Structural Attack against Graph Based Android Malware Detection

Kaifa Zhao  
The Hong Kong Polytechnic University, China  
kaifa.zhao@connect.polyu.hk

Hao Zhou  
The Hong Kong Polytechnic University, China  
cshaoz@comp.polyu.edu.hk

Yulin Zhu  
The Hong Kong Polytechnic University, China  
yulinzh@polyu.edu.hk

Xian Zhan  
The Hong Kong Polytechnic University, China  
chichoxian@gmail.com

Kai Zhou  
The Hong Kong Polytechnic University, China  
kaizhou@polyu.edu.hk

Jianfeng Li  
The Hong Kong Polytechnic University, China  
jfli.xjtu@gmail.com

Le Yu  
The Hong Kong Polytechnic University, China  
cslyu@comp.polyu.edu.hk

Wei Yuan  
Huazhong University of Science and Technology, China  
yuanwei@hust.edu.cn

Xiapu Luo∗  
The Hong Kong Polytechnic University, China  
csxluo@comp.polyu.edu.hk

ABSTRACT

Malware detection techniques achieve great success with deeper insight into the semantics of malware. Among existing detection techniques, function call graph (FCG) based methods achieve promising performance due to their prominent representations of malware’s functionalities. Meanwhile, recent adversarial attacks not only perturb feature vectors to deceive classifiers (i.e., feature-space attacks) but also investigate how to generate real evasive malware (i.e., problem-space attacks). However, existing problem-space attacks are limited due to their inconsistent transformations between feature space and problem space.

In this paper, we propose the first structural attack against graph-based Android malware detection techniques, which addresses the inverse-transformation problem [1] between feature-space attacks and problem-space attacks. We design a Heuristic optimization model integrated with Reinforcement learning framework to optimize our structural Attack (HRAT). HRAT includes four types of graph modifications (i.e., inserting and deleting nodes, adding edges and rewiring) that correspond to four manipulations on apps (i.e., inserting and deleting methods, adding call relation, rewiring). Through extensive experiments on over 30k Android apps, HRAT demonstrates outstanding attack performance on both feature space (over 90% attack success rate) and problem space (up to 100% attack success rate in most cases). Besides, the experiment results show that combining multiple attack behaviors strategically makes the attack more effective and efficient.

CCS CONCEPTS

• Security and privacy  →  Software security engineering; Software reverse engineering.

KEYWORDS

Structural attack; Android malware detection; Function call graph

ACM Reference Format:

1  INTRODUCTION

Android malware detection (AMD) [2–17] has attracted much attention from both industry and academia as there are already around 69.84 million Android malware [18]. Existing detection techniques are usually based on classification methods, which extract features from benign and malicious apps and then train the detection model. In particular, many systems [2–10, 19–22] use static analysis to extract features, such as requested permissions [2–4] and function call relations [6]. Among these systems, FCG-based methods [5–10] achieve promising performance as FCGs contain rich semantic information, e.g., calling relations. In a FCG, each node represents a method and each directed edge represents a calling relationship between two methods. The state-of-the-art malware detection tools extract potential malicious features (e.g., sensitive APIs) from FCGs to represent malware’s malicious behaviors. For example, Malscan [5] extracts centralities of sensitive nodes in FCGs to train classifiers. Mamadroid [6] abstracts the nodes of FCGs into different states and uses the transition probability between states as features. Cai et al. [9] use the graph neural network to extract features from FCGs for malware detection.

Recent studies [5, 23] demonstrate the possibility of attacking FCG-based detection methods [5, 6] through adversarial samples. Unfortunately, their methods are almost limited to perturbing extracted feature vectors from FCGs, i.e., feature attacks. By contrast, in this paper, we investigate the vulnerability of FCG-based detection methods from their attack surfaces relevant to edges and nodes [24–30] and propose a new structural attack method.

Figure 1 shows the differences between feature attacks and structural attacks. The former adds perturbations to extracted feature vectors whereas the latter directly modifies the nodes and edges.
in graphs. Graph-based detection methods extract features from graphs and use these features to train classifiers for malware detection. By contrast, structural attacks directly modify the graph features and thus they are more intrinsic and effective [31, 32]. Besides, as nodes in FCGs correspond to methods in software and edges correspond to call relations, structural attacks could address the inverse-transformation problems between feature-space attacks and problem-space attacks [1].

Existing feature attacks [1, 23, 33–35] generated adversarial samples by adding perturbations to extracted feature vectors. For instance, Grosse et al. [33] apply Jacobian matrix to modify features in Drebín [2] to generate adversarial examples and achieve 69% evasion rate. Android HIV [23] aims to optimize their attack process, which perturbs the features in Mamadroid [6] and Drebín, through several algorithms (e.g., C&W based methods [23]). Recent studies designed problem-space attacks to generate real adversarial malware [1, 23]. Unfortunately, to preserve the semantics of generated adversarial malware, existing problem-space attacks are restricted to inserting non-functional methods or calls [1, 23]. Due to the inverse feature-mapping problems, the feature mapping functions between problem space and feature space are neither injective nor surjective [30]. Thus, existing problem-space attacks have to take extra processes to deal with side effects [1, 23].

In this paper, we propose a novel and practical structural attack method against FCG-based AMD systems, which tackles the limitations (L1-3) of existing attack methods. **L1-system-specific attack methods.** Existing attack approaches [1, 23, 33–35], especially those problem-space ones [1, 23], highly depend on the feature extraction methods of target systems. For example, Android HIV [23] implements problem-space attack on Mamadroid by transforming the features (i.e., call probability) to invocation numbers. Hence, if the features change, the transformation relation must also be adjusted accordingly. By contrast, since our structural attack perturbs the graph structures rather than feature vectors, feature extraction methods have no impact on our attack flow. That is, our structural attack is general to all graph-based systems. **L2-limited software modification operations.** To maintain functional consistency, existing App modification methods are limited to inserting dead code, such as no-op API calls [1, 23, 36]. By contrast, we design four types of software manipulation operations: inserting methods, removing methods, adding call relations and rewiring call relations (§4). **L3-inconsistent transformation relation.** The adversarial App generation methods of existing approaches are guided by the transformation from feature perturbations to App modifications [1, 23]. Although they can randomly modify features as needed when perturbing features, they can only insert no-op code to restricted methods [1, 23] when generating adversarial apps. To bridge this gap, we design each structural attack action by considering the characteristics of apps (§3.2). As nodes and edges in the CFGs correspond to methods and call relations in apps, respectively, our attack method ensures that the modification of the graph is consistent with the manipulations of apps.

Although our structural attack solves the aforementioned limitations in existing methods, there remain two challenges in our system design, i.e., how to determine a manipulation operation type and how to select the most effective objects (nodes or edges) to modify. To this end, we design a Heuristic optimization integrated Reinforcement learning ATack (HRAT) algorithm (§3.3) to solve them. HRAT consists of two phase: a) determining an action type according to current graph state and b) selecting optimal edges or nodes to conduct the modifications on the graph. Leveraging reinforcement learning, HRAT learns how to select effective action types based on current graph state 3.3 through interacting with the target environment [37]. Then, with determined action type, HRAT uses gradient search to select the most influential edges/nodes for modification. In this way, HRAT learns the modification sequences on the target graph, which allows the modified App to bypass the detection. Finally, HRAT automatically generates adversarial apps based on graph modification sequences.

Our major contributions are summarized as follows:

- **A Novel Structural Attack.** To our best knowledge, we propose the first structural attack on Android malware detection systems. It includes four types of graph modification operations and uses reinforcement learning to optimize the attack process.
- **Fill Research Gap.** Our method fills the gap that adversarial features cannot be effectively mapped to real apps. Besides, since our attack method works on graphs directly, it could be extended to other graph-based detection systems.
- **A Useful Tool.** We develop an automated tool that can modify the structure of Android Apps without affecting the original functionality. HRAT can modify apps according to graph modification sequences. (We release the code and data to other researchers by responsibly sharing a private repository. The project website with instructions to request access is at: [https://sites.google.com/view/hrat/](https://sites.google.com/view/hrat/))
- **Valid Evasion Effects.** We evaluate the effectiveness of our attack on two latest AMD systems and one enhancement system. The results of extensive experiments show that our attack can achieve over 90% of overall attack success rate in feature space and up to 100% attack success rate in problem space.

**Roadmap.** The remainder of the paper is organized as follows: §2 introduces fundamental concepts; §3 presents HRAT, the first structural attack on AMD; §4 introduces our APK manipulation tool; §5 evaluates the effectiveness of HRAT; §6 provides analysis for the applications of HRAT on other systems and discusses potential limitations; §7 reviews relevant work; §8 concludes the paper.

### 2 PRELIMINARY

In this section, we present the necessary knowledge on our target Android malware detection (AMD) systems (i.e., Malscan [5]...
and Mamadroid [6]) and one AMD enhancement method (i.e., APIGraph [11]). Both of the two AMD tools use FCGs to detect Android malware. Besides, we present basic knowledge about reinforcement learning to ease the explanation of our methods later.

2.1 Target Systems

Malscan. Malscan extracts FCGs (G) from Apps and uses centralities [38, 39] of sensitive nodes in G as features. Those sensitive nodes correspond to Android’s sensitive APIs, which reflect the malicious properties [5, 40] of APKs. Let c(G) denote the centralities of all nodes in G. The label of target G in Malscan is formulated as:

\[ y = f \left( l_{cen} \times c \left( G \right) \right), \]

where \( l_{cen} = [i_1, ..., i_N] \in R^{1 \times N} \) is the sensitive index of nodes in G, N is the number of nodes, i_k \in [0, 1] and i_k = 1 means that node k is sensitive; G \in R^{N \times N} is the adjacent matrix of FCG, f(·) is the pre-trained classifier (i.e., kNN in Malscan).

Mamadroid. Mamadroid [6] extracts function call relations from FCGs as features. Based on the characteristic of Android method signature, it first abstracts methods into different states according to the package name or family name. By doing so, it will be resilient to API changes in Android framework [6]. Next, Mamadroid extracts the probabilities between states, i.e., families or packages of target methods, as features and trains classifiers like kNN to detect malware. The label of G is represented as:

\[ y = f \left( T_{cp} \left( S \times G \right), S = \{s_{ij}\} \in R^{N \times N}, \right) \]

where N is the number of nodes in G and N_k is the number of states in Mamadroid, s_{ij} \in [0, 1] and s_{ij} = 1 denotes that node j belongs to state i. T_{cp}(·) is the call probabilities among states, and f(·) is pre-trained classifier.

APIGraph. APIGraph [11] is a framework to enhance AMD. It enhances the representation ability of features and uses the characteristics of Android to aggregate APIs with the similar semantics. Specifically, APIGraph first collects API documents from Android official website and then builds the connections among APIs using a relation graph. Based on the relation graph, a graph embedding algorithm is applied to get each API’s embedding vector. APIGraph uses a clustering algorithm, like k-means, to aggregate APIs with similar semantics into one cluster. To enhance the target AMD, APIGraph uses a specific API to denote all APIs in one cluster during the feature extraction process.

2.2 Reinforcement Learning

Reinforcement learning (RL) [37] learns what to do and how to take actions based on current situations, through interacting with the target environment, and maximizes the reward from the environment’s feedback. Different from supervised learning, which learns from the training set, corresponding knowledge (labels of samples in the training set), and an external supervisor (objective function), RL learns without prior knowledge. Unlike unsupervised learning, which targets leveraging a hidden structure in the unlabeled data set, such as the distributions or representations, RL targets maximizing a reward signal by interacting with the target environment. Although evolutionary algorithms (EAs) could approach RL problems, EAs are more suitable to solve problems whose policy space is small and can be structured or the problems whose learning agent cannot accurately sense the environment. Moreover, EAs neither learn from the environment nor formulate the relation between actions and environment’s states [37].

Reinforcement learning contains four elements: an action set, a state set, a reward function and a policy model. The state set stores all the possible states of the target environment. For example, in the maze problem, the state set saves all possible positions of the player in the maze. Action set contains all the actions that the learner can take. For instance, in the maze problem, it includes the directions the player can move forward at each step. Reward function defines the environment feedback for current state. For example, in the maze problem, if the player walks out of the maze, the greatest reward is given. In middle states, which refer to any positions between the entrance and the exit, the closer the player is to the exit, the higher the reward will be given. Policy model defines how current action influences current state. An essential property of reinforcement learning is that the problem to be solved should conform to the Markov Decision Process (MDP) [41]. Only when this condition is met, the action and reward in reinforcement learning can be formulated as a function of the current state.

3 ATTACK MODEL

This section presents our threat model (§3.1), attack formulation (§3.2), and optimization process (§3.3). We also theoretically prove the effectiveness and advantages of our structural attack (§3.4).

3.1 Threat Model

In our attack scenario, the adversary possesses the white-box access to target systems. That is, the adversary has access to dataset, feature space and model parameters of target systems. This setting follows Kerckhoffs’ principle [1] and ensures that a defense does not rely on “security by obscurity” by unreasonably assuming some properties of the defense that can be kept secret [42]. Our attack aims at modifying the function call graph of a malicious App to evade target system’s detection. Two FCG-based AMD, i.e., Malscan [5] and Mamadroid [6], and one AMD enhancement method, namely APIGraph [11], are used to evaluate the effectiveness of our attack. We select Malscan and Mamadroid because they are the state-of-the-art FCG-based AMDs (§7) and published in top conferences, i.e., ASE 2019 and NDSS 2017 respectively, with influential impact. They report outstanding malware detection performance (98% detection accuracy for Malscan and 99% F1 score for Mamadroid).

3.2 Attack Formulation

Our structural attack \( \mathcal{K} \) takes in FCG (adjacent matrix) and modifies the nodes and edges in the graph to deceive the detection systems. Our attack is defined as:

\[ \tilde{G} = \mathcal{K} (G) = G + \delta, \]

where G is input graph, \( \tilde{G} \) is the adversarial graph, and \( \delta \) is the perturbations to the adjacent matrix of graph. We use \( f(G, \theta) \) to denote a malware detection system, which takes in G and uses the pre-trained parameters \( \theta \) to determine the category to which G belongs. Since our attack only modifies the structure of \( \tilde{G} \) without changing parameters \( \theta \), the detection system can be simplified to
\( f(G) \). The goal of our attack is to modify \( G \) to make the system \( f(G) \) misclassify \( G \) as benign, i.e., \( f(G) \neq f(\hat{G}) = f(K(G)) \).

To ensure that the modified App works normally and preserves the same functionality as the original one, we design four types of modification actions, namely adding edges, rewiring, inserting nodes and deleting nodes, which takes into account the characteristics of Apps. Next, we first introduce the definition of constraints and then describe the definitions and properties of each action.

**Definition 1.** (Constraints) \( C = \{c_1, ..., c_n\} \in R^{N_a \times 1} \) where \( N_a \) is the number of nodes in graph \( G \), \( c_i \in \{0, 1\} \) denotes the modifiability of node \( i \). \( c_i = 1 \) indicates that the node \( i \) is modifiable, otherwise \( c_i = 0 \).

Definition 1 defines the edges and nodes that cannot be modified during our attack process. They refer to the scenarios when some specific methods in Apps (§4.1) cannot be modified. For example, we cannot modify Android framework APIs used in App [23].

**Definition 2.** (Adding edge) An adding edge action \( A_2 \) involves two nodes \( \hat{A}_2 = \{v_{beg}, v_{tar}\} \), where \( v_{beg}, v_{tar} \notin C \) are the caller and callee of the edge to be added respectively. The adding edge operation builds an invocation relation from \( v_{beg} \) to \( v_{tar} \).

**Definition 3.** (Rewiring) Rewiring removes an edge from the graph and finds another intermediate node to maintain the connectivity of nodes in the deleted edge. A rewiring action \( A_3 \) involves three nodes \( \hat{A}_3 = \{v_{beg}, v_{end}, v_{mid}\} \), where \( v_{beg}, v_{end} \notin C \) are the caller and callee in deleted edge respectively and \( v_{mid} \notin C \) is the intermediate node. The rewiring action deletes the edge between \( v_{beg} \) and \( v_{end} \), and creates a new edge from \( v_{beg} \) to \( v_{mid} \), and from \( v_{mid} \) to \( v_{end} \).

**Definition 4.** (Inserting node) An inserting node action \( A_4 \) involves one node \( A_4 = \{\hat{v}_{call}er\} \). It creates a new node \( \hat{v}_{call}er \) on the graph and then builds an invocation relation from \( \hat{v}_{call}er \notin C \) to \( \hat{v}_{call}er \).

**Definition 5.** (Deleting node) A deleting node action \( A_5 \) involves three types of node \( \hat{A}_5 = \{v_{var}, \hat{v}_{call}er, \hat{\hat{v}}_{call}er\} \), where \( v_{var} \notin C \) denotes a node to be removed, \( \hat{v}_{call}er \notin C \) is the set of nodes that call \( v_{var} \), and \( \hat{\hat{v}}_{call}er \) is the set of nodes called by \( v_{var} \). Deleting node action deletes nodes \( v_{var} \) in the graph and build call relations from all nodes in \( \hat{v}_{call}er \) to each node in \( \hat{\hat{v}}_{call}er \).

According to Definition 5, when a node is deleted, the connectivity of the remaining nodes in the graph keeps unchanged.

So far, we have defined all four operations of the attack. Then, our attack process is formulated as:

\[
K(G) \equiv (a_1, a_2, ..., a_m) G, \\
\text{where } a_i \in \{ \hat{A}_{\text{arc}}, \hat{A}_{\text{revwi}}, \hat{A}_{\text{im}}, \hat{A}_{\text{dn}} \}.
\]
18 if which greatly saves memory. Since current state only depends on $\epsilon$ (a two layer fully connected neural network). With probability $\epsilon$.

Execute action

return

Perform gradient descent:

$y = y_1 + y_2 + \cdots + y_t$

Store action in action sequence:

$Q$ and rewards) for further learning with an action-value network.

17 structural attack. We first initialize (line 1-2) a memory list with capacity $N$, probability $\epsilon$, maximum modification times $M$, feature-space transformation $T$, node constraints $C$.

Output: action sequence $a_{seq}$ adversarial graph $G'$

Initialize replay memory $D$ to capacity $N$

Initialize action-value network $Q$ with random weights $\theta$

Ob $f_{adv}(G) = \sum_{i=1}^{m} \epsilon_i \cdot \sigma(||x_i - T(G)||_2)$

$y_t = f(G), y_p = f(G), G_t = G$

while $y_p = y_t$ and $i < M$

$\text{tmp} = \text{random\_probability}$

if $\text{tmp} < \epsilon$

$\text{else}$

$\text{Calculate gradient of each edge: } \delta g = \nabla G f_{\text{adv}}(G_i)$

Execute action $a_1$ on $G_i$: $G_{i+1}, r_i = \text{ATT\_OBJ}(G_i, \delta g, C)$

$D \leftarrow (T(G_i), a_i, r_i, G_{i+1})$

$y_j = \begin{cases} r_j, & \text{if } i \text{ terminates at } j + 1 \\ r_j + \max_a Q(G_{i+1}, a'; \theta), & \text{otherwise} \end{cases}$

Perform gradient descent: $\nabla g = y_j - Q(G_i, a_i; \theta)$

Store action in action sequence: $a_{seq} \leftarrow a_i$

$y_p = f(G_{i+1}), i++$

if $y_p \neq y_t$

return $a_{seq}, G'$

- Easy handling: The feature vector has a fixed length and is suitable for training the policy network using states, actions and rewards. We do not use adjacent matrix because how to dynamically handle the changes of graph scale is out of the scope of this paper.

- Easy storing: It is easy to store the feature vector. Since the FCG of an App can have hundreds of thousands and even millions of nodes and edges, a sufficiently large amount of memory is required if the intermediate graphs are stored directly. Fortunately, the features extracted from the graph by these target systems are usually a one-dimensional vector with tens of thousands of dimensions, which greatly saves memory. Since current state only depends on the previous state and action (§3.3), storing the latest graph structure is sufficient for determining the next action.

3.3.2 Policy Model. The policy model determines the attack action that consists of determining an action type and selecting attack objects. HRAT determines an attack action based on reinforcement learning, specifically deep Q-learning [43].

Algorithm 1 sketches the flow of deep Q-learning process for structural attack. We first initialize (line 1-2) a memory list with capacity $N$ to store the experience (i.e., the action type, graph state and rewards) for further learning with an action-value network $Q$ (a two layer fully connected neural network). With probability $\epsilon$ (0.95 as default [43]), HRAT determines the action type using $Q$; otherwise, HRAT randomly selects one action type (line 7-10). With determined action type, HRAT uses gradient search to select the optimal attack objects (i.e., nodes/edges). If the target system uses kNN, which is non-differentiable, as its classifier, we first transform kNN into a differentiable version [44]:

$$\text{Ob}_{f_{adv}}(G) = \arg \min_{\epsilon} \sum_{i=1}^{m} \omega_i \cdot \sigma(||x_i - T(G)||_2),$$  \hspace{1cm} (5)

where $m$ is number of instances in training set. $\omega_i = 1$ if the label of $x_i$ equals $y_{\text{adv}}$. otherwise $\omega_i = -1$. $\sigma(x) = \frac{1}{1 + e^{-x}}$ is a sigmoid function, and $T(\cdot)$ is the transformation function from FCGs to feature vectors in the target system. Differentiating the objective function with respect to the edges in graph $G_i$, we obtain the gradient of each edge (line 11). With action type and gradient of each edge, HRAT conducts current action on $G_i$ and obtains the modified graph $G_{i+1}$ and the corresponding reward § 3.3.4 (line 12). Next, HRAT stores the experience (line 13) for further learning and optimizing $Q$ (line 13-15). Each step’s experience is potentially used in weight updates, which allows for greater data efficiency [43] (Algorithm 1 line 7-8). Besides, HRAT stores features $T(G)$ and latest $N$ experience tuples in the replay memory. This setting that stores feature vectors instead of graph saves memory to a great extent and stores the interaction experience. Next, we will introduce how to utilize gradient search to guide the modification of the graph according to each action type.

- Adding edge. We first calculate the gradient of each edge in the adjacent matrix. Then, we extract the gradients of all adding edge actions and select the edge, denoted as a two-tuple of nodes (\{v\_beg, v\_end\}), with maximum gradients to add. Notably, when we select \{v\_beg, v\_end\}, if one node is in the constraints C, i.e., v\_beg $\in$ C or v\_end $\notin$ C, we select the edge with the second-largest gradient until both nodes are not in C (line 5-9 of Algorithm 2). In this way, we can guarantee that the edges in the output modification sequence, $a_{seq}$, can be modified in the App.

- Rewiring. We first select the edge, \{v\_beg, v\_end\}, with maximum gradient to remove (line 4-6 of Algorithm 3). Similar to adding edges, we need to make sure $v_{\text{beg}} \notin C$. It worth noticing that we do not modify the callee (v\_end) during manipulation. Then, to maintain the App’s functionality according to Definition 3, we select an intermediate node, $v_{\text{mid}}$, that has no connection to $v_{\text{beg}}$ and $v_{\text{end}}$ and is not in $C$, with the maximum gradient sum of $\{v_{\text{beg}}, v_{\text{mid}}\}$ and \{v\_mid, v\_end\} (line 8-13 of Algorithm 3).

- Inserting node. We create a new method (v\_new) and calculate the gradient from each manipulable node (v\_candi $\notin$ C) to v\_new. Then, the edge \{v\_candi, v\_new\} with maximum gradient is built (line 2-4 of Algorithm 4). Building edge from an existing node to the inserted node guarantees that the static analysis will not exclude the inserted node (method) as dead code. Finally, we update $C$ by adding $v_{\text{new}}$ with a modifiable index (line 6 of Algorithm 4).

- Deleting node. When deleting a node (v\_tar), we need to maintain the connectivity between the methods calling v\_tar and the methods called by the v\_tar. According to Definition 5, the gradient (g(·)) of node i, is defined as:

$$g(i) = \sum_{j} v_{ij} \cdot g(v_{ij}) + \sum_{j} v_{ij} \cdot \sum_k (1 - v_{jk}) \cdot g(v_{jk}),$$  \hspace{1cm} (6)

where $v_{ij} = 1$ denotes existing connections between node $i$ and $j$, and vice versa. The first item calculates the gradient sum of all edges to node $i$. The second item calculates the gradient sum of all
edges from node $j$, which invokes node $i$, to node $i$, which are called by node $i$. For the node that has maximum gradient and is not in constraint set, we remove it from the adjacent matrix and build the connections from each of its caller to all of its callee (line 3 of Algorithm 5). Finally, we update $\eta_t$ by removing $\eta_{tar}$.

3.3.3 Action Space. The action space stores the operations to modify the graph. Each action is represented as a four-tuple which stores the action type (the first element) and action objects (the remaining three elements).

3.3.4 Reward Function. The reward function evaluates the effect of the selected action on the current state, i.e., graph. Since the goal of our attack is to make the system’s decision on the modified graph different from the decision on the original graph $f(\hat{G}) \neq f(G)$ by modifying graphs as few as possible, our reward function is designed as follows:

$$R(s_t, a_t) = \begin{cases} 1 & \text{if } f(\hat{G}) \neq f(G) \\ -(\Delta N_{\text{node}} + \Delta N_{\text{edge}}) & \text{if } f(\hat{G}) = f(G) \end{cases},$$

(7)

where $\Delta N_{\text{node}}$ and $\Delta N_{\text{edge}}$ denote the differences between the number of nodes and the number of edges in the current graph and the original graph, respectively.

This reward function assesses the impacts of attack action types and attack objects. The reward of adding edge is -1 because only one edge is modified no matter which edge the gradient search selects. The reward of rewiring and inserting nodes are -3 and -2, respectively. For removing nodes, the reward depends on the node that gradient search selects. Removing nodes deletes one nodes $\eta_{tar}$ from $G$ and builds connections between each of $\eta_{tar}$’s callers to all of $\eta_{tar}$’s callee. The reward of removing one node is:

$$R(\cdot)_{rn} = 1 + N^r_{\text{callers}} - N^r_{\text{callees}},$$

(8)

where $N^r_{\text{callers}}$ and $N^r_{\text{callees}}$ are the number of callers and callees of $\eta_{tar}$, respectively.

3.4 Structural Attack Analysis

We first analyze how graph-based algorithms learn and extract features from FCGs. Malscan uses centralities of sensitive nodes (sensitive APIs [40]) in FCGs to represent graph semantics. Malscan demonstrates that the centrality quantifies the importance of sensitive nodes in graphs, and sensitive nodes can be used to characterize the Apps’ malicious behaviors. Mamadroid uses the function call probability in FCGs as features. To unify the size of the extracted vector, Mamadroid abstracts functions into different clusters based on the families or packages.

According to the algorithms adopted by target systems, we analyze the influence of each structural modification on them. For Malscan, we take degree centrality ($d_{cen}$) as an example:

$$d_{cen} = \frac{d_i}{N_o - 1},$$

(9)

where $d_i$ denotes the degree of node $i$ and $N_o$ denotes the number of nodes in a FCG. When we add one edge (method call) between node $i$ and any of the other node, the $d_{cen}$ increases and then degree centrality of node $i$ increases; and vice versa for deleting one edge; when we add one node to a FCG, $N_o$ increases and then degree centrality of node $i$ decreases and vice versa for deleting one node. For Mamadroid, the function call probability ($f_{cp}$) is calculated by:

$$f_{cp}(i,j) = \frac{f_N(i,j)}{\sum_{k=1}^{N} f_N(i,k)},$$

(10)

where $f_N(i,j)$ denotes the number of callers from state $i$ to $j$, $N$ is the total number of states (i.e., the number of method families). In this way, when we insert one edge from state $i$ to $j$, the numerator and denominator increase by 1 and the $f_{cp}(i,j)$ increases and vice versa for deleting one edge; when we delete one nodes $k$ from state $i$, all callers to $k$ will lost. To preserve the functionality, those callers integrate the code of $k$ and invoke the callees of $k$. In this way, $f_N(i,j)$ decreases, $\sum_{k=1}^{N} f_N(i,k)$ increases and then the probability from state $i$ to $j$ decreases.

Structural attacks bridge the gap between feature-space attacks, which only perturb feature vectors to deceive the classifier, and problem-space attacks, which generate adversarial objects. It is also known as the inverse feature-mapping problem [1]. Existing problem-space attacks [1, 23, 30] are limited by inverse feature-mapping problem (i.e., the optimized feature-space attacks cannot be perfectly mapped into problem-space attacks), which can also cause side-effect and decreases the attack success rate. Structural attacks modify the nodes and edges in one FCG that contains the methods and call relations in an App. Hence, our four FCG modification actions correspond to the manipulations on Apps (§4).

4 ANDROID APPLICATION MANIPULATION

Our Android App modifier (APPMOD) automatically manipulates an App according to the graph modification sequence following two principles: a) Functional Consistency: the App’s functionality before and after modifications should be consistent; b) Valid Modifications: the inserted code will not be identified and removed by static analysis. Specifically, static analysis can detect dead code that will be never executed [1] and remove it. In this case, the graph extracted from the manipulated App will not include the corresponding nodes or edges, i.e., the modifications are invalid.

The prototype of HRAT (i.e., APPMOD) is built on soot [45]. APPMOD modifies Apps using soot, which translates Android bytecode to an intermediate representation without the need of Apps’ source code. According to method signatures, as shown in Figure 2, APPMOD locates the methods in App and conducts the modifications. Figure 3 sketches the work flow of APPMOD. Next, we introduce how APPMOD implements the four manipulation operations: adding function call (adding edge), rewiring function calls (rewiring), inserting method (inserting node), deleting method (deleting node). We first introduce how to determine the constraints, which define whether the methods are modifiable by soot.

4.1 Constraints Determination

Constraints list the methods in APPs and their modifiability. When HRAT modifies nodes and edges in FCGs, the constraints guides HRAT to only modify modifiable methods and call relations. We determine unmodifiable methods according to the properties of
Figure 3: Work flow of APPMOD

![Diagram of work flow of APPMOD](image)

Figure 4: Pseudo code of adding function call.

```plaintext
1. packageName1.className1 returnType1
caller(parameterList1){
  callee(parameterList2, True);
  ... raw caller method body ...
}

(a) Modified target caller.

1. packageName2.className2 returnType2
callee(parameterList2, True){
  ... raw callee method body ...
}

(b) Modified callee.

1. returnType1 oriCaller(parameterList1){ ...
2. tmp = callee(parameterList2);
3. tmp = callee(parameterList2, False);

(c) Modified callers of original callee.

Android, Flowdroid [46], and soot. The following methods are unmodifiable:

- **Framework APIs**: framework APIs are defined and implemented in Android system rather than the App so they are unmodifiable.
- **Lifecycle methods**: lifecycle methods can be invoked by Android system. If we delete or add connection to the lifecycle method, such modifications may lead to App crash.
- **Flowdroid methods**: Since FlowDroid introduces additional methods (e.g., fake main method [47]) to facilitate the analysis, the FCGs will include these methods. But it is worth noting that the Apps under investigation do not include such auxiliary methods.

### 4.2 Adding Function Calls

This operation manipulates two methods (i.e., a `caller` and a `callee`) and all statements that invoke `callee`. Figure 4 shows the pseudo code of adding a function call, where the code in blue will be inserted and the code in gray will be removed. To add a function call, we insert an extra parameter `FLAG` in `callee` (line 1 of Figure 4b), and then insert a statement to invoke `callee` in `caller`'s method body. To keep the functional consistency, APPMOD inserts a conditional expression on `FLAG` in `callee`'s method body (line 2 of Figure 4a). If `FLAG=True`, it indicates the invocation is an inserted call, and `callee` directly returns a default value that has the same type as that of `callee`'s return value (line 2-3 of Figure 4b). If `FLAG=False`, it indicates the invocation is an original call, and the `callee` runs as usual. Besides, to maintain the functional consistency, APPMOD modifies all statements that initially invoke `callee` and sets `FLAG` to False (e.g., line 2-3 in Figure 4c).

### 4.3 Rewiring Function Calls

According to Definition 3, we implement rewiring by modifying three methods: a `caller`, a `callee`, and an intermediate method (`imm`). This operation includes three steps: a) build call relations between `imm` and `callee`, b) replace call relations between `caller` and `callee`, and c) update the original call statements.

- **Step a.** Figure 5b illustrates a modified intermediate method. APPMOD adds an extra parameter `FLAG` to `imm`'s parameter list to determine whether the invocation is an original one or an intermediate invocation (i.e., `caller` invokes `imm`). If it is an original invocation, `imm` runs its original method body (line 6 of Figure 5b); otherwise, it invokes `callee` (line 3-5 of Figure 5b). Note that the `imm`'s return type and `callee`'s return type may be inconsistent. To handle this issue, APPMOD introduces a new data type (i.e., `newType`) in Figure 5a) that includes both `imm`'s and `callee`'s return types. When `imm` needs to return data, APPMOD assigns the original data to the object of `newType` and then returns the object to caller (line 8 in Figure 5b). To ensure that `callee` works as original, we extend the parameter list of `imm` and pass the `callee`'s variables used for invoking `callee` to `imm` when invoking `imm` (line 3 of Figure 5c).

- **Step b.** APPMOD modifies `callee`'s function body and replaces the statements that originally invoke `callee` with new statements to invoke the intermediate method. Then, APPMOD sets `FLAG` to `True` to indicate that the invocation is an original one. Then, `callee` obtains the return value of `callee` (line 3 of Figure 5c).

- **Step c.** As the method signature (i.e., parameter list and return type) of `imm` is changed, APPMOD finds all statements that invoke `imm` and updates them coordinately. Specifically, APPMOD first locates the invocation statements and adds the parameters of `callee` with their default value, e.g., `0` for `Integer` (line 4 of Figure 5c), to the end of the `imm`'s parameter list. Then, APPMOD sets `FLAG` to `False` that indicates an original invocation (line 4 of Figure 5d). In this way, when those methods invoke `imm`, `imm` runs as usual (line 6 of Figure 5b) and the functional consistency of `imm` is preserved.
4.4 Inserting Methods
This operation first creates a new method and then finds one existing method (i.e., caller) to invoke it. APPMOD creates a method that performs simple mathematical calculations and returns the results (Figure 6a). Then, APPMOD inserts an invocation statement in caller’s method body (line 3 in Figure 6b). The caller gets the returned value of inserted invocation, and uses the returned value to perform mathematical calculations in caller’s method body (line 4 in Figure 6b), so that the inserted method will neither be excluded as dead code nor affect the App’s functionalities.

4.5 Deleting Methods
This operation removes the target method (tarmethod) and modifies all methods that invoke it. Figure 7 shows the pseudo code of deleting tarmethod. First, APPMOD finds all methods that invoke tarmethod and locates the corresponding invocation statements. Then, APPMOD replaces the invocation statements with tarmethod’s method body. More specifically, APPMOD firstly creates local variables (lv_caller) in caller that include tarmethod’s parameter variables (pv_tar) and local variables (lv_tar). Then, APPMOD assigns the variables used for invoking tarmethod to pv_tar (line 2-3 in Figure 7b). Since we recreate tarmethod’s variables in caller, which are different from those originally used by tarmethod, APPMOD needs to rewrite the tarmethod’s statements rather than directly copying the statements from tarmethod to callers. For example, APPMOD uses soot’s APIs newAssignStmt(), newInvokeStmt() to rewrite the assign statements and invoke statements with newVar, respectively. Moreover, if tarmethod has a return value, APPMOD replaces the return statement with an assignment statement (line 4 of Figure 7a to line 5 of Figure 7b). Since the return statements will end the invocation, if the precondition of one return statement is met, the program will end the invocation directly. Besides, one method may contain multi return statements. To avoid affecting the invocation logic, when APPMOD replaces one return statement, APPMOD inserts a goto statement to let the program jump to the next statement of the statements (i.e., line 5 in Figure 7b) that initially invoke tarmethod.

5 EVALUATION
Figure 8 sketches the attack flow of HRAT. Given an App, HRAT first extracts its FCG ① and generates a graph modification sequence (GMS) ② where each item indicates a perturbation on the FCG’s nodes or edges. Then, we convert a GMS to the manipulation sequence where each item denotes a manipulation on App ③. APPMOD modifies the App according to the manipulation sequence ④ and repackage it ⑤. Finally, we evaluate whether target systems (i.e., Malscan, Mamadroid and APIGraph enhanced Malscan) can detect the adversarial malware ⑥. We evaluate the performance of HRAT by answering the following five research questions.

• RQ1: Effectiveness analysis. How effective is HRAT against the state-of-the-art AMD techniques?
• RQ2: Modification efficiency comparisons. Compared with other attack methods, how is the modification efficiency of HRAT?
• RQ3: Effectiveness of IMA. How effective is the intervention method action (IMA)?
• RQ4: Resilience to code obfuscation. How is the resilience of HRAT to malware with different obfuscation techniques?
• RQ5: Functional consistency assessment. Do the adversarial app generated by HRAT preserve the functionality as the original one?

Dataset. We adopt the dataset that includes 11,613 benign Apps and 11,583 malicious Apps from 2011 to 2018 in Malscan [5] to evaluate HRAT (for RQ1-3&5). All Apps are collected from AndroZoo [48] and each sample has been detected by several antivirus systems in...
Table 1: ASRs of HRAT towards Malscan, Mamadroid and APIGraph enhanced Malscan

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Training</th>
<th>Testing</th>
<th>Init ASR</th>
<th>Rela ASR</th>
<th>Abs ASR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Malscan</td>
<td>TRo</td>
<td>TE0</td>
<td>82.50%</td>
<td>91.31%</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TE1</td>
<td>94.23%</td>
<td>96.71%</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TE2</td>
<td>97.83%</td>
<td>93.86%</td>
<td>100%</td>
</tr>
<tr>
<td>APIGraph + MalScan</td>
<td>TRo</td>
<td>TE0</td>
<td>89.67%</td>
<td>97.50%</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TE1</td>
<td>99.57%</td>
<td>98.93%</td>
<td>100%</td>
</tr>
<tr>
<td>Mamadroid</td>
<td>TRo</td>
<td>TE0</td>
<td>71.42%</td>
<td>87.88%</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TE1</td>
<td>99.94%</td>
<td>94.95%</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TE2</td>
<td>100%</td>
<td>88.79%</td>
<td>100%</td>
</tr>
</tbody>
</table>

5.1 RQ1: Effectiveness Analysis

Experimental Setup. We divide the dataset into training sets and testing sets. As HRAT aims to modify malware to evade the target detection system, the testing sets only contain malware. Since modifying misclassified malware makes no sense, we use the pre-trained classifiers to exclude those samples. We design three dataset [5] settings for effectiveness analysis. In the first setting, the training dataset (TRo) and the testing dataset (TEo) are collected during the same period. In the second setting, the testing data (TE1 and TE2) was collected after the training set. This setting emulates the situation that the malware detectors are trained with known malware and use pre-trained classifiers to detect malware. In this case, since there may be concept drift [53] in the malware samples, detectors are suggested to re-train their classifiers to deal with new malware. The third setting uses the latest malware to train the classifier (TR2) and use older malware (TE1) for testing. In this setting, we aim at evaluating whether our attack can renew outdated malware. We use each training set to train target AMDs (i.e., Malscan, APIGraph enhanced Malscan and Mamadroid). Given a malware in testing sets, we use HRAT to modify it and then evaluate whether it can evade target AMDs and record each ASR.

We also compare the performance of HRAT and that of other approaches. One is Android HIV [23], which is the state-of-the-art attack against Mamadroid and also considers problem-space attacks. Since Android HIV has not been released to the public, we implement Android HIV by strictly following the description and configuration in [23]. We also design attack strategies based on evolutionary algorithms [54] (i.e., simulated annealing-based structural attack, SACK, hill-climbing-based structural attack, HACK, and evolutionary programming-based structural attack, EPACK, see Appendix E) as baseline algorithms for comparisons with reinforcement learning adopted by HRAT.

Results. Table 1 lists the results of effectiveness comparison of different attacks. Init ASR represents the ratio of malware that escapes detection after at most 500 modifications have been applied. According to our analysis in §3.4, the characteristics of malicious Apps will eventually be diluted so that they will be regarded as benign ones as long as HRAT keeps adding useless vertices. But the unlimited modifications will increase the attack’s computational complexity. We applied HRAT to randomly selected 50 Apps, which fail to escape the detection after 500 modifications, without the restriction on the number of modification until they can escape the
Due to the limitations of soot and flowdroid [1], some Apps cannot be successfully re-packaged. Thus, we utilize Rela ASR to denote the ratio of Apps that can evade the detection at algorithm level but cannot be re-packaged successfully. It is worth noting that the failure of App re-packaging is usually due to the anti-re-packaging strategies used by those Apps instead of our manipulations.

Comparing the Init ASR and Rela ASR of Malscan and APIGraph enhanced Malscan in Table 1, we can see that HRAT is more effective on APIGraph enhanced detector than the original detector. The reason is that the APIs that cannot be modified originally may become modifiable because APIGraph uses a unique API to express APIs with similar functionalities. Moreover, we observe that it is easier to dilute the characteristics of malware to evade the detection if the number of features is smaller. For example, the feature number of Mamadroid (121) is much smaller than that of Malscan (43,972), and the Init ASR on Mamadroid in Table 1 is better than Malscan.

Table 2 illustrates the ASR of different algorithms using TRo for training and TE1 for testing. We can see that for feature-space attack (Init ASR), Android HIV can achieve over 95% attack success rate. However, for problem-space attack, Android HIV’s performance drops to 37% because Android HIV can only modify a limited number of methods in malware and cannot keep consistency between perturbations on features and modifications on Apps. Moreover, the initial ASRs of SACK, HACK and EPACK are lower than that of reinforcement learning based algorithm, because evolutionary algorithms guide structural attacks to randomly select attack action type. Benefiting from structural attack, EA-based methods achieve the same absolute ASRs as reinforcement learning guided attack.

Answer to RQ1: HRAT achieves up to 99.94% ASR within 500 modifications. Without restriction on the number of modifications, HRAT can achieve 100% ASR of problem-space attack.

5.2 RQ2: Modification Efficiency Comparison

Experimental Setup. We measure the modification efficiency of an attack using the number of modifications required to let a malicious App evade the detection. We compare the modification efficiency of HRAT and that of Android HIV and SACK (see Appendix E). Note that HRAT and SACK work against both Malscan and Mamadroid whereas Android HIV only targets Mamadroid.

Results. Figure 9 shows the cumulative distribution (CDF) of the required number of modifications for evasion. We can see that even SACK achieves comparative ASR with HRAT (§5.1), SACK requires more modifications to make target malware escape detection. Specifically, When attacking Malscan through SACK, more than 10% of malware needs more than 50 modifications. By contrast, this ratio is only about 5% under HRAT’s attack. When attacking Mamadroid through HRAT, 90% of the malware needs at most 50 modifications to evade the detection. However, SACK requires at least 150 modifications to achieve the same ratio. This difference may be caused by the different learning strategies of those two algorithms. More precisely, HRAT uses reinforcement learning to learn and decide the action type by interacting with the target environment whereas SACK randomly selects the action type and decides whether to adopt the action by checking if the selected action has a positive impact. For Android HIV, nearly 30% of adversarial malware escapes detection within 80 modifications and only 65% of malware deceives Mamadroid within 300 modifications.

Answer to RQ2: HRAT needs fewer number of modifications than other methods to let malicious Apps evade the detection.

5.3 RQ3: Effectiveness of IMA

5.3.1 Ratio of individual actions. To evaluate the effectiveness of individual manipulation action (IMA), we compute the ratio of each attack action to all modifications applied to all adversarial Apps that successfully evade the target systems (i.e., Mamadroid, Malscan and APIGraph enhanced Malscan) under the aforementioned data sets and configurations. For example, to obtain the ratio of insert node in Malscan_TRo_TE1, we first count the number of insert node (denoted as N_{in}) applied to those adversarial Apps and the total
Table 3: Comparisons between individual attack strategies and HRAT

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Init ASR</th>
<th>Rela ASR</th>
<th>Abs ASR</th>
<th>Avg. Mod</th>
</tr>
</thead>
<tbody>
<tr>
<td>HRAT</td>
<td>94.23%</td>
<td>96.71%</td>
<td>100%</td>
<td>5.58</td>
</tr>
<tr>
<td>Add edge</td>
<td>96.97%</td>
<td>82.31%</td>
<td>100%</td>
<td>11.03</td>
</tr>
<tr>
<td>Rewiring</td>
<td>67.14%</td>
<td>79.58%</td>
<td>100%</td>
<td>5.74</td>
</tr>
<tr>
<td>Insert node</td>
<td>81.48%</td>
<td>98.65%</td>
<td>100%</td>
<td>14.68</td>
</tr>
<tr>
<td>Delete node</td>
<td>74.17%</td>
<td>93.76%</td>
<td>100%</td>
<td>8.06</td>
</tr>
<tr>
<td>Malscan HRAT</td>
<td>99.94%</td>
<td>94.95%</td>
<td>100%</td>
<td>34.55</td>
</tr>
<tr>
<td>Add edge</td>
<td>93.77%</td>
<td>94.90%</td>
<td>100%</td>
<td>57.75</td>
</tr>
<tr>
<td>Rewiring</td>
<td>83.95%</td>
<td>75.53%</td>
<td>100%</td>
<td>27.64</td>
</tr>
<tr>
<td>Insert node</td>
<td>96.76%</td>
<td>98.29%</td>
<td>100%</td>
<td>64.26</td>
</tr>
<tr>
<td>Delete node</td>
<td>78.49%</td>
<td>83.17%</td>
<td>100%</td>
<td>23.24</td>
</tr>
<tr>
<td>Mamadroid</td>
<td>99.57%</td>
<td>98.93%</td>
<td>100%</td>
<td>1.63</td>
</tr>
<tr>
<td>Add edge</td>
<td>91.51%</td>
<td>98.00%</td>
<td>100%</td>
<td>3.39</td>
</tr>
<tr>
<td>Rewiring</td>
<td>95.61%</td>
<td>85.46%</td>
<td>100%</td>
<td>2.57</td>
</tr>
<tr>
<td>Insert node</td>
<td>92.20%</td>
<td>95.80%</td>
<td>100%</td>
<td>2.81</td>
</tr>
<tr>
<td>Delete node</td>
<td>94.62%</td>
<td>96.07%</td>
<td>100%</td>
<td>4.29</td>
</tr>
</tbody>
</table>

number of modifications (denoted as $N_m$) and then compute the ratio as $N_{an}/N_m$.

**Results.** Figure 10 illustrates the ratio of each attack action in adversarial samples. The deeper the shade, the greater the ratio. We can see that for evading Malscan, inserting nodes and deleting nodes actions account for a large proportion (over 26%) because adding edges and deleting nodes can effectively decrease the degree centrality of nodes ($\S$ 3.4). It is consistent with the analysis of Malscan in [5] that suggests the degree centralities of benign Apps is smaller than that of malware. For APIGraph enhanced Malscan, rewiring action accounts for a large ratio (over 27%). The reason may be that as APIGraph clusters methods with similar semantics into one class, rewiring action can effectively decrease the degree centrality of target clusters by replacing the connections between different clusters with connections within on clusters. As Mamadroid abstracts methods into different families and uses invocation probabilities as features, inserting nodes to specific families could be more effective to perturb the features. Besides, as the action ratios of all actions exceed 20%, it suggests that HRAT actively selects attack actions to achieve the trade-off between ASR and modification efficiency.

5.3.3 **ASR of individual actions.** To evaluate whether malicious Apps can escape the detection with only one type of attack action, we compare the ASR of HRAT and that of IMA attacks against aforementioned detectors. TRO and TE1 is adopted as training and testing set, respectively. For each attack action, we just use it to perturb the structure of target graphs and record its ASR. We also compute the average number of modifications ($\AvgMod$) required by each IMA by first counting the number of modifications ($N_{mod}$) applied to $N_{ad}$ adversarial samples that successfully evade the detectors in both feature space and problem space and then calculating the ratio of $N_{mod}/N_{ad}$.

**Results.** Table 3 shows that using adding edges alone to attack Malscan can achieve over 90% Init ASR but requires 11.03 average modifications that is nearly double of the number of modifications required by HRAT (5.58). Regarding Mamadroid, both adding edges and inserting nodes can achieve over 90% Init ASR in feature space, while rewiring and deleting nodes only achieves 83.95% and 78.49% ASR, respectively. However, adding edges and inserting nodes require more average modifications than rewiring and deleting nodes. Combing different attack actions together, HRAT achieves nearly 100% Init ASR against Mamadroid with much better modification efficiency. As APIGraph clusters similar APIs into one class to enhance target AMD, perturbations on fewer APIs can let malware escape the enhanced detector. In other words, when APIGraph enhances target systems, it also introduces new vulnerability. As all IMAs follow our graph structure modifications, their performance of problem-space attacks is the same as that of feature space attacks.

5.3.4 **Effectiveness of HRAT on malware that fail to escape detection using individual actions.** To evaluate whether combining multiple attack actions is more effective than individual attack actions, we use HRAT to modify Apps that fail to evade the target systems (i.e., Malscan, APIGraph enhanced Malscan and Mamadroid) using individual attack actions. If $N_e$ Apps fail to deceive target systems using individual actions but $N_o$ out of $N_e$ Apps successfully escape the detection under HRAT’s modifications, we define the effective ratio as $N_e/N_o$ to quantify HRAT’s effectiveness.

**Results.** Figure 12 shows the effective ratio of HRAT over Apps that fail to evade detection systems using individual attack actions. For attacking Malscan, since adding nodes action achieves comparative Init ASR with HRAT, the malware that fails to deceive Malscan using adding nodes has overlapping with malware that fails to evade the detection under HRAT’s modification, leading to 37.5% effective ratio. For Mamadroid and enhanced detector, as HRAT achieves nearly 100% Init ASR (Table 3), HRAT can also achieve nearly 100% and around 90% effective ratio, respectively.

**Answer to RQ4:** Individual attack actions are not always effective in attacking all AMDs. Combining multiple attack actions, HRAT is much more effective and modification efficient.

5.4 **RQ4: Resilience to Obfuscation Techniques**

**Experimental Setup.** To evaluate the resilience of HRAT against malware adopting different obfuscation techniques, we use a dataset in [50], which includes malware from different families obfuscated by three different obfuscation techniques (i.e., identifier renaming, string encryption and reflection). We use TRO as the basic training set. For each malware family, we randomly select 100 samples from the dataset and add them to the training set to improve classifier’s performance. To construct the testing set, we randomly select 500 samples that are correctly identified as malware by target systems. We apply HRAT to above malware for evading target AMD systems (i.e., Malscan and Mamadroid), and define evasion rate = $N_e/N_t$, where $N_e$ is the number of malware that escapes the detection, and $N_t$ is the number of test samples, to quantify the resiliency of HRAT against these commonly used obfuscation techniques.

**Results.** As shown in Table 4, the evasion rate of Malscan and Mamadroid are 100%, meaning that our approach is resilient to identifier renaming. The reason is that the renaming obfuscation can only change the names of parameters or identifiers into meaningless strings or hash values [51], which do not affect the structure of FCGs. For string encryption, the evasion rates of Malscan and Mamadroid are 93.94% and 87.11%, respectively, meaning that this obfuscation technique can affect our approach. Our manual analysis reveals that string encryption may affect the graph structure because malware may use encrypted string to replace original invocation statement and restore them at run-time [50], and thus some
nodes and edges are missed during the modification. For example, when deleting a node, if the callee's name is encrypted in one call relation, HRAT fails to replace the method invocation statement with the deleted method body in the corresponding method. It may cause HRAT to break the functionality of target malware and make the performance of problem-space attack (evasion rate) lower than 100%. If reflection is used, the evasion rate of Malscan and Mamadroid are 92.08% and 91.30%, respectively, meaning that reflection may affect our approach. Reflection may make it difficult to conduct static analysis on Apps and thus some invocation relations may be missed.

### Answer to RQ4: If an obfuscation technique neither affects the structure of FCGs nor impedes the static analysis and manipulation on Apps, it will not affect our approach.

### 5.5 RQ5: Functional Consistency Assessment

We conduct static analysis and dynamic analysis to assess whether the adversarial Apps generated by HRAT preserves the functionality as the original ones. We randomly select 40 malicious Apps and apply HRAT to them. During this process, we also insert log into the modified methods of original and modified apps to collect information for assessment.

- **Static analysis assessment.** For these App pairs, we conduct static analysis on them to ensure that the modifications have been correctly imposed. Specifically, we check whether the added invocations and methods exists, whether the deleted methods are correctly modified. Besides, we also compare the scale of nodes and edge of FCGs and extracted features of modified App and those obtained by HRAT.

- **Dynamic analysis assessment.** For the dynamic assessment, we install the Apps before and after modifications on two Android virtual machines (AVMs) with the same configuration, respectively. For Apps (35/40) that have been modified by less than ten times, we manually analyze their FCGs and 33/35 Apps show the same functionality. Two of thirty five Apps crashed during the testing process whereas their original Apps run smoothly. By manually inspecting these two Apps, we find that when HRAT modifies the methods called by reflection, it cannot make the corresponding modifications to the reflection invocation and thus leads to App crash. Thus, we cannot verify their functional consistency. For other Apps tested using Monkey, the coverage rate drops to 24.87% as Monkey randomly interacts with the App to trigger methods. For these trigger path, we find all modifications do not affect the functionality.

### Answer to RQ5: HRAT keeps functional consistency of the modified apps in most cases. It may break the functionalities of apps using obfuscation techniques that hinder the static analysis.

### 6 DISCUSSION

#### 6.1 Defense against HRAT

We evaluate two potential approaches for defending against HRAT, including adversarial training and ensemble learning [56], which have been used to defend against adversarial attacks in other domain (e.g., computer version).

Adversarial retraining is regarded as one of the most effective defense methods against adversarial attacks [57]. We randomly select 500 samples that evade the detection in both feature and problem space and divide them into training and testing set. Figure 11 shows that with the increase of training ratio (i.e., the ratio retraining samples to all adversarial samples), ASR drops accordingly. However, Mamadroid trained with TR0 dataset cannot achieve good defense performance even enough adversarial samples are available. It may be due to the limited ability of Mamadroid to learn extracted features for malware detection. Similarly, when retraining ratio is less than 0.5, the attack success rate of Mamadroid TE1 and TE2 remains more than 40%. It is worth noting that Malscan enhanced by APIGraph only needs a small number of retraining samples to successfully detect adversarial Apps. For attacking Malscan, when the training ratio is limited to 10% to 40%, the ASR drop dramatically while the training ratio grows to more than 50%, the defense
effectiveness does not improve a lot. This could be caused by limitation of Malscan’s defense performance as the F1-score of raw Malscan’s detect performance is limited to 98.8% [5]. In summary, retraining would be an effective defense method against HRAT if there are enough adversarial samples for retraining. However, it is worth noting that if many samples are used for retraining, the time consumption and computational workload will also increase.

We also evaluate the effectiveness of ensemble learning (see Appendix A). The results show that it could be a promising defense strategy for Mamadroid. However, it is not always effective for Malscan. Therefore, different defense strategies should be adopted for different detection systems to defend against HRAT.

6.2 Other Limitations

HRAT’s optimization progress leads to high computational consumption, because given a new graph HRAT needs to calculate the gradient of each edge to select the influential nodes or edges. In future work, we will investigate the transferability of HRAT. In other words, we will first use HRAT to attack one detection system and generate adversarial malware, and then check whether the generated malware can escape another detection system. Besides, to address the limitation of gradient related methods, we will explore involving information entropy in modification selection.

Since HRAT relies on static analysis, its attack may break APK’s dynamic features, such as reflection [50], dynamic class loading, etc. We can add related methods into constraints and set them as unmodifiable to avoid modifying such dynamic features. Moreover, HRAT could not handle heavily obfuscated malware [58–62], such as packaged apps, because static analysis may just access the Dex file of the shell rather than real functional Dex file.

7 RELATED WORK

Adversarial attacks against AMD. Many attack methods [1, 23, 33–35, 63, 64] have been proposed to evade AMD, which can be categorized into feature-space attacks [1, 23, 33–35, 63, 65] and problem-space attacks [1, 23, 35].

Feature-space attacks modify the features to deceive the classifiers used by AMD. Grosse1 et al. [33] propose to replace the classifier in Drebin with a neural network to improve detection performance. They also propose Jacobian matrix-based (JM) methods to modify the features to escape the detection. Similarly, Shahpasand et al. [63] use a generative adversarial neural network to generate adversarial features for evading Drebin.

Recently, researchers propose problem-space attacks to generate real adversarial examples. Android HIV [23] builds the transformation relation between feature modification and APK manipulation to transform features (i.e., invocation probability) to the number of invocation. Then, Android HIV applies optimization algorithms to guide perturbations on feature vector. Based on feature perturbations, Android HIV inserts the corresponding numbers of call relations between target methods to generate adversarial malware. Pierazzi et al. [1] extract benign components from the training set and then insert those benign components into malicious samples to generate adversarial malware. Although these studies aim to generate real APK, they just insert no-op code. Li et al. [35] attack Drebin by inserting and removing components, in which removal is achieved by renaming components, in the feature vectors. However, their modification only renames the corresponding string content to deceive the feature extraction methods.

Graph structural attacks. Graph attacks [24–29, 66] can be categorized into poisoning attacks and evasion attacks. Poisoning attacks [24, 25] insert triggers into graphs to poison training data. When a test graph contains the specific structure, the trigger will be activated and the graph will be classified as a predefined class. Existing poisoning attacks [24, 25] only include inserting or deleting edges in their trigger generation methods and assume that attackers can access the training data [1].

Evasion attacks [26–29, 32] modify the structure of graphs or the features of nodes to evade the detection. Except for simply inserting and deleting edges, ReWatt [26] proposes rewiring edges to preserve the graph characteristics. ReWatt uses reinforcement learning to select the most influential edges for modification. The middle nodes in ReWatt’s rewiring are restricted to the second order neighbors of nodes in the original edge to make their modification “unnoticeable” to GCN. Wang et al. [27] deceive the GCN classifier by inserting nodes selected by generative adversarial neural network. Note that existing studies usually ignore deleting nodes. Moreover, existing attacks against malware detection only insert dead code.

8 CONCLUSION

We propose a novel structural attack against FCG-based Android malware detection systems. By leveraging the correlation between FCG and software, our attack bridges the gap between feature-space attacks and problem-space attacks. Through interacting with target systems, our attack is more effective and modification efficient than heuristic methods. Our extensive experiments demonstrate that combining multiple attack actions is more effective than applying single action. Moreover, our methodology can also be applied to platforms other than Android.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their insightful comments. This work was partially supported by Hong Kong RGC Project (No. PolyU15223918) and PolyU Internal Fund (No. BE3U, No. ZVQ8) and the National Natural Science Foundation of China (No. 61571205).
A ENSEMBLE LEARNING BASED DEFENSE METHODS

Ensemble learning has been used to defend against adversarial attacks [56]. To evaluate the defense effectiveness of ensemble learning algorithms, we adopt four ensemble learning algorithms: Bagging, Adaboost, gradient boosting decision tree, and Voting. Bagging [57] forms a class of algorithms which will be trained using a random subset of the original training set and then aggregate their individual predictions to get a final prediction. Adaboost [68] utilizes a sequence of weak learners on repeatedly modified versions to improve the performance of weak classifiers. Gradient boosting decision tree [69] (GBDT) integrates a set of regression trees that use a generalization of boosting to arbitrary differentiable loss functions. Voting strategy simply combines different machine learning classifiers and uses vote to predict the class labels. We use sklear [69] to implement those embedding algorithms and the parameters of each algorithm are set as default (see Table 6).

To conduct the experiments, we use the ensemble learning algorithms to replace the kNN classifier in original detection systems. Then, we use those pre-trained ensemble classifier to identify the label of adversarial malware. Table 5 shows that ensemble learning for APIGraph enhanced system cannot effectively defend HRAT’s attack. For Malscan, Adaboost and GBDT are promising defense strategies which could benefit from their boosting strategies. Since Boosting strategy use weighted methods to combine weak classifiers, the ensemble classifier is supposed to integrate the advantages of weak classifiers, thus making the defense more effective. For Mamadroid, ensemble learning could achieve at least 60% ASR decrease. As the feature number in Mamadroid is small, the learning strategy of bagging (using subset of training set) can effectively exclude outliers in training set.

Table 5: Evaluation of ensemble learning based defense methods

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Bagging</th>
<th>Adaboost</th>
<th>GBDT</th>
<th>Voting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Malscan TE0</td>
<td>91.50%</td>
<td>18.42%</td>
<td>15.79%</td>
<td>100%</td>
</tr>
<tr>
<td>Malscan TE1</td>
<td>92.31%</td>
<td>15.03%</td>
<td>13.99%</td>
<td>100%</td>
</tr>
<tr>
<td>APIGraph</td>
<td>100%</td>
<td>99.84%</td>
<td>97.28%</td>
<td>100%</td>
</tr>
<tr>
<td>Mamadroid TE0</td>
<td>14.86%</td>
<td>32.32%</td>
<td>33.19%</td>
<td>34.20%</td>
</tr>
<tr>
<td>Mamadroid TE1</td>
<td>30.91%</td>
<td>36.75%</td>
<td>35.14%</td>
<td>24.64%</td>
</tr>
<tr>
<td>Mamadroid TE2</td>
<td>14.00%</td>
<td>31.88%</td>
<td>32.64%</td>
<td>36.25%</td>
</tr>
</tbody>
</table>

B OBJECTS SEARCH ALGORITHMS FOR EACH ATTACK ACTION TYPE

Algorithm 2 to 5 illustrate how HRAT searches optimal objects for modification through the gradient search.

Algorithm 2: Adding edge

Input: current graph G, node constraints C

Output: action sequence a_seq, new graph G' |
1 a_seq ← [−1, −1, −1, −1];
2 a_type ← max{NN(s_i)};
3 calculate the gradient d of adding edge in G;
4 d = argsort(beam);
5 for each edge (v_beg, v_end) in beam do
6 if v_beg ∈ C and v_end ∈ C then
7 connect v_beg and v_end in G; G' = G;
8 a_seq = [0, v_beg, v_end, −1];
9 break;
10 return a_seq, G';

Table 6: Parameter settings of ensemble algorithms

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Parameter settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagging</td>
<td>base_estimator: INN</td>
</tr>
<tr>
<td></td>
<td>max_samples: 0.5</td>
</tr>
<tr>
<td></td>
<td>max_features: 0.5</td>
</tr>
<tr>
<td>Adaboost</td>
<td>base_estimator: INN</td>
</tr>
<tr>
<td>GBDT</td>
<td>n_estimators: 100</td>
</tr>
<tr>
<td></td>
<td>learning_rate: 1.0</td>
</tr>
<tr>
<td></td>
<td>max_depth: 1</td>
</tr>
<tr>
<td></td>
<td>random_state: 0</td>
</tr>
<tr>
<td>Voting</td>
<td>base_estimators: SVM, INN, DT</td>
</tr>
<tr>
<td></td>
<td>voting_majority_voting</td>
</tr>
</tbody>
</table>

SVM: support vector machine; INN: 1st nearest neighbors; DT: decision tree.
Algorithm 3: Rewiring

**Input:** current graph $G$, node constraints $C$  
**Output:** action sequence $a_{seq}$, new graph $G'$

1. $v_{end} \leftarrow -1, v_{tar} \leftarrow -1$;  
2. calculate the gradient $\partial$ of deleting edge in $G$;  
3. $\partial = \text{argsort}(\partial)$;  
4. for each edge $(v_{beg}, v_{end})$ in $\partial$ do  
   if $(v_{beg} \notin C)$ then  
      break;  
7. find the intermediate node $v_{mid}$ linking $v_{beg}$ and $v_{beg}$ with the maximum gradient;  
8. if $v_{mid} \notin C$ then  
   disconnect $v_{beg}$ and $v_{end}$ in $G$;  
   connect $v_{beg}$ and $v_{mid}$ in $G$;  
   connect $v_{beg}$ and $v_{mid}$ in $G; G' = G$;  
12. $a_{seq} = \{1, v_{beg}, v_{end}, v_{mid}\}$;  
13. break;  
14. return $a_{seq}, G'$;

Algorithm 4: Inserting node

**Input:** current graph $G$, node constraints $C$  
**Output:** action sequence $a_{seq}$, new graph $G'$

1. insert a new node $v_n$ to $G$;  
2. calculate the gradient $\partial$ of all edges to $v_n$;  
3. find the edge $(v_{beg}, v_{end})$ with the largest gradient and $v_{beg} \notin C$;  
4. connect $v_{beg}$ and $v_{mid}$ in $G$;  
5. $a_{seq} = \{2, v_{beg}, v_{mid}, v_{end}\}$;  
6. update $C$;  
7. return $a_{seq}, G'$;

Algorithm 5: Deleting node

**Input:** current graph $G$, node constraints $C$  
**Output:** action sequence $a_{seq}$, new graph $G'$

1. calculate the gradient $\partial$ of all nodes;  
2. find the nodes $v_{tar} \notin C$ with largest gradient;  
3. remove $v_{tar}$ in $G$; connect each caller of $v_{tar}$ to all callees of $v_{tar}$ in $G$; $G' = G$;  
4. $a_{seq} = \{3, v_{tar}, -1, -1\}$;  
5. update $C$;  
6. return $a_{seq}, G'$;

C EFFECTIVENESS OF HRAT ON MALWARE THAT CANNOT EVADE DETECTION UNDER INDIVIDUAL ATTACK ACTION

Figure 12 demonstrates the effectiveness of HRAT over malware that fails to escape detection under individual attack action.

D WHY USING REINFORCEMENT LEARNING?

Our structural attack aims at modifying malware to escape target systems’ detection with minimal modifications: modifying the fewest edges and nodes. Our attack considers four types of attack actions and the decision of each action consists of determining an action type and selecting attack objects.

For greedy algorithm that chooses locally optimal solutions and concatenates them together to approximate the optimal global solution, it will always select adding edge at each step because adding edge modifies one edge each time, rewiring modifies two edges, inserting nodes modifies one edge and one node, and deleting nodes modifies at least one node and one edge. However, adding edges alone cannot always achieve optimal perturbations. For example, if decreasing the degree centrality of certain nodes is locally optimal for the state, adding edges cannot achieve it because adding edge can only increase the degree centralities. Besides, our experimental results 5.3 also demonstrate that adding edges cannot achieve desirable attack performance.

Evolutionary algorithm (EA) effectively solves problems whose policy spaces (i.e., attack action set) are small or can be structured [37]. In our scenario, the policy space is enormous (number of nodes and edges in target graph) and cannot be structured because each attack action consists of a) selecting an action type and b) determining attack objects. The determination of b) depends on a), and a) is related to previous actions ($\S$ 3.3).

HRAT uses reinforcement learning, specifically deep Q-learning, to determine the attack action on the target graph. Our policy model involves two parts. The first part uses a neural network to learn the relation between state and action type with the loss function based on reward. Then, with the determined action type, HRAT uses gradient search to determine attack objects. Besides, to evaluate the effectiveness of each attack action, our reward function, i.e., Equation 7, is designed as the number of modified nodes and edges. This reward function evaluates the impact of both action type and attack objects. For example, adding one edge, the reward value is -1 as only one edge is added, no matter which edge the gradient search selects. For rewiring, the reward is -3 because one edge is
removed and 2 edges are added. For inserting nodes, the reward is 2 because one node and one edge are inserted. For deleting nodes, the value of reward depend on the node the gradient search selects. Since deleting nodes will remove one node from the target graph and build the connections from each of the deleted node’s caller to all of its callees, the reward value depends on the number of deleted nodes’ callers and callees.

E EVOLUTIONARY ALGORITHMS BASED STRUCTURAL ATTACK

We also evaluate evolutionary algorithms, specifically simulated annealing [54], hill-climbing [70] and evolutionary programming[71], for comparisons. Evolutionary algorithms are suitable for scenarios whose policy space is small or can be structured [37]. However, in our attack scenario, the policy space, i.e., the attack action set, is enormous and unable to be structured (Appendix D). We adopt the ideal of evolutionary algorithms and adapt it to our scenarios. Combining gradient search, we design three evolutionary algorithms based structural attacks: 1) Simulated Annealing based structural attaCK(SACK), 2) Hill-climbing based structural AttaCK (HACK) and 3) Evolutionary Programming based structural AttaCK (EPACK). Next, we use HACK as an example to introduce how we use evolutionary algorithms to guide a structural attack.

In the initialization phase, we set the initial temperature, the target temperature, and the temperature drop ratio. To initialize the state, we randomly select an action type from four candidate types and then use the gradient search to modify the graph based on the selected action type. In the original simulated annealing (SA) algorithm, each step should randomly select a state from the neighbors of the state. Intuitively, the neighbors should be obtained by 1) extracting the corresponding representation feature for the latest graph; 2) conducting all possible modifications to the graph; 3) extracting features from all modified graphs – candidate state set. Then, SA uses a predefined distance formula, such as Euclidean distance, to calculate distance between latest features and candidate

Algorithm 6: SACK: simulated annealing based structural attack

| Input: Target classifier f, training dataset \{Xt, Yt\}, target FCG G, feature-space transformation T, maximum modification times M, node constraints C, cooling ratio r |
| Output: action sequence a_seq, adversarial graph G' |

1. Initial and final temperature \(T_i\) and \(T_f\)
2. Initialize: \(i = 0\), \(T = T_i\), \(G_i = G\), \(y_t = f(G), y_p = f(G)\), \(cost_i = min(dist(f(G), X_t))\)
3. while \(T > T_f\) and \(i < M\) and \(y_t \neq y_p\) do
   4. \(a_i = random\_type\)
   5. Calculate gradient of each edge: \(a_G = \nabla_{G_i} Obj_{jade}(G_i)\)
   6. Execute action \(a_i\) on \(G_i\): \(G_{i+1} = ATT\_OBJ(G_i, a_G, C)\)
   7. \(cost_{i+1} = min(dist(f(G_{i+1}), X_t))\)
   8. \(prob = \exp(- (cost_{i+1} - cost_i) / T)\)
   9. if \(cost_{i+1} < cost\) or rand_num < prob then
      10. Store action in action sequence: \(a_{seq} \leftarrow a_i\)
      11. \(G_{i+1} = G_{i+1}, cost = cost_{i+1}\)
      12. \(T = T \times r\)
      13. \(y_p = f(G_{i+1}), \ i++\)
   14. if \(y_p \neq y_t\) then
      15. return \(a_{seq}, G'\)

Algorithm 7: HACK: hill-climbing based structural attack

| Input: Target classifier f, training dataset \{Xt, Yt\}, target FCG G, maximum modification times M, node constraints C |
| Output: action sequence a_seq, adversarial graph G' |

1. Initialize: \(i = 0\), \(G_i = G\), \(y_t = f(G), y_p = f(G)\), \(cost_i = min(dist(f(G), X_t))\)
2. while \(i < M\) and \(y_t \neq y_p\) do
   3. \(a_i = random\_type\)
   4. Calculate gradient of each edge: \(a_G = \nabla_{G_i} Obj_{jade}(G_i)\)
   5. Execute action \(a_i\) on \(G_i\): \(G_{i+1} = ATT\_OBJ(G_i, a_G, C)\)
   6. \(cost_{i+1} = min(dist(f(G_{i+1}), X_t))\)
   7. if \(cost_{i+1} < cost\) then
      8. Store action in action sequence: \(a_{seq} \leftarrow a_i\)
      9. \(G_{i+1} = G_{i+1}, cost = cost_{i+1}\)
     10. \(y_p = f(G_{i+1}), \ i++\)
   11. if \(y_p \neq y_t\) then
      12. return \(a_{seq}, G'\)

Algorithm 8: EPACK: evolutionary programming based structural attack

| Input: Target classifier f, training dataset \{Xt, Yt\}, target FCG G, maximum modification times M, node constraints C, number of action types \(n_a\) |
| Output: action sequence a_seq, adversarial graph G' |

1. Initialize: \(i = 0\), \(G_i = G\), \(y_t = f(G), y_p = f(G)\), \(cost_i = min(dist(f(G), X_t))\)
2. while \(i < M\) and \(y_t \neq y_p\) do
   3. Randomize mutation probability: \(p_m = p_{i1}, ..., p_{i_{na}}\)
   4. \(G_{i+1} = G_i, tmp\_act\)
   5. for \(a_i\) from \(0\) to \(a_i < n_a\) do
     6. if \(p_a > 1/n\) then
        7. Calculate gradient of each edge: \(a_G = \nabla_{G_i} Obj_{jade}(G_{i+1})\)
        8. Execute action \(a_i\) on \(G_i\):
        9. \(G_{i+1} = ATT\_OBJ(G_{i+1}, a_G, C)\)
        10. \(tmp\_act \leftarrow a_i\)
   11. if \(cost_{i+1} < cost\) then
      12. Store action in action sequence: \(a_{seq} \leftarrow tmp\_act\)
      13. \(G_{i+1} = G_{i+1}, cost = cost_{i+1}\)
     14. \(y_p = f(G_{i+1}), \ i++\)
   15. if \(y_p \neq y_t\) then
      16. return \(a_{seq}, G'\)
state set, and select the nearest neighbor. However, this method is not feasible because even without adding nodes, the number of all possible modifications is very large. Assuming that there are $N$ nodes in the target graph, the number of all possible modifications is:

$$N_{\text{mod}} = \binom{1}{N} \times \left( \binom{1}{N \times N} \times \binom{1}{(N-2) \times (N-2)} \right) \times \binom{1}{N}$$  \hspace{1cm} (12)$$

where $\binom{j}{i}$ is combination. The first item calculates the number of possible situations of adding edges, the second item calculates the number of possible situations of rewiring and the third for deleting nodes. Considering a malware that has 1000 nodes and 20% of nodes are modifiable, we have over 12 quadrillion possible modifications. Besides, as we can insert arbitrary numbers of nodes, the scale of the candidate solution set is infinite. Thus, we randomly select an action type and use gradient search to conduct the action on the graph, which also ensures the fairness of comparison with HRAT. We define the cost as the nearest distance from modified graph to benign graphs in the training set (Algorithm 8 line 7). This setting follows the intuition that our target is to device kNN classifiers. Then, if the latest solution is better than previous one, i.e., $\text{cost}_{i+1} < \text{cost}_i$, or the with the probability less than $\exp\left(-\frac{(\text{cost}_{i+1} - \text{cost}_i)}{T}\right)$, SACK will adopt the attack action. Otherwise, SACK continues to modify the graph based on previous state. Different from SACK, HACK adopts all actions that have positive impacts on the state. In each epoch, EPACK randomly generates the mutation probability of each attack action. Then, EPACK adopts all attack actions whose mutation probability is larger than $1/n_a$, $n_a$ is the number of all action types.

F INFLUENCE OF KEY PARAMETERS

Two key parameters will influence the effectiveness and efficiency of HRAT: a) probability, which determines how likely HRAT adopts an action type learned by deep Q-network or randomly selects an action type; and b) memory capacity, which determines the frequency that HRAT interacts with the environments. We evaluate the parameter influence of HRAT on Mamadroid and Malscan with 500 randomly selected malware. Figure 13 shows the influence of these two key parameters on HRAT. We can see that our attack on Mamadroid is not sensitive to the parameters. For Malscan, as storage capacity increases, ASR drops, because the increase of storage capacity means that the frequency of interaction between HRAT and the environment is reduced and thus the system cannot better determine its behavior based on the environment. Similarly, this will also lead to more modifications with the increase of memory capacity. Since HRAT takes random attack action type with $1 - \text{Probability}$, when the Probability decreases, the probability of our system taking random attack behavior will increase. This will result in a decrease in the ASR of the system and an increase in the number of modifications taken.

For other parameters in our system, such as maximum modification times $M$, $m$ in Eq 5, we set $M$ to 500 and $m$ as 75 which follows the settings in [44]. We set the value of $M$ to 500, because as $M$ increases, ASR will also be increased, but the optimization time consumption will also increase. The results of our experiments in § 5.1 demonstrate that if the number of $M$ is not limited, the App can be successfully attacked in the feature space. Similarly, the CDF