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
yulinzhu@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 combing multiple attack behaviors strategically makes the attack more effective and efficient.

CCS CONCEPTS

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

*The corresponding author.

CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8454-4/21/11…$15.00
https://doi.org/10.1145/3460120.3485387

KEYWORDS

Structural attack; Android malware detection; Function call graph

ACM Reference Format:
Kaifa Zhao, Hao Zhou, Yulin Zhu, Xian Zhan, Kai Zhou, Jianfeng Li, Le Yu, Wei Yuan, and Xiapu Luo. 2021. Structural Attack against Graph Based Android Malware Detection. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3460120.3485387

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 relationships. 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

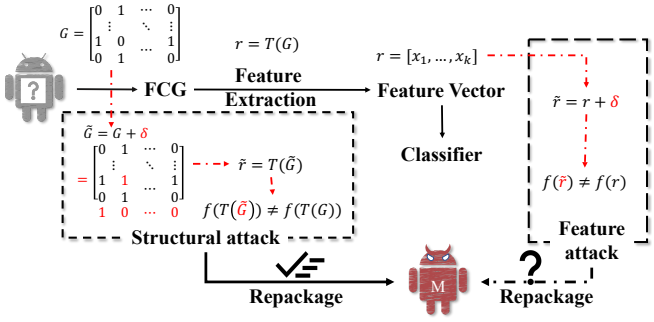

Figure 1: Feature attacks vs. structural attack

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 Drebin [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 Drebin, 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 ATtack (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.)
- **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:

$$y = f\left(I_{cen} \times c\left(G\right)\right), \tag{1}$$

where $I_{cen} = [i_1, ..., i_N] \in R^{1\times N}$ 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, $f(\cdot)$ 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 call 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:

$$y = f\left(T_{cp}\left(S \times G\right)\right), S = \{s_{i,j}\} \in R^{N_s\times N}, \tag{2}$$

where N is the number of nodes in G and N_s is the number of states in Mamadroid, $s_{i,j} \in \{0, 1\}$ and $s_{i,j} = 1$ denotes that node j belongs to state i, $T_{cp}(\cdot)$ is the call probabilities among states, and $f\left(\cdot\right)$ 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:

$$\tilde{G} = \mathcal{K}(G) = G + \delta, \tag{3}$$

where G is input graph, $\tilde{G}$ is the adversarial graph, and δ 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 θ to determine the category to which G belongs. Since our attack only modifies the structure of G without changing parameters θ, 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., $f(G) \neq f(\tilde{G}) = f(\mathcal{K}(G))$.

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)* $\mathbb{C} = [c_1, .., c_{N_n}] \in R^{N_n \times 1}$ *where* N_n *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]. For better formulation, we use $c_n \notin \mathbb{C}$ to denote $c_n = 1$, i.e., the node n is modifiable.

Definition 2. *(Adding edge) An adding edge action* A_a *involves two nodes* $A_a = \{v_{beg}, v_{tar}\}$, *where* $v_{beg}, v_{tar} \notin \mathbb{C}$ *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_r *involves three nodes* $A_r = \{v_{beg}, v_{end}, v_{mid}\}$, *where* $v_{beg}, v_{end} \notin \mathbb{C}$ *are the caller and callee in deleted edge respectively and* $v_{mid} \notin \mathbb{C}$ *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 3 aims to preserve the App's functionality. That is, after deleting an edge that denotes a call relationship (e.g., $A \Rightarrow B$), we must find an intermediate node C to maintain the connection between A and B (i.e., modify the call relationship to $A \Rightarrow C \Rightarrow B$). Although it is possible to insert multiple nodes between A and B to maintain their connectivity after removing the edge between A and B, HRAT inserts one intermediate node in each rewiring action for the ease of implementation. Note that inserting multiple nodes can be achieved by several rewiring operations.

Definition 4. *(Inserting node) An inserting node action* A_i *involves one node* $A_i = \{v_{caller}\}$. *It creates a new node* v_{nce} *on the graph and then builds an invocation relation from* $v_{caller} \notin \mathbb{C}$ *to* v_{nce}.

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

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:

$$\begin{aligned} &\mathcal{K}(G) \Leftrightarrow (a_1, a_2, ..., a_m)\, G, \\ &\text{where } a_i \in \{A_{ae}, A_{rewi}, A_{in}, A_{dn}\}. \end{aligned} \tag{4}$$

3.3 Heuristic Optimized Reinforcement Learning based Structural Attack

Our structural attack process consists of a sequence of attack actions on a target graph. Specifically, the decision of each attack action involves two phases: *determining an action type* and *selecting attack objects*. The former resolves the attack action type (adding edge, rewiring or inserting/deleting node), while the latter determines the specific edges or nodes to be modified according to the action type. Our target is to find the modification sequence with minimum modifications on graph rather than the hidden structure, e.g., the distributions of dataset, in the target graph. Since supervised or unsupervised learning cannot optimize our attack process [37], we leverage reinforcement learning [26, 37, 43] – an iterative learning algorithm that learns how to take actions based on current environment (Appendix D details why we use reinforcement learning rather than other algorithms).

For a target system $f(G)$ that takes in graph G and outputs the decision of G, HRAT aims to modify the structure of the graph, $\hat{G} = \mathcal{K}(G)$ so that the output ($f(\hat{G})$) differs from the original one, i.e., $f(\hat{G}) \neq f(G)$. Following reinforcement learning, our structural attack progress is described as a series of decision-making processes: $P = \{\mathcal{S}, \mathcal{A}, \mathcal{R}, \pi\}$, where $\mathcal{S} = \{s_t\}$ is the state set that contains intermediate and final state of target environment, $\mathcal{A} = \{a_t\}$ is the set of actions that consists of all possible actions in state s_t, $\mathcal{R}$ is a reward function that evaluates the reward of taking action a_t on state s_t, and π is the deterministic policy that describes how to determine a_t and how the action a_t changes the state s_t .

HRAT adopts one attack action at each step to modify target graph until target systems flip their decision on the graph to benign. The structure of the current graph depends on the state of all previous graphs and the modification actions on the graph, i.e., $s_t \mapsto \pi(s_{t-1}, a_{t-1}, ..., s_0, a_0)$. According to HRAT's termination condition (i.e., the target system regards the FCG as benign), the intermediate states s_{mid} are all predicted to the same label as the original FCG. Thus, every step in our attack can be regarded as modifying a new graph. For example, when crafting malware to deceive the target system, HRAT extracts FCG and modifies its structure (methods and call relations). After one modification, HRAT can regard the next modification as modifying a new version of the malware. In other words, current state only depends on latest state and action, i.e., $\pi(s_{t-1}, a_{t-1}, ..., s_0, a_0) = \pi(s_{t-1}, a_{t-1})$, which satisfies the Markov Decision Process (*MDP*) [37].

To choose the influential attack action in each step, we use reinforcement learning to learn the attack process. Reinforcement learning optimizes the decision-making process by continually interacting with the target environment and obtaining rewards. We will introduce each element of reinforcement learning in HRAT.

3.3.1 State Space. The state space stores all intermediate states during the attack process. HRAT chooses feature vectors to store extracted features from corresponding intermediate graphs. For example, when HRAT attacks Malscan, the state space stores all intermediate centrality features of sensitive APIs. HRAT only stores the graph (i.e., adjacent matrix) after the latest modification to facilitate the next modification on the graph. This setting has the following advantages:

Algorithm 1: Deep Q-learning for structural attack

Input: Target classifier f, training dataset $\{X_t, Y_t\}$, target FCG G, memory capacity N, probability ϵ, maximum modification times M, feature-space transformation T, node constraints $\mathbb{C}$

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

1 Initialize replay memory D to capacity N
2 Initialize action-value network Q with random weights θ
3 $Obj_{adv}(G) = \sum_{i=1}^{m} \omega_i \cdot \sigma\left(\|x_i - T(G)\|_2\right)$
4 $y_t = f(G), y_p = f(G), G_i = G$
5 **while** $y_p == y_t$ *and* $i < M$ **do**
6 $\quad tmp = random\_probability$
7 $\quad$ **if** $tmp < \epsilon$ **then**
8 $\quad\quad a_i = argmax_a Q(G_i, a; \theta)$
9 $\quad$ **else**
10 $\quad\quad a_i = random\_action$
11 $\quad$ Calculate gradient of each edge: $\partial_G = \nabla_{G_i} Obj_{adv}(G_i)$
12 $\quad$ Execute action a_i on G_i: $G_{i+1}, r_i = ATT\_OBJ(G_i, \partial_{G_i}, \mathbb{C})$
13 $\quad D \leftarrow (T(G_i), a_i, r_i, T(G_{i+1}))$
14 $\quad y_j = \begin{cases} r_j, \text{ if } i \textit{ terminates at } j+1 \\ r_j + \max_{a'} Q\left(G_{i+1}, a'; \theta\right), otherwise \end{cases}$
15 $\quad$ Perform gradient descent: $\nabla_\theta \left(y_j - Q(G_i, a_i; \theta)\right)$
16 $\quad$ Store action in action sequence: $a_{seq} \leftarrow a_i$
17 $\quad y_p = f(G_{i+1}), \quad i++$
18 **if** $y_p \neq y_t$ **then**
19 $\quad$ **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 ϵ (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]:

$$Obj_{adv}(G) = \arg min_\delta \sum_{i=1}^{m} \omega_i \cdot \sigma\left(\|x_i - T(G)\|_2\right), \tag{5}$$

where m is number of instances in training set. $\omega_i = 1$ if the label of x_i equals y_{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 $\mathbb{C}$, i.e., $v_{beg} \in \mathbb{C}$ or $v_{end} \in \mathbb{C}$, we select the edge with the second-largest gradient until both nodes are not in $\mathbb{C}$ (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 $v_{beg} \notin \mathbb{C}$. 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_{mid} that has no connection to v_{beg} and v_{end} and is not in $\mathbb{C}$, with the maximum gradient sum of $\{v_{beg}, v_{mid}\}$ 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 ($v_{candi} \notin \mathbb{C}$) 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 $\mathbb{C}$ by adding v_{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 ($g(\cdot)$) of node i, is defined as:

$$g(i) = \sum_j v_{ij} \cdot g(v_{ij}) + \sum_j (v_{ji} \cdot \sum_k (1 - v_{jk}) \cdot g(v_{jk})), \tag{6}$$

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 $node_k$, 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 $\mathbb{C}$ by removing v_{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 $f(\hat{G}) = f(\mathcal{K}(G)) \neq f(G)$ by modifying graphs as few as possible, our reward function is designed as follows:

$$\mathcal{R}(s_t, a_t) = \begin{cases} 1 & \text{if } f(\hat{G}) \neq f(G) \\ -(\triangle N_{node} + \triangle N_{edge}) & \text{if } f(\hat{G}) = f(G) \end{cases}, \quad (7)$$

where $\triangle N_{node}$ and $\triangle N_{edge}$ 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 v_{tar} from G and builds connections between each of v_{tar}'s callers to all of v_{tar}'s callee. The reward of removing one node is:

$$\mathcal{R}(\cdot)_{rn} = 1 + N_{caller}^{v_{tar}} \cdot N_{callee}^{v_{tar}}, \quad (8)$$

where $N_{caller}^{v_{tar}}$ and $N_{callee}^{v_{tar}}$ are the number of callers and callees of v_{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:

$$d_{cen\_i} = \frac{d_i}{N_v - 1}, \quad (9)$$

where d_i denotes the degree of node i and N_v 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 deg_i 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_v 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:

```
1. <packageName.className returnType
   methodName(paraList)>
```

Figure 2: Android method signature in soot

$$f_{cp}(i, j) = \frac{f_N(i, j)}{\sum_{k=1}^{N} f_N(i, k)}, \quad (10)$$

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 delete 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 ony modify modifiable methods and call relations. We determine unmodifiable methods according to the properties of

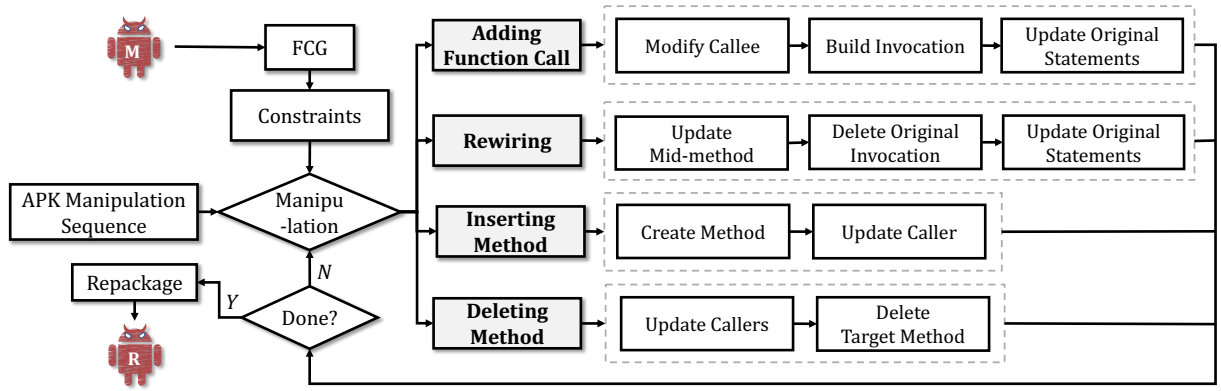

Figure 3: Work flow of APPMOD

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

(a) Modified target caller.

```
1.  packageName2.className2 returnType2
    callee(parameterList2, FLAG){
2.   if (FLAG == True){
3.      return defaultValueofReturnType2;
4.   }else{
5.      ... raw callee method body ... }}
```

(b) Modified callee.

```
1.  returnTypei oriCaller(paraListi){ ...
2.    tmp = callee(paraList2);
3.    tmp = callee(paraList2, False);...}
```

(c) Modified callers of original callee.

Figure 4: Pseudo code of adding function call.

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 *caller*'s variables used for invoking *callee* to *imm* when invoking *imm* (line 3 of Figure 5c).

• **Step b).** APPMOD modifies *caller*'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, *caller* 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.

```
1.  class newType{
2.   returnType1 rt1;  returnType2 rt2; }
```

(a) Insert new data type.

```
1.   returnType1newType imm(paraL3,
    paraL2, FLAG){
2.    vNt = new newType;                ...
3.    if (FLAG == True){
4.        vRt.rt2 = callee(paraL2);
5.        return vRT; }
6.    else{ raw caller method body; } ...
7.    return vRt1;    vNt.rt1 = vRt1;
8.    return vNT;  }
```

(b) Modified intermediate method.

```
1.  returnType2 caller(paraL){  ...
2.     v1 = callee(paraL2);
3.     vtmp = imm(paraL3, paraL2, True);
4.     v1 = vtmp.rt2;            ...    }
```

(c) Modified caller.

```
1.  returnType2 oneImmCaller(paraL1){ ...
2.     tmp = imm(VarL3);
3.     vNt = new newType;
4.     vNt = imm(VarL3,defaultRt2,False);
5.     tmp = vNt.rt1;    ... }
```

(d) Modified original callers of intermediate method.

Figure 5: Pseudo code of rewiring.

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.

```
1.  packName.className int
    newMethod_i(int i1,int i2){
2.      int i3;    i3 = i1 + i2;
3.      return i3; }
```

(a) Create a new method.

```
1.  returnType caller(paraList){
2.      int i;
3.      i = newMethod_i(1, 1);
4.      i = i + 1;
5.      ... raw caller method body  ... }
```

(b) Modified caller to invoke new method.

Figure 6: Pseudo code of inserting methods.

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

```
1.  returnType1 tarMethod(para1, para2){
2.      var1 = para1;  var2 = para2;  ...
3.      { method body of tarMethod }  ...
4.      return var3;                   }
```

(a) Target method to be deleted.

```
1.  returnType2 oneOfCaller(paraL){ ...
2.      v_rt = tarMethod(v1, v2);
3.      var1 = v1;        var2 = v2;
4.      { method body of tarMethod }
5.      v_rt = var3;        ...       }
```

(b) One of modified methods that calls target method.

Figure 7: Pseudo code of deleting methods.

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 *newAssignStemt()*, *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 individual modification 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

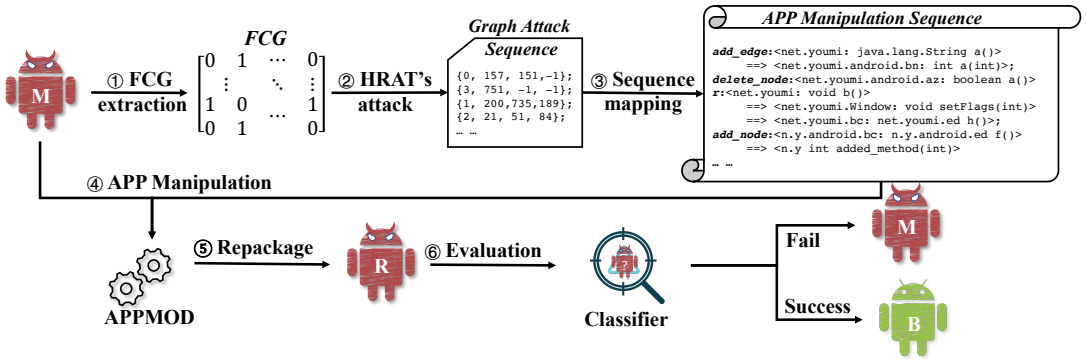

Figure 8: Work flow of HRAT

Table 1: ASRs of HRAT towards Malscan, Mamadroid and APIGraph enhanced Malscan

Algorithm	Training	Testing	Init ASR	Rela ASR	Abs ASR
Malscan	TRo	TEo	82.50%	91.31%	100%
		TE1	94.23%	96.71%	100%
		TE2	97.83%	93.86%	100%
	TR2	TE1	91.50%	97.81%	100%
APIGraph + MalScan	TRo	TEo	89.67%	97.50%	100%
		TE1	99.57%	98.93%	100%
Mamadroid	TRo	TEo	71.42%	87.88%	100%
		TE1	99.94%	94.95%	100%
		TE2	100%	88.79%	100%

VirusTotal [49] to determine their label. For RQ4&5, we use a dataset from previous work [50] to evaluate the effectiveness of HRAT on malware using different obfuscation. This dataset includes Apps from different malware families, and 6586 malware are obfuscated by variable renaming [51], 1090 malware are obfuscated with string encryption [52] and 1172 malware includes reflection [50].

Metrics. To evaluate the effectiveness of the attacks on both feature-space and problem-space, we use three types of attack success rates (ASRs), i.e., *Init_ASR*, *Rela_ASR* and *Abs_ASR*, as our evaluation metrics, which are defined as follows.

$$\begin{aligned} Init\_ASR &= N_g/N \\ Rela\_ASR &= N_p/N_g \\ Abs\_ASR &= N_s/N_p \end{aligned} \tag{11}$$

• *Initialization ASR* (*Init_ASR*) evaluates HRAT's effectiveness on feature space. Given N malware samples, HRAT changes the FCGs of N_g malware samples with at most 500 modifications and make them successfully escape the detection.The threshold (i.e., 500 modifications) selection can refer to RQ2. We found that nearly 100% malware samples can escape detection within 500 manipulations by HRAT. Note that not all modified FCGs can be repackaged due to the anti-repackage protection [20].

• *Relative ASR* (*Rela_ASR*) reflects the rate of successful repackaged malware. That is, among N_g malware samples, N_p samples can be successfully repackaged into App files.

• *Absolute ASR* (*Abs_ASR*) evaluates HRAT's effectiveness on problem space, i.e., whether the repackaged samples can evade the detection and keep the functionality. Among N_p repackaged malware samples, N_s samples run successfully and evade the detector.

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 dateset (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

Table 2: Effectiveness comparison of different attacks

Systems	Algorithms	Init ASR	Rela ASR	Abs ASR
Malscan	HRAT	**94.23%**	96.71%	100%
	SACK	**87.13%**	62.56%	100%
	HACK	75.80%	97.89%	100%
	EPACK	76.63%	94.21%	100%
Mamadroid	HRAT	**99.94%**	94.95%	100%
	SACK	81.05%	84.30%	100%
	HACK	87.40%	81.01%	100%
	EPACK	72.60%	99.17%	100%
	Android HIV	**96.02%**	87.64%	**37.67%**

detection. The result shows that these Apps can successfully evade the detection after more modifications (i.e., from 635 to 4091).

Due to the limitations of *soot* and *flowdroid* [1], some Apps cannot be successfully repackaged. Thus, we utilize *Rela_ASR* to denote the ratio of Apps that can evade the detection at algorithm level but cannot be repackaged successfully. It is worth noting that the failure of App repackaging is usually due to the anti-repackaging 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

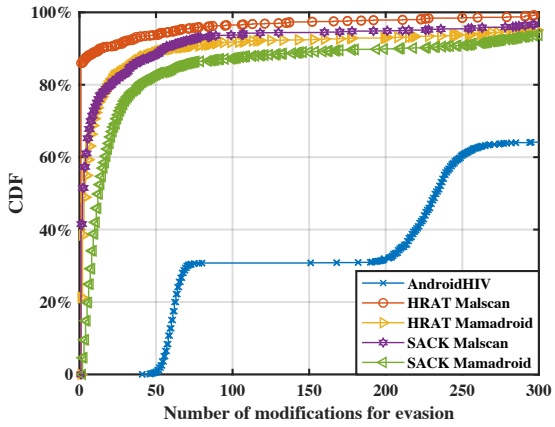

Figure 9: CDF of the required number of modifications

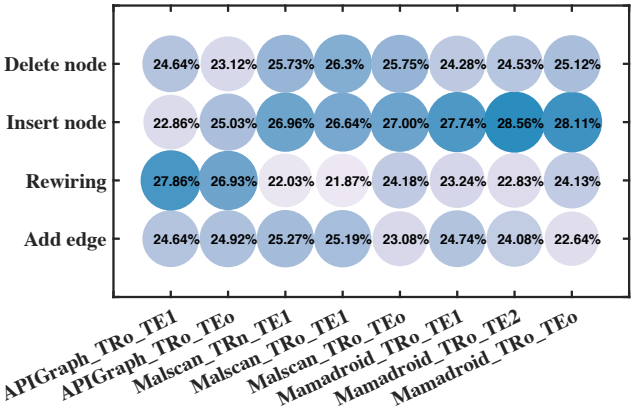

Figure 10: The ratio of each attack action.

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_{an}) applied to those adversarial Apps and the total

Table 3: Comparisons between individual attack strategies and HRAT

Algorithm		Init ASR	Rela ASR	Abs ASR	Avg. Mod
Malscan	HRAT	**94.23%**	96.71%	100%	**5.58**
	Add edge	**96.97%**	82.31%	100%	11.03
	Rewiring	67.14%	79.58%	100%	**7.54**
	Insert node	81.48%	98.65%	100%	14.68
	Delete node	74.17%	93.76%	100%	8.06
Mamadroid	HRAT	**99.94%**	94.95%	100%	**34.55**
	Add edge	93.77%	94.90%	100%	57.75
	Rewiring	83.95%	75.53%	100%	**27.64**
	Insert node	**96.76%**	98.29%	100%	64.26
	Delete node	78.49%	83.17%	100%	**23.24**
APIGraph + MalScan	HRAT	**99.57%**	98.93%	100%	**1.63**
	Add edge	91.51%	98.00%	100%	3.39
	Rewiring	**95.61%**	85.46%	100%	**2.57**
	Insert node	92.20%	95.80%	100%	2.81
	Delete node	94.62%	96.07%	100%	4.29

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 (§ 3.4). It is in 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.

5.3.2 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. More precisely, 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 (*Avg. Mod*) 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* requires 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.

5.3.3 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_s Apps fail to deceive target systems using individual actions but N_h out of N_s Apps successfully escape the detection under HRAT's modifications, we define the effective ratio as N_h/N_s 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 RQ3: 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 $evasion\ rate$s of Malscan and Mamadroid are 93.94% and 87.11%, respectively, meaning that this obfuscation technique can affect our approach. Our manual anlaysis 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

Table 4: Evasion rate of AMD systems by adversarial apps whose original apps belong to different families and adopt three different obfuscation techniques.

	Without attack		With attack	
	Malscan	Mamadroid	Malscan	Mamadroid
Renaming	0.00%	0.00%	100%	100%
Encryption	0.00%	0.00%	93.94%	87.11%
Reflection	0.00%	0.00%	92.08%	91.30%

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 refection 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 (number of nodes and edge) of FCGs and extracted features of modified App and those obtained by HRAT.

Results. The results show that the FCGs extracted from modified Apps are the same as the FCGs computed by the algorithm. Besides, the number of nodes and edges in the FCGs and extracted features from modified Apps are also the same as those calculated 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 to learn how to trigger the modified methods. Then, two authors of this paper conduct the same operations to run an App pair on the two AVMs and record the run-time UI. To check whether the modified methods are triggered, we insert *log* functions to the modified methods to print the method's all parameters and the *callers*'s signature. For example, when testing *adding function call*, we insert log to print the parameters of *callee* and its callers signature between line 1 and 2 of Figure 4b. To check whether the modified App works the same as the original one, we insert *log* to print the parameters of methods to be modified in original App. In this way, besides manually checking whether the user interface gives the same feedbacks (e.g., same activity transition, same pop-up window, same text rendered on window, etc.), we also compare the values of the methods' parameters before and after modification. For Apps (5/40) that are hard to find the activation paths because of heavy code obfuscation, we use a popular Android testing tool Monkey [55] to conduct the dynamic exploration on them. We configure Monkey to let it ignore crashes and timeouts and set the during of each event to 300ms. The execution time is set to 20 minutes. We collect the `logs` to identify the invoked methods and check whether the Apps before and after modification have the same functionality by comparing the log information.

Result. For Apps that have been examined by we manually, the coverage rate ($\frac{N_{triggered\_methods}}{N_{modified\_methods}}$) of modified Apps is 100%, because we have pre-analyzed their FCGs, and 33/35 Apps show the same user interface with original Apps. Specifically, when we conduct the same operations on the Apps, both modified Apps and original Apps output the same UI feedback. Besides, the log messages show the excepted results . For example, for inserted nodes, the expected invocation sequence exist in the corresponding caller. However, 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 TRo 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

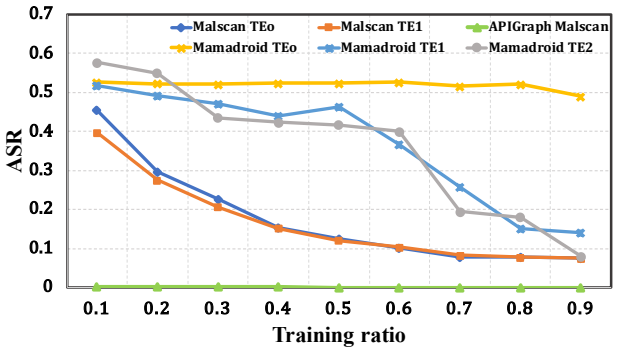

Figure 11: Attack success rate against retraining.

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).

REFERENCES

[1] F. Pierazzi, F. Pendlebury, J. Cortellazzi, and L. Cavallaro. Intriguing properties of adversarial ml attacks in the problem space. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1332–1349. IEEE, 2020.

[2] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. E. R. T. Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, volume 14, pages 23–26, 2014.

[3] J. Li, L. Sun, Q. Yan, Z. Li, W. Srisa-An, and H. Ye. Significant permission identification for machine-learning-based android malware detection. *IEEE Trans Industr Inform*, 14(7):3216–3225, 2018.

[4] T. Kim, B. Kang, M. Rho, S. Sezer, and E. G. Im. A multimodal deep learning method for android malware detection using various features. *IEEE Trans. Inf. Forensics Secur.*, 14(3):773–788, 2018.

[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)*, pages 139–150. IEEE, 2019.

[6] E. Mariconti, L. Onwuzurike, P. Andriotis, E. De Cristofaro, G. Ross, and G. Stringhini. MAMADROID: Detecting Android Malware by Building Markov Chains of Behavioral Models. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2017.

[7] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu. Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 659–674, 2015.

[8] K. Chen, P. Liu, and Y. Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *Proceedings of the 36th International Conference on Software Engineering*, pages 175–186, 2014.

[9] M. Cai, Y. Jiang, C. Gao, H. Li, and W. Yuan. Learning features from enhanced function call graphs for android malware detection. *Neurocomputing*, 423:301–307, 2021.

[10] MW. Hall and K. Kennedy. Efficient call graph analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(3):227–242, 1992.

[11] 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*, pages 757–770, 2020.

[12] Y. Ye, S. Hou, L. Chen, J. Lei, W. Wan, J. Wang, Q. Xiong, and F. Shao. Out-of-sample node representation learning for heterogeneous graph in real-time android malware detection. In *28th International Joint Conference on Artificial Intelligence (IJCAI)*, 2019.

[13] TS. John, T. Thomas, and S. Emmanuel. Graph convolutional networks for android malware detection with system call graphs. In *2020 Third ISEA Conference on Security and Privacy (ISEA-ISAP)*, pages 162–170. IEEE, 2020.

[14] MK. Alzaylaee, SY. Yerima, and S. Sezer. Dl-droid: Deep learning based android malware detection using real devices. *Comput Secur*, 89:101663, 2020.

[15] L. Shi, J. Ming, J. Fu, G. Peng, D. Xu, K. Gao, and X. Pan. Vahunt: Warding off new repackaged android malware in app-virtualization's clothing. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 535–549, 2020.

[16] S. Hou, Y. Ye, Y. Song, and M. Abdulhayoglu. Hindroid: An intelligent android malware detection system based on structured heterogeneous information network. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1507–1515, 2017.

[17] L. Xue, Y. Zhou, T. Chen, X. Luo, and G. Gu. Malton: Towards on-device non-invasive mobile malware analysis for {ART}. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 289–306, 2017.

[18] Statista. Development of new android malware worldwide from june 2016 to march 2020, 2021.

[19] X. Zhan, L. Fan, T. Liu, S. Chen, L. Li, H. Wang, Y. Xu, X. Luo, and Y. Liu. Automated third-party library detection for android applications: Are we there yet? In *ASE*, 2020.

[20] X. Zhan, L. Fan, S. Chen, F. Wu, T. Liu, X. Luo, and Y. Liu. Atvhunter: Reliable version detection of third-party libraries for vulnerability identification in android applications. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1695–1707, 2021.

[21] L. Yu, X. Luo, C. Qian, S. Wang, and H. Leung. Enhancing the description-to-behavior fidelity in android apps with privacy policy. *IEEE Trans. Softw. Eng.*, 44, 2018.

[22] M. Fan, X. Luo, J. Liu, M. Wang, C. Nong, Q. Zheng, and T. Liu. Graph embedding based familial analysis of android malware using unsupervised learning. In *Proc. ICSE*, 2019.

[23] X. Chen, C. Li, D. Wang, S. Wen, J. Zhang, S. Nepal, Y. Xiang, and K. Ren. Android hiv: A study of repackaging malware for evading machine-learning detection. *IEEE Trans. Inf. Forensics Secur.*, 15:987–1001, 2019.

[24] Z. Xi, R. Pang, S. Ji, and T. Wang. Graph backdoor. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[25] Z. Zhang, J. Jia, B. Wang, and NZ. Gong. Backdoor attacks to graph neural networks. In *Proceedings of the 26th ACM Symposium on Access Control Models and Technologies*, pages 15–26, 2021.

[26] Y. Ma, S. Wang, T. Derr, L. Wu, and J. Tang. Attacking graph convolutional networks via rewiring. *arXiv preprint arXiv:1906.03750*, 2019.

[27] X. Wang, M. Cheng, J. Eaton, CJ. Hsieh, and F. Wu. Attack graph convolutional networks by adding fake nodes. *arXiv preprint arXiv:1810.10751*, 2018.

[28] Y. Sun, A. Valente, S. Liu, and D. Wang. Preserve, promote, or attack? gnn explanation via topology perturbation. *arXiv preprint arXiv:2103.13944*, 2021.

[29] H. Dai, H. Li, T. Tian, X. Huang, L. Wang, J. Zhu, and L. Song. Adversarial attack on graph structured data. In *International conference on machine learning*, pages 1115–1124. PMLR, 2018.

[30] E. Quiring, A. Maier, and K. Rieck. Misleading authorship attribution of source code using adversarial learning. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 479–496, 2019.

[31] B. Wang and NZ. Gong. Attacking graph-based classification via manipulating the graph structure. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2023–2040, 2019.

[32] Y. Zhu, Y. Lai, K. Zhao, X. Luo, M. Yuan, J. Ren, and K. Zhou. Binarizedattack: Structural poisoning attacks to graph-based anomaly detection. *arXiv preprint arXiv:2106.09989*, 2021.

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

[34] H. Berger, C. Hajaj, and A. Dvir. When the guard failed the droid: A case study of android malware. *arXiv preprint arXiv:2003.14123*, 2020.

[35] D. Li and Q. Li. Adversarial deep ensemble: Evasion attacks and defenses for malware detection. *IEEE Trans. Inf. Forensics Secur.*, 15:3886–3900, 2020.

[36] I. Rosenberg, A. Shabtai, L. Rokach, and Y. Elovici. Generic black-box end-to-end attack against state of the art api call based malware classifiers. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 490–510. Springer, 2018.

[37] RS. Sutton and AG. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[38] LC. Freeman. Centrality in social networks conceptual clarification. *Social networks*, 1(3):215–239, 1978.

[39] L. Katz. A new status index derived from sociometric analysis. *Psychometrika*, 18(1):39–43, 1953.

[40] KWY. Au, YF. Zhou, Z. Huang, and D. Lie. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228, 2012.

[41] E. Levin, R. Pieraccini, and W. Eckert. Using markov decision process for learning dialogue strategies. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP'98 (Cat. No. 98CH36181)*, volume 1, pages 201–204. IEEE, 1998.

[42] N. Carlini, A. Athalye, N. Papernot, W. Brendel, J. Rauber, D. Tsipras, I. Goodfellow, A. Madry, and A. Kurakin. On evaluating adversarial robustness. *arXiv preprint arXiv:1902.06705*, 2019.

[43] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[44] C. Sitawarin and D. Wagner. On the robustness of deep k-nearest neighbors. In *2019 IEEE Security and Privacy Workshops (SPW)*, pages 1–7. IEEE, 2019.

[45] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing java bytecode using the soot framework: Is it feasible? In *International conference on compiler construction*, pages 18–34. Springer, 2000.

[46] S. Arzt. Static data flow analysis for android applications. 2017.

[47] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, YT. Le, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[48] K. Allix, TF. Bissyandé, J. Klein, and TY. Le. Androzoo: Collecting millions of android apps for the research community. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 468–471. IEEE, 2016.

[49] VirusTotal. Virustotal - free online virus, malware and url scanner, 2021.

[50] S. Dong, M. Li, W. Diao, X Liu, J. Liu, Z. Li, F. Xu, K. Chen, X. Wang, and K. Zhang. Understanding android obfuscation techniques: A large-scale investigation in the wild. In *International conference on security and privacy in communication systems*, pages 172–192. Springer, 2018.

[51] Proguard. https://www.preemptive.com/dotfuscator/pro/userguide/en/protection_obfuscation_renaming.html, 2017.

[52] Dexguard. https://www.guardsquare.com, 2017.

[53] F. Barbero, F. Pendlebury, F. Pierazzi, and L. Cavallaro. Transcending transcend: Revisiting malware classification in the presence of concept drift. *arXiv preprint arXiv:2010.03856*, 2020.

[54] PJM. Van Laarhoven and EHL. Aarts. Simulated annealing. In *Simulated annealing: Theory and applications*, pages 7–15. Springer, 1987.

[55] Google-Monkey. Google Monkey, 2021.
[56] T. Strauss, M. Hanselmann, A. Junginger, and H. Ulmer. Ensemble methods as a defense to adversarial perturbations against deep neural networks. *arXiv preprint arXiv:1709.03423*, 2017.
[57] IJ. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.
[58] Y. Zhang, X. Luo, and H. Yin. DexHunter: toward extracting hidden code from packed Android applications. In *Proc. ESORICS*, 2015.
[59] L. Xue, H. Zhou, X. Luo, Y. Zhou, Y. Shi, G. Gu, F. Zhang, and HM. Au. Happer: Unpacking android apps via a hardware-assisted approach. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1641–1658. IEEE, 2021.
[60] L. Xue, Y. Yan, L. Yan, M. Jiang, X. Luo, D. Wu, and Y. Zhou. Parema: An unpacking framework for demystifying vm-based android packers. 2021.
[61] L. Xue, H. Zhou, X. Luo, L. Yu, D. Wu, Y. Zhou, and X. Ma. Packergrind: An adaptive unpacking system for android apps. *IEEE Trans. Softw. Eng.*, 2020.
[62] H. Zhou, T. Chen, H. Wang, L. Yu, X. Luo, T. Wang, and W. Zhang. Ui obfuscation and its effects on automated ui analysis for android apps. In *Proc. ASE*, 2020.
[63] M. Shahpasand, L. Hamey, D. Vatsalan, and M Xue. Adversarial attacks on mobile malware detection. In *2019 IEEE 1st International Workshop on Artificial Intelligence for Mobile (AI4Mobile)*, pages 17–20. IEEE, 2019.
[64] L. Zhang, P. Liu, and YH. Choi. Semantic-preserving reinforcement learning attack against graph neural networks for malware detection. *arXiv preprint arXiv:2009.05602*, 2020.
[65] S. Hou, Y. Fan, Y. Zhang, Y. Ye, J. Lei, W. Wan, J. Wang, Q. Xiong, and F. Shao. αcyber: Enhancing robustness of android malware detection system against adversarial attacks on heterogeneous graph based model. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, pages 609–618, 2019.
[66] K. Zhou and Y. Vorobeychik. Robust collective classification against structural attacks. In *Conference on Uncertainty in Artificial Intelligence*, pages 250–259. PMLR, 2020.
[67] R. Kohavi and GH. John. Wrappers for feature subset selection. *Artif Intell*, 97(1-2):273–324, 1997.
[68] Y. Freund and RE. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *J Comput Syst Sci*, 55(1):119–139, 1997.
[69] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in Python. *J Mach Learn Res*, 12:2825–2830, 2011.
[70] DB. Skalak. Prototype and feature selection by sampling and random mutation hill climbing algorithms. In *Machine Learning Proceedings 1994*, pages 293–301. Elsevier, 1994.
[71] X. Yao, Y. Liu, and G. Lin. Evolutionary programming made faster. *IEEE Trans. Evol. Comput.*, 3(2):82–102, 1999.

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* [67] 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 and uses 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 sklearn [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

	Bagging	Adaboost	GBDT	Voting
Malscan TEo	91.50%	18.42%	15.79%	100%
Malscan TE1	92.31%	15.03%	13.99%	100%
APIGraph +Malscan	100%	99.84%	97.28%	100%
Mamadroid TEo	14.86%	32.32%	33.19%	34.20%
Mamadroid TE1	30.91%	36.75%	35.14%	24.64%
Mamadroid TE2	14.00%	31.88%	32.64%	36.25%

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 $\mathbb{C}$
Output: action sequence a_{seq}, new graph G'

$a_{seq} \leftarrow [-1, -1, -1, -1]$;
$a_{type} \leftarrow max(NN(s_t))$;
calculate the gradient ∂ of adding edge in G;
$\partial = argsort(\partial)$;
for *each edge* (v_{beg}, v_{end}) *in* ∂ **do**
  if $(v_{beg} \notin \mathbb{C}$ *and* $v_{end} \notin \mathbb{C}$ **then**
    connect v_{beg} and $v_{end} in G$; $G' = G$;
    $a_{seq} = [0, v_{beg}, v_{end}, -1]$;
    break;
return a_{seq}, G' ;

Table 6: Parameter settings of ensemble algorithms

Algorithms	Parameter settings
Bagging	base_estimator: 1NN max_samples: 0.5 max _features: 0.5
Adaboost	base_estimator: 1NN n_estimators:100
GBDT	n_estimators: 100 learning_rate: 1.0 max_depth: 1 random_state: 0
Voting	base_estimators: SVM, 1NN, DT voting:majority voting

SVM: support vector machine; 1NN: 1st nearest neighbors
DT: decision tree

Algorithm 3: Rewiring

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

$v_{end} \leftarrow -1, v_{tar} \leftarrow -1$;
calculate the gradient ∂ of deleting edge in G;
$\partial = argsort(\partial)$;
for *each edge* (v_{beg}, v_{end}) *in* ∂ **do**
  if $(v_{beg} \notin \mathbb{C}$ **then**
    break;
find the intermediate node v_{mid} linking v_{beg} and v_{beg} with the maximum gradient;
if $z_{mid} \notin \mathbb{C}$ **then**
  disconnect v_{beg} and v_{end} in G;
  connect v_{mid} and v_{end} in G;
  connect v_{beg} and v_{mid} in G;$G' = G$;
  $a_{seq} = [1, v_{beg}, v_{end}, v_{mid}]$;
  break;
return a_{seq}, G' ;

Algorithm 4: Inserting node

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

insert a new node v_n to G;
calculate the gradient ∂ of all edges to v_n;
find the edge (v_{beg}, v_n) with the largest gradient and $v_{beg} \notin \mathbb{C}$;
connect v_{beg} and v_n in G; $G' = G$;
$a_{seq} = [2, v_{beg}, v_n, -1]$;
update $\mathbb{C}$;
return a_{seq}, G' ;

Algorithm 5: Deleting node

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

calculate the gradient ∂ of all nodes;
find the nodes $v_{tar} \notin \mathbb{C}$ with largest gradient;
remove v_{tar} in G; connect each caller of v_{tar} to all callee of v_{tar} in G; $G' = G$;
$a_{seq} = [3, v_{tar}, -1, -1]$;
update $\mathbb{C}$;
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.

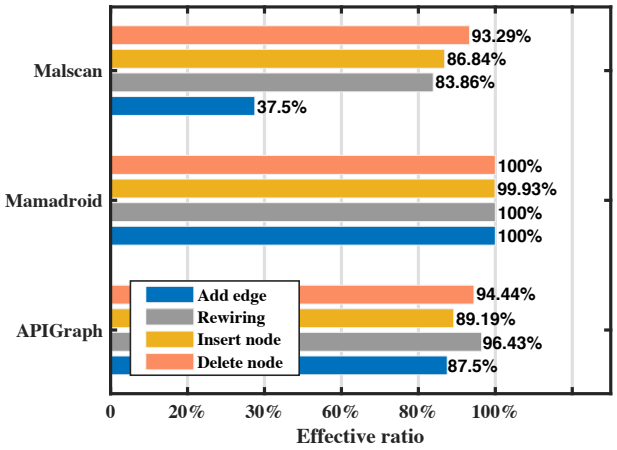

Figure 12: Effectiveness of HRAT over malware that fails to escape detection under individual attack action

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 concretes 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 (§ 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, for 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

Algorithm 6: SACK: simulated annealing based structural attack

Input: Target classifier f, training dataset $\{X_t, Y_t\}$, target FCG G, feature-space transformation T, maximum modification times M, node constraints $\mathbb{C}$, cooling ratio r

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

Initial and final temperature T_i and T_f
Initialize: $i = 0,\ \ T = T_i, G_i = G,\ y_t = f(G), y_p = f(G),$ $cost_i = min(dist(f(G), X_t))$
while $T > T_f$ *and* $i < M$ *and* $y_t == t_p$ **do**
  $a_i = random\_type$
  Calculate gradient of each edge: $\partial_G = \nabla_{G_i} Obj_{adv}(G_i)$
  Execute action a_i on G_i: $G_{i+1} = ATT\_OBJ(G_i, \partial_{G_i}, \mathbb{C})$
  $cost_{i+1} = min(dist(f(G_{i+1}), X_t))$
  $prob = exp(-(cost_{i+1} - cost_i)/T)$
  if $cost_{i+1} < cost$ *or* $rand\_num < prob$ **then**
    Store action in action sequence: $a_{seq} \leftarrow a_i$
    $G_i = G_{i+1},\ cost = cost_{i+1}$
  $T = T * r$
  $y_p = f(G_i),\ \ i++$
if $y_p \neq y_t$ **then**
  return a_{seq}, G'

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 revolutionary 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

Algorithm 7: HACK: hill-climbing based structural attack

Input: Target classifier f, training dataset $\{X_t, Y_t\}$, target FCG G, maximum modification times M, node constraints $\mathbb{C}$

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

Initialize: $i = 0,\ \ G_i = G,\ y_t = f(G), y_p = f(G),$ $cost_i = min(dist(f(G), X_t))$
while $i < M$ *and* $y_t == t_p$ **do**
  $a_i = random\_type$
  Calculate gradient of each edge: $\partial_G = \nabla_{G_i} Obj_{adv}(G_i)$
  Execute action a_i on G_i: $G_{i+1} = ATT\_OBJ(G_i, \partial_{G_i}, \mathbb{C})$
  $cost_{i+1} = min(dist(f(G_{i+1}), X_t))$
  if $cost_{i+1} < cost$ **then**
    Store action in action sequence: $a_{seq} \leftarrow a_i$
    $G_i = G_{i+1},\ cost = cost_{i+1}$
  $y_p = f(G_i),\ \ i++$
if $y_p \neq y_t$ **then**
  return a_{seq}, G'

Algorithm 8: EPACK: evolutionary programming based structural attack

Input: Target classifier f, training dataset $\{X_t, Y_t\}$, target FCG G, maximum modification times M, node constraints $\mathbb{C}$, number of action types n_a

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

Initialize: $i = 0,\ \ G_i = G,\ y_t = f(G), y_p = f(G),$ $cost_i = min(dist(f(G), X_t))$
while $i < M$ *and* $y_t == t_p$ **do**
  Randomize mutation probability: $p_m = p_1, ..., p_{n_a}$
  $G_{i+1} = G_i, tmp_{act}$
  for $a_i = 0$ *to* $a_i < n_a$ **do**
    if $p_{a_i} > 1/n$ **then**
      Calculate gradient of each edge: $\partial_G = \nabla_{G_i} Obj_{adv}(G_{i+1})$
      Execute action a_i on G_i: $G_{i+1} = ATT\_OBJ(G_{i+1}, \partial_{G_{i+1}}, \mathbb{C})$
      $tmp_{act} \leftarrow a_i$
  $cost_{i+1} = min(dist(f(G_{i+1}), X_t))$
  if $cost_{i+1} < cost$ **then**
    Store action in action sequence: $a_{seq} \leftarrow tmp_{act}$
    $G_i = G_{i+1},\ cost = cost_{i+1}$
  $y_p = f(G_i),\ \ i++$
if $y_p \neq y_t$ **then**
  return a_{seq}, G'

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

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:

$$N_{pmod} = C^1_{N\times N} \times (C^1_{N\times N} \times C^1_{(N-2)\times(N-2)}) \times (C^1_N) \quad (12)$$

, where C^j_i is combination. The first item calculates the number of possible situations of *adding edges*, the second item calculates the 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., $cost_{i+1} < cost_i$, or the with the probability less than $exp(-(cost_{i+1} - cost_i)/T)$, 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 $1 - Probability$, 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

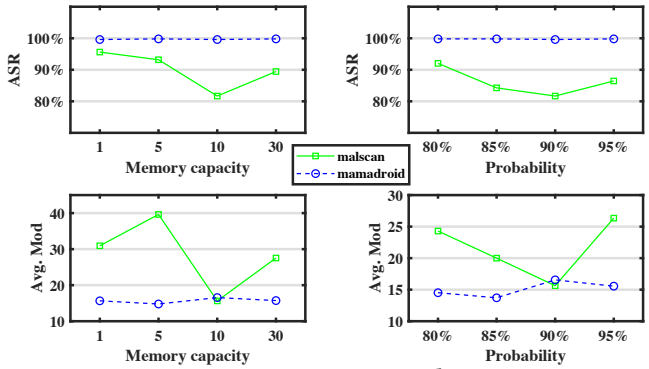

Figure 13: Parameter analysis

of the number of App modifications § 5.2 also shows that most Apps can be successfully modified by at most 50 times. For m, it is only used to solve the distance between the target graph and the m samples in the training set during the optimization process. When evaluating whether the algorithm was successfully attacked, we still used all the samples from the original training set.