Object-oriented design heuristics," 1996.
[45] T. Berger, D. Lettner, J. Rubin, P. Grünbacher, A. Silva, M. Becker, M. Chechik, and K. Czarnecki, "What is a feature? a qualitative study of features in industrial software product lines," in *Proceedings of the 19th international conference on software product line*, 2015, pp. 16–25.
[46] D. Lettner, K. Eder, P. Grünbacher, and H. Prähofer, "Feature modeling of two large-scale industrial software systems: Experiences and lessons learned," in *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, 2015, pp. 386–395.
[47] B. Behringer, J. Palz, and T. Berger, "Peopl: projectional editing of product lines," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 563–574.
[48] S. Ghosh, M. Halappanavar, A. Tumeo, A. Kalyanaraman, H. Lu, D. Chavarria-Miranda, A. Khan, and A. Gebremedhin, "Distributed louvain algorithm for graph community detection," in *2018 IEEE international parallel and distributed processing symposium (IPDPS)*. IEEE, 2018, pp. 885–895.
[49] A. Ilyas, S. Santurkar, D. Tsipras, L. Engstrom, B. Tran, and A. Madry, "Adversarial examples are not bugs, they are features," *Advances in neural information processing systems*, vol. 32, 2019.
[50] X. Zhang, Y. Zhang, M. Zhong, D. Ding, Y. Cao, Y. Zhang, M. Zhang, and M. Yang, "Enhancing state-of-the-art classifiers with api semantics to detect evolved android malware," in *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, 2020, pp. 757–770.
[51] 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*, 2012, pp. 217–228.
[52] A. Paszke, S. Gross, S. Chintala, and G. Chanan, "Pytorch," 2022, https://github.com/pytorch/pytorch.
[53] VirusShare, "Virusshare - sharing of malware samples," 2022, https://virusshare.com.
[54] VirusTotal, "Virustotal - free online virus, malware and url scanner," 2022, https://www.virustotal.com.
[55] 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*, 2016, pp. 468–471.
[56] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.
[57] AMCDroid, "Amcdroid online artefact," 2023, https://zenodo.org/record/7886463#.ZFEPjHZByUk.
[58] G. Meng, Y. Xue, C. Mahinthan, A. Narayanan, Y. Liu, J. Zhang, and T. Chen, "Mystique: Evolving android malware for auditing anti-malware tools," in *Proceedings of the 11th ACM on Asia conference on computer and communications security*, 2016, pp. 365–376.
[59] Kaspersky, "Roaming mantis reaches europe," 2022, https://securelist.com/roaming-mantis-reaches-europe/105596/.
[60] A. Hota and P. Irolla, "Deep neural networks for android malware detection." in *ICISSP*, 2019, pp. 657–663.
[61] H. Cai, N. Meng, B. Ryder, and D. Yao, "Droidcat: Effective android malware detection and categorization via app-level profiling," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 6, pp. 1455–1470, 2018.
[62] D. Zou, Y. Wu, S. Yang, A. Chauhan, W. Yang, J. Zhong, S. Dou, and H. Jin, "Intdroid: Android malware detection based on api intimacy analysis," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 3, pp. 1–32, 2021.
[63] L. Li, D. Li, T. F. Bissyandé, J. Klein, H. Cai, D. Lo, and Y. Le Traon, "Automatically locating malicious packages in piggybacked android apps," in *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 2017, pp. 170–174.