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

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

***Abstract*—Graph-based detection methods leveraging Function Call Graph (FCG) 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 40.9%. Notably, FCGHUNTER achieves a 100% success rate on robust models (e.g., AdaBoost with MalScan), where baselines achieve less than 24% 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 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], FD-VAE [11]), and graph-based detection (e.g., MalScan [8], MaMaDroid [9]). Graph-based methods have emerged as a particularly promising alternative [12], offering superior performance compared to string-based ones [8], [10], [12]. Specifically, such methods use features extracted from the FCG of an Android package kit (APK)'s smali code (i.e., the intermediate representation of an APK after compilation [13]), 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 [14], [15], [16], [17], [18], 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 [19]. To this end, adversarial attack methods [20], [21], [22], [23], [24] 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 [25], [26], [27].

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 [21], [8], [24]. 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 [28], [23]. Essentially, these attacks involve modifying the smali code of an APK such that its FCG can be affected. HRAT [28], 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 [23] 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 [12] (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

Shiwen Song and Xiaofei Xie are with Singapore Management University, Singapore. E-mail: swsong@smu.edu.sg; xfxie@smu.edu.sg.

Ruitao Feng is with the Faculty of Science and Engineering, Southern Cross University, Australia. E-mail:ruitao.feng@scu.edu.au.

Qi Guo is with Tianjin University, China. E-mail: bxguoqi@tju.edu.cn.

Sen Chen is with the College of Cryptology and Cyber Science, Nankai University, China. E-mail: senchen@nankai.edu.cn.

*Corresponding author.

TABLE I: The Scope of Existing Adversarial Attack Methods.

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

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

feedback mechanisms for better exploitation. However, current methods are often restricted to specific mutation types, such as only adding edges [23], or they perform multiple but simplistic perturbations [28], limiting the generation of diverse FCGs. Concerning feedback mechanisms, existing approaches like HRAT [28] predominantly rely on gradient information, which is costly and unobtainable in non-differentiable ML classifiers like Random Forest or K-nearest neighbors (KNN). These 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], [12] (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 so that the modified sample can bypass detection. 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 [29], 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 outperforming state-of-the-art methods (i.e., HRAT and BagAmmo) by at least 40.9%. 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 [30].

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 [13]), 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 [12], i.e., MalScan, MaMaDroid, and APIGraph.

MalScan [8] emphasizes the importance of 21,986 critical system API calls within function-level call graphs. To encode

the FCG into a fixed-length feature vector, MalScan first identifies which of these predefined system APIs are actually invoked in the given APK (i.e., have corresponding nodes in the FCG). For each identified API node, it computes a structural importance score using one of six centrality metrics. For the rest of APIs not used in the APK, the corresponding feature values are set to zero.

Specifically, it defines four basic centrality metrics: *Degree*, *Katz*, *Closeness*, and *Harmonic*, each offering unique insights into a node's significance. For example, *Degree Centrality* measures a node's importance based on the number of direct connections it has. In a directed FCG, the degree feature for a node v is the sum of its in-degree and out-degree, i.e.,

$$C_D(v) = \frac{\text{degree}(v)}{N-1} \tag{1}$$

where $degree(v)$ represents the number of edges connected to node v, and N is the number of nodes in the FCG.

In addition, MalScan employs two aggregated centrality metrics to enhance feature robustness: *Average* and *Concentrate*. In the *Average* mode, for each identified API node, the four basic centrality scores are computed and averaged to produce a single representative score, resulting in a 21,986-dimensional feature vector. In contrast, the *Concentrate* mode preserves all four basic centrality scores for each identified API node and concatenates them into a single vector, resulting in a 21,986 × 4-dimensional representation.

MaMaDroid [9] extracts behavioral features by constructing a first-order Markov chain over abstracted API call transitions derived from FCGs. Each API call in the FCG is abstracted into a corresponding high-level state, defined at either the *family level* (11 coarse-grained categories such as `android`, `java`, etc.) or the *package level* (446 fine-grained packages such as `android.accounts`, `android.content`, etc.). Based on this abstraction, the FCG is traversed to extract transitions between consecutive states. These transitions are aggregated into a transition matrix that reflects the empirical frequencies of state-to-state invocations.

For a Markov chain with state space S, the transition probability from state j to k is defined as:

$$P_{jk} = \frac{O_{jk}}{\sum_{i \in S} O_{ji}} \tag{2}$$

where O_{jk} denotes the number of observed transitions from state j to k in the graph. The resulting transition matrix is row-normalized so that each row sums to 1.

To obtain a vector representation, the transition matrix is flattened into a $|S|^2$-dimensional vector, resulting in a 121-dimensional vector for family-level abstraction and a 198,916-dimensional vector for package-level abstraction.

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.

In our implementation, we cluster the 442 API packages from MaMaDroid's abstraction (excluding two special categories: `self-defined` and `obfuscated`) into 50 semantic groups using the pretrained API knowledge graph. Including the two special categories, this yields a total of 52 abstracted states. We then construct a 52×52 transition matrix based on the formula 2. After row normalization, the matrix is flattened into a 2,704-dimensional feature vector, which serves as the final input embedding for downstream detection models.

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. Such architectural diversity allows us to investigate how different model types respond to adversarial attacks, revealing variations in their robustness characteristics.

Deep learning. Multi-Layer Perceptron (MLP) is a basic DL model widely used in AMD and adversarial attacks [31], [32], [33], [23]. 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. Due to its differentiable and continuous nature, MLP is susceptible to gradient-based attacks [34].

Instance-based learning. The KNN algorithm, a typical instance-based method commonly used in AMD [35], [21], [28], [33], [23], classifies data points by measuring distances (typically using Euclidean [36] 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. Unlike MLP, KNN outputs discrete class labels based on local neighborhood decisions, potentially offering enhanced robustness against gradient-based attacks [37].

Ensemble-based learning. Random Forest and AdaBoost [21], [9], [33], [23] 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. Unlike MLP's smooth continuous outputs, ensemble models make decisions through voting mechanisms (i.e., majority voting for Random Forest, weighted voting for AdaBoost), creating discrete decision boundaries that may be more robust against gradient-based adversarial attacks [38].

We note that the dependence of adversarial effectiveness on the choice of ML algorithms is a general phenomenon across domains [37], [38]. In FCG-based AMD, this effect is further amplified by the sparsity of predefined API features, where only a small subset of APIs are activated per APK. Consequently, perturbations must act on very limited feature dimensions, making adversarial attacks particularly challenging in this domain.

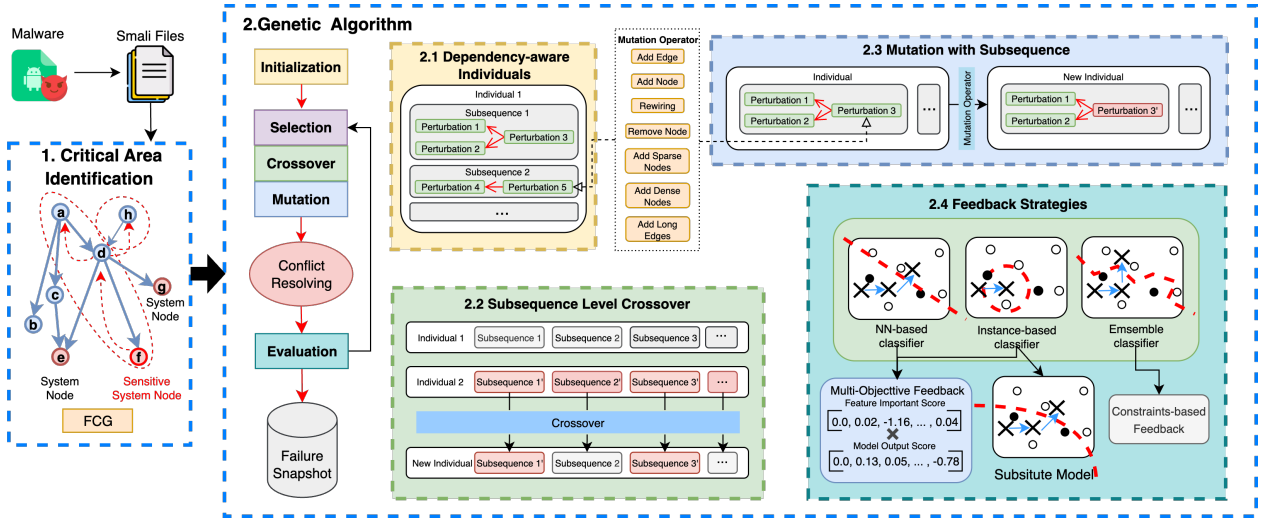


Fig. 1: Overview of FCGHUNTER.

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 the FCG from an APK and calculate the embedding of the FCG, respectively.

Inspired by prior works on adversarial testing [28], [23], we define the AMD testing problem in the problem space as follows: for a given malware sample m, find FCG-level perturbations $\delta \in \Delta$ such that:

$$\begin{aligned} &M(E(G(m))) \neq M(E(G(m) \oplus \delta)) \\ &\text{subject to: } F(m) = F(m + reverse(\delta)) \end{aligned} \tag{3}$$

where Δ represents all possible perturbations in the FCG space, $\oplus$ denotes the operation of applying perturbations to the FCG, $reverse(\cdot)$ transforms FCG-level perturbations into corresponding code-level modifications, $+$ denotes applying these code modifications to the original APK, and F represents the functionality 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; (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 pinpoint the *critical areas* (see §V) comprising nodes and edges that significantly influence model predictions, thus effectively reducing the search 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 (see § VI-A). 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 that cannot be applied to the original FCG. 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 (see § VI-B).
- **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 (see § VI-C) 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 (see § VI-D): 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.

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 [28], 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 call 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 [23] and unreachable conditions [28] 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 $v_1, v_2, \ldots, v_k$ to the G *at once*. To affect the area around an existing node $a \in V_u$, edges $(a, v_1), (a, v_2), \ldots, (a, v_k)$ 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 $v_1, v_2, \ldots, v_k$ are added sequentially, creating the edges $(a, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, 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-E).

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[1]:

- *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

[1]More detailed illustration and code change examples can be found in [28].

and insert an $if\text{-}else$ 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 redirect an existing call from function a to function c through 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 insert a long edge by adding k intermediate functions between an existing function a and a function c, thereby 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 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.

Operator Constraints and Safety Guarantees. To guarantee the safety of our modifications and preserve program integrity, these seven operators enforce the following constraints during the execution:

- *System Method Protection*: Methods from the Android framework are never modified or removed, as their implementations are externally defined and cannot be altered by the APK.
- *Lifecycle Method Protection*: Android lifecycle methods (e.g., `onCreate`, `onDestroy`) form a special category of user-defined methods that override framework callbacks. Modifying these methods or their call relationships may disrupt the system's implicit invocation mechanisms, potentially causing application crashes or unexpected behavior.
- *JVM Special Method Protection*: Constructor methods (i.e., `<init>`) and static initializers (i.e., `<clinit>`) are excluded from modification, as they are managed by the JVM and essential for correct object and class initialization.
- *Cycle Prevention*: Operators that add new edges include cycle detection checks to maintain the acyclic property of the FCG, avoiding infinite recursion and potential deadlocks.
- *Inheritance Safety Checks*: Methods containing `invoke-virtual` instructions in their function body are excluded from *Remove Node* operations to prevent semantic violations during inlining, as such methods may be overridden by subclasses.

Any operation that violates these constraints is rejected, and the GA explores alternative perturbation strategies, demonstrating the practical importance while maintaining attack effectiveness.

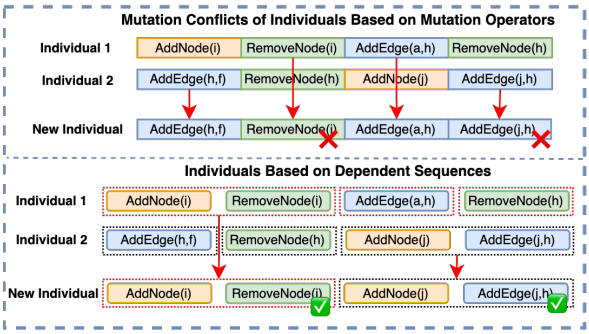


Fig. 2: Crossover with/without Dependency Strategy.

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 $(o_1, o_2, \ldots, o_n)$, 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 subsequences. 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 [39]) 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 [40] 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.

Note that, during the GA initialization, the initial sequence $(o_1, o_2, \ldots, o_n)$ 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 [41], [23]. 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 [29], 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:

$$\begin{aligned} fitness1(I) &= M(E(G+I)) \\ fitness2(I) &= -\sum_{i=0}^{n-1} SHAP(M,G,I)_i \cdot (E(G)_i - E(G+I)_i) \end{aligned} \tag{4}$$

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:

$$\begin{aligned} &fitness1(x) > fitness1(y) \vee \\ &fitness1(x) == fitness1(y) \wedge fitness2(x) > fitness2(y) \end{aligned} \tag{5}$$

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:

$$fitness1(I) = \frac{1}{x}\sum_{i=1}^{k}(M(I)_i^m - M(I)_i^b)$$
$$fitness2(I) = -\sum_{i=0}^{n-1} SHAP(M', G, I)_i \cdot (E(G)_i - E(G+I)_i) \quad (6)$$

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 5.

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) [42]. 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:

$$fitness(I) = \sum_{c \in C} SAT(G, M, I, c) \quad (7)$$

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**: How effective is FCGHunter in attacking target models under concept drift scenarios?
- **RQ3**: What is the performance of FCGHunter?
- **RQ4**: 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 follow standard data collection methodologies described in prior works [28], [23], [43]. Benign samples are obtained from AndroZoo [44] (with a VirusTotal [45] score of 0), and malware samples from VirusShare [46] (with a VirusTotal score above 4). We evaluate FCGHunter against baselines under two dataset settings: (1) standard datasets with stable distribution; and (2) concept drift scenarios (i.e., temporal and ratio-based) with distribution shifts.

Datasets under Stable Distribution. The collected dataset, denoted as *SD*, 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. 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.

Datasets under Concept Drift Scenarios. To comprehensively evaluate the robustness of FCGHunter, we construct additional datasets that simulate realistic concept drift scenarios. These settings are motivated by prior studies [47], [48], [49], which emphasize the importance of evaluating detection models under both temporal and ratio shifts.

Temporal Drift Settings. Temporal drift refers to scenarios where the training data precedes the testing data in chronological order, simulating practical deployment environments. To ensure sufficient training data and capture diverse drift scenarios, we construct three temporal settings, denoted as *TS-1*, *TS-2*, and *TS-3*, by gradually expanding the training set to include data from 3, 4, and 5 consecutive years, respectively. Each year contributes 1,600 samples with 800 benign and 800 malware samples for training, and the test set is drawn from the immediate following year using an 80:20 train-test split, and the attack set consists of 120 real malware samples from the same test year. Details are summarized in Table II.

Ratio Drift Settings. Ratio drift is a form of label distribution shift that reflects the class imbalance commonly observed between benign and malware samples in real-world Android datasets. Motivated by prior studies [47], [43], [50], which report a typical benign-to-malware ratio of around 10:1 in practice, we adopt this imbalance level for our ratio drift scenario. To ensure comparability with the stable setting, we keep the temporal span fixed from 2018 to 2023. In the training set, we maintain 800 benign samples per year and reduce malware samples to 80 per year. The test set also follows the same 10:1 ratio, containing 200 benign and 20 malware samples per year. The attack set includes 20 real malware samples per year, consistent with *SD*. This dataset configuration is referred to as the *RS*, as shown in Table II.

More details about dataset are available on our website [39].

Target Models. To ensure a systematic and comprehensive evaluation of the testing methods, we construct a diverse set of 40 ML-based Android malware detection (AMD) models by combining a wide range of feature types and classifiers. Specifically, we use 8 types of features, including the Degree,

TABLE II: Dataset Configurations for Evaluation.

Setting	Training Set		Test Set		Attack Set	
	Years	Sample Number	Years	Sample Number	Years	Sample Number
Stable Distribution (SD)	2018–2023	(800 B + 800 M) × 6	2018–2023	(200 B + 200 M) × 6	2018–2023	20 M × 6
Temporal Shift-1 (TS-1)	2018–2020	(800 B + 800 M) × 3	2021	600 B + 600 M	2021	120 M
Temporal Shift-2 (TS-2)	2018–2021	(800 B + 800 M) × 4	2022	800 B + 800 M	2022	120 M
Temporal Shift-3 (TS-3)	2018–2022	(800 B + 800 M) × 5	2023	1000 B + 1000 M	2023	120 M
Ratio Shift (RS)	2018–2023	(800 B + 80 M) × 6	2018–2023	(200 B + 20 M) × 6	2018–2023	20 M × 6

Note: B = Benign samples, M = Malware samples.

TABLE III: Detection Performance (F1 Score) of the ML-based AMD Methods under Different Drift Scenarios.

	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**
SD	0.94	0.93	0.94	0.92	0.91	0.95	0.94	0.94	0.92	0.91	0.95	0.83	0.93	0.91	0.90	0.95	0.93	0.93	0.92	0.90
TS-1	0.92	0.88	0.88	0.91	0.93	0.94	0.90	0.91	0.90	0.93	0.93	0.89	0.91	0.90	0.92	0.94	0.90	0.91	0.90	0.93
TS-2	0.90	0.90	0.90	0.92	0.93	0.95	0.92	0.91	0.89	0.93	0.93	0.87	0.92	0.92	0.93	0.90	0.92	0.93	0.92	0.93
TS-3	0.92	0.95	0.94	0.92	0.95	0.95	0.94	0.94	0.90	0.94	0.96	0.79	0.95	0.92	0.94	0.96	0.95	0.94	0.91	0.95
RS	0.86	0.94	0.85	0.74	0.91	0.93	0.93	0.85	0.66	0.88	0.94	0.80	0.86	0.67	0.85	0.87	0.94	0.85	0.73	0.89
	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**
SD	0.95	0.94	0.94	0.91	0.91	0.95	0.94	0.94	0.92	0.90	0.93	0.92	0.91	0.88	0.92	0.92	0.93	0.93	0.88	0.88
TS-1	0.94	0.91	0.91	0.90	0.93	0.95	0.91	0.91	0.90	0.94	0.91	0.87	0.88	0.89	0.92	0.93	0.89	0.90	0.86	0.91
TS-2	0.91	0.91	0.92	0.92	0.93	0.92	0.92	0.92	0.92	0.94	0.90	0.88	0.86	0.89	0.92	0.94	0.91	0.91	0.88	0.93
TS-3	0.96	0.95	0.95	0.91	0.94	0.97	0.95	0.95	0.92	0.95	0.92	0.91	0.91	0.89	0.92	0.96	0.95	0.95	0.90	0.95
RS	0.95	0.94	0.86	0.69	0.87	0.93	0.94	0.86	0.75	0.91	0.86	0.89	0.79	0.55	0.84	0.94	0.91	0.83	0.45	0.89

Katz, Harmonic, Closeness, Average, and Concentrate features from MalScan [8], the family level from MaMaDroid [9], and the package level from APIGraph. Each feature set is paired with five widely used ML-based classifiers: MLP [51], KNN-1 [52], KNN-3, Random Forest (RF) [53], and AdaBoost (AB) [54]. The initial detection performance (measured by F1 score) of these models across four evaluation scenarios is summarized in Table III.

Baselines. We selected three baselines for comparison: a random testing approach and two state-of-the-art adversarial attack methods [28], [23] 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 [23], which has not released its code, we replicated its algorithm based on the descriptions provided in their paper. We denote it as BagAmmo* to clearly distinguish it from the original implementation. To ensure a fair comparison with FCGHUNTER, we configured the 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 [28], although the code is available [55], 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 [56], [57], which restricts its use with tree-based models (i.e., RF and AB) and memory-intensive features (i.e., Average and Concentrate).

Regarding the Random Attack, it randomly generates basic 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 [39] and GitHub [30].

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.

$$ASR = \frac{N_a}{N_m},\ PR = \frac{1}{N_a}\sum_{i=1}^{N_a}\delta_i,\ ASGG = \frac{1}{G}\sum_{g=1}^{G}N_g \tag{8}$$

where N_a is the number of malware that can successfully bypass the AMDs, N_m is the total number of seed malware, $\delta_i = \frac{P_{add,i}}{P_{ori,i}}$ 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.

TABLE IV: Attack Success Rates of FCGHUNTER and Baselines under Stable Distribution.

	MalScan (Degree)						MalScan (Katz)						MalScan (Harmonic)						MalScan (Closeness)					
	MLP	KNN-1	KNN-3	RF	AB	AVG	MLP	KNN-1	KNN-3	RF	AB	AVG	MLP	KNN-1	KNN-3	RF	AB	AVG	MLP	KNN-1	KNN-3	RF	AB	AVG
Ours	**0.82**	**0.73**	**0.76**	**0.78**	**1.00**	***0.82***	**0.77**	**0.68**	**0.69**	**0.94**	**0.96**	***0.81***	**0.82**	**0.90**	**0.83**	**1.00**	**1.00**	***0.91***	**0.88**	**0.97**	**0.84**	**0.91**	**1.00**	***0.92***
HRAT	0.17	0.26	0.11	-	-	*0.18*	0.02	0.04	0.04	-	-	*0.03*	0.02	0.16	0.07	-	-	*0.08*	0.09	0.39	0.34	-	-	*0.27*
BagAmmo*-M	0.58	0.57	0.50	0.13	0.10	*0.38*	0.24	0.23	0.16	0.25	0.20	*0.22*	0.70	0.66	0.61	0.43	0.20	*0.52*	0.77	0.67	0.59	0.13	0.24	*0.48*
BagAmmo*-G	0.65	0.61	0.28	0.14	0.06	*0.35*	0.43	0.22	0.05	0.23	0.20	*0.23*	0.73	0.67	0.58	0.39	0.19	*0.51*	0.74	0.64	0.56	0.11	0.30	*0.47*
Random	0.59	0.62	0.56	0.19	0.03	*0.40*	0.02	0.14	0.08	0.24	0.21	*0.14*	0.59	0.67	0.57	0.08	0.14	*0.41*	0.63	0.58	0.58	0.13	0.26	*0.44*
Initial Error	0.03	0.08	0.03	0.11	0.03	*0.06*	0.01	0.01	0.03	0.23	0.17	*0.09*	0.02	0.13	0.06	0.06	0.11	*0.08*	0.08	0.27	0.16	0.06	0.11	*0.14*
	MalScan (Average)						**MalScan (Concentrate)**						**Mamadroid**						**APIGraph**					
	MLP	KNN-1	KNN-3	RF	AB	AVG	MLP	KNN-1	KNN-3	RF	AB	AVG	MLP	KNN-1	KNN-3	RF	AB	AVG	MLP	KNN-1	KNN-3	RF	AB	AVG
Ours	**0.90**	**0.92**	**0.83**	**0.91**	**1.00**	***0.91***	**0.87**	**0.91**	**0.78**	**0.94**	**0.96**	***0.89***	**0.96**	**0.99**	**0.93**	**1.00**	**0.98**	***0.97***	**0.83**	**0.84**	**0.84**	**0.78**	**0.72**	***0.80***
HRAT	-	-	-	-	-	-	-	-	-	-	-	-	0.06	0.16	0.06	-	-	*0.09*	0.11	0.18	0.08	-	-	*0.12*
BagAmmo*-M	0.75	0.63	0.58	0.38	0.21	*0.51*	0.72	0.66	0.57	0.23	0.23	*0.48*	0.58	0.76	0.71	0.56	0.13	*0.55*	0.81	0.79	0.78	0.20	0.50	*0.62*
BagAmmo*-G	0.72	0.62	0.57	0.19	0.14	*0.45*	0.73	0.60	0.55	0.29	0.10	*0.45*	0.80	0.65	0.53	0.49	0.33	*0.56*	0.79	0.74	0.70	0.29	0.47	*0.60*
Random	0.61	0.62	0.58	0.22	0.06	*0.42*	0.66	0.60	0.58	0.26	0.07	*0.43*	0.09	0.12	0.06	0.29	0.14	*0.14*	0.47	0.14	0.07	0.18	0.16	*0.20*
Initial Error	0.02	0.06	0.04	0.14	0.03	*0.06*	0.02	0.05	0.04	0.06	0.04	*0.04*	0.03	0.08	0.03	0.16	0.00	*0.06*	0.06	0.09	0.04	0.13	0.07	*0.08*

Note: Bold indicates the maximum ASR in each method. Italic underlined values denote the average ASR across ML-based classifiers.

TABLE V: Attack Success Rates of FCGHUNTER and Baselines under Concept Drift Scenarios.

Setting	Method	MalScan (Degree)				MalScan (Concentrate)				Mamadroid				APIGraph			
		MLP	KNN-1	AB	AVG	MLP	KNN-1	AB	AVG	MLP	KNN-1	AB	AVG	MLP	KNN-1	AB	AVG
	Ours	**0.72**	**0.60**	**0.71**	***0.68***	0.73	**0.63**	**0.77**	***0.71***	**0.78**	**0.76**	**0.85**	***0.80***	0.76	**0.77**	**0.78**	***0.77***
	HRAT	0.10	0.16	-	*0.13*	0.08	0.11	-	*0.10*	0.10	0.13	-	*0.12*	0.14	0.14	-	*0.14*
TS-1	**BagAmmo*-M**	0.50	0.21	0.07	*0.26*	0.71	0.57	0.08	*0.45*	0.73	0.72	0.08	*0.51*	**0.82**	0.64	0.30	*0.59*
	BagAmmo*-G	0.48	0.20	0.08	*0.25*	**0.76**	0.55	0.07	*0.46*	**0.78**	0.71	0.08	*0.52*	0.78	0.58	0.40	*0.59*
	Random	0.60	0.23	0.03	*0.29*	0.28	0.13	0.06	*0.16*	0.04	0.03	0.29	*0.12*	0.34	0.08	0.18	*0.20*
	Initial Error	0.03	0.11	0.05	*0.06*	0.02	0.10	0.13	*0.08*	0.08	0.10	0.07	*0.08*	0.09	0.08	0.13	*0.10*
	Ours	**0.95**	**0.75**	**0.73**	***0.81***	**0.90**	**0.75**	**0.82**	***0.82***	0.91	**0.90**	**0.94**	***0.92***	0.93	**0.92**	**0.90**	***0.92***
	HRAT	0.15	0.22	-	*0.19*	0.10	0.14	-	*0.12*	0.15	0.12	-	*0.14*	0.09	0.05	-	*0.07*
TS-2	**BagAmmo*-M**	0.65	0.65	0.18	*0.49*	0.83	0.73	0.19	*0.58*	0.93	0.88	0.34	*0.72*	**0.96**	0.90	0.57	*0.81*
	BagAmmo*-G	0.46	0.63	0.09	*0.39*	0.81	0.73	0.20	*0.58*	**0.93**	0.76	0.51	***0.73***	0.93	0.83	0.60	*0.82*
	Random	0.22	0.65	0.18	*0.35*	0.70	0.66	0.18	*0.51*	0.47	0.09	0.43	*0.33*	0.63	0.07	0.48	*0.39*
	Initial Error	0.12	0.06	0.06	*0.08*	0.03	0.07	0.08	*0.06*	0.02	0.08	0.03	*0.04*	0.01	0.01	0.14	*0.05*
	Ours	**0.88**	**0.81**	**0.68**	***0.79***	**0.87**	**0.79**	**0.75**	***0.80***	**0.92**	**0.88**	**0.84**	***0.88***	0.93	**0.90**	**0.86**	***0.90***
	HRAT	0.18	0.23	-	*0.20*	0.13	0.16	-	*0.15*	0.10	0.07	-	*0.09*	0.07	0.06	-	*0.07*
TS-3	**BagAmmo*-M**	0.68	0.68	0.09	*0.48*	0.82	0.78	0.08	*0.56*	0.63	**0.88**	0.19	*0.57*	0.93	0.85	0.28	*0.69*
	BagAmmo*-G	0.66	0.65	0.08	*0.46*	0.78	0.77	0.07	*0.54*	0.87	0.81	0.18	*0.62*	**0.94**	0.88	0.55	*0.79*
	Random	0.66	0.69	0.10	*0.48*	0.68	0.68	0.08	*0.48*	0.07	0.07	0.19	*0.11*	0.67	0.12	0.28	*0.36*
	Initial Error	0.03	0.03	0.01	*0.02*	0.06	0.08	0.01	*0.05*	0.03	0.05	0.15	*0.08*	0.01	0.03	0.08	*0.04*
	Ours	**0.86**	**0.79**	**0.75**	***0.80***	**0.88**	**0.78**	**0.83**	***0.83***	0.85	**0.85**	**0.88**	***0.86***	0.83	**0.84**	**0.98**	***0.88***
	HRAT	0.26	0.20	-	*0.23*	0.18	0.15	-	*0.17*	0.15	0.20	-	*0.18*	0.15	0.13	-	*0.14*
RS	**BagAmmo*-M**	0.66	0.57	0.09	*0.44*	0.74	0.60	0.29	*0.54*	**0.88**	0.68	0.43	*0.66*	**0.87**	0.82	0.69	*0.79*
	BagAmmo*-G	0.66	0.66	0.08	*0.47*	0.76	0.72	0.22	*0.57*	0.84	0.79	0.53	*0.72*	0.85	0.81	0.72	*0.79*
	Random	0.36	0.53	0.13	*0.34*	0.63	0.63	0.08	*0.45*	0.64	0.34	0.47	*0.48*	0.62	0.63	0.61	*0.62*
	Initial Error	0.13	0.19	0.11	*0.14*	0.11	0.09	0.08	*0.09*	0.12	0.18	0.14	*0.15*	0.10	0.09	0.14	*0.11*

Note: Bold indicates the maximum ASR in each method. Italic underlined values denote the average ASR across ML-based classifiers.

B. RQ1: Effectiveness

Setup. We conduct this evaluation under the *SD* setting to provide a standard training-testing environment environment. 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 [28], [23]. 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. For consistency and fairness, we configure the Random attack identically to the baseline methods, using 300 perturbations per iteration and up to 100 iterations in total.

Results & Analysis. As presented in Table IV, our attack method consistently outperforms all baselines, achieving an average ASR of 87.9%, which is at least 40.9% higher than the best-performing baseline, BagAmmo*-M (47.0%). In comparison, BagAmmo*-G achieves an average ASR of 45.3%, the Random Attack reaches 32.3%, and HRAT performs the worst with only 12.8%. Additionally, the Initial Error baseline shows an average ASR of only 7.6%, confirming that successful attacks are non-trivial.

(1) Baseline Analysis: We found that HRAT generally performs poorly across most models, often yielding an ASR close to zero after removing the initial error, 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 [58] and its results.[2]

This poor performance can be attributed to two key limitations: coarse feedback signals and inaccurate perturbation selection mechanisms. First, the reward in HRAT is designed to penalize large graph modifications by assigning negative rewards proportional to the number of nodes and edges changed. This encourages minimal perturbation but does not explicitly guide the attack towards the model's decision boundary, leading to coarse and potentially misleading optimization signals. Second, in HRAT, once the action type is determined by the Q-network, the attack object (i.e., which node or edge to modify) is selected by computing the gradient of the target model's loss with respect to the graph structure (i.e., the adjacency matrix of the current graph). Although modifying a single edge in the graph appears to be a small change, the binary nature of the adjacency matrix causes this to correspond to a large and non-smooth change in the model's input space. As a result, the computed gradients are often unstable and unreliable, making it difficult to accurately localize effective perturbations. The combination of coarse reward signals and unreliable gradient-based selection creates a cascading effect: early suboptimal actions lead to increasingly poor graph states, from which recovery becomes difficult due to the sequential nature of reinforcement learning, unlike population-based GA methods that maintain diverse solution candidates.

For BagAmmo*, we observed that the choice of surrogate model (i.e., GCN and MLP) has minimal impact, with ASR differences typically under 0.06. Thus, we focus on its performance across different target models. From a feature perspective, BagAmmo* achieves higher ASRs on MaMaDroid and APIGraph than on MalScan. This is because it relies on a single perturbation operator (i.e., adding/removing edges), which directly affects edge-based features like MaMaDroid and APIGraph. In contrast, MalScan features are derived from node-level graph metrics (e.g., degree centrality), which are less sensitive to such simple edge modifications. From a model perspective, BagAmmo* performs reasonably well on simple models (i.e., MLP, KNN-1 and KNN-3) but struggles on ensemble models (i.e., RF and AB). This exposes another limitation of coarse reward signals. Although BagAmmo*'s confidence-based feedback is more principled than HRAT 's penalty-based approach, it remains too coarse for ensemble methods. Since ensemble models use majority voting, their output probabilities are less affected by small changes (e.g., adding one edge). As a result, the feedback lacks sufficient granularity to guide the GA, leading it to local optima. Overall, the combination of a single perturbation type and feedback with limited directionality makes it difficult for BagAmmo* to handle complex feature types and model architectures.

[2]https://github.com/reproducibility-sec/reproducibility/blob/main/sheet1.csv

Finding #1: *Existing methods are notably ineffective, particularly on MalScan and ensemble models, with results that are close to random testing. This ineffectiveness stems from coarse feedback mechanisms and limited perturbation strategies.*

(2) Robustness Analysis: We further analyze the robustness of different features and model architectures by examining ASR variations under our unified attack setup. From the perspective of features, MalScan (Harmonic, Closeness and Average) and MaMaDroid consistently exhibit high average ASRs, typically exceeding 91% 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 average ASRs often below 82%, likely due to the complexity of perturbing these features effectively; From the model perspective, ensemble classifiers (i.e., RF and AB) generally exhibit stronger robustness (i.e., lower ASR from baselines) compared to single models (i.e., MLP, KNN-1 and KNN-3). Nevertheless, our attack still achieves high ASRs (over 90%) on ensemble models when paired with susceptible features in MalScan (Harmonic, Closeness and Average), showcasing its capability to handle challenging models. By contrast, ASRs drop to below 80% on more resilient combinations, such as ensemble models with APIGraph features, reflecting the increased difficulty in attacking these robust setups.

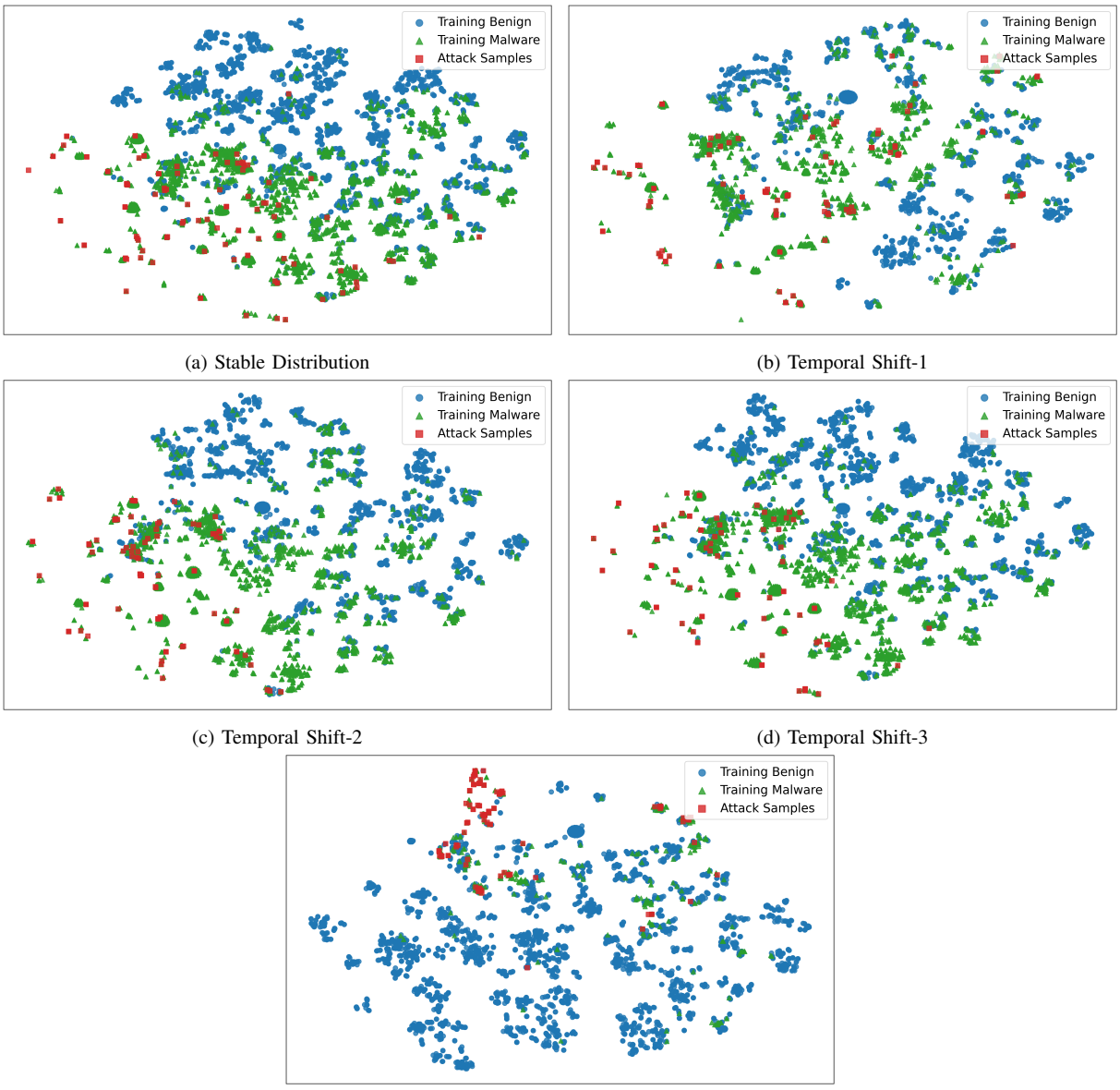
Finding #2: *MalScan (Harmonic, Closeness and Average) and MaMaDroid are less robust, while MalScan (Degree and Katz) and APIGraph are more robust; Ensemble models (i.e., RF and AB) generally show greater robustness.*

Answer to RQ1: FCGHUNTER, *with an average ASR of 87.9%, outperforms baselines by at least 40.9% 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: Effectiveness under Concept Drift

Setup. Compared to RQ1, which evaluates effectiveness under standard static distributions, RQ2 focuses on assessing attack effectiveness under concept drift scenarios (TS-1 to TS-3 and RS in Table II). The baseline setup remains consistent with § VII-B. Given the increased experimental complexity and time cost, we select representative target models based on their performance in Table III and RQ1 results.

Specifically, among the KNN variants, we select KNN-1 due to its wider adoption [8], [9], [28] and slightly better performance compared to KNN-3 in Table III. Between the two ensemble models, AB is chosen over RF because RF consistently underperforms in the RS setting (all F1 scores below 0.74), while AB demonstrates stronger performance across all four settings (all F1 scores above 0.84). Based on these observations, we choose *MLP, KNN-1, and AB* as representative architectures. For feature selection, we choose



(a) Stable Distribution

(b) Temporal Shift-1

(c) Temporal Shift-2

(d) Temporal Shift-3

(e) Ratio Shift

Fig. 3: An Example of Dataset Distributions Across Different Scenarios Based on the APIGraph Feature.

MalScan (Degree) and MalScan (Concentrate) based on their stronger robustness observed in § VII-B among the six variants of MalScan. To ensure diversity, we also include *MaMaDroid and APIGraph*, which represent feature extraction paradigms different from MalScan.

Results & Analysis. As shown in Table V, our method outperforms all baselines in most configurations under four concept drift scenarios, achieving an average ASR of 82.3% compared to BagAmmo*-G (58.1%), BagAmmo*-M (57.1%), Random Attack (35.4%), and HRAT (14.0%). Also, the Initial Error baseline yields only 7.7% ASR, confirming that attack success is not incidental. Next, we provide a detailed analysis to understand these results.

(1) Comparative Effectiveness Analysis: Our approach demonstrates particular strength in challenging scenarios such as MalScan features with tree-based models (e.g., AB), where we consistently achieve higher ASRs than all baselines. We attribute it to our method's diverse perturbation strategies and fine-grained fitness guidance, which effectively navigate complex decision boundaries of robust ensemble models. These patterns are consistent with RQ1 (§ VII-B). In contrast, in specific configurations such as APIGraph and MaMaDroid with MLP, BagAmmo* variants occasionally match or slightly outperform our method. This can be attributed to three factors: (1) MaMaDroid and APIGraph features abstract edge-level structural information, making them highly sensitive

TABLE VI: Perturbation Rates of Successful Attack Samples.

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

to BagAmmo*'s simple edge perturbations, consistent with RQ1 observations in § VII-B; (2) APIGraph represents more aggregated features, which creates a smoother optimization landscape that mitigates the local optima issues typically encountered by GA-based methods. (3) Our SHAP-assisted fitness function assumes consistent training-test distributions, which weakens under concept drift. While our multi-objective approach (§ VI-D) provides some mitigation through model confidence, this distributional assumption still contributes to the occasional performance gaps.

(2) Distribution Drift Impact Analysis: To analyze how different types of concept drift affect attack effectiveness across methods, we visualize the data distributions from Table II using APIGraph, as shown in Fig. 3.

We make three key observations about drift impact: (1) Across all drift scenarios, attack samples (red) exhibit substantial overlap with training malware samples (green), indicating that graph-based features maintain structural consistency despite distributional changes. This overlap confirms that attack effectiveness reflects genuine evasion capabilities rather than model degradation under drift. (2) In the TS-1 setting, limited training data (3 years) leads to sparse coverage of the feature space. Attack samples tend to reside farther from benign regions, making them harder to misclassify and explaining the consistent ASR drop across all methods compared with Table IV. (3) By contrast, TS-2 and TS-3 have enough training samples, and show highly consistent benign (blue) and malware (green) distributions, along with similar ASR outcomes. This suggests that a one-year temporal shift introduces limited structural drift in graph-based features, preserving detector stability across adjacent years.

Answer to RQ2: FCGHUNTER *consistently outperforms all baselines across four concept drift scenarios, achieving an average ASR of 82.3%, which is at least 24.2% higher than the best-performing baseline. These results demonstrate the strong adaptability of* FCGHUNTER *under realistic, dynamic conditions.*

D. RQ3: Performance

1) Perturbation Rates across Target Models:

Setup. To evaluate how much FCGHUNTER perturbs original samples, we compute the average PR across all successful adversarial samples for each target model under the SD setting.

Results & Analysis. As shown in Table VI, successful adversarial samples on MLP and KNNs-based models exhibit relatively low PR, while higher PR on ensemble models. Specifically, MalScan variants such as Harmonic, Closeness, Average, and Concentrate achieve PRs below 0.01 under both MLP and KNN-3. However, PRs under KNNs are slightly higher compared to MLP, suggesting MLP's less robustness. In contrast, tree-based models (i.e., RF and AB) generally exhibit higher PRs, except in the MalScan (Harmonic) setting, where perturbations remain low. While a few cases show PRs exceeding 10, these typically result from unusually large original samples. For instance, in the MalScan (Katz) with AB configuration, 50% of adversarial samples have PRs below 3, and 73% remain under 10, indicating that extreme values are outliers. Overall, higher PRs indicate that ensemble models are more challenging to attack than KNNs and MLP models, yet the scale of applied perturbations in successful attacks remains within a reasonably acceptable range.

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 over 40 iterations using MalScan (Degree) with MLP, the fastest target model. Under the SD setting, we selected 20 malware samples from 2019 and conducted comparative experiments with and without the dependency analysis. To prevent premature termination of these samples, the evaluation phase of the GA was omitted.

Results & Analysis. The green and orange lines in Figure 4 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 [59].

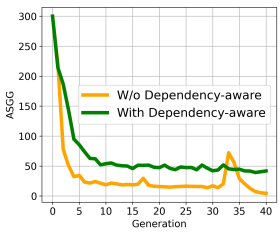


Fig. 4: Impact of Dependency-aware Strategy.

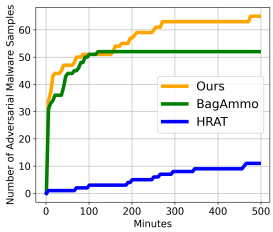


Fig. 5: Runtime Efficiency Comparison with Other State-of-the-Art Methods.

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 model under the SD setting. These models and features were chosen because the baselines (e.g., HRAT) achieved the highest ASR with them.

Results & Analysis. Figure 5 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 high 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 exampless, as shown in Figure 5.

Answer to RQ3: *Successful attacks exhibited relatively low perturbation rates, indicating high attack efficiency. Notable efficiency was demonstrated in solving operator conflicts, thereby ensuring diversified sequences to avoid premature convergence.* FCGHUNTER *has also better runtime compared to other methods.*

TABLE VII: Ablation Configuration Across Target Models.

Model	Cri	ASN	ADN	ALE	Dep	Int	Sur	SAT
MLP	✔	✘	✘	✘	✔	✔	✘	✘
KNN-1	✔	✘	✘	✘	✔	✔	✔	✘
KNN-3	✔	✘	✘	✘	✔	✔	✔	✘
Random Forest	✔	✔	✔	✔	✘	✘	✘	✔
AdaBoost	✔	✔	✔	✔	✘	✘	✘	✔

Note: ✔ indicates the component is applicable and included in the ablation; ✘ indicates it is not applicable and excluded.

E. RQ4: Ablation Studies

Setup. To assess the effectiveness of key components in FCGHUNTER, we quantified the ASR reduction across 40 target models upon the removal of specific components under the SD setting.

The ablated components include critical area identification (§ V, Cri), the dependency-aware strategy (§ VI-B, Dep), and different fitness functions (§ VI-D). For MLP models, the fitness function uses interpretation-based scores (Int); for KNN models, it includes additional surrogate-derived scores (Sur); and for ensemble models, it adopts a constraint-satisfaction-based score (SAT) that analyzes their internal decision paths.

In addition, three complex perturbation operators (§ VI-A), including Add Sparse Nodes (ASN), Add Dense Nodes (ADN), and Add Long Edges (ALE) are exclusively applied to ensemble models (i.e., RF and AB) due to their strong robustness. Since these operators do not involve removal operations, they are less likely to trigger gene conflicts. Therefore, the dependency-aware strategy (Dep) becomes less critical for these models.

As summarized in Table VII, MLP and KNN models are evaluated with Cri, Dep, and Int (with Sur additionally for KNNs), while ensemble models are evaluated with Cri, ASN, ADN, ALE, and SAT. The experimental parameters are consistent with those used in RQ1 (§ VII-B).

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

(1) Impact of Critical Area Identification (Cri): Removing the Cri causes noticeable ASR drops across most models and features, highlighting its universal role in guiding effective perturbations. From the feature perspective, MalScan (Katz and Concentrate) show the largest ASR drops upon removing Cri, with drops up to 0.29 for MalScan (Katz) with MLP. As discussed in RQ1 (§ VII-B), these features are particularly robust and typically require targeted, high-impact perturbations to succeed. Without Cri to narrow the search space to the most effective subgraphs, the attack is prone to spending its perturbation budget on less influential regions, leading to failure. In contrast, APIGraph shows minimal impact from removing Cri (drops below 0.05), likely due to its aggregated feature nature where most regions are already relatively effective for perturbation. By contrast, removing Cri causes consistent ASR drops across different classifiers, ranging from 0.12 to 0.17. This consistency reflects the model-agnostic nature of Cri. Its effectiveness arises from identifying feature-relevant subgraphs based on predefined sensitive APIs during feature extraction, independent of the downstream classifier.

TABLE VIII: 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	-Cri	-ASN	-ADN	-ALE	-SAT	-Cri	-ASN	-ADN	-ALE	-SAT
MalScan (Degree)	-0.11	**-0.08** ↓	-0.12	-0.12	**-0.03** ↓	-0.05	**-0.03** ↓	-0.15	**-0.09** ↓	-0.14	-0.16	-0.11	-0.01	**0.00** ↓	**-0.19** ↓	-0.10	-0.09	**-0.19** ↑	-0.02	-0.56	-0.09
MalScan (Katz)	**-0.29** ↑	**-0.60** ↑	**-0.56** ↑	**-0.23** ↑	-0.17	-0.11	-0.11	**-0.23** ↑	**-0.50** ↑	-0.12	-0.19	**-0.19** ↑	-0.05	-0.13	-0.38	**-0.22** ↑	-0.15	-0.01	-0.28	-0.48	**-0.15** ↑
MalScan (Harmonic)	-0.10	**-0.08** ↓	-0.07	-0.18	-0.17	-0.18	-0.13	-0.18	-0.13	-0.15	**-0.25** ↑	-0.14	-0.01	**0.00** ↓	-0.41	-0.18	-0.13	**0.00** ↓	-0.01	-0.60	-0.11
MalScan (Closeness)	-0.16	-0.21	-0.08	-0.18	-0.18	-0.16	-0.21	-0.17	-0.13	-0.13	-0.15	-0.11	-0.01	-0.02	-0.35	-0.12	-0.12	-0.06	-0.03	-0.25	-0.11
MalScan (Average)	-0.20	-0.15	-0.12	-0.18	-0.18	-0.18	**-0.28** ↑	-0.20	-0.18	**-0.18** ↑	-0.21	-0.17	-0.03	-0.02	-0.23	-0.15	-0.14	**0.00** ↓	-0.13	**-0.18** ↓	-0.09
MalScan (Concentrate)	-0.18	-0.15	-0.11	**-0.23** ↑	**-0.27** ↑	**-0.23** ↑	**-0.28** ↑	-0.14	-0.11	-0.09	-0.16	**-0.19** ↑	**-0.06** ↑	**-0.14** ↑	-0.61	-0.19	**-0.16** ↑	-0.05	**-0.50** ↑	-0.58	-0.15
MaMaDroid	-0.18	-0.20	-0.16	-0.18	-0.22	-0.19	-0.20	-0.14	-0.22	-0.18	-0.16	-0.14	**0.00** ↓	**0.00** ↓	**-0.75** ↑	-0.13	-0.12	**0.00** ↓	**0.00** ↓	**-0.78** ↑	-0.14
APIGraph	**-0.04** ↓	-0.13	**-0.04** ↓	**-0.05** ↓	-0.05	**-0.03** ↓	**-0.03** ↓	**-0.01** ↓	-0.18	**-0.03** ↓	**-0.04** ↓	**-0.04** ↓	**0.00** ↓	**0.00** ↓	-0.68	**-0.09**↓	**-0.02** ↓	**0.00** ↓	**0.00** ↓	-0.72	**-0.07**↓
AVG	-0.16	-0.20	-0.16	-0.17	-0.16	-0.14	-0.16	-0.15	-0.19	-0.13	-0.17	-0.14	-0.02	-0.04	-0.45	0.15	-0.12	-0.04	-0.12	-0.52	-0.11

Note: For each feature row, ↑ marks the setting with the largest ASR drop (strongest impact), and ↓ marks the setting with the smallest drop. The last row reports the average ASR drop across all features.

(2) Impact of Dependency-aware Strategy (Dep): The Dep component mitigates gene conflicts to preserve population diversity during GA optimization. Removing Dep reduces ASR across configurations, confirming its utility. However, unlike Cri, the impact patterns are less consistent across different settings, reflecting the context-dependent nature of gene conflicts, which means gene conflicts do not occur uniformly across all optimization scenarios. Notably, MalScan (Degree) consistently exhibits minimal sensitivity to Dep removal (drops below 0.09). This suggest that degree-based features, which capture local structural properties, benefit more from direct connectivity manipulation than from diverse population exploration.

(3) Impact of Fitness Function Variants: Removing fitness function variants (Int, Sur, and SAT) results in ASR drops across all scenarios, confirming their utility for breaking local optima in GA. From the feature perspective, APIGraph consistently shows minimal sensitivity (drops below 0.09). This may indicate that for the aggregated feature, the original prediction probabilities already provide sufficient optimization signals, potentially reducing the need for fine-grained fitness components.

(4) Impact of Complex Perturbation Operators: 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.

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.*

To further analyze the generality of complex operators, we extended the ablation study by applying them to simpler models (MLP, KNN-1, and KNN-3). Specifically, we incorporate

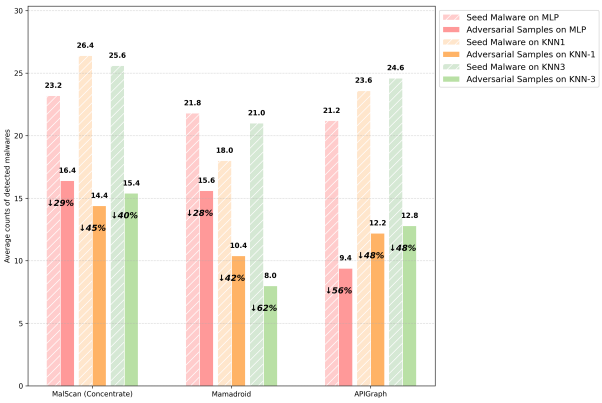


Fig. 6: Detection Score Difference Between Malware and Corresponding Adversarial Samples.

all seven mutation operators into a full configuration, denoted as *All Operators*, and then perform three additional ablations by selectively removing each of the complex operators (i.e., ASN, ADN, ALE) in turn. We mainly focus on two representative feature sets, MalScan (Concentrate) and MaMaDroid. The results in Table IX report the relative performance changes of these ablations with respect to the *All Operators* baseline.

Compared with the RQ1 results in Table IV, incorporating complex operators into these simple models leads to slight decreases in ASR. This indicates that large perturbations are not always beneficial for these classifiers, which can be effectively bypassed by the basic mutations. In many cases, removing a complex operator even yields marginal improvements, suggesting that smaller operators could provide incremental search steps for GA optimization. By contrast, robust ensemble models typically require stronger perturbations to overcome their inherent stability, making complex operators more valuable in those settings. Therefore, we only use these complex operators to ensemble models.

Answer to RQ4: *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.*

TABLE IX: Evaluation of Complex Operators on Single Models.

	MLP				KNN-1				KNN-3			
	All Operators	-ASN	-ADN	-ALE	All Operators	-ASN	-ADN	-ALE	All Operators	-ASN	-ADN	-ALE
MalScan (Concentrate)	0.81	+0.01	+0.02	+0.03	0.88	+0.03	-0.03	+0.03	0.79	+0.07	+0.05	+0.08
MaMaDroid	0.91	0.00	+0.03	-0.08	0.87	-0.01	-0.01	-0.01	0.86	+0.03	+0.03	+0.04

Note: Numbers under -ASN/-ADN/-ALE denote relative changes w.r.t. the "All Operators" baseline. Positive values indicate improvements, negative values indicate decreases.

VIII. Discussions

A. Evaluation on Real-world AMD.

To explore the robustness issues present in real-world models, we selected VirusTotal [45], a leading platform for malware analysis, as our target [23], [24], [33]. We adopt the VirusTotal detection count, which refers to the number of antivirus engines that classify a sample as malicious. This count serves as a proxy signal and is widely used in prior works [60], [47] to label malware samples. We randomly selected 10 successful adversarial samples from each of the 9 attack scenarios, covering three classifiers (MLP, KNN-1, KNN-3) and three feature representations (MalScan, MaMaDroid, and APIGraph). For each adversarial sample, we retrieved its corresponding original malware and computed the VirusTotal detection counts before and after the attack. We then calculated the average detection counts for both original and adversarial samples in each scenario.

Figure 6 presents the results in 18 bars (9 pairs), where each pair compares the average detection scores of the seed malware and its adversarial counterpart under a specific model-feature combination. 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 [39].

While our experimental setting follows prior works [23], [61], it is important to note that the detection of the original malware may be influenced by prior submissions to the engines, i.e., the original malware sample becomes known malware and tends to be detectable by more VirusTotal engines over time [62]. This could bias the comparison between the detection results of the original samples and their variants. Nevertheless, we emphasize that these observations reflect two complementary aspects of real-world challenges: on one hand, detection algorithms may exhibit robustness weaknesses when confronted with adversarial attacks (i.e., variants of known malware cannot be detected); on the other hand, commercial AMD engines may react slowly to newly generated variants, leaving a time window where adversaries can evade detection. Together, these findings highlight the urgent need for more robust and proactive defenses.

B. 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 [63]. From the model perspective, adversarial retraining has been a popular strategy to boost ML model robustness [25], [26], [64]. We further evaluated whether adversarial samples generated by FCGHunter can improve model robustness via retraining. Specifically, we enhanced the original training set under SD setting with 90 adversarial samples and tested the improved model's resilience using an independent set of 60 adversarial samples, aiming to verify the mitigation of previously identified vulnerabilities. During the retraining process, we varied the number of adversarial samples in the training set from 10 to 90 to observe changes in the ASR from the test samples.

As shown in Figure 7, as adversarial samples are introduced into the training set, there is a sharp decline in ASR for all features. This decrease is most pronounced within the first 10 samples added, indicating that retraining with even a small number of adversarial samples significantly enhances model robustness. However, different features show different levels of resilience. For example, features like MaMaDroid exhibit unstable or fluctuating ASR performance even with increased adversarial training, which may indicate the model is susceptible to overfitting and forgetting previously correct patterns due to MaMaDroid's family features being quite low-dimensional (i.e., 121 dimensions). Similarly, the issue also appears in APIGraph, although it is less severe than in MaMaDroid. In contrast, MalScan variants demonstrate more stable because they capture a broad and complex set of behaviors or characteristics, making them less prone to drastic shifts in performance with the introduction of adversarial examples. Thus, during retraining, it is crucial to choose a suitable ratio of adversarial samples for different feature types to mitigate this issue. In summary, the adversarial examples generated by FCGHunter could be effectively used to enhance the resilience of ML-based models.

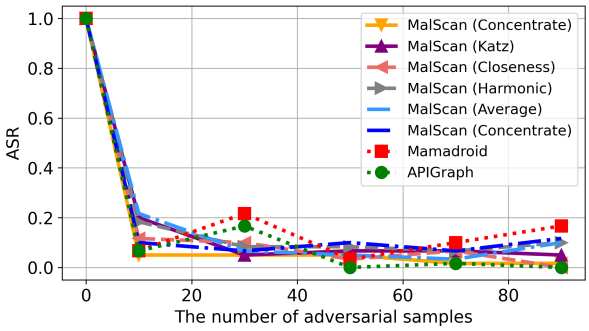


Fig. 7: Detection Score Difference Between Malware and Corresponding Adversarial Samples.

C. 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-E. 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.

D. Robustness of Mutation Operators Against Preprocessing.

Mutation operators are often vulnerable to static preprocessing techniques such as dead code elimination [65], [66], [67] and control-flow simplification [68], [69]. At the code modification level, our mutation operators reduce to three fundamental strategies, namely *AddEdge*, *AddNode*, and *RemoveNode*, which serve as the building blocks for more complex operators.

Both *AddNode* and *AddEdge* perturbations introduce semantically reachable code paths. For *AddNode*, although the inserted function may return unused values, removing such calls requires non-trivial def-use and data-flow analysis to determine that these calls have no observable behavioral impact. However, this is an expensive process that is not typically deployed in AMD pipelines. *AddEdge* supports multiple implementations. While simplistic forms such as IF(FALSE) are susceptible to branch pruning during static simplification, more resilient variants like wrapping the call within a try-catch block generate realistic control-flow patterns that are less likely to be flagged or eliminated. This technique has been shown to be effective in previous work [23].

In contrast, *RemoveNode* performs inline expansion of the target function into its callers. While this introduces duplicated basic blocks, these blocks include meaningful control and data flows, including downstream function calls. Detecting and removing them would require basic-block-level code similarity analysis such as clone detection across inline contexts, which is both time consuming and uncommon in post-compilation Android workflows.

Notably, all mutation operations are performed at the intermediate representation (IR) level extracted post-compilation using Androguard. The resulting modified code is structurally embedded and is preserved through standard repackaging processes. Since repackaged malware is rarely subject to full recompilation or IR-level optimization, these mutations remain intact and operational.

In summary, while no attack strategy can claim complete immunity against all possible preprocessing transformations, our design prioritizes semantic reachability, realistic code patterns, and IR-level modification to mitigate the impact of current static simplification techniques.

E. Scope of Threat Model.

Traditional concealment techniques, such as manifest modification [33], code obfuscation [70], and repacking [71], primarily aim to bypass rule-based or pattern-matching detectors by disguising or altering program artifacts. For example, manifest modification may insert redundant permissions, declare unused hardware features, or register extra components to mislead simple pattern checks; code obfuscation often renames classes, methods, and variables to disguise semantic meaning and break signature matching; and repacking typically involves embedding malicious payloads into benign applications or repackaging existing malware with new cryptographic signatures to evade detection. By contrast, adversarial attacks against AI-based malware detection do not attempt to hide the malicious logic itself. They introduce small, semantics-preserving perturbations to code structure changes that can mislead graph-based ML classifiers. Consequently, the underlying malicious behavior may remain observable to rule-based heuristics, yet a ML-based detector that normally achieves near-perfect accuracy (e.g., 95%) can still be bypassed.

We focus on an emerging attack surface unique to AI-driven detectors, which anticipates how adversaries may exploit robustness vulnerabilities in machine learning systems to evade detection as these systems are increasingly deployed in real-world malware defenses.

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 [58]. 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 [72] versus FlowDroid [73]), and the models evaluated; 3) releasing our code, models, and seed malware to facilitate verification and replication of our findings [39].

The other potential threat arises from static preprocessing techniques (e.g., advanced dead code elimination), which could in principle reduce the effectiveness of our perturbations. Although our design emphasizes semantic reachability, realistic control/data flows, and IR-level modification (see § VIII-D), more advanced preprocessing might still filter out some injected code. Looking ahead, we also envision future extensions such as opaque predicates (e.g., `if (time < 0)`), which could further increase semantic complexity and harden perturbations against advanced preprocessing.

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 [74], [21], [75], [28], [33], [23] 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 [74], [21], [76] or interpretability-assisted techniques [77], [24], 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 [21] 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 [28], [23]. Zhao et al. [28] 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. [23] 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 [78], [79], [24], [80]. For example, Amich et al. [78] leverages SHAP to guide adversarial example crafting against ML models, seeking meaningful perturbations to aid in assessing the system's robustness. Sun et al.[24] utilize SHAP to guide attacks on string-based detectors, identifying critical API permissions and inserting uncalled functions. Similarly, Yu et al.[80] 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.

ACKNOWLEDGMENTS

This research is partially supported by the National Research Foundation, Singapore, the Cyber Security Agency

under its National Cybersecurity R&D Programme (NCRP25-P04-TAICeN), and the Ministry of Education, Singapore under its Academic Research Fund Tier 2 (Proposal ID: T2EP20223-0043). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore, Cyber Security Agency of Singapore and the Ministry of Education, Singapore. We would also like to thank Dr. Lili Quan for her valuable external guidance and insightful suggestions throughout this research.

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] H. Li, S. Zhou, W. Yuan, X. Luo, C. Gao, and S. Chen, "Robust android malware detection against adversarial example attacks," in *Proceedings of the Web Conference 2021*, 2021, pp. 3603–3612.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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. 20 297–20 308, 2020.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] Open-source dataset and code of this paper. [Online]. Available: https://anonymous.4open.science/r/FCGHUNTER/

[31] 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.

[32] 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.

[33] 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.

[34] K. Ren, T. Zheng, Z. Qin, and X. Liu, "Adversarial attacks and defenses in deep learning," *Engineering*, vol. 6, no. 3, pp. 346–360, 2020.

[35] M. Kakavand, M. Dabbagh, and A. Dehghantanha, "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.

[36] P.-E. Danielsson, "Euclidean distance mapping," *Computer Graphics and image processing*, vol. 14, no. 3, pp. 227–248, 1980.

[37] Y. Harbi, K. Medani, C. Gherbi, Z. Aliouat, and S. Harous, "Roadmap of adversarial machine learning in internet of things-enabled security systems," *Sensors*, vol. 24, no. 16, p. 5150, 2024.

[38] H. Chen, H. Zhang, D. Boning, and C.-J. Hsieh, "Robust decision trees against adversarial examples," in *International Conference on Machine Learning*. PMLR, 2019, pp. 1122–1131.

[39] More details about this paper. [Online]. Available: https://sites.google.com/view/fcghunter

[40] 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.

[41] 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.

[42] 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.

[43] J. Zheng, J. Liu, A. Zhang, J. Zeng, Z. Yang, Z. Liang, and T.-S. Chua, "Maskdroid: Robust android malware detection with masked graph representations," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 331–343.

[44] Androzoo. [Online]. Available: https://androzoo.uni.lu/

[45] Virustotal. [Online]. Available: https://www.virustotal.com

[46] Virusshare. [Online]. Available: https://www.virusshare.com

[47] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro, "{TESSERACT}: Eliminating experimental bias in malware classification across space and time," in *28th USENIX security symposium (USENIX Security 19)*, 2019, pp. 729–746.

[48] D. Arp, E. Quiring, F. Pendlebury, A. Warnecke, F. Pierazzi, C. Wressnegger, L. Cavallaro, and K. Rieck, "Dos and don'ts of machine learning in computer security," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3971–3988.

[49] J. Liu, J. Zeng, F. Pierazzi, L. Cavallaro, and Z. Liang, "Unraveling the key of machine learning solutions for android malware detection," *arXiv preprint arXiv:2402.02953*, 2024.

[50] T. Chow, M. D'Onghia, L. Linhardt, Z. Kan, D. Arp, L. Cavallaro, and F. Pierazzi, "Breaking out from the tesseract: Reassessing ml-based malware detection under spatio-temporal drift," *arXiv preprint arXiv:2506.23814*, 2025.

[51] 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.

[52] E. Fix and J. L. Hodges, "Discriminatory analysis: Nonparametric discrimination: Small sample performance," 1952.

[53] L. Breiman, "Random forests," *Machine learning*, vol. 45, pp. 5–32, 2001.

[54] 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.

[55] Hrat's code. [Online]. Available: https://sites.google.com/view/hrat

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Proceedings of the 13th international conference on mining software repositories*, 2016, pp. 468–471.

[61] L. De Rose, G. Andresini, A. Appice, and D. Malerba, "Olivander: a counterfactual-based method to generate adversarial windows pe malware," *Data Mining and Knowledge Discovery*, vol. 39, no. 5, p. 46, 2025.

[62] L. Wang, H. Wang, T. Zhang, H. Xu, G. Meng, P. Gao, C. Wei, and Y. Wang, "Android malware family labeling: Perspectives from the industry," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 2176–2186.

[63] 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.

[64] Y. Chen, Z. Ding, and D. Wagner, "Continuous learning for android malware detection," in *32nd USENIX Security Symposium (USENIX)*, 2023, pp. 1127–1144.

[65] M. Rodriguez-Cancio, B. Combemale, and B. Baudry, "Automatic microbenchmark generation to prevent dead code elimination and constant folding," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 132–143.

[66] T. Theodoridis, M. Rigger, and Z. Su, "Finding missed optimizations through the lens of dead code elimination," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 697–709.

[67] M. Chen, G. Li, L.-I. Wu, and R. Liu, "Dce-llm: Dead code elimination with large language models," *arXiv preprint arXiv:2506.11076*, 2025.

[68] X.-C. Wen, Y. Chen, C. Gao, H. Zhang, J. M. Zhang, and Q. Liao, "Vulnerability detection with graph simplification and enhanced graph representation learning," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2275–2286.

[69] X. Cheng, J. Ren, and Y. Sui, "Fast graph simplification for path-sensitive typestate analysis through tempo-spatial multi-point slicing," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 494–516, 2024.

[70] M. Hammad, J. Garcia, and S. Malek, "A large-scale empirical study on the effects of code obfuscations on android apps and anti-malware products," in *Proceedings of the 40th international conference on software engineering*, 2018, pp. 421–431.

[71] H. Ma, S. Li, D. Gao, D. Wu, Q. Jia, and C. Jia, "Active warden attack: On the (in) effectiveness of android app repackage-proofing," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 5, pp. 3508–3520, 2021.

[72] Androguard. [Online]. Available: https://github.com/androguard/androguard

[73] Flowdroid. [Online]. Available: https://github.com/secure-software-engineering/FlowDroid

[74] 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.

[75] 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.

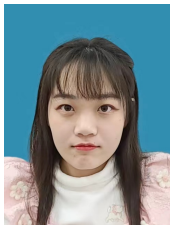
[76] J. Zhang, C. Zhang, X. Liu, Y. Wang, W. Diao, and S. Guo, "Shadowdroid: practical black-box attack against ml-based android malware detection," in *2021 IEEE 27th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2021, pp. 629–636.

[77] 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.

[78] 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.

[79] 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.

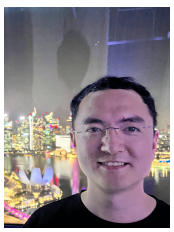
[80] J. Yu, W. Guo, Q. Qin, G. Wang, T. Wang, and X. Xing, "{AIRS}: Explanation for deep reinforcement learning based security applications," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 7375–7392.



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 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 AI4Sec&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 a Professor at the College of Croptology and Cyber Science, Nankai University, China. His research focuses on software security and software supply chain security. He got six ACM SIGSOFT Distinguished Paper Awards. More information is available on his website.