

# FCGHUNTER: Towards Evaluating Robustness of Graph-Based Android Malware Detection

Song Shiwen, Xiaofei Xie, Ruitao Feng, Qi Guo, and Sen Chen

**Abstract**—Graph-based detection methods leveraging Function Call Graphs (FCGs) have shown promise for Android malware detection (AMD) due to their semantic insights. However, the deployment of malware detectors in dynamic and hostile environments raises significant concerns about their robustness. While recent approaches evaluate the robustness of FCG-based detectors using adversarial attacks, their effectiveness is constrained by the vast perturbation space, particularly across diverse models and features. To address these challenges, we introduce FCGHUNTER, a novel robustness testing framework for FCG-based AMD systems. Specifically, FCGHUNTER employs innovative techniques to enhance *exploration* and *exploitation* within this huge search space. Initially, it identifies critical areas within the FCG related to malware behaviors to narrow down the perturbation space. We then develop a dependency-aware crossover and mutation method to enhance the *validity* and *diversity* of perturbations, generating diverse FCGs. Furthermore, FCGHUNTER leverages multi-objective feedback to select perturbed FCGs, significantly improving the search process with interpretation-based feature change feedback. Extensive evaluations across 40 scenarios demonstrate that FCGHUNTER achieves an average attack success rate of 87.9%, significantly outperforming baselines by at least 44.7%. Notably, FCGHUNTER achieves a 100% success rate on robust models (e.g., AdaBoost with MalScan), where baselines achieve only 11% or are inapplicable.

**Index Terms**—Android Malware Detection, Function Call Graph, Robustness Testing.

## I. INTRODUCTION

ANDROID malware, such as those designed to steal users' privacy or device resources, has become a major threat to mobile security [1], [2], [3]. This growing threat, fueled by the popularity and openness of the Android platform, has driven the development of various detection methods. In recent years, machine learning (ML)-based approaches have been widely applied in Android malware detection (AMD), demonstrating promising results by leveraging static features of applications [4], [5], [6], [7], [8], [9], [10]. These methods can be mainly divided into two categories, i.e., string-based detection (e.g., Drebin [4]), and graph-based detection (e.g., MalScan [8]). Graph-based methods have emerged as a particularly promising alternative [11], offering superior performance compared to string-based ones [8], [10], [11]. Specifically, such methods use features extracted from the Function Call Graph (FCG) of an Android package kit (APK)'s smali code (i.e., the intermediate representation of an APK after compilation [12]), which offers deep semantic insights into app behaviors and effectively identifies malicious patterns.

However, ML-based applications are widely recognized for their susceptibility to robustness issues [13], [14], [15], [16], [17], which can lead to severe consequences, particularly in

safety- and security-critical contexts like autonomous driving and malware detection. For instance, attackers can make subtle modifications to malware, preserving its malicious intent while enabling it to evade detection. To address this, robust testing is essential before deploying ML models in dynamic and potentially hostile environments [18]. To this end, adversarial attack methods [19], [20], [21], [22], [23] have been developed to rigorously evaluate model robustness. These evaluations help developers identify vulnerabilities, providing insights for improving robustness, such as retraining models with adversarial samples generated during testing [24], [25], [26].

There are two main kinds of attacks in graph-based AMD: *feature-level attacks* and *code-level attacks*. Feature-level attacks, which directly perturb the features of an APK (i.e., the model's input), can achieve high success rates [20], [8], [23]. However, these perturbations often do not realistically reflect APK modifications, thereby compromising the fidelity of robustness assessments. In contrast, code-level attacks alter the APK's smali code, indirectly changing its features used for detection. These attacks, conducted directly on the APK, are more realistic but inherently more complex due to the discontinuous nature of the perturbation space.

Recent studies have begun to explore code-level adversarial attacks [27], [22]. Essentially, these attacks involve modifying the smali code of an APK such that its FCG can be affected. HRAT [27], the pioneering work, introduced a set of FCG-level perturbation operators that can be translated into semantically consistent code-level perturbations. Furthermore, a deep Q-network (DQN) is used to guide the perturbation generation. Meanwhile, BagAmmo [22] employs a genetic algorithm (GA) that simulates targeted classifiers with a surrogate model and modifies the FCG by inserting non-executable code, optimizing the attack process. Despite these advancements, their effectiveness is still limited, particularly when facing relatively robust scenarios [11] (e.g., MalScan [8]).

The primary challenge lies in the *vast perturbation space* in the APK, where potential modifications to an FCG can be infinite, complicating the search for adversarial perturbations. To effectively navigate this, a variety of perturbation operators is necessary for enhanced exploration, alongside precise feedback mechanisms for better exploitation. However, current methods are often restricted to specific mutation types, such as only adding edges [22], or they perform multiple but simplistic perturbations [27], limiting the generation of diverse FCGs. Concerning feedback mechanisms, existing approaches like HRAT [27] predominantly rely on gradient information, which is costly and unobtainable in non-differentiable ML classifiers like Random Forest or K-nearest neighbors (KNN). These

TABLE I: The Scope of Existing Adversarial Attack Methods.

Tool	FCG-based	Deep Learning	Instance Algorithm		Ensemble Models	
	AMD	MLP	KNN-1	KNN-3	Random Forest	AdaBoost
HRAT	MalScan	○	●	○	○	○
	MaMaDroid	○	●	○	○	○
	APIGraph	○	●	○	○	○
BagAmmo	MalScan	○	○	○	○	○
	MaMaDroid	●	●	●	●	●
	APIGraph	●	●	●	●	●

Note: (●) for full consideration and (○) for no consideration.

challenges become more severe when AMDs employ diverse features and models. Additionally, we observe that current methods are mainly applied and evaluated only in limited scenarios (as depicted in Table I) and fail to be effective in scenarios with more robust features [8], [11] (e.g., MalScan) and popular models (e.g., Random Forest).

Motivated by these issues, this paper introduces FCGHUNTER, a testing method specifically designed to assess the robustness of FCG-based malware classifiers across various feature types and models. FCGHUNTER optimizes a sequence of perturbations to the original FCG, such that the malware can bypass detection after the modifications. Specifically, FCGHUNTER tackles the exploration and exploitation challenge in the vast perturbation space through several innovative strategies: 1) it narrows the search space by pinpointing critical areas of the FCG based on sensitive system APIs; 2) it incorporates diverse perturbation operators, including three novel types (e.g., *Adding Long Edges*) that significantly impact FCG features for better exploration; 3) it introduces a dependency-aware mutation representation and a conflict-resolving strategy, ensuring the feasibility of the sequence of perturbations; and 4) for optimal exploitation, FCGHUNTER employs a multi-objective optimization. Except for the model output feedback, a novel interpretation-based feedback, utilizing the SHAP method [28], is proposed to prioritize perturbations that significantly affect crucial features, thus improving the effectiveness of the whole search.

Technically, FCGHUNTER is implemented within a genetic algorithm framework. Each individual in the population is represented as a sequence of perturbations, where each gene is not just a single perturbation but a sub-sequence of dependent perturbations. These sub-sequences, containing highly interdependent perturbations, are considered together during crossover and mutation processes to ensure the validity of the generated FCG. Following this step, individuals are selected based on interpretation-assisted fitness scores that evaluate the effectiveness of perturbations in evading detection. If conflicts arise, FCGHUNTER resolves them by adjusting or removing the conflicting perturbations, ensuring that the best candidates are retained for further evolution.

To demonstrate the effectiveness of FCGHUNTER, we conducted comprehensive experiments on 40 distinct target models, incorporating eight types of graph embeddings and five different ML classifiers. To the best of our knowledge, we are the first to evaluate AMD systems across such a broad and diverse range, covering all the scenarios outlined in Table I. FCGHUNTER achieves an average attack success rate of 87.9% across these detection models, significantly outper-

forming state-of-the-art methods (i.e., HRAT and BagAmmo) by at least 44.7%. Our experiments also confirm the usefulness of the key components in FCGHUNTER. Based on the transferability of different models, we also applied FCGHUNTER to evaluate the robustness of black-box models (i.e., VirusTotal) in the real world, revealing the robustness issues of such models.

In summary, our main contributions are as follows:

- We expose the challenges presented by current approaches for attacking three widely-used ML model types: deep neural networks, k-nearest neighbors, and decision trees, each trained with distinct feature sets. Our analysis reveals that existing methods have limitations in certain scenarios, particularly regarding the models and feature types.
- We propose a novel robustness testing framework, incorporating dependency-aware mutation and multi-objective optimization, which can effectively evaluate different kinds of graph-based Android malware detectors. Our approach generates adversarial samples while preserving the malicious functionalities of the malware, leveraging diverse perturbation operators for enhanced exploration and precise feedback mechanisms for optimal exploitation.
- We conduct comprehensive experiments across 40 target models, spanning five distinct model and eight feature sets, which demonstrate the effectiveness of FCGHUNTER. We have made our dataset and code publicly available [29].

## II. GRAPH-BASED ANDROID MALWARE DETECTION

Graph-based detection leverages features extracted from the FCG of an APK’s smali code (i.e., the intermediate representation of an APK after compilation [12]), which captures the runtime behavior semantics of the application. The FCG is then transformed into a feature vector via graph embedding, which is subsequently used for binary classification to determine whether the application exhibits malicious behaviors. Next, we will introduce the main FCG-based methods, which include three graph embedding techniques and three widely used ML-based classifier types.

### A. Graph Embedding Methods

This step involves deriving a vector from an APK’s FCG, where nodes represent functions or abstract entities and edges depict call relationships, to capture crucial structural and behavioral patterns for classification. In the following, we will briefly introduce the three commonly used features [11], i.e., MalScan, MaMaDroid, and APIGraph.

**MalScan** [8] emphasizes the importance of 21,986 critical system API calls within function-level FCGs. It employs four centrality metrics: *Degree*, *Katz*, *Closeness*, and *Harmonic*, each offering unique insights into a node’s significance, and two combined centrality metrics: *Average* and *Concentrate*, to enhance the robustness of the extracted features.

**MaMaDroid** [9] employs a Markov chain, represented as a transition matrix, to model call probabilities between abstract states (e.g., family names) within function-level FCGs, effectively characterizing app behavior. Consequently, it provides two graph embedding modes: family-level with 11 states and package-level with 446 states.

**APIGraph** [10] utilizes a knowledge graph built upon the official Android API documentation to group APIs with similar functionalities or usage contexts through clustering. Therefore, it not only abstracts the representation of FCGs but also significantly reduces feature dimensions. For instance, it can reduce the dimensions in MaMaDroid’s package mode.

### B. ML-based Classifiers

After obtaining feature vectors via the graph embedding, ML-based methods are employed for the binary classification (i.e., malware or not). There are three commonly used types of classifiers: deep learning (DL), instance-based learning, and ensemble-based learning.

**Deep learning.** Multi-Layer Perceptron (MLP) is a basic DL model widely used in AMD and adversarial attacks [30], [31], [32], [22]. MLP learns nonlinear relationships between input features and output class labels, and it outputs a continuous score from 0 to 1 that indicates the probability of a sample being malicious, using a sigmoid activation function.

**Instance-based learning.** The KNN algorithm, a typical instance-based method commonly used in AMD [33], [20], [27], [32], [22], classifies data points by measuring distances (typically using Euclidean [34] metrics) to the nearest training samples. For each query, it selects the  $k$  closest samples (e.g.,  $k = 1$ ) and assigns a class based on the majority label among these neighbors.

**Ensemble-based learning.** Random Forest and AdaBoost [20], [9], [32], [22] effectively combine multiple learning algorithms to enhance both performance and robustness. Random Forest, an ensemble of decision trees, consolidates decisions through majority voting, thus mitigating the influence of any single, potentially biased model. AdaBoost sequentially applies a series of weak learners to progressively modified datasets, thereby incrementally improving the performance of initially weak classifiers.

## III. PROBLEM FORMULATION

Given a target AMD system represented by model  $M(\cdot)$ , which classifies input APKs as either benign or malware, we use  $G$  and  $E$  to represent the functions that extract FCG from the APK and calculate the embedding of the FCG, respectively. For a given malware  $m$ , the problem of AMD testing is to calculate the FCG perturbations  $\delta \in \Delta$  such that:

$$M(E(G(m))) \neq M(E(G(m) + \delta)) \wedge F(m) = F(m + reverse(\delta)) \quad (1)$$

where  $\Delta$  represents all possible perturbations on the vast space of FCG and  $F$  represents the functionalities of the APK. The formulation sets forth three critical requirements for calculating the perturbation: 1) the perturbation should be reversible, allowing it to be mirrored at the smali code level (i.e.,  $m + reverse(\delta)$ ); 2) the perturbation does not affect the functionality; and 3) the feature should be alerted sufficiently to change the final prediction outcome. Addressing this problem necessitates an effective optimization-based method to search and apply these perturbations effectively.

## IV. OVERVIEW OF FCGHUNTER

Figure 1 illustrates the main workflow of FCGHUNTER, which includes identifying critical areas of the FCG and optimizing perturbations within these areas using a GA.

**Step 1:** Initially, FCGHUNTER extracts the FCG from the malware’s smali files. Given the challenge of navigating the vast perturbation space within an FCG, we first identify the critical area relevant to malware behaviors for effectively reducing the vast perturbation space.

**Step 2:** FCGHUNTER employs a GA to optimize perturbations in the identified critical area. For better exploration in the perturbation space, we incorporate seven semantics-preserving mutation operators on FCGs. Note that these mutation operators can be translated to code-level mutation that does not affect the original functionality. The optimization aims to identify a *sequence* of operators that orderly perturbs the FCG. Each individual in the GA population represents a perturbation sequence, enabling the mutation of diverse FCGs.

- Step 2.1: However, directly applying crossover and mutation at the level of perturbation operators to generate offspring may lead to *invalid* perturbations. For example, an *Add Edge* operator may become infeasible if its prerequisite node has been removed by an earlier operator within the same sequence. To address this issue, we perform a dependency analysis and group dependent operations into sub-sequences, ensuring the validity of the perturbation sequence.
- Steps 2.2 and 2.3: Crossover and mutation processes are then performed at the level of sub-sequences to ensure the dependency. Additionally, we propose a conflict-resolving mechanism to address any conflicts within a sequence after crossover and mutation.
- Step 2.4: The new individuals are evaluated to calculate their fitness, selecting the best candidates for the next iteration or stopping if optimal conditions are met in the current iteration. To obtain more useful feedback, we design model-specific and explanation-based fitness functions: a multi-objective score for MLP classifiers, a surrogate model approach for instance-based classifiers, and a constraint-based solution for decision tree classifiers. If a perturbation sequence successfully bypasses the target model when applied to the FCG, it is recorded as a failure test (i.e., an adversarial sample). The sequence is finally applied to alter the APK’s smali code, resulting in a malware that can be misclassified as “benign”.

## V. STEP 1: CRITICAL AREA IDENTIFICATION

To mitigate the issue of search space explosion, we propose a specialized critical area identification method designed for FCG-based embeddings. This method efficiently pinpoints nodes and edges sensitive to perturbations that have notable impacts on detector outcomes, thereby reducing the search space during GA optimization.

An FCG is obtained through static analysis of the smali code from the decompiled APK. Nodes in the FCG are categorized as system nodes (SDK-defined functions, i.e., APIs) and user nodes (user-defined functions). Malware often invokes some sensitive system APIs to achieve malicious objectives (e.g., accessing user contacts). In other words, malicious calls typically occur from user nodes to system nodes. Therefore, FCG-based AMD typically prioritizes the user function calls that can invoke system APIs.

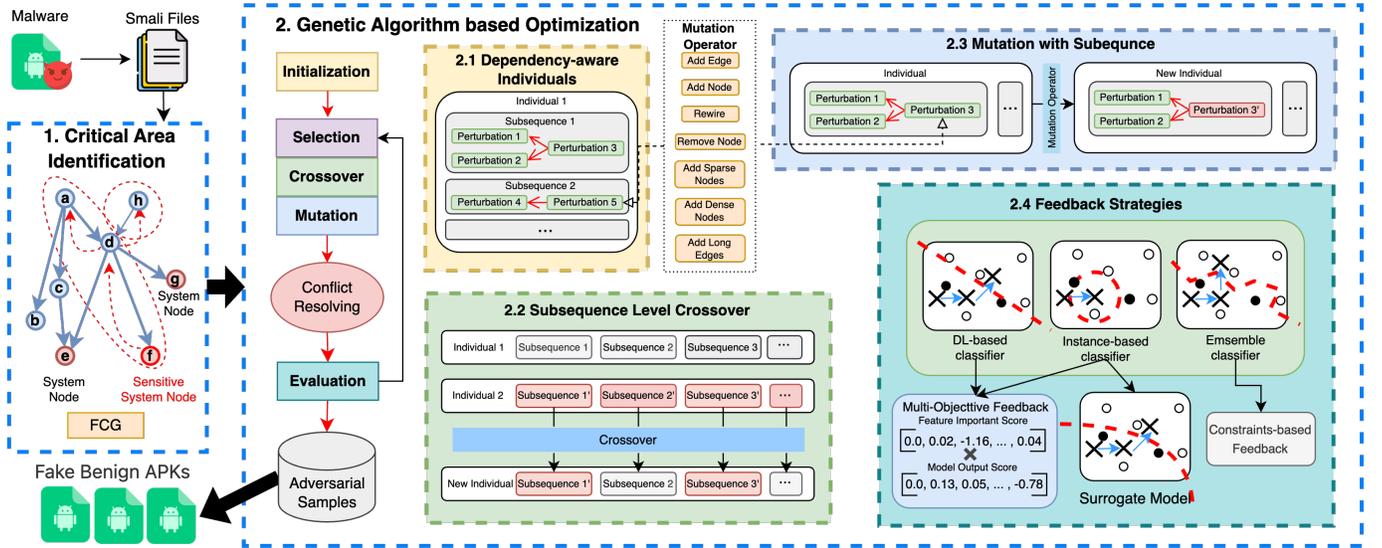


Fig. 1: Overview of FCGHUNTER.

To locate user function calls that can invoke critical system APIs (e.g., 21,986 sensitive APIs in MalScan and 11 family states in MaMaDroid), we first identify the nodes representing these critical system APIs in the graph, and then perform a backward traversal from these nodes to identify preceding nodes and edges, defining these connected regions as the *critical area*. Note that the perturbation can only be performed in the user functions. This area will be used for the subsequent GA-based optimization process.

## VI. STEP 2: PERTURBATION OPTIMIZATION

### A. Basic Perturbation Operators

Given an FCG  $G = (V, E)$ , the nodes  $V$  include both system nodes  $V_s$  and user nodes  $V_u$  and an edge  $(v_1, v_2) \in E$  shows the calling relationship between the two functions (with the caller  $v_1$  and the callee  $v_2$ ). In an FCG, user nodes can act as callers or callees, while system nodes can only serve as callees. To modify the FCG, we integrate seven semantic-preserving and code-level perturbation operators. The first four operators are based on prior work [27], and we briefly introduce these four operators:

- **Add Node:** This operator creates a new function  $i$  and selects a user node  $a \in V_u$  to invoke  $i$ . Consequently, a new node  $i$  and new edge  $(a, i)$  are added to  $G$ . The new function  $i$  is designed to perform non-functional operations (e.g., basic mathematical calculations) to ensure that it does not affect the overall functionality.
- **Add Edge:** This operator establishes a new calling between two existing functions,  $i \in V_u$  and  $f \in V$ , resulting in a new edge  $(i, f)$  within  $G$ . To maintain original functionality, strategies such as using *try-catch* blocks [22] and unreachable conditions [27] ensure that the callee  $f$  is never invoked, even though the calling is in the  $G$ .
- **Rewire:** This operator removes an existing edge  $(a, d)$  and selects a user node  $h \in V_u$  as an intermediary, adding two new edges:  $(a, h)$  and  $(h, d)$ , where  $(a, h)$  is  $a$ 's invocation to  $h$ , and  $(h, d)$  is  $h$ 's invocation to  $d$ . Special branches are

added to related functions to ensure the original invocations of  $a$  and  $d$  remain unaffected.

- **Remove Node:** This operator removes a user node  $d \in V_u$ . For maintain functionality, it identifies all original callers  $\{h | (h, d) \in E\}$  and replaces the invocation statements with  $d$ 's function body in the code. Correspondingly, the edges  $\{(h, d) \in E | h \in V_u\}$  are removed, and new edges  $\{(h, v) | (h, d) \in E \wedge (d, v) \in E\}$  are added to the  $G$ , where  $h$  are the original callers of  $d$  and  $v$  are its callees.

However, these operators are very basic and insufficient for modifying features, particularly those in robust models (e.g., MalScan, which is sparser than others), often leading the GA toward local optima. Therefore, we introduce three new perturbation operators designed to substantially affect features:

- **Add Sparse Nodes:** This operator adds  $k$  nodes  $v_1, v_2, \dots, v_k$  to the  $G$  at once. To affect the area around an existing node  $a \in V_u$ , edges  $(a, v_1), (a, v_2), \dots, (a, v_k)$  are added. This dilutes the centrality of other nodes and redistributes the influence across the  $G$ .
- **Add Dense Nodes:** This operator first performs the *Add Sparse Nodes* operator, then adds new edges  $\{(v_i, v_j) | i < j \wedge i, j \in [1, k]\}$  to the  $G$ . This effectively creates a dense subgraph, decreasing the relative importance of other paths in the  $G$ , which is particularly impactful for path-based analysis methods (e.g., Katz in MalScan).
- **Add Long Edges:** This operator adds  $m$  long edges between two existing nodes  $a \in V_u$  and  $f \in V_s$ . For each long edge,  $k$  new nodes  $v_1, v_2, \dots, v_k$  are added sequentially, creating the edges  $(a, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$ , and finally an edge  $(v_k, f)$  to create a path between  $a$  and  $f$ . This increases the number of paths leading to  $f$  in  $G$ , significantly boosting its centrality in the network.

These three operators are based on the combinations of four basic operators, thus still ensuring the functional integrity of the APK. Differently, they introduce a greater magnitude of perturbation for affecting the graph's features by allowing for the adjustment of parameters (e.g.,  $k$ ), which increases the

population’s diversity and helps avoid the risk of GA falling into local optima (see results in §VII-D).

**Translating FCG-Based Mutation to Code-Level Perturbation.** It is essential to convert FCG-based mutations into code-level modifications. These modifications should be repackaged into an APK that retains the same functionalities as the original. Specifically, the mutation in the FCG can be mapped to corresponding changes in the code as follows<sup>1</sup>:

- *Add Node*: We introduce a new function (i.e., node  $i$ ) in the code that does not affect the original functionality (e.g., only printing or simple calculation like  $int\ j = j + 1$ , and then returns  $j$ ). The existing function (i.e., the user node  $a$  in FCG) is modified to call this new function, but it does not process or utilize any of the returned value, thereby preserving the semantics of original function.
- *Add Edge*: We add an invocation from a user function  $a$  to any other function  $b$ . To guarantee the functionality of original function  $a$ , we can prevent the actual execution of function  $b$  by introducing a *condition* parameter in  $b$  and insert an *if-else* statement in its function body. When the function  $a$  invokes  $b$ , *condition* is set to *true*, causing  $b$  to return a value immediately, without executing the original logic in  $b$ . For invocations from  $b$ ’s original callers, *condition* is set to *false*, allowing  $b$  to execute its original logic, thereby preserving the original functions.
- *Rewire*: We replace an existing call from function  $a$  to function  $c$  with an intermediary function  $b$ , so that the call flow becomes  $a \rightarrow b \rightarrow c$ . To achieve this, we replace  $a$ ’s call to  $c$  with a call to  $b$  and add a call to  $c$  within  $b$ . To ensure  $b$ ’s original callers remain unaffected, we apply a strategy similar to *Add Edge*, i.e., using a *condition* parameter.
- *Remove Node*: We delete function  $a$ , which results in the removal of all calling relationships involving  $a$  in the original graph. To ensure that the program logic remains unaffected, we copy  $a$ ’s function body into all its caller functions as an inline code implementation. Consequently, in the final graph, direct connections are established between  $a$ ’s original callers and its callees.
- *Add Sparse Nodes*: We insert  $k$  functions simultaneously, all of which are called by a single existing function  $a$ . To maintain original program semantics, similar operations as in the *Add Node* process are applied.
- *Add Dense Nodes*: We start by performing the same operation as in *Add Sparse Nodes*. Then, for the newly added  $k$  functions, we sequentially connect them with calls. Throughout this process, we apply the same method as in the *Add Edge* operation to ensure that program semantics remain unchanged.
- *Add Long Edges*: Suppose we only insert a long edge, we insert  $k$  intermediate functions between an existing function  $a$  and function  $c$ , creating a nested call sequence. Essentially, this establishes a chain of function calls, where  $a$  calls the first intermediate function, which in turn calls the next, and so on, until reaching  $c$ . These intermediate  $k$  functions are newly added and serve solely as proxies, relaying the call

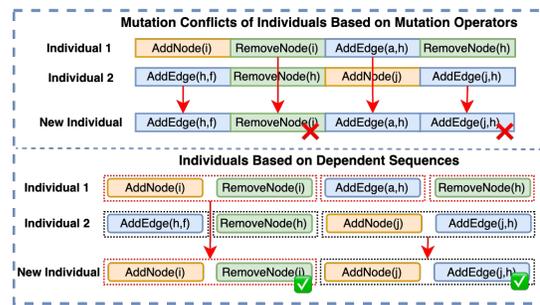


Fig. 2: Crossover with/without Dependency Strategy.

from  $a$  to  $c$  without affecting any other existing functions or altering the program’s original functionality.

Note that our approach mainly utilizes FCG-based mutations instead of arbitrary direct code mutations (e.g., transforming  $m = m * 2$  to  $m = m << 1$ ). This is because we focus on FCG-based AMDs that rely solely on the features of the FCG. Arbitrary code mutations may not always impact the FCG, and therefore, might not effectively influence robustness.

### B. Step 2.1: Individual Representation

To initialize the GA’s population, a simple way is to initialize each individual as a sequence of perturbation operators  $(o_1, o_2, \dots, o_n)$ , where  $o_i$  denotes one of the seven available perturbation operators. The sequence is then applied to the FCG  $G$  to generate a new FCG  $G'$ .

However, dependencies among operations often result in the generation of invalid perturbation sequences during crossover and mutation processes. For example, as shown in the top of Figure 2, two sequences of individuals undergo crossover, creating an infeasible sequence in the new individual. Without the *AddNode(i)* operator, the *RemoveNode(i)* operator becomes infeasible as the node  $i$  does not exist in the  $G$ . This issue can also arise during mutation, which greatly affects the testing efficiency.

To address this issue, we propose a dependency-aware representation to avoid operator conflicts during crossover and mutation processes. The approach involves a preliminary dependency analysis, grouping dependent operations into sub-sequences. Then crossover and mutation are performed at the level of sub-sequence.

To identify the dependent perturbations, we develop a greedy-based method (see the detailed algorithm on the website [35]) to check for dependencies between current operator  $o$  and the existing sub-sequence  $Seq$ . If  $o$  shares dependencies with any operators in  $Seq$ , it is added to that group; otherwise,  $o$  is placed into a new sub-sequence, indicating no dependencies with existing groups. Specifically, dependency checking involves a use-def analysis [36] where the target nodes  $V'$  and edges  $E'$  created by an operator  $o'$  (the definition) are examined against the usage in operation  $o$ . If  $o$  utilizes any nodes or edges defined by  $o'$ , a dependency exists. Taken the example in the bottom part of Figure 2, *AddNode(i)* defines node  $i$ , and *RemoveNode(i)* uses node  $i$ , establishing a dependency that necessitates grouping these operations together as an atomic operation to ensure safe crossover and mutation.

<sup>1</sup>More detailed illustration and code change examples can be found in [27].

Note that, during the GA initialization, the initial sequence  $(o_1, o_2, \dots, o_n)$  is guaranteed to be valid through the on-the-fly check. Specifically, starting with the initial graph  $G$ , a valid sub-sequence is randomly selected that can feasibly be applied to the current state of  $G$ . After applying this sub-sequence, the  $G$  is updated to  $G'$ . Subsequent operators are then chosen based on this updated  $G'$ , ensuring each selected operator remains feasible.

During GA iterations, the generation of individuals differs significantly from this initial process. Instead of being generated on-the-fly, the entire sequences in individuals are first constructed (with crossover and mutation) and then evaluated later. It introduces the problem in maintaining the feasibility of each operator within the sequence. Our dependency analysis is designed to mitigate this challenge in crossover and mutation.

### C. Step 2.2 and 2.3: Crossover and Mutation

Once dependency-aware individuals are established, crossover and mutation processes are conducted at the sub-sequence level. This approach is crucial for maintaining the integrity of dependent operators within each individual.

**Crossover.** Sub-sequences that contain dependent operators are randomly selected either to be retained or removed in their entirety from the new individual during the crossover process. This method helps prevent conflicts that could arise from breaking apart interdependent operations (see step 2.2 in Figure 1).

**Mutation.** As shown in step 2.3 of Figure 1, a sub-sequence is randomly chosen, and one of three types of mutations is applied: *adding*, *removing*, or *updating*. *Adding* involves inserting a new operator at a randomly selected position within a random sub-sequence, while *removing* deletes the operator at that position. *Updating* involves replacing an existing operator within the sub-sequence with another random operator.

Due to the possibility of disturbing the dependency of the sequence, we then perform an on-the-fly dependency check for operators. If a mutation renders subsequent operators infeasible, such as by altering dependent nodes or edges, we use a fix strategy. Problematic operators may be modified to fit the new context (e.g., changing an edge or node) or removed from the sequence. If the fix fails, the mutation is abandoned.

### D. Step 2.4: Evaluation and Selection

After crossover and mutation in the GA, we obtain new individuals as offspring. The fitness function is crucial for selecting superior individuals from the offspring.

The typical fitness function in adversarial attacks uses the model's output to decrease the prediction probability of the current class or increase that of the target class [37], [22]. However, relying solely on model output may not be effective in the context of AMD attacks, especially due to the non-differentiability of instance-based models and decision trees. Specifically, it can lead to premature convergence (i.e., all yield similar probability values), particularly if the model consistently exhibits high confidence in classifying certain samples as malicious.

1) *Fitness for MLP Model:* To overcome this challenge, we introduce an additional guidance mechanism based on feature interpretation, i.e., SHAP [28], a popular technique for understanding feature importance. Features with positive SHAP values positively contribute to the prediction, whereas negative SHAP values indicate a negative contribution.

When the GA encounters local optima without observable changes in model output, SHAP values allow us to monitor feature-level changes, offering a finer-grained criterion for selection. Specifically, *if an individual increases the value of features with negative contributions or decreases the value of features with positive contributions, the prediction is closer to failure*, even if the probability output remains unchanged.

We define a multi-objective fitness function as follows:

$$\begin{aligned} fitness1(I) &= M(E(G+I)) \\ fitness2(I) &= -\sum_{i=0}^{n-1} SHAP(M, G, I)_i \cdot (E(G)_i - E(G+I)_i) \end{aligned} \quad (2)$$

where  $I$  is a given individual (i.e., perturbations),  $G$  is the original FCG,  $E(G)$  is the embedding vector of the  $G$  with length  $n$ ,  $M(E(G+I))$  is the probability of the benign class and  $SHAP(M, G, I)$  represents the SHAP values of features.

The *fitness1* evaluates the model's target class probability. The *fitness2* measures the potential for classification changes, with  $SHAP(M, G, I)_i$  indicating the direction (positive or negative) and  $E(G)_i - E(G+I)_i$  quantifying the change in the  $i$ -th feature value due to perturbation  $I$ .

**Dominance and Selection.** We define a dominance relation for selection based on the two scores. An individual  $x$  is said to dominate another one  $y$  if and only if:

$$\begin{aligned} fitness1(x) &> fitness1(y) \vee \\ fitness1(x) &== fitness1(y) \wedge fitness2(x) > fitness2(y) \end{aligned} \quad (3)$$

We prioritize individuals with higher benign probability scores or, when scores are equal, those that modify feature values in the most beneficial direction.

2) *Fitness for Instance-based Model:* For instance-based learning models (e.g. KNN), the main challenge is the lack of gradient information. To approximate gradients for KNN, we employ a surrogate model (e.g., an MLP model), which facilitates the use of an interpretation-based approach (see §VI-D1) alongside the model output.

For a KNN model  $M$ , where the adversarial challenge is to manipulate the instance such that it resembles benign samples more closely than malware samples, we train a surrogate model  $M'$ . This allows us to derive a dual-score fitness function, as follows:

$$\begin{aligned} fitness1(I) &= \frac{1}{x} \sum_{i=1}^k (M(I)_i^m - M(I)_i^b) \\ fitness2(I) &= -\sum_{i=0}^{n-1} SHAP(M', G, I)_i \cdot (E(G)_i - E(G+I)_i) \end{aligned} \quad (4)$$

where  $k$  represents the number of neighbors considered in KNN.  $M(I)_i^m$  denotes the distance to the  $i$ -th nearest malware sample, and  $M(I)_i^b$  denotes the distance to the  $i$ -th nearest benign sample.

The *fitness1* aims to increase the similarity to benign neighbors and decrease the similarity to malware neighbors,

effectively manipulating the prediction of the adversarial example. The second fitness function  $fitness2$ , similar to that used for target MLP models (§ VI-D1), utilizes SHAP values estimated by the surrogate model  $M'$  to assess the impact of perturbations on feature importance. The selection follows the dominance relation defined in Equation 3.

3) *Fitness for Ensemble Model*: Ensemble models like Random Forest determine output probabilities through a voting process among numerous decision trees, each selecting a subset of features for decision nodes (tree split nodes) [38]. Altering tree-based model outputs during testing is challenging, especially when only a few or none of the selected decision features are present in the target sample. Consequently, changes in the overall model output (i.e., the decision of the majority of trees) are unlikely if key decision features remain unaffected.

To overcome this challenge, we directly examine the constraints associated with the decision features. By analyzing the decision paths of all decision trees, we identify all possible feature constraints that could result in a benign output, as our goal is to have the target model misclassify the malware as benign. We will eliminate the constraints that conflict with those from other decision trees. Finally, our objective is to maximize the number of constraints that the perturbed inputs can satisfy, thereby increasing the likelihood of a benign classification. The fitness function is defined as follows:

$$fitness(I) = \sum_{c \in C} SAT(G, M, I, c) \quad (5)$$

where  $C$  represents all the constraints that can potentially lead to a benign output, and  $SAT$  determines whether a given constraint  $c \in C$  is satisfied (1) or not (0). The optimization process aims to generate perturbations that maximize the number of satisfied constraints.

## VII. EVALUATION

We aim to evaluate the effectiveness of FCGHUNTER by answering the following research questions.

- **RQ1**: How effective is FCGHUNTER compared to others?
- **RQ2**: What is the performance of FCGHUNTER?
- **RQ3**: How does each component of FCGHUNTER impact the overall effectiveness?

### A. Experimental Setup

**Dataset.** Since the datasets used in previous studies are not publicly available, we adhere to the very common methodologies described in prior works [27], [22] to collect datasets. The collected dataset includes 12,000 samples with 6,000 benign and 6,000 malware samples, divided into an 80:20 ratio for training and testing the models. To ensure representativeness, these collected samples are evenly distributed across six years, from 2018 to 2023, with 1,000 benign and 1,000 malware samples per year. Similar to previous works [27], [22], benign samples are sourced from AndroZoo [39] (VirusTotal [40] score of 0) and malware samples from VirusShare [41] (VirusTotal score above 4). To assess robustness, we additionally collected 120 true malware samples from the same six-year

period (20 samples per year) as test seeds. These seed samples are distinct from the initial set of 6,000 malware samples. More details about dataset are available on our website [35].

**Target Models.** We invested significant efforts to include a wide range of models and features, ensuring a systematic and comprehensive evaluation of the testing methods. Specifically, we trained 40 (8×5) ML-based AMD models built upon 8 types of features, including the Degree, Katz, Harmonic, Closeness, Average, and Concentrate features from MalScan [8], the family level from MaMaDroid [9], and the package level from APIGraph, alongside 5 ML-based classifiers: MLP [42], KNN-1 [43], KNN-3, Random Forest (RF) [44], and AdaBoost (AB) [45]. The performance of target models can be found on our website [35].

**Baselines.** We selected three baselines for comparison: a random testing approach and two state-of-the-art adversarial attack methods [27], [22] for AMD. Due to the limited applicability of these state-of-the-art baselines or the unavailability of the code, we extended or re-implemented them based on the descriptions provided in their respective papers.

Specifically, for BagAmmo [22], which has not released its code, we replicated its algorithm based on the descriptions provided in their paper. To ensure a fair comparison with FCGHUNTER, we configured BagAmmo in a white-box setting, where feedback is obtained directly from the target model alongside a surrogate model. In the configuration referred to as *BagAmmo-G*, we used a GCN surrogate model (the original model in this baseline) trained on ground truth data. Additionally, we experimented with using an MLP (the same surrogate model as in our method) in place of their original GCN, designated as *BagAmmo-M*. For HRAT [27], although the code is available [46], it primarily addresses attacks utilizing Degree and Katz centrality metrics for MalScan on the KNN-1 model. We expanded its application to encompass a wider array of target scenarios. However, HRAT is limited by GPU memory constraints and the need for classifier differentiability [47], [48], which restricts its use with tree-based models and memory-intensive features such as Average and Concentrate. Regarding the Random attack, it randomly generates perturbation operators and evaluates their effects when applied to the FCG. Further details about the baselines, including the source code, are available on our website [35] and GitHub [29].

**Metrics.** We employed three widely used metrics: Attack Success Rates (ASR), Perturbation Rates (PR) and Average Number of Survival Genes per Generation (ASGG). ASR measures the effectiveness of attack methods. PR quantifies the relative increase in graph components (i.e., nodes and edges) of adversarial samples compared with the original malware. Considering the potential conflicts that can result in certain infeasible perturbations (i.e., genes in the individuals), ASGG is designed to assess the count of genes that remain feasible (referred to as the ‘survival genes’) following crossover and mutation in each generation.

$$ASR = \frac{N_a}{N_m}, PR = \frac{1}{N_a} \sum_{i=1}^{N_a} \delta_i, ASGG = \frac{1}{G} \sum_{g=1}^G N_g \quad (6)$$

where  $N_a$  is the number of malware that can successfully bypass the AMDs,  $N_m$  is the total number of seed malware,

TABLE II: Attack Success Rates of FCGHUNTER and Baselines.

	MalScan (Degree)					MalScan (Katz)					MalScan (Harmonic)					MalScan (Closeness)				
	MLP	KNN-1	KNN-3	RF	AB	MLP	KNN-1	KNN-3	RF	AB	MLP	KNN-1	KNN-3	RF	AB	MLP	KNN-1	KNN-3	RF	AB
<b>Ours</b>	<b>0.82</b>	<b>0.73</b>	<b>0.76</b>	<b>0.78</b>	<b>1.00</b>	<b>0.77</b>	<b>0.68</b>	<b>0.69</b>	<b>0.94</b>	<b>0.96</b>	<b>0.82</b>	<b>0.90</b>	<b>0.83</b>	<b>1.00</b>	<b>1.00</b>	<b>0.88</b>	<b>0.97</b>	<b>0.84</b>	<b>0.91</b>	<b>1.00</b>
HRAT	0.14	0.18	0.08	-	-	0.01	0.03	0.01	-	-	0.01	0.03	0.01	-	-	0.01	0.12	0.18	-	-
BagAmmo-M	0.58	0.57	0.50	0.02	0.07	0.24	0.23	0.16	0.01	0.03	0.70	0.66	0.61	0.37	0.09	0.77	0.67	0.59	0.13	0.13
BagAmmo-G	0.65	0.61	0.28	0.03	0.06	0.43	0.22	0.05	0.00	0.03	0.73	0.67	0.58	0.33	0.08	0.74	0.64	0.56	0.05	0.08
Random	0.59	0.62	0.56	0.19	0.03	0.02	0.08	0.08	0.11	0.04	0.59	0.67	0.57	0.08	0.03	0.63	0.58	0.58	0.13	0.04

	MalScan (Average)					MalScan (Concentrate)					Mamadroid					APIGraph				
	MLP	KNN-1	KNN-3	RF	AB	MLP	KNN-1	KNN-3	RF	AB	MLP	KNN-1	KNN-3	RF	AB	MLP	KNN-1	KNN-3	RF	AB
<b>Ours</b>	<b>0.90</b>	<b>0.92</b>	<b>0.83</b>	<b>0.91</b>	<b>1.00</b>	<b>0.87</b>	<b>0.91</b>	<b>0.78</b>	<b>0.94</b>	<b>0.96</b>	<b>0.96</b>	<b>0.99</b>	<b>0.93</b>	<b>1.00</b>	<b>0.98</b>	<b>0.83</b>	<b>0.84</b>	<b>0.84</b>	<b>0.78</b>	<b>0.72</b>
HRAT	-	-	-	-	-	-	-	-	-	-	0.03	0.08	0.03	-	-	0.05	0.09	0.04	-	-
BagAmmo-M	0.75	0.63	0.58	0.24	0.18	0.72	0.66	0.57	0.17	0.19	0.58	0.76	0.71	0.40	0.13	0.81	0.79	0.78	0.07	0.43
BagAmmo-G	0.72	0.62	0.57	0.05	0.11	0.73	0.60	0.55	0.23	0.06	0.80	0.65	0.53	0.33	0.33	0.79	0.74	0.70	0.16	0.40
Random	0.61	0.62	0.58	0.08	0.03	0.66	0.60	0.58	0.20	0.03	0.09	0.12	0.03	0.13	0.14	0.47	0.14	0.03	0.05	0.09

$\delta_i = \frac{P_{add,i}}{P_{ori,i}}$  is the perturbation ratio for the  $i$ -th successful sample, reflecting the proportion of added nodes and edges,  $G$  is the total number of generations and  $N_g$  is the total number of surviving genes in the  $g$ -th generation.

**B. RQ1: Effectiveness**

**Setup.** We initialize each population with 100 individuals for 40 generations, with each individual initializing with 300 perturbation operations. This configuration follows established precedents in the literature. According to BagAmmo, the best attack success rate is achieved at 40 generations. Meanwhile, HRAT identifies 300 as the optimal number of perturbations. To enhance HRAT’s performance and ensure fair comparisons, we increased the number of random initializations in HRAT from 16 to 100. The configuration for the Random attack remains consistent with the baselines previously described, involving 300 perturbations per iteration and a maximum number of 100 iterations.

**Results & Analysis.** As presented in Table II, the results demonstrate that our attack method outperforms the baselines significantly, the results demonstrate that our attack method significantly outperforms the baselines, achieving an average ASR of 87.9%, which is at least 44.7% higher than BagAmmo-M (43.2%), BagAmmo-G (41.2%), Random Attack (28.8%), and HRAT (7.8%).

(1) **Baseline Analysis:** We found that HRAT generally performs poorly across most models, often yielding an ASR close to zero, especially in MalScan (Katz, Harmonic), MaMaDroid, and APIGraph, where it is ineffective. Furthermore, the results of MalScan (Degree and Katz) with KNN-1 show a significant discrepancy compared to the claims in their paper, with similar doubts raised in this survey [49] and its results.<sup>2</sup> Our analysis indicates that a primary limitation of HRAT lies in its RL reward mechanism, which relies on coarse model scores and suffers from inaccuracies in gradient estimation on the adjacency matrix. This lack of precise, granular feedback hinders the model’s ability to fine-tune perturbations across complex feature types and target models. BagAmmo-M and BagAmmo-G perform well with MaMaDroid and APIGraph feature types across MLP, KNN-1, and KNN-3 models, especially with MLP. However, their effectiveness drops significantly on tree-based models and MalScan, where they perform comparably to Random Attack. This reduction stems primarily from reliance on a single mutation operator and feedback based solely on coarse-grained

<sup>2</sup>https://github.com/reproducibility-sec/reproducibility/blob/main/sheet1.csv

TABLE III: Perturbation Rates of Successful Attack Samples.

	MLP	KNN-1	KNN-3	RF	AB
<b>MalScan (Degree)</b>	0.04	2.23	0.04	5.17	2.29
<b>MalScan (Katz)</b>	0.17	2.88	0.19	4.68	>10
<b>MalScan (Harmonic)</b>	<0.01	0.81	<0.01	<b>0.02</b>	<b>0.01</b>
<b>MalScan (Closeness)</b>	<0.01	0.52	<0.01	0.68	1.58
<b>MalScan (Average)</b>	<0.01	<0.01	<0.01	5.27	<b>8.07</b>
<b>MalScan (Concentrate)</b>	<0.01	1.70	<0.01	>10	>10
<b>Mamadroid</b>	0.45	0.45	0.14	>10	4.85
<b>APIGraph</b>	1.31	0.26	0.14	>10	3.07

scores from surrogate and target models. Although it claims GCN’s strengths with graph-structured data [22], adapting to tree-based models in AMD is challenging, as these models typically lack the relational structure that GCNs leverage. Please note that we observed a discrepancy between our evaluation results and those reported in the original paper. We have made extensive efforts to investigate this issue and confirmed our results, as discussed in Section IX and the explanations on our website [35].

**Finding #1:** Existing methods are notably ineffective, particularly on MalScan and ensemble models, with results that are close to random testing.

(2) **Robustness Analysis:** There are noticeable variations in ASR across different feature types. From the perspective of features, MalScan (Closeness and Average) and MaMaDroid consistently exhibit high ASRs, typically exceeding 90% across most classifiers, suggesting these features may be more susceptible to attacks. Conversely, features like MalScan (Degree and Katz) and APIGraph demonstrate greater robustness, with ASRs often below 80%, likely due to the complexity of perturbing these features effectively; While ensemble models (i.e., RF and AB) generally show more robustness than single models (i.e., MLP and KNNs), our method still achieves high ASRs on MalScan (Harmonic) and MalScan (Closeness) features, proving its effectiveness even against complex models. However, in ensemble models, our method achieves relatively lower ASRs, notably less than 80% on APIGraph, reflecting their resilience against attacks.

**Finding #2:** MalScan (Closeness and Average) and MaMaDroid are less robust, while MalScan (Degree and Katz) and APIGraph are more robust; Ensemble models (i.e., RF and AB) generally show greater robustness.

**Answer to RQ1:** FCGHUNTER, with an average ASR of 87.9%, outperforms baselines by at least 44.7% in white-box attacks, achieving higher ASR across diverse models. The results highlight the persistent vulnerability of current FCG-based ML models to adversarial attacks, emphasizing the need for robustness testing.

### C. RQ2: Performance

#### 1) Perturbation Rates across Target Models:

**Setup.** To measure the perturbation degree that FCGHUNTER applies to original samples, we calculated the average PR across all adversarial samples for each target model.

**Results & Analysis.** As shown in Table III, the successful adversarial samples on MLP and KNNs-based models exhibit relatively low PR, while higher PR on ensemble models. Specifically, MalScan (Harmonic, Closeness, Average, and Concentrate) models achieved PRs below 0.01 with both MLP and KNN-3. However, PRs for KNNs are slightly higher compared to MLP, suggesting MLP’s less robustness. PRs for tree-based models (i.e., RF and AB) are notably higher, except for MalScan (Harmonic), where they are lower. Instances of PRs exceeding 10 are observed, typically due to exceptionally large samples. For instance, in MalScan (Katz) under AB, 50% of the samples are below 3, and 73% are below 10.

**Finding #3:** Single models (i.e., MLP and KNNs) are generally easier to bypass, requiring fewer perturbations. In contrast, ensemble models (i.e., RF and AB) combine predictions from multiple models, making them more robust and necessitating greater perturbations to compromise.

#### 2) Survival Genes during GA Iterations:

**Setup.** To assess the usefulness of the dependency-aware strategy (see § VI-B) in reducing mutation conflicts, we calculated the ASGG after 40 iterations. Specifically, we randomly selected 20 malware and conducted experiments with and without the dependency analysis for comparative analysis.

**Results & Analysis.** The green and orange lines in Figure 3 represent FCGHUNTER’s ASGG with and without the dependency-aware strategy, respectively. Throughout the iterations, the green line consistently maintains a higher ASGG compared to the orange line, with the difference doubling after the fifth generation. The orange line shows a noticeable bump between generations 30 and 35. Our analysis indicates that this increase can be attributed to the probabilistic introduction of a significant number of new genes by the mutation, leading to pronounced fluctuations. Following this bump, the ASGG quickly declines due to the absence of the dependency-aware strategy capable of preemptively resolving gene conflicts. In contrast, the green line remains more stable throughout the generations. This stability suggests that the strategy helps preserve the number of viable genes within the population, which could prevent premature convergence of the GA optimization. Stability in the gene pool is crucial because significant diminishment in genetic variety can impede the GA’s ability to generate new and potentially more effective individuals [50].

**Finding #4:** The dependency-aware strategy protects critical genes from mutation conflicts, thereby preserving genetic diversity and preventing premature convergence by maintaining a sufficient number of viable perturbations.

#### 3) Runtime Performance Compared with Others:

**Setup.** To assess the performance of the testing, we monitored the number of adversarial samples generated within 500 minutes on MalScan (Degree) and KNN-1 models. These models

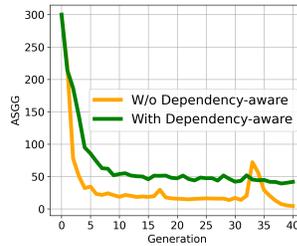


Fig. 3: Impact of Dependency-aware Strategy.

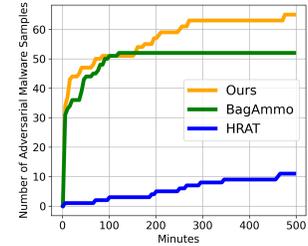


Fig. 4: Runtime Efficiency Comparison with Other State-of-the-Art Methods.

and features were chosen because the baselines (e.g., HRAT) achieved the highest ASR with them.

**Results & Analysis.** Figure 4 demonstrates that FCGHUNTER (orange line) detects more successful attacks than both BagAmmo (green line) (i.e., BagAmmo-G) and HRAT (blue line). HRAT’s runtime efficiency is notably low because each perturbation requires the computation of gradient information from the FCG and target model. Although FCGHUNTER and BagAmmo efficiently completed most attacks within a short period, exhibiting exceptional runtime efficiency, BagAmmo’s progress stalls after approximately 110 minutes, indicating premature convergence, likely due to the limited variety in its single operators (i.e., only involve adding edges) and the lack of directional feedback from target models. In contrast, the persistent growth of FCGHUNTER highlights its ability to continuously explore the huge search space and identify viable solutions.

For further analysis, we also calculated the average time cost per iteration of mutant generation, with the following results: 0.34s for BagAmmo, 18.72s for HRAT, and 0.67s for FCGHUNTER. HRAT takes significantly longer due to the heavy gradient calculation. Compared to single-objective BagAmmo, the selection process in our method is slightly slower. However, our dependency-aware mutation and multi-objective optimization are well worth it, as they ultimately lead to a significant reduction in the overall time cost required to detect adversarial examples, as shown in Figure 4.

**Answer to RQ2:** FCGHUNTER is efficient in generating failure cases (i.e., more failures within the same time), adding fewer perturbations (i.e., lower PR), and maintaining a higher number of valid perturbations (i.e., higher ASGG).

### D. RQ3: Ablation Studies

**Setup.** To assess the effectiveness of key components in FCGHUNTER, we quantified the reduction in ASR across 40 target models upon the removal of the specific component. Specifically, key components include critical area identification (§ V) as *Cri*, dependency-aware strategy (§ VI-B) as *Dep*, and fitness function (§ VI-D). Additionally, three new perturbation operators (§ VI-A), including Add Sparse Nodes (*ASN*), Add Dense Nodes (*ADN*), and Add Long Edges (*ALE*), are used mainly for ensemble models (i.e., RF and AB). The fitness function for MLP models incorporates interpretation-based scores (*Int*), whereas, for KNN

TABLE IV: Ablation Studies on the Key Components of FCGHUNTER.

	MLP			KNN-1				KNN-3				Random Forest			AdaBoost		
	-Cri	-Dep	-Int	-Cri	-Dep	-Int	-Sur	-Cri	-Dep	-Int	-Sur	-ASN	-ADN	-ALE	-ASN	-ADN	-ALE
MalScan (Degree)	-0.11	-0.08	-0.12	-0.12	-0.03	-0.05	-0.03	-0.15	-0.09	-0.14	-0.16	-0.01	0.00	-0.19	<b>-0.19</b>	0.02	-0.56
MalScan (Katz)	<b>-0.29</b>	<b>-0.60</b>	<b>-0.56</b>	<b>-0.23</b>	-0.17	-0.11	-0.11	<b>-0.23</b>	<b>-0.50</b>	-0.12	-0.19	-0.05	-0.13	-0.38	-0.01	-0.28	-0.48
MalScan (Harmonic)	-0.10	-0.08	-0.07	-0.18	-0.17	-0.18	-0.13	-0.18	-0.13	-0.15	<b>-0.25</b>	-0.01	0.00	-0.41	0.00	-0.01	-0.60
MalScan (Closeness)	-0.16	-0.21	-0.08	-0.18	-0.18	-0.16	-0.21	-0.17	-0.13	-0.13	-0.15	-0.01	-0.02	-0.35	-0.06	-0.03	-0.25
MalScan (Average)	-0.20	-0.15	-0.12	-0.18	-0.18	-0.18	<b>-0.28</b>	-0.20	-0.18	<b>-0.18</b>	-0.21	-0.03	-0.02	-0.23	0.00	-0.13	-0.18
MalScan (Concentrate)	-0.18	-0.15	-0.11	<b>-0.23</b>	<b>-0.27</b>	<b>-0.23</b>	<b>-0.28</b>	-0.14	-0.11	-0.09	-0.16	<b>-0.06</b>	<b>-0.14</b>	-0.61	-0.05	<b>-0.50</b>	-0.58
MaMaDroid	-0.18	-0.20	-0.16	-0.18	-0.22	-0.19	-0.20	-0.14	-0.22	-0.18	-0.16	0.00	0.00	<b>-0.75</b>	0.00	0.00	<b>-0.78</b>
APIGraph	-0.04	-0.13	-0.04	-0.05	-0.05	-0.03	-0.03	-0.01	-0.18	-0.03	-0.04	0.00	0.00	-0.68	0.00	0.00	-0.72

models, it includes additional scores derived from a surrogate model ( $Sur$ ). To summarize, the evaluated components for MLP and KNNs include  $Cri$ ,  $Dep$ , and  $Int$ , with  $Sur$  additionally assessed for KNNs. For ensemble models, we focus on the impact of  $ASN$ ,  $ADN$ , and  $ALE$ . The experimental parameters are consistent with those in RQ1 (§ VII-B).

**Results & Analysis.** Table IV displays the ASR discrepancies resulting from the removal of individual components, compared to the complete configuration in Table II.

First, the removal of certain components results in notable reductions in ASR, emphasizing their critical role in the effectiveness of FCGHUNTER. For instance, removing  $Cri$  and  $Dep$  significantly affects the ASR in MalScan (Katz) models more than in other models. The removal of  $Cri$  always results in the largest drops, and under the KNN-1 and KNN-3 configurations, the removal of  $Dep$  leads to decreases as high as 0.60 and 0.50, respectively. Due to its inherent robustness, MalScan (Katz) requires substantial perturbations (Table III) and effective dependency management to maintain a viable number of genes in the population (Figure. 3), which prevents premature convergence of the GA, as noted in § VII-C2. Moreover, the  $Sur$  is critical for KNN models, where its removal leads to significant ASR reductions to other components. This suggests that KNN classifiers may rely more heavily on specific interpretations or feature relations that the surrogate model helps to exploit.

Second, certain features and models demonstrate minimal impact from the removal of components. For instance, API-Graph shows notably smaller ASR declines, all below 0.04, when  $Cri$ ,  $Int$ , and  $Sur$  are removed. This suggests that APIGraph has inherent robustness, as noted in § VII-B. Its robustness can be attributed to high-level feature abstraction, making it less sensitive to perturbations affecting fewer nodes or less critical connections within the FCG’s structure.

Third, the impact of new perturbation operators varies significantly across ensemble models. The removal of  $ALE$  demonstrates substantial ASR drops, especially in the MaMaDroid and APIGraph models. This is primarily because these models are based on features constructed from the call relationships between functions, making them highly sensitive to substantial changes in edges, particularly those directed toward system functions. Although the node-adding based operators,  $ASN$  and  $ADN$ , aim to reduce the centrality of malicious nodes in MalScan graphs,  $ALE$  more significantly disrupts node centrality by altering graph structures, i.e., adding edges that modify path lengths and node connectivity (§VI-A). This change impacts the graph’s overall centrality more drastically than simply adjusting nodes.

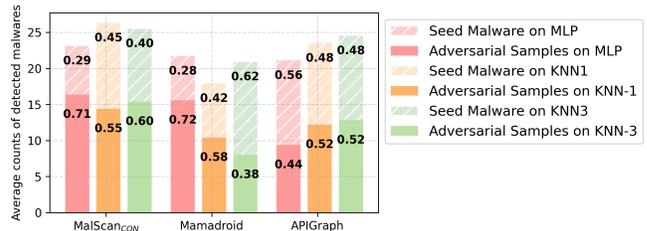


Fig. 5: Detection Score Difference Between Malware and Corresponding Adversarial Samples.

**Finding #5:** Edge-based perturbations tend to be more effective than node-based ones, impacting the most robust models by altering the graph’s features significantly.

**Answer to RQ3:** Each component in FCGHUNTER plays a distinct role in enhancing the ASR for different models. Specifically,  $Cri$  and  $Dep$  significantly boost ASR in MalScan (Katz), while  $Sur$  is crucial for KNN models. In ensemble models,  $ALE$  proves to be highly influential.

## VIII. DISCUSSIONS

**(1) Evaluation on Real-world AMD.** To explore the robustness issues present in real-world models, we selected VirusTotal [40], a leading platform for malware analysis, as our target [22], [23], [32]. We randomly selected 10 adversarial samples from each scenario, with original detection scores ranging from 4 to 45, and compared their detection scores before and after our attack. Figure 5 shows the change in detection scores for the original malware and their adversarial counterparts as analyzed by VirusTotal. Notably, the detection scores (which indicate the level of maliciousness) dropped significantly (by over 28%), suggesting potential weak robustness in the real-world AMD systems of several vendors. This drop might be due to these systems relying on detection algorithms that have robustness issues. Further analysis shows that the impact of adversarial samples varies across different classifiers. KNN-3 experiences the greatest impact (average 50%) from a model perspective, while APIGraph has the greatest impact (average 51%) from a feature perspective. This suggests a preference for relatively robust features and models in real-world scenarios, but they still cannot withstand strong adversarial attacks (e.g., FCGHUNTER). To further understand why FCGHUNTER performs effectively in real-world black-box systems (i.e., VirusTotal), we analyzed the transferability across different features and models. For more detailed results, please refer to our website [35].

**(2) Robustness Enhancement via Retraining.** The experimental results reinforce the need for continued enhancements

in AMD robustness. From the feature perspective, integrating robust features such as MalScan (Katz) and APIGraph to comprehensively represent malware behaviors proves to be a promising approach. This method, akin to merging static and dynamic features, leverages the distinct advantages of each to provide a complete depiction of potential threats [51]. From the model perspective, retraining has been a popular strategy to boost ML model robustness [24], [25], [52]. In supplementary experiments, we retrained models with adversarial examples generated by FCGHUNTER, which could significantly decrease the ASR (see detailed results on the website [35]). Additionally, our studies suggest that ensemble models serve as more robust classifiers in FCG-based AMD environments.

**(3) High-efficiency Strategies for Ensemble Models.** The perturbation rates for ensemble models were observably high (§VII-B), partially due to a limited number of modifiable nodes (i.e., user function calls) in certain malware. This led us to propose the new node-based mutation operators (i.e., ASN and ADN). However, we found that their effectiveness was still limited to specific scenarios. These results indicate that the mutation operators could still be improved, especially by developing edge-based perturbation variants to potentially increase effectiveness on ensemble models, as demonstrated by the success of edge-based ALE in §VII-D. Moreover, we can also attempt to develop some interpretation-based feedback at the operator level, which is capable of identifying more influential and less frequently altered operations.

## IX. THREATS TO VALIDITY

The selection of the dataset, including the training dataset and the seed malware, poses a threat to validity. We address this threat by adhering to established data selection protocols and collecting a diverse range of APKs from 2018 to 2023. Similarly, to ensure the validity and representativeness of the seed malware, we randomly selected these seeds from the past six years and varied their sizes to maintain diversity. To the best of our knowledge, our dataset spans a recent six-year range, providing broader coverage compared to the baselines.

The selection of models presents another potential threat to validity, as the results may vary in different AMD models. To mitigate it, we have endeavored to include a broad range of categories, incorporating features with various granularity from the FCG and multiple machine learning models with distinct decision mechanisms. To the best of our knowledge, our evaluations are the most comprehensive concerning various FCG-based features and models.

The replication and extension of baselines pose another threat to validity. Notably, significant discrepancies persist between our results and those reported in the original papers. Actually, the reproducibility issues have also been acknowledged by existing works [49]. We addressed this threat carefully by: 1) meticulously reviewing the code and consulting with their authors to clarify ambiguous parts; 2) engaging in discussions with the authors about the discrepancies, attributing potential causes to differences in datasets, the APK extraction tools used (e.g., Androguard [53] versus FlowDroid [54]), and the models evaluated; 3) releasing our code, models, and seed malware to facilitate verification and replication of our findings [35].

Finally, employing the interpretation-based method (i.e., SHAP) for interpreting model decisions could pose a threat to validity. SHAP values may not always accurately reflect the influence of different inputs in models. Additionally, alternative interpretation methods could be considered. To mitigate this, we did not rely solely on interpretation scores; instead, the model’s output served as the primary and dominant feedback. Our extensive evaluation also demonstrates the overall usefulness of this approach. In future work, we plan to explore the impact of various interpretation methods on FCGHUNTER.

## X. RELATED WORK

**ML-based Android Malware Detection.** ML techniques have gained significant traction in the domain of Android malware detection, leveraging diverse feature extraction and embedding methodologies, such as string-based [4], [5], image-based [6], [7], graph-based [8], [9], [10]. For instance, Drebin [4] utilizes static strings such as permissions and API calls extracted from APKs, employing Support Vector Machines (SVM) for classification. Addressing string obfuscation, RevealDroid [5] resorts to byte-code extraction for consistent classification with Drebin. However, string/image-based approaches often lack semantic information, prompting a shift towards more sophisticated techniques like Function Call Graph (FCG) representations, exemplified by MalScan [8]. MalScan represents FCG from the smali code as a social network and employs k-nearest neighbors (KNN) as the classifier, offering improved robustness and efficacy.

**Adversarial Attacks for Robustness Evaluation.** Related works [55], [20], [56], [27], [32], [22] have primarily focused on various techniques for generating adversarial examples to evaluate the robustness of malware detectors. Abundant adversarial attacks on string-based detectors are relatively simple and straightforward features using one-hot encoding, like Drebin [4]. By using gradient-based methods [55], [20], [57] or interpretability-assisted techniques [58], [23], features can be directly modified and mapped back to the code, leading to high attack success rates. However, adversarial attacks on graph-based detectors face the problem-feature reverse challenge. This work [20] focused on the feature level but struggled to maintain the functional integrity of the APK. Two recent studies have shifted towards exploring code-level attacks, employing heuristic search algorithms [27], [22]. Zhao et al. [27] exploited gradient information to estimate perturbation locations and directions. However, discrete gradient estimation errors on binary graphs may cause reinforcement learning to proceed in the wrong direction. Li et al. [22] utilized single perturbations and scores from a surrogate model to guide the attack process. However, it easily falls into local optima due to the vast perturbation space created by sparse features like MalScan, resulting from a lack of precise feedback and diversified operators. Other researchers are exploring more efficient adversarial attack techniques to improve robustness evaluations of AMD methods, such as using interpretation-assisted feedback [59], [60], [23], [61]. For example, Amich et al. [59] leverages SHAP to guide adversarial example crafting against ML models, seeking meaningful perturbations to aid in

assessing the system's robustness. Sun et al.[23] utilize SHAP to guide attacks on string-based detectors, identifying critical API permissions and inserting uncalled functions. Similarly, Yu et al.[61] propose step-level interpretability feedback for deep reinforcement learning in security, aiding in identifying critical steps.

Compared to these studies, which focus on proposing adversarial attacks, we concentrate on testing the robustness of graph-based malware detectors through adversarial graph/sample generation and providing findings to enhance robustness. Additionally, our method does not require the model to be differentiable and significantly broadens the range of target features and models.

## XI. CONCLUSION

In this paper, we introduce a method to evaluate the robustness of FCG-based AMD systems. This method incorporates dependency-aware mutation strategies and utilizes innovative interpretation-based fitness functions to effectively guide perturbation optimization within an FCG. Our experiments demonstrate superior performance across diverse 40 scenarios, and achieve an average attack success rate of 87.9%, significantly outperforming baseline methods. Furthermore, our findings offer valuable insights for enhancing model robustness in future developments, and we also provide an in-depth discussion on the benefits of adversarial retraining.

## REFERENCES

- [1] R. Chatterjee, P. Doerfler, H. Orgad, S. Havron, J. Palmer, D. Freed, K. Levy, N. Dell, D. McCoy, and T. Ristenpart, "The spyware used in intimate partner violence," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 441–458.
- [2] G. Suarez-Tangil and G. Stringhini, "Eight years of rider measurement in the android malware ecosystem," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 1, pp. 107–118, 2020.
- [3] Z. Sun, R. Sun, L. Lu, and A. Mislove, "Mind your weight (s): A large-scale study on insufficient machine learning model protection in mobile apps," in *30th USENIX security symposium (USENIX)*, 2021, pp. 1955–1972.
- [4] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of android malware in your pocket," in *Network and Distributed System Security Symposium (NDSS)*, vol. 14, 2014, pp. 23–26.
- [5] J. Garcia, M. Hammad, and S. Malek, "Lightweight, obfuscation-resilient detection and family identification of android malware," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 26, no. 3, pp. 1–29, 2018.
- [6] X. Xiao and S. Yang, "An image-inspired and cnn-based android malware detection approach," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1259–1261.
- [7] B. Yuan, J. Wang, D. Liu, W. Guo, P. Wu, and X. Bao, "Byte-level malware classification based on markov images and deep learning," *Computers & Security*, vol. 92, p. 101740, 2020.
- [8] Y. Wu, X. Li, D. Zou, W. Yang, X. Zhang, and H. Jin, "Malscan: Fast market-wide mobile malware scanning by social-network centrality analysis," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 139–150.
- [9] L. Onwuzurike, E. Mariconti, P. Andriotis, E. D. Cristofaro, G. Ross, and G. Stringhini, "Mamadroid: Detecting android malware by building markov chains of behavioral models (extended version)," *ACM Transactions on Privacy and Security (TOPS)*, vol. 22, no. 2, pp. 1–34, 2019.
- [10] 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 (CCS)*, 2020, pp. 757–770.
- [11] C. Gao, G. Huang, H. Li, B. Wu, Y. Wu, and W. Yuan, "A comprehensive study of learning-based android malware detectors under challenging environments," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE)*, 2024, pp. 1–13.
- [12] 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*, 2017, pp. 1507–1515.
- [13] K. Pei, Y. Cao, J. Yang, and S. Jana, "Deepxplore: Automated whitebox testing of deep learning systems," in *proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 1–18.
- [14] X. Xie, L. Ma, F. Juefei-Xu, M. Xue, H. Chen, Y. Liu, J. Zhao, B. Li, J. Yin, and S. See, "Deephunter: a coverage-guided fuzz testing framework for deep neural networks," in *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*, 2019, pp. 146–157.
- [15] J. Kim, R. Feldt, and S. Yoo, "Guiding deep learning system testing using surprise adequacy," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1039–1049.
- [16] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 39–57.
- [17] J. Wang, J. Chen, Y. Sun, X. Ma, D. Wang, J. Sun, and P. Cheng, "Robot: Robustness-oriented testing for deep learning systems," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 300–311.
- [18] F. Barbero, F. Pendlebury, F. Pierazzi, and L. Cavallaro, "Transcending transcend: Revisiting malware classification in the presence of concept drift," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 805–823.
- [19] 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)*. IEEE, 2019, pp. 17–20.
- [20] 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 Transactions on Information Forensics and Security (TIFS)*, vol. 15, pp. 987–1001, 2019.
- [21] G. Sriramanan, S. Addepalli, A. Baburaj et al., "Guided adversarial attack for evaluating and enhancing adversarial defenses," *Advances in Neural Information Processing Systems*, vol. 33, pp. 20297–20308, 2020.
- [22] H. Li, Z. Cheng, B. Wu, L. Yuan, C. Gao, W. Yuan, and X. Luo, "Black-box adversarial example attack towards fcg based android malware detection under incomplete feature information," in *32nd USENIX Security Symposium (USENIX)*, 2023, pp. 1181–1198.
- [23] R. Sun, M. Xue, G. Tyson, T. Dong, S. Li, S. Wang, H. Zhu, S. Camtepe, and S. Nepal, "Mate! are you really aware? an explainability-guided testing framework for robustness of malware detectors," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2023, pp. 1573–1585.
- [24] J. Chen, D. Wang, and H. Chen, "Explore the transformation space for adversarial images," in *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy*, 2020, pp. 109–120.
- [25] N. Mani, M. Moh, and T.-S. Moh, "Defending deep learning models against adversarial attacks," *International Journal of Software Science and Computational Intelligence (IJSSCI)*, vol. 13, no. 1, pp. 72–89, 2021.
- [26] Y. Zhou, X. Zhang, J. Shen, T. Han, T. Chen, and H. Gall, "Adversarial robustness of deep code comment generation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 4, pp. 1–30, 2022.
- [27] K. Zhao, H. Zhou, Y. Zhu, X. Zhan, K. Zhou, J. Li, L. Yu, W. Yuan, and X. Luo, "Structural attack against graph based android malware detection," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021, pp. 3218–3235.
- [28] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," in *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS)*, 2017, pp. 4768–4777.
- [29] Open-source dataset and code of this paper. [Online]. Available: <https://anonymous.4open.science/r/FCGHUNTER/>
- [30] W. Yuan, Y. Jiang, H. Li, and M. Cai, "A lightweight on-device detection method for android malware," *IEEE Transactions on Systems, Man, and Cybernetics: systems*, vol. 51, no. 9, pp. 5600–5611, 2019.

- [31] C. Li, X. Chen, D. Wang, S. Wen, M. E. Ahmed, S. Camtepe, and Y. Xiang, "Backdoor attack on machine learning based android malware detectors," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 5, pp. 3357–3370, 2021.
- [32] P. He, Y. Xia, X. Zhang, and S. Ji, "Efficient query-based attack against ml-based android malware detection under zero knowledge setting," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2023, pp. 90–104.
- [33] M. Kakavand, M. Dabbagh, and A. Dehghantaha, "Application of machine learning algorithms for android malware detection," in *Proceedings of the 2018 International Conference on Computational Intelligence and Intelligent Systems*, 2018, pp. 32–36.
- [34] P.-E. Danielsson, "Euclidean distance mapping," *Computer Graphics and image processing*, vol. 14, no. 3, pp. 227–248, 1980.
- [35] More details about this paper. [Online]. Available: <https://sites.google.com/view/fcghunter>
- [36] T. B. Tok, S. Z. Guyer, and C. Lin, "Efficient flow-sensitive interprocedural data-flow analysis in the presence of pointers," in *Proceedings of the 15th International Conference on Compiler Construction (CC)*. Springer, 2006, pp. 17–31.
- [37] Y. Ma, S. Wang, T. Derr, L. Wu, and J. Tang, "Graph adversarial attack via rewiring," in *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining (KDD)*, 2021, pp. 1161–1169.
- [38] Y.-Y. Song and L. Ying, "Decision tree methods: applications for classification and prediction," *Shanghai archives of psychiatry*, vol. 27, no. 2, p. 130, 2015.
- [39] Androzo. [Online]. Available: <https://androzo.uni.lu/>
- [40] Virustotal. [Online]. Available: <https://www.virustotal.com>
- [41] Virusshare. [Online]. Available: <https://www.virusshare.com>
- [42] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel, "Adversarial examples for malware detection," in *Proceedings of the 22nd European Symposium on Research in Computer Security (ESORICS)*. Springer, 2017, pp. 62–79.
- [43] E. Fix and J. L. Hodges, "Discriminatory analysis: Nonparametric discrimination: Small sample performance," 1952.
- [44] L. Breiman, "Random forests," *Machine learning*, vol. 45, pp. 5–32, 2001.
- [45] 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*. PMLR, 2018, pp. 1115–1124.
- [46] Hrat's code. [Online]. Available: <https://sites.google.com/view/hrat>
- [47] M. Alzantot, Y. Sharma, S. Chakraborty, H. Zhang, C.-J. Hsieh, and M. B. Srivastava, "Genattack: Practical black-box attacks with gradient-free optimization," in *Proceedings of the genetic and evolutionary computation conference*, 2019, pp. 1111–1119.
- [48] J. Chen, D. Zhou, J. Yi, and Q. Gu, "A frank-wolfe framework for efficient and effective adversarial attacks," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 34, no. 04, 2020, pp. 3486–3494.
- [49] D. Olszewski, A. Lu, C. Stillman, K. Warren, C. Kitroser, A. Pascual, D. Ukirde, K. Butler, and P. Traynor, "get in researchers; we're measuring reproducibility": A reproducibility study of machine learning papers in tier 1 security conferences," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2023, pp. 3433–3459.
- [50] G. Squillero and A. Tonda, "Divergence of character and premature convergence: A survey of methodologies for promoting diversity in evolutionary optimization," *Information Sciences*, vol. 329, pp. 782–799, 2016.
- [51] M. Anupama, P. Vinod, C. A. Visaggio, M. Arya, J. Philomina, R. Raphael, A. Pinhero, K. Ajith, and P. Mathiyalagan, "Detection and robustness evaluation of android malware classifiers," *Journal of Computer Virology and Hacking Techniques*, vol. 18, no. 3, pp. 147–170, 2022.
- [52] Y. Chen, Z. Ding, and D. Wagner, "Continuous learning for android malware detection," in *32nd USENIX Security Symposium (USENIX)*, 2023, pp. 1127–1144.
- [53] Androguard. [Online]. Available: <https://github.com/androguard/androguard>
- [54] Flowdroid. [Online]. Available: <https://github.com/secure-software-engineering/FlowDroid>
- [55] 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)*. IEEE, 2020, pp. 1332–1349.
- [56] D. Li and Q. Li, "Adversarial deep ensemble: Evasion attacks and defenses for malware detection," *IEEE Transactions on Information Forensics and Security (TIFS)*, vol. 15, pp. 3886–3900, 2020.
- [57] J. Zhang, C. Zhang, X. Liu, Y. Wang, W. Diao, and S. Guo, "Shadowdroid: practical black-box attack against ml-based android malware detection," in *2021 IEEE 27th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2021, pp. 629–636.
- [58] G. Severi, J. Meyer, S. Coull, and A. Oprea, "Explanation-guided backdoor poisoning attacks against malware classifiers," in *30th USENIX security symposium (USENIX)*, 2021, pp. 1487–1504.
- [59] A. Amich and B. Eshete, "Eg-booster: explanation-guided booster of ml evasion attacks," in *Proceedings of the Twelfth ACM Conference on Data and Application Security and Privacy*, 2022, pp. 16–28.
- [60] M. Liu, X. Liu, A. Yan, Y. Qi, and W. Li, "Explanation-guided minimum adversarial attack," in *International Conference on Machine Learning for Cyber Security*. Springer, 2022, pp. 257–270.
- [61] J. Yu, W. Guo, Q. Qin, G. Wang, T. Wang, and X. Xing, "{AIRS}: Explanation for deep reinforcement learning based security applications," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 7375–7392.



**Shiwen Song** received the B.E. degree in software engineering from Henan University, China, in 2020, and the M.E. degree in software engineering from Nanjing University, China, in 2023. She is currently pursuing the Ph.D. degree with the school of computing and information systems, Singapore Management University, starting in 2024. Her research interests include malware detection, program analysis, and Android forensics.



**Xiaofei Xie** is an Assistant Professor and Lee Kong Chian Fellow at Singapore Management University. He obtained his Ph.D from Tianjin University and won the CCF Outstanding Doctoral Dissertation Award (2019) in China. Previously, he was a Wallenberg-NTU Presidential Postdoctoral Fellow at NTU. His research mainly focuses on the quality assurance of both traditional software and AI-enabled software. He has published top-tier conference/journal papers in the areas of software engineering, security and AI, focusing on the use

of AI for software testing and the testing and security of AI systems. In particular, he has received four ACM SIGSOFT Distinguished Paper Awards and a APSEC Best Paper Award.



**Ruitao Feng** is a Lecturer at Southern Cross University, Australia. He received the Ph.D. degree from the Nanyang Technological University. His research centers on security and quality assurance in software-enabled systems, particularly A14Sec&SE. This encompasses learning-based intrusion/anomaly detection, malicious behavior recognition for malware, and code vulnerability detection.



**Qi Guo** received the B.E. degree from Shanghai Jiao Tong University, China, in 2010, and the M.E. degree from Tianjin University, China. He is currently pursuing the Ph.D. degree with the college of intelligence and computing, Tianjin University, China, since 2019. His research interests include code retrieval and code completion. His papers have been published in top-tier software engineering conferences, such as ICSE.



**Sen Chen** (Member, IEEE) is an Associate Professor at the College of Intelligence and Computing, Tianjin University, China. Before that, he was a Research Assistant Professor at Nanyang Technological University, Singapore. His research focuses on software and system security. He got six ACM SIGSOFT Distinguished Paper Awards. More information is available on [.](#)