Black-box Adversarial Example Attack towards FCG Based Android Malware Detection under Incomplete Feature Information

Heng Li[1], *Zhang Cheng*[1,3], *Bang Wu*[1], *Liheng Yuan*[1], *Cuiying Gao*[1], *Wei Yuan*[1], *Xiapu Luo*[2]
[1] *Huazhong University of Science and Technology*
[2] *The Hong Kong Polytechnic University*
[3] *NSFOCUS Technologies Group Co., Ltd*

Abstract

The function call graph (FCG) based Android malware detection methods have recently attracted increasing attention due to their promising performance. However, these methods are susceptible to adversarial examples (AEs). In this paper, we design a novel *black-box* AE attack towards the FCG based malware detection system, called BagAmmo. To mislead its target system, BagAmmo purposefully perturbs the FCG feature of malware through inserting "never-executed" function calls into malware code. The main challenges are two-fold. First, the malware functionality should not be changed by adversarial perturbation. Second, the information of the target system (e.g., the graph feature granularity and the output probabilities) is absent.

To preserve malware functionality, BagAmmo employs the *try-catch trap* to insert function calls to perturb the FCG of malware. Without the knowledge about feature granularity and output probabilities, BagAmmo adopts the architecture of generative adversarial network (GAN), and leverages a multi-population co-evolution algorithm (i.e., Apoem) to generate the desired perturbation. Every population in Apoem represents a possible feature granularity, and the real feature granularity can be achieved when Apoem converges.

Through extensive experiments on over 44k Android apps and 32 target models, we evaluate the effectiveness, efficiency and resilience of BagAmmo. BagAmmo achieves an average attack success rate of over 99.9% on MaMaDroid, APIGraph and GCN, and still performs well in the scenario of concept drift and data imbalance. Moreover, BagAmmo outperforms the state-of-the-art attack SRL in attack success rate.

1 Introduction

Occupying about 85% of the global mobile operating system market, Android has become the main target of mobile malware in the world. A recent security report shows that on average, about 10000 new mobile malware samples were captured per day [20]. The rapidly increasing of malware poses severe threats to Android users [30, 37, 49], e.g., privacy leakage and economic losses. To tackle this problem, a variety of machine learning based Android malware detection methods have been designed to identify malware based on their features [3, 23, 25, 35, 41, 55, 58, 64, 66, 67]. As a common feature for Android malware detection, Function Call Graph (FCG) [23, 25, 41, 55, 58, 66, 67] (e.g., frequent subgraph [15] and E-FCG [8]) provides important clues for understanding how Android apps work. In an FCG, every node represents a function or an *abstracted* function (e.g., class, package or family), and every edge denotes the calling relationship between caller and callee. As depicted in Fig. 1,

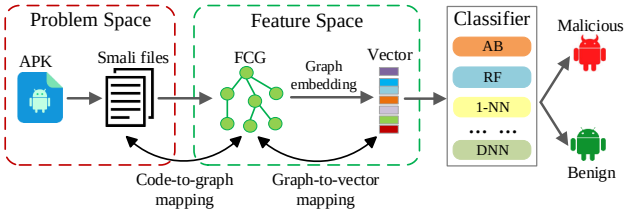

Figure 1: FCG based Android malware detection framework.

the FCG based Android malware detection usually consists of three steps. First, the FCG feature (e.g., frequent subgraph) is extracted from the Android Package (APK) file. Second, the FCG is transformed into a feature vector, i.e., graph embedding. Third, the feature vector is processed for malware prediction. Existing studies [41, 58, 67] demonstrate that the FCG based Android malware detection methods can achieve promising performance.

Unfortunately, the FCG based malware detection is susceptible to adversarial examples (AEs) [9, 40, 48, 50, 51, 62], which are generated by imposing well-crafted adversarial perturbations on normal examples to induce misclassification. To evade detection, an adversary just needs to manipulate a malicious app by elaborately modifying (e.g., inserting non-functional function calls) and repackaging its code. Although malware manipulation takes place in problem space (depicted

by the first box in Fig. 1), it changes the FCG (e.g., adding new edges) and perturbs the feature vector in feature space (described by the second box in Fig. 1). Once the perturbation helps the feature vector stride over the target classifier's decision boundary, the repackaged malware will evade detection. Up to now, a variety of AE attacks towards Android malware detection have been proposed to produce evasive Android malware. Most of them [21, 24, 27, 33, 34] direct at non-graph features (i.e., syntax features) based detection models that use binary feature vectors for app classification. Recently, increasing attention has been paid to the AE attacks towards graph feature (i.e., semantic feature) based detection models [6] [10]. For example, Bostani *et al.* [6] leverage random search to find optimal perturbation for APK files in a black-box setting. Chen *et al.* [10] propose a method to exert optimal perturbations on Android APK files.

Up to now, how to produce Android malware to circumvent the FCG based detection is still an open issue. This motivates us to investigate the generation of AEs to fight against the FCG based Android malware detection. In practice, building evasive malware needs to consider the following realistic problems that have not been well addressed.

(1) *Malware functionality preservation*. The malware manipulation should be able to mislead its target classifier in the premise of malware functionality preservation.

(2) *Problem-feature space gap*. Since the feature vector in feature space cannot be directly perturbed, adversaries have to modify malware code in problem space and expect their modification brings about the desired adversarial perturbation on feature vector.

(3) *Strict black-box setting*. For adversaries, the target classifier is a strict black box and its architecture, parameters and output probabilities are all unknown.

(4) *Feature information absence*. Adversaries cannot get the feature used by their target classifier, i.e., the FCG and the feature vector obtained by graph embedding (denoted in the second box of Fig. 1). Moreover, a detection system may use one of several possible feature granularities, e.g., class level, package level and family level (as discussed in Subsection 2.1). In practice, the feature granularity information is often unavailable to adversaries.

To overcome the above challenges, we design a **b**lack-box **a**ttacks towards FC**G** based **A**ndroid **m**alware detection with **m**ulti-population co-ev**o**lution, termed BagAmmo. BagAmmo works under the *incomplete feature information* condition, which means adversaries do not know the granularity of the FCG feature used by their target system. Our main tasks include designing a malware manipulation technique used in problem space, and developing an algorithm to derive adversarial perturbation in feature space. BagAmmo constructs a dedicated Generative Adversarial Network (GAN) and employs its *generator* to generate candidate manipulations under the guidance of its *discriminator*. The generator is implemented by our proposed **A**dversarial multi-**p**opulation c**o**-**e**volution algorith**m** (Apoem). BagAmmo iteratively queries its target detection system with manipulated samples, and gradually learns the desired manipulation from a sequence of query-reply pairs. BagAmmo uses the following techniques to overcome the above challenges.

(1) BagAmmo leverages a novel malware manipulation method "*try-catch trap*" to insert never-executed function calls into malware code for functionality preservation.

(2) BagAmmo maps the FCG into a feature vector, which transfers the impacts of malware manipulation into feature space and hence bridges the problem-feature space gap.

(3) To overcome the challenge of strict black box, the discriminator substitutes the target classifier and guides the generator to figure out the desirable manipulation rapidly.

(4) In Apoem, every population corresponds to a possible feature granularity. Owning to the cooperative evolution among populations, Apoem converges to the real feature granularity under incomplete feature information.

Our main contributions are summarized as follows.

- We propose a novel black-box AE attack BagAmmo towards the FCG based Android malware detection. BagAmmo does not require complete information about feature space, and hence it is a broad-spectrum attack with strong generalizability.
- We theoretically analyze why Apoem can mitigate the prematurity problem that often plagues the evolution algorithms.
- We conduct extensive experiments on three state-of-the-art (SOTA) malware detection methods MaMaDroid [41], API-Graph [67] and GCN [65] with five classifiers (e.g., RF and DNN) under three feature granularities. BagAmmo surpasses the SOTA attack (i.e., reinforcement learning based method SRL) in our experiments. It achieves an average attack success rate of over **99.9%** on all **32** target detection systems. Our experiments also confirm the BagAmmo's attack efficiency and resilience to concept drift and data imbalance.

Roadmap. The remainder of the paper is organized as follows: §2 introduces preliminaries; §3 presents the problem formulation; §4 discusses how to manipulate the malware; §5 describes the algorithm of perturbation generation; §6 gives the performance evaluation; §7 reviews relevant work; §8 provides the limitations and discussion.

2 Preliminaries

2.1 Features for Android malware detection

In this subsection, we focus on the static features that are obtained prior to app execution and widely used in Android malware detection. Earlier studies devote more attention to syntax features, e.g., requested permissions [14,35,70] , intent actions [18,44,64], Inter-Component Communications (ICCs) [4, 17] and API calls [3, 47]. Recently, semantic features [41, 58, 67] (e.g., FCGs) have attracted increasing attention. They

can characterize the behavior and functionality of apps, and hence achieve promising performance.

As the most common semantic feature, FCGs are often constructed based on smali files. A function or an *abstracted* function denoted by its function name (e.g., java.lang.StrictMath: max()), class name (e.g., java.lang.StrictMath), package name (e.g., java.lang), or family name (e.g., java) can be used to represent a node in an FCG. Therefore, there exist four feature granularities in FCGs , i.e., function level, class level, package level and family level, as shown in Fig. 2. The features with finer granularities (e.g., class level) usually have a more complex graph structure, causing heavier computational overhead and requiring dimensionality reduction [41].

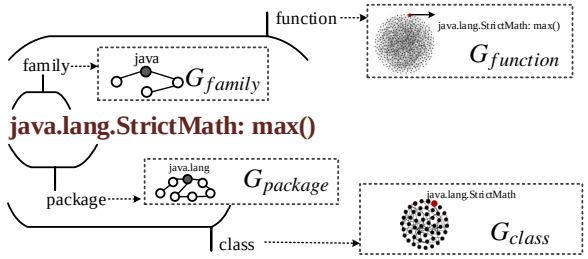

Figure 2: Different granularities of the FCG.

Clearly, the knowledge about the feature granularity of the target system is helpful for adversaries to generate AEs. However, this prior knowledge is hard to obtain in practice. Hence, we put forward the incomplete feature information assumption, assuming that adversaries do not know the feature granularity of the target system.

2.2 FCG based Android malware detection

Here we introduce three state-of-the-art FCG based detection methods, which will act as the target detection systems in our experiments.

Mamadroid. Mamadroid [41] considers the package-level or family-level FCGs as its features. More specifically, it adopts 340 packages and 11 families. To extract a feature vector from an FCG, Mamadroid constructs a Markov chain with the transition probabilities among packages or families. The extracted feature vectors are then used to train a classifier (e.g., KNN and SVM) for app classification.

APIGraph. Different from Mamadroid, APIGraph [67] is a general framework for further enhancing the performance of the graph based Android malware detection methods. It employs a clustering algorithm (e.g., K-means) to aggregate the nodes (i.e., functions) of an FCG, based on the similarity among their semantics. It then uses a specific function to represent all functions in every cluster. Finally, APIGraph builds a new FCG with coarser granularity, in which every node denotes a cluster of functions and every edge indicates the call between two clusters. Experiments show that the new FCG can result in better classification performance.

GCN. Graph Convolutional Network (GCN) is a powerful graph embedding method, which can be utilized to detect malware. For instance, the GCN is used to convert the control flow graph into a feature vector for malware detection in [65][1]. In Section 6, we will apply the GCN to the FCG based Android malware detection.

While these methods have achieved impressive results, they are susceptible to adversarial examples. The existence of adversarial examples is attributed to the problem that the decision boundaries of classification models are non-ideal [26, 52]. This problem becomes more serious in Android malware detection since the static analysis methods cannot precisely model the malware behavior. Therefore, the existing Android malware detection systems are not really secure [2].

3 Problem formulation

Here we first introduce the system and threats considered in our work, and then propose an attack formulation to guide the design of black-box AE attacks.

3.1 System & Threat

Fig. 1 depicts the FCG based Android malware detection system considered in this work. Suppose an adversary launches a black-box AE attack towards this system to produce real evasive malware. To this end, the adversary first gets the classes.dex file from an APK file, and further decompiles it into a series of smali files, as shown in Fig. 3. The adversary manipulates the smali code according to its perturbation, and rebuilds the code to obtain a new APK file. The adversary then queries the detection system with the generated malware sample, utilizes the received binary decision (i.e., benign or malicious) to update its perturbation, and then rebuilds a new malware sample. The above procedure is repeated until a real evasive malware is obtained.

The adversary only knows that the target system uses FCG feature for malware detection. However, the adversary does not know the feature granularity and the graph embedding method used by the target system. Moreover, the adversary has no information about the architecture, the parameters and the output probabilities of the target classifier. As for the defender, it can use static analysis and white list based defenses to resist evasive malware. In addition, the defender may raise alarms once the number of queries from a user is unusually large[2].

[1] [65] mainly studies how to attack malware detectors, although it proposes a GCN based malware detection method.

[2] Our experiments indicate that our method only needs dozens of queries to generate the perturbations that can successfully attack the target model. Moreover, our method can further reduce the number of perturbations by conducting more queries (e.g., several hundreds of queries). To accelerate

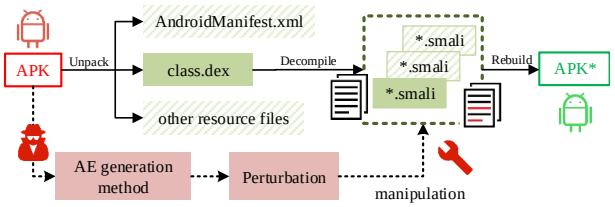

Figure 3: Overview of the AEs generation.

3.2 Attack formulation

For convenience, we first use s and m to refer to the malware sample and the manipulation, respectively. We then use two functions $\mathcal{M}_G(\cdot)$ and $\mathcal{M}_V(\cdot)$ to denote the code-to-graph mapping and the graph-to-vector mapping shown in Fig. 1, respectively. Through manipulating the malware sample s with m, the adversary changes the input graph from $G = \mathcal{M}_G(s)$ to $\widetilde{G} = \mathcal{M}_G(s+m)$, where G and $\widetilde{G}$ represent the original input FCG and the perturbed input FCG, respectively. Suppose $L(\cdot)$ denotes the label (i.e., benign or malicious) predicted by the target classifier. Then, the desired adversarial manipulation m^* can be derived by solving the following problem:

$$L(\mathcal{M}_V(\mathcal{M}_G(s))) \neq L(\mathcal{M}_V(\mathcal{M}_G(s+m^*))) \quad (1)$$

under the constraint of malware functionality preservation.

The above formulation points out two tasks for us: 1) designing a manipulation technique to modify malware code while preserving malware functionality, and 2) developing an adversarial perturbation generation algorithm to realize m^*. Due to the challenges of problem-feature space gap and strict black-box setting, $\mathcal{M}_G(\cdot)$ and $\mathcal{M}_V(\cdot)$ are actually unknown to the adversary. Hence it is extremely hard to derive the desired adversarial perturbation in one shot. This motivates us to develop an evolutionary algorithm (i.e., Apoem) to gradually find the desired perturbation. We will discuss how to fulfill the above two tasks in Sections 4 and 5, respectively.

Furthermore, it is noted that a variety of graph adversarial attack models [5, 12, 40, 48, 57, 62, 71] have been proposed in the community of machine learning. Although these methods offer inspirations to us, they cannot be directly applied to our attack for two reasons. First, graph adversarial attack models launch attacks from feature space. However, the attack against Android malware detection cannot directly access feature space, and has to indirectly affect feature space through manipulating malware code in problem space. Second, our attack needs to meet practical requirements (i.e., **R1**-**R4** discussed in Subsection 4.1), which are absent in existing graph adversarial attacks. Therefore, specialized study is needed for malware adversarial attack design.

the attack process, we provide a substitute network to fit the target model. The related experiments can be found in Section 6.3

4 Malware manipulation

In this section, we first introduce the common requirements and the existing techniques [10, 45, 65] of malware manipulation, and then propose a new malware manipulation technique.

4.1 Background of malware manipulation

Although the manipulation on malware is intuitively simple, the challenges come from the following requirements.

R1: Functional Consistency. The malware functionality should keep consistently before and after manipulation.

R2: All-granularities influence. Since the feature granularity (e.g., family level and package level) of malware detection is unknown, malware manipulation should be able to affect the features of all granularities [41].

R3: Resilience to static analysis. Malware manipulation should not be hindered by static analysis inspection [3] [13, 42], and cannot completely rely on dead codes (i.e., unreachable instruction blocks).

R4: Non-stationary perturbation. Manipulation should be non-stationary and cannot be restricted to a fixed set of operations (e.g., a pre-determined white list [65]), to reduce the risk of being identified.

Existing manipulation methods are summarized below.

Inserting dead codes: To maintain functional consistency, [10] chooses to insert dead codes (e.g., no-op calls) into smali files. Unfortunately, these codes can be easily detected and filtered, violating the requirement **R3**. For example, [15] proposes a weighted sensitive-API-call-based Android malware family classification method, which can resist the impact of no-op calls.

Adding valueless calls: [10] creates user-defined classes and adds valueless calls (i.e., invoking empty functions) into them. However, these calls may be susceptible to static analysis and cannot attack the class-granularity FCGs, violating the requirements **R2** and **R3**. For example, the Android malware detection method proposed in [64] does not use self-defined functions as feature. Hence, this method is not influenced by the valueless calls inserted by adversaries.

Adding functions from a white list: To change FCGs, the authors of [65] add a function coming from a predetermined white list. However, once adversarial examples are captured, the white list will be revealed and adversarial attacks may fail. Please refer to requirement **R4**.

Opaque predicates: [45] leverages opaque predicates to insert new APIs for malware detection evasion. Specifically, this method constructs obfuscated conditions where the outcome is always known in design phase but the truth value is difficult or impossible to determine by static analysis. Hence this method can effectively resist static analysis. However, it

[3] In this work, the static analysis mainly refers to the program analysis techniques that only examine the source code but do not execute the program.

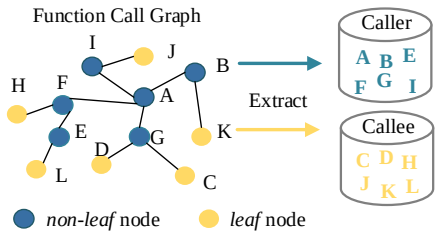

Figure 4: Selecting callers and callees from an FCG.

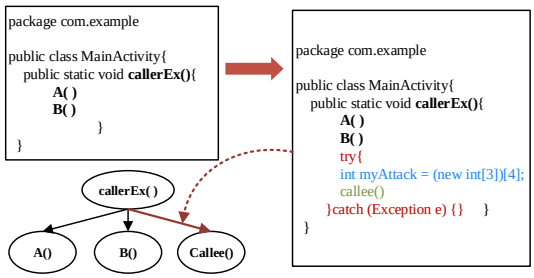

Figure 5: An example of try-catch trap.

may introduce some undesired functions (e.g., the *random* function), which impose unexpected impacts on FCGs.

4.2 The proposed manipulation method

Here we design a new malware manipulation method to modify smali code. Clearly, we cannot remove nodes or edges from FCGs, according to the requirement **R1**. Hence, we only consider adding (or inserting) nodes or edges. However, adding isolated nodes (i.e., the functions that are not invoked or do not invoke others) is not recommended for two reasons. First, the isolated nodes are easily detected by static analysis (e.g., some program analysis techniques that perform redundant code elimination would remove unreachable code [22]). Second, adding nodes usually cannot impact feature space, since lots of malware detectors utilize edges (instead of nodes) for classification. As a result, we select adding edges (i.e., calls) in our manipulation method. Then the rest of the problem includes: how to create *candidate* edges, how to select desirable edges from the candidate edges, and how to insert the selected edges. In this section, we only consider the first and the third problems. The second problem will be solved in Section 5.

(1) How to create candidate edges?

Up to now, how to impose all-granularities influence (required by **R2**) on FCG with incomplete feature information (i.e., the feature granularity is unknown) has not been thoroughly studied. To tackle this problem, we propose to create an edge between two nodes of any type by adding a function call between a caller and a callee. This method changes the FCG no matter what kind of feature granularity is used. Then the problem becomes how to determine the caller and the callee for every candidate edge. Due to the requirement **R4**, we cannot utilize a white list to generate callers and callees. Instead, we propose to generate them from the functions used by malware itself. In this way, we can ensure that the candidate edges created for different malware are diverse, hence satisfying the requirement **R4**.

Now we study where to place the added edges. An FCG consists of *non-leaf* nodes and *leaf* nodes, as depicted in Fig. 4. The non-leaf nodes are user-defined functions, and the leaf nodes correspond to Android standard functions (e.g., $java/io/File;->exists()$) or the user-defined functions that do not invoke others. In our method, non-leaf nodes (i.e., user-defined functions) are selected as callers, since they are easily inserted with new function calls. Leaf nodes are chosen as callees, since invoking a function that does not invoke others will not trigger unintended calls. Here we avoid generating unintended calls because they may further impose perturbations on the FCG, which is beyond our expectation. Furthermore, we supply more discussion on callee selection in Appendix 10.1. Now we can use the above method to create candidate edges. In Section 5, we will propose an algorithm to select the most desirable edges for manipulation.

(2) How to insert selected edges?

We assume that the desirable edges have been selected, and study how to insert the corresponding function calls into smali files under the requirements of **R1** and **R3**. Our proposed method is called **try-catch trap**. It first inserts a try-catch block into the caller, and places the statement of invoking callee in its try block. It then adds several statements in front of this function call statement. These statements are used to trigger a pre-selected exception (e.g., an arithmetic exception). Now we analyze why this method works. First, it inserts a function call statement in smali files, hence changing the FCG by adding a new edge. Second, the function call statement is never executed, hence preserving malware functionality. For illustration, Fig. 5 gives an example of try-catch trap. Suppose the codes in the left box come from a malware sample. The function *callerEX()* is selected as our caller. We place a try-catch block in this function, and invoke the function callee() after the blue statement is executed. In this way, we can add a new edge into the FCG, as shown in Fig. 5. When the try-catch block is executed, an exception of *IndexOutOfBoundsException* will be thrown, and the statement of function call will be skipped over. In summary, our method can be considered as a variant of opaque predicates. It carefully constructs obfuscated conditions that are difficult to determine during static analysis, hence possessing the ability to resist static analysis.

The main steps of inserting function calls are briefly described in Appendix 10.3.

5 Adversarial perturbation generation

In Subsection 4.2, we propose the question of how to select

desirable edges from the candidate edges. To answer this question, we develop a novel GAN model and the algorithm Apoem to find the desired adversarial perturbation.

5.1 Challenges & Solutions

We first introduce the main procedure of BagAmmo below.

(1) Given a pre-selected malware sample, BagAmmo finds some callers and callees from the smali codes, and uses them to create a set of candidate edges, as discussed in Section 4. With candidate edges, BagAmmo generates a variety of samples through manipulating malware, and sends them (i.e., queries) to its target system for malware detection.

(2) The target model sends back a reply for a query. In our strict black-box setting [68], every reply contains only the binary classification outcome (i.e., malicious or benign).

(3) Through learning from the query-reply pairs, BagAmmo gradually recognizes the most desirable edges that can successfully induce misclassification.

The main challenges in designing BagAmmo include: 1) the feature granularity of the target model is unknown, 2) a large number of queries are usually required in the strict black-box attack scenario [4] [1, 36]. Our countermeasures are briefly explained below.

Surmising feature granularity. Our adversarial multi-population co-evolution algorithm, i.e., Apoem, uses one population to represent a possible feature granularity. The multiple populations, corresponding to multiple possible feature granularities, cooperatively evolve until the population corresponding to the real feature granularity keeps alive but the others fade away. In this way, BagAmmo can accurately identify the feature granularity used by its target model, as will be shown in Subsection 5.3.

Reducing the number of queries. BagAmmo constructs a novel *substitute* model to simulate its target model. The substitute model is trained with the samples generated by Apoem and labeled by the target model. As will be shown in Subsection 5.4, once the substitute model is well trained, BagAmmo only needs to attack it instead of the target model, hence greatly reducing the number of queries.

5.2 The overview of BagAmmo

Following the architecture of GANs, BagAmmo adopts a generator and a discriminator that are cooperatively trained.

Generator: The generator is responsible for generating perturbations, i.e., the new edges added into the FCG. It is implemented with an adversarial multi-population co-evolution algorithm (i.e., Apoem).

Discriminator: The discriminator is introduced to stimulate the generator to improve its perturbations. It is implemented with a GCN, acting as a substitute network [10] to simulate the target model.

Training: In each round of model training, the generator modifies the malware's code and sends the rebuilt malware to the target model or the substitute model for malware detection. BagAmmo makes a choice between the target model and the substitute model with a variable probability p. After receiving the queries, the target model sends back its replies, i.e., the binary decisions. With the query-reply pairs, BagAmmo trains the substitute model and guides its generator to improve its generated perturbations. The probability p keeps growing as the number of rounds increases, to decrease the number of queries sent to the target model.

5.3 Adversarial Multi-population co-evolution

The main challenge faced by the generator is that the real feature granularity is unknown. To facilitate the understanding, we consider the case where the target system uses the family-level feature but we perturb the class-level feature. In this case, we will fall into a huge search space, hence prolonging model training time and requiring more queries. To alleviate this problem, BagAmmo uses the Apoem algorithm to surmise the real feature granularity. Apoem follows the general framework of evolutionary algorithms, but it introduces cooperation among multiple populations to speed up convergence. Along with the evolution, the population corresponding to the real feature granularity gradually stands out from the crowd. In the following, we first describe the main components of Apoem depicted in the red block of Fig. 6, and then discuss how to use these components to generate the desired perturbation.

(1) Population & Individual. A population represents a collection of generated AEs under a certain feature granularity. For example, the family-level population consists of the AEs generated under the assumption that the target classifier uses a family-level FCG as its input. Apoem adopts multiple populations, each of which corresponds to one possible feature granularity (i.e., family, package and class). Each individual in a population gives a perturbation that can be imposed on the original FCG[5], i.e., the set of edges added into the FCG. As shown in Fig. 7 (a), the above graph denotes the original FCG, and the below graph represents an adversarial example. Accordingly, the perturbation, i.e., the edge set $(A \to E, B \to D)$, is considered as an individual. We use $x_r^{(i,j)}$ to refer to the j-th individual of the i-th population in the r-th generation of Apoem. We have $x_r^{(i,j)} = \{e_1^{(i,j)}, e_2^{(i,j)}, ..., e_n^{(i,j)}\}$,

[4] In this scenario, both $\mathcal{M}_G(\cdot)$ and $\mathcal{M}_V(\cdot)$ mentioned in Subsection 3.2 are unknown. Moreover, the reply of the black-box model contains only the binary classification outcome (e.g., many malware detection websites [54] only sends back a binary decision instead of class probabilities).

[5] Strictly speaking, an individual refers to an adversarial example in a population. However, the difference between adversarial example and malicious example is perturbation. Hence we use the perturbation to represent an individual.

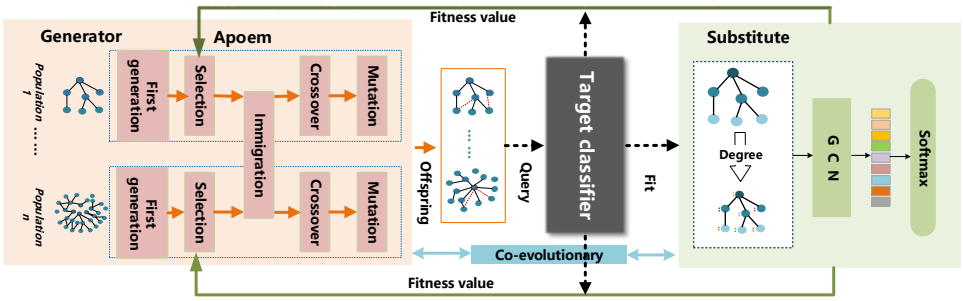

Figure 6: The model architecture of BagAmmo.

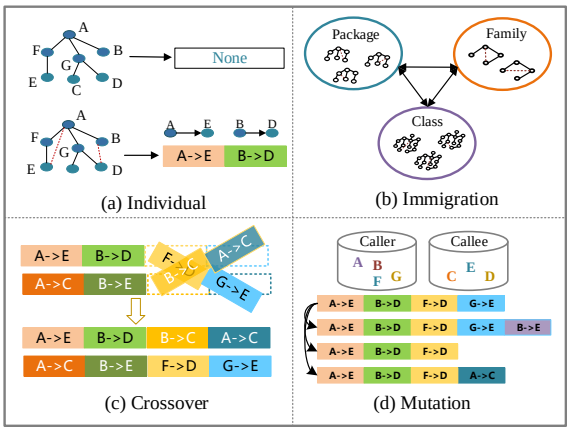

Figure 7: How multiple populations cooperatively evolve?

where $e_k^{(i,j)}$ $(1 \leq k \leq n)$ is the added edge. In the initial phase, we need to collect sufficient individuals to build the populations. Therefore, we randomly perturb the original FCG, and get a set of individuals for each population.

(2) Fitness & Selection. Apoem employs the metric fitness to select superior individuals and eliminate inferior individuals. This metric reflects the aggressivity and the invisibility of an AE. Its calculation takes into account two factors: threat degree T and perturbation amount L. The threat degree is measured according to the output of the target model $F(\cdot)$ or the substitute model $S(\cdot)$[6]. For an individual x, the threat degree is defined as:

$$T = \begin{cases} 1 - F(x) & \text{if target model is used} \\ 1 - S(x) & \text{if substitute model is used} \end{cases} \quad (2)$$

The perturbation amount is calculated as the number of added edges. Furthermore, Apoem introduces the *elitist* selection strategy [53] to pass on the good genes of individuals to the next generation, through retaining the fittest individuals and eliminating the others.

[6]For the target or substitute model, the input is detected to be malicious when its output $F(x)$ or $S(x)$ equals to or approaches 1.

(3) Immigration. In general, the individuals with high fitness have a greater chance of producing better offsprings. To produce more high-quality individuals, Apoem leverages the immigration operation to transfer individuals with high fitness within one population into other populations. Accordingly, the superior individuals immigrate to different populations, making all populations cooperatively evolve to generate better AEs. There exist two kinds of immigration in Apoem: fine-to-coarse (e.g., from class level to family level) and coarse-to-fine (e.g., from family level to class level), as shown in Fig. 7 (b). We first consider the fine-to-coarse case where one individual in the class-level population is immigrated into the package-level population. In this case, the name of the packages related to the perturbation (e.g., java.lang.StrictMath->java.lang) is retained and the individual containing only package names is then put into the package-level population. Now we consider the coarse-to-fine case where the individual from the package-level population is injected into the class-level population. Since a package may contain multiple classes, we randomly select one class used by malware code to replace the package and then put the individual containing class names into the class-level population.

(4) Crossover. Apoem leverages crossover to randomly swap genes from two parents to produce offsprings. More specifically, K pairs of individuals are randomly chosen from a population as parents, and half of the perturbation in every pair is exchanged to produce two offsprings, as shown in Fig. 7 (c). Suppose the parents are $x_r^{(i,j_1)} = \{e_1^{(i,j_1)}, e_2^{(i,j_1)}, e_3^{(i,j_1)}, e_4^{(i,j_1)}\}$ and $x_r^{(i,j_2)} = \{e_1^{(i,j_2)}, e_2^{(i,j_2)}, e_3^{(i,j_2)}, e_4^{(i,j_2)}\}$, where $e_k^{(i,j)}$ is an added edge (e.g., A->E) in Fig. 7 (c) . The offsprings derived by crossover are $x_{r+1}^{(i,j_1)} = \{e_1^{(i,j_1)}, e_2^{(i,j_1)}, e_3^{(i,j_2)}, e_4^{(i,j_2)}\}$ and $x_{r+1}^{(i,j_2)} = \{e_1^{(i,j_2)}, e_2^{(i,j_2)}, , e_3^{(i,j_1)}, e_4^{(i,j_1)}\}$, respectively.

(5) Mutation. Apoem employs mutation to bring new changes to a population. As depicted in Fig. 7 (d), there are three possible mutation modes: 1) randomly adding function calls on the existing perturbation, 2) randomly reducing existing perturbation, and 3) randomly exchanging existing perturbations. They can be mathematically expressed

as $x_{r+1}^{(i,j)} = \{e_1^{(i,j)}, ..., e_n^{(i,j)}, e_{n+1}^{(i,j)}\}$, $x_{r+1}^{(i,j)} = \{e_1^{(i,j)}, ..., e_{n-1}^{(i,j)}\}$, and $x_{r+1}^{(i,j)} = \{e_1^{(i,j)}, ..., e_{n-1}^{(i,j)}, e_{n+1}^{(i,j)}\}$, respectively.

5.4 Substitute model

Apoem only knows the binary decision of its target model, making it hard to accurately evaluate individuals. To overcome this challenge, we design a novel substitute model to simulate the target model, and provide Apoem with approximate class probabilities.

The inputs of our substitute model are *function-level* FCGs generated according to the perturbation produced by the generator. We use a GCN (i.e., Graph Convolutional Network) to extract features from the substitute model, as shown in the green block in Fig. 6. GCNs extend convolution to graph data, and they are good at utilizing structural information and node information to fulfill graph-related machine learning tasks. However, the main obstacle of applying GCNs to our task is the absence of node property. That is, FCGs do not provide property information for their nodes. To alleviate this problem, we propose to use **out degree** and **in degree** of a node as its features.

Now we briefly explain how to use a GCN to extract features from the inputs. The GCN has multiple convolutional layers. Each layer aggregates node properties using a propagation rule, and the aggregated features are then processed by the next layer. Accordingly, we can obtain a feature vector to represent the FCG using iterative computation.

5.5 Algorithm design

Apoem aims to conglutinate multi-population co-evolution mechanism and substitute model to cooperatively generate adversarial perturbations. Its main procedure is given in Algorithm 1. In this algorithm, F is the target classifier, N is the maximum number of individuals in a population, and r_{max} denotes the maximum number of generations. In every iteration, Apoem first randomly selects the target or the substitute model (lines 3-6), calculates fitness for every individual (lines 8-12), then retains high-rate individuals based on fitness (line 13), and finally conducts immigration, crossover and mutation (line 14). In addition, the substitute model should be trained when it is selected, as denoted by lines 15-17. The individual with the highest fitness is outputted when Apoem terminates. Below we summarize the important considerations for Apoem.

(1) *How to implement co-evolution in Apoem?*
The co-evolution in Apoem is two-fold. On one hand, the generator and the discriminator cooperate with each other to improve the generated perturbation. On the other hand, multiple populations cooperatively evolve through immigration.
(2) *How to avoid premature convergence?*
When the genes of some high-rate individuals quickly dominate the population [32], premature convergence occurs and evolutionary algorithms converge to a local optimum. Apoem can mitigate premature convergence owing to the cooperation among populations. Through immigration, different populations share their good genes and further promote their evolution. Meanwhile, immigration also helps the populations jump out from local optimum traps. Our theoretical analyses are given in Appendix 10.2.
(3) *When to terminate our algorithm?*
There are three stopping criteria for Apoem. First, all the offsprings cannot induce misclassification on the target model anymore. Second, the perturbation amount does not decrease within several continuous rounds. Third, the maximum number of rounds is reached.
(4) *How to modify the APK according to the output?*
The output of our algorithm is the caller-callee function pairs. According to the output, we use the try-catch trap mentioned in Section 4 to insert the callee function into the caller function, in order to implement adversarial perturbation. The implementation details can be found in Appendix 10.3.

Algorithm 1: The Apoem Algorithm

Input: The FCG G of a given APP
1 Population initialization;
2 **for** r *in* r_{max} **do**
3 **if** $(r-3)/r_{max} > random(0,1)$ **then**
4 Is_substitute = 1;
5 **else**
6 Is_substitute = 0;
7 **for** *each* P^i **do**
8 **if** *Is_substitute = 1* **then**
9 Get fitness $T(x_r^{(i,j)})$ from substitute model;
10 **else**
11 Get fitness $T(x_r^{(i,j)})$ from target model;
12 Get $L(x_r^{(i,j)})$ for every individuals;
13 Select top N individuals according to $T(x_r^{(i,j)})$ and $L(x_r^{(i,j)})$ in turn ;
14 Immigration(); Crossover(); Mutation();
15 **if** *Is_substitute = 1* **then**
16 Get the result from the target model $F(x_r^{(i,j)})$;
17 Train substitute model with $F(x_r^{(i,j)})$ and $x_r^{(i,j)}$;
18 Determine whether algorithm should terminates;

6 Experiments

In this section, we conduct extensive experiments to evaluate BagAmmo by answering the following research questions:
RQ1: Effectiveness. Does BagAmmo successfully attack the SOTA Android malware detection methods?
RQ2: Evolution. How do multiple populations in Apoem

evolve?

RQ3: Efficiency. Does the substitute model help to decrease queries and improve attack efficiency?

RQ4: Overhead. Is there a trade-off between manipulation overhead and attack success rate?

RQ5: Resilience. Is BagAmmo still effective when there exists concept drift or data imbalance?

RQ6: Functionality. Does our adversarial perturbation change the functionality of malware?

Datasets. Our dataset contains 21399 benign samples and 22975 malicious samples, which come from Androzoo[7], Faldroid dataset [15] and Drebin dataset [3]. Every sample collected from Androzoo is detected by VirusTotal [54]. Only when a sample is detected to be malicious by more than four antivirus systems, we label it as malware. The details of our dataset are provided in Appendix 10.4.

Furthermore, our experiments adopt two configurations to evaluate BagAmmo. According to the first configuration, we use 10-fold cross-validation to train the target models. To evaluate the attack methods, we randomly choose 100 malicious examples (not included in the training data of target model) that can be correctly classified by target models for evasive malware generation. For the second, we divide the dataset according to the years that Android apps emerge as discussed in Section 6.5. The newly-emerged malware samples are used for test, while the old data are used in training.

Finally, we also consider the scenarios of concept drift and data imbalance in Subsection 6.5. In the scenario of concept drift, 17685 samples (8,017 benign examples and 9,668 malicious examples) from Androzoo are grouped by production year (from 2016 to 2020) and used to train target models. In the scenario of data imbalance, we randomly disarrange samples and set the benign-malicious ratio to 10:1, following the experimental setting in [2].

Target Model. We choose three SOTA malware detection methods (i.e., MaMadroid [41], APIGraph [67] and GCN [65]) as our target system. In MaMadroid and APIGraph, we employ Random Forest (RF) [7], AdaBoost (AB) [11], 1-Nearest Neighbor (1-NN) [19], 3-Nearest Neighbor (3-NN) and Dense Neural Network (DNN) as the target classifier, respectively. Similar to [65], we use a two-layer DNN as the target classifier in the GCN-based method.

Metric. We use attack success rate (**ASR**), average perturbation ratio (**APR**), and the number of interaction rounds (**IR**) to evaluate BagAmmo. ASR corresponds to the ratio of the number of successfully generated AEs (denoted by $N_{success}$) to the number of malicious examples used for AE generation (denoted by N_{total}), i.e., $ASR = N_{success}/N_{total}$. APR is the ratio of the number of added edges (denoted by E_{added}) to the total number of edges (denoted by E_{total}), i.e., $APR = E_{added}/E_{total}$. IR is defined as the number of interactions between our attack model and the target model.

[7] https://androzoo.uni.lu/

6.1 RQ1: Effectiveness

Experimental Setup. To verify the attack effectiveness of BagAmmo, we use BagAmmo to attack the 32 target models[8] mentioned above, and calculate ASR, APR and IR on every target model.

Furthermore, we also compare BagAmmo with three attack methods, i.e., SRL [65], SRL_N and Random Insertion (RI). To our knowledge, SRL is the SOTA malware AE generation method [9]. Since SRL requires knowing the class probabilities outputted by the target model, we modify its reward function and create a variant of SRL (i.e., SRL_N) that only relies on binary outputs. The RI attack method is also introduced from [65], and it randomly inserts non-functional functions.

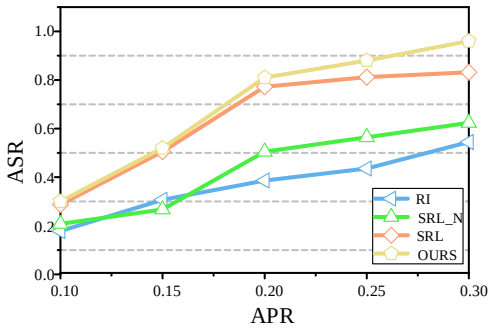

Figure 8: Comparison with SOTA methods.

Results & Analyses. Table 1 reflects the attack performance of BagAmmo on MaMaDroid, APIGraph and GCN under various feature granularities. First, BagAmmo achieves an average ASR of 99.9% over 32 target models, hence confirming the effectiveness of BagAmmo. Second, when attacking the family-granularity classifier, BagAmmo achieves the lowest APR and IR (i.e., 0.071 and 10.936). This indicates that although the family-granularity FCG speeds up malware detection through reducing input complexity, it still improves the efficiency of BagAmmo by reducing search space.

Fig. 8 compares BagAmmo, SRL, SRL_N and RI with respect to ASR under various APRs. Not surprisingly, RI performs worst in our experiments due to its poor search strategy. SRL performs better than SRL_N, because SRL has access to class probabilities, which is more valuable than binary decisions. It is worth noting that BagAmmo still outperforms SRL (e.g., its ASR is 4% higher when APR is 0.2), although BagAmmo cannot utilize class probabilities. The above results confirm that, under a certain number of perturbations,

[8] Our experiments use 2 traditional FCG-based feature extraction methods (MaMaDroid and APIgraph), 3 feature levels (class, family and package) and 5 target classifiers (RF, AB, etc.). Furthermore, 1 GCN feature extraction method is considered with 2 feature levels (family and package). Hence, there are $32 = 2 \times 3 \times 5 + 1 \times 2$ classifiers.

[9] Note that SRL works on control flow graph instead of FCG. To apply SRL to the FCG based Android malware detection, we design a non-functional API list (instead of non-functional instruction list), which contains 17 non-functional APIs.

Table 1: Effectiveness of BagAmmo towards MaMaDroid, APIGraph and GCN.

Classifier\Level		Family			Package			Class		
		ASR	APR	IR	ASR	APR	IR	ASR	APR	IR
MaMaDroid	RF	1.000	0.021	8.670	1.000	0.049	13.640	1.000	0.083	12.490
	DNN	0.990	0.149	11.130	1.000	0.134	16.730	1.000	0.153	15.907
	AB	1.000	0.066	10.270	1.000	0.072	14.300	1.000	0.118	15.460
	1-NN	1.000	0.031	7.000	1.000	0.109	11.630	1.000	0.060	10.960
	3-NN	1.000	0.037	9.390	1.000	0.142	13.380	1.000	0.072	10.770
APIGraph	RF	1.000	0.039	11.260	1.000	0.098	14.930	1.000	0.040	9.530
	DNN	1.000	0.132	14.370	1.000	0.096	18.630	1.000	0.168	12.566
	AB	1.000	0.093	14.510	0.990	0.131	18.350	1.000	0.067	12.250
	1-NN	1.000	0.058	11.190	1.000	0.089	14.040	1.000	0.012	6.910
	3-NN	1.000	0.085	11.570	1.000	0.105	13.770	1.000	0.019	7.780
GCN	DNN	1.000	0.205	11.610	1.000	0.104	17.320	-	-	-

our method generates a better combination of the added edges that are more deceptive to detectors, as compared to the other methods.

6.2 RQ2: Evolution

Experimental Setup. In this subsection, we use experiments to analyze the effects of multi-population co-evolution mechanism. First, we want to show that this mechanism can overcome the challenge of unknown feature granularity. To this end, we compare our method with the single-population methods in attacking MaMaDroid. The single-population methods rely on one single population corresponding to class, package and family level, denoted by BagAmmo-C, BagAmmo-P and BagAmmo-F, respectively. We also randomly select a malware sample and take a close look at these methods' attack processes.

Second, we want to know whether the correct feature granularity is found by our method. We then record the survival number of each population and analyze how these populations evolve. In this experiment, we choose MaMaDroid with an RF classifier as our target model, and use family-level feature granularity in malware detection.

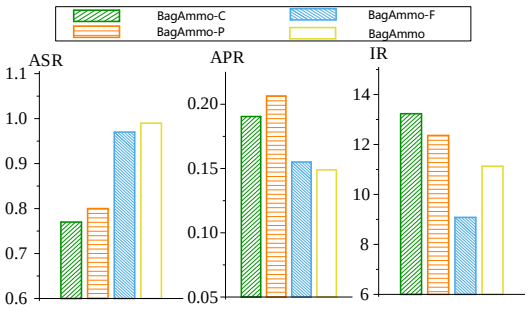

Figure 9: Performance comparison with single-population.

Results & Analyses. For comparison, we choose family-level feature granularity and evaluate BagAmmo and single-population methods on **all** test samples. The results are shown in Fig. 9. It can be seen that BagAmmo performs best and achieves the highest ASR with the lowest APR. BagAmmo-C and BagAmmo-P perform worst since they use a false feature granularity. Surprisingly, BagAmmo performs better than BagAmmo-F (i.e., 2% higher in ASR and 0.06 lower in APR). This is because the introduction of multiple populations helps to avoid premature convergence and approach a global optimum. However, it may result in more interactions with the target model. This accounts for why BagAmmo has a higher IR than BagAmmo-F.

Now we randomly select a malware sample, and use it to generate an AE to attack 5 classifiers under family-level feature granularity. The attack processes of all methods are depicted in Fig. 10. In this figure, the vertical axis represents the perturbation ratio of all methods, and the horizontal axis shows the IR values. If a curve exhibits an evident decreasing trend and falls below a low threshold, we can conclude that the corresponding method succeeds in generating an AE and defeating the target model. As for those curves keeping horizontal (e.g., the green curve in the first subfigure), the corresponding methods fail to generate AEs. Fig.10 shows that the perturbation ratio of multi-population always has a satisfactory decreasing trend, hence confirming the effects of multi-population co-evolution. Furthermore, using a single population may cause premature convergence to a local optimum, as indicated by Fig. 10-(1). However, BagAmmo effectively mitigates this problem using multiple populations. Theoretical analyses are given in Appendix 10.2.

Finally, we verify the multi-population co-evolution method converges to the real feature granularity from a different perspective. We show the survival proportion (i.e., the ratio between the number of alive individuals and the total number of individuals) of different populations in Fig. 11. At the beginning, the perturbations are randomly added, and the

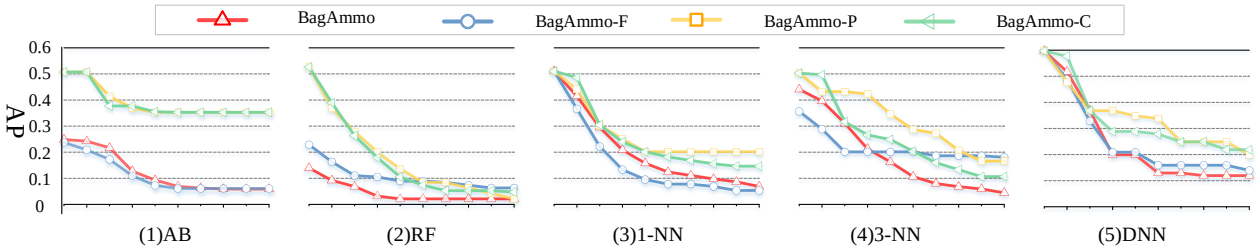

Figure 10: Multi-population vs. single-population.

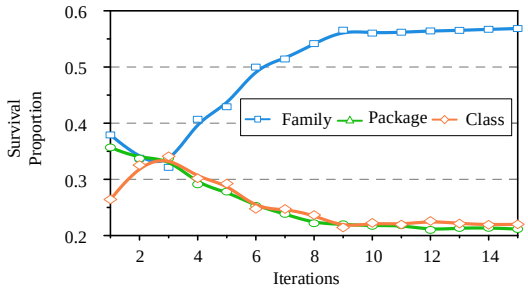

Figure 11: The changing trend of survival proportion.

survival proportion of different populations are irregular. However, as the number of queries increases, the family population and class population gradually fall to a low level. Contrarily, the survival proportion of the population corresponding to the correct feature granularity (i.e., family level) gradually rises to a high level. This phenomenon also confirms the effects of multi-population co-evolution.

6.3 RQ3: Efficiency

Experimental Setup. We conduct *ablation studies* to verify the effects of the substitute model in decreasing queries and improving attack efficiency. For comparison, we remove the substitute model and guide the multi-population co-evolution algorithm only using the target model. This method is called BagAmmo-Without-S. Then we use BagAmmo and BagAmmo-Without-S to manipulate the same APK file, and compare their performance.

Results & Analyses. Substitute model's effects are shown in Fig. 12, where the solid and the dotted lines represent BagAmmo and BagAmmo-Without-S, respectively. The vertical axis reflects perturbation ratio, and the horizontal axis indicates the number of queries. It can be seen that in all cases, BagAmmo always has a higher convergence speed. Moreover, BagAmmo always requires fewer queries before the perturbation ratio is kept below a certain threshold (e.g., 0.1). Note that the difference between two methods in the initial phase is relatively small. It is because the substitute model has not been well trained in this phase. However, after the substitute model is well trained with sufficient data[10], BagAmmo performs more efficiently and exhibits its advantage.

Fig. 13 compares BagAmmo and BagAmmo-Without-S in terms of IR. Its top-half part gives the results on the family-level FCG based MaMadroid, while the bottom-half part shows the results on the package-level FCG based MaMadroid. The horizontal axis indicates various classifiers (e.g., AB, RF and 1-NN). We can draw two conclusions from this figure. First, the package-level classifier is more difficult to attack. This is because package-level FCGs contain much more nodes than family-level FCGs, resulting in a larger search space for BagAmmo. Second, using the substitute model reduces the number of queries in almost all cases and helps enhance the attack efficiency.

6.4 RQ4: Manipulation Overhead

Experimental Setup. Here we study the number of code modifications (i.e., manipulation overhead) required to generate a real evasive malware. We use experimental results to reflect the relationship between ASR and the allowed perturbation ratio.

Result & Analysis. Experimental results are shown in Fig. 14. In this figure, the horizontal axis represents the allowed perturbation ratio, and the vertical axis gives the cumulative distribution function (CDF) of ASR. It can be observed that the ASR keeps rising with the increase of the allowed perturbation ratio. In practice, a larger perturbation ratio results in larger computational overhead for adversaries. Therefore, there exists a trade-off between manipulation overhead and attack success ratio. Moreover, the 3-NN classifier is more robust than the 1-NN classifier. It is because the 3-NN classifier considers more data than 1-NN classifier when classifying a sample, which makes it distinguish benign and malicious apps easier.

[10]In general, the training accuracy of the substitute model arises as the number of iteration rounds increase. However, the increasing trend of training accuracy is not strictly monotonic, because the training data used in the iterations are different.

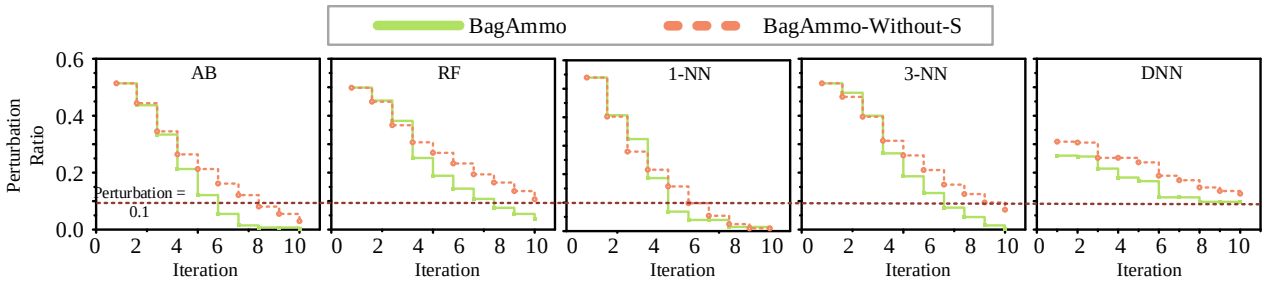

Figure 12: With substitute model vs. without substitute model.

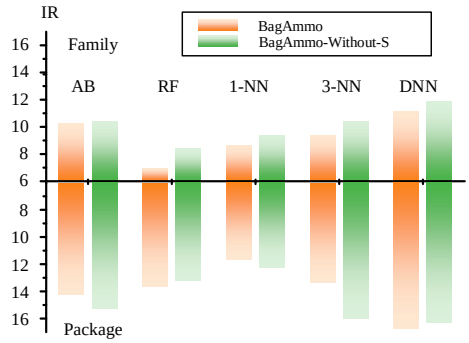

Figure 13: Can substitute model reduce queries?

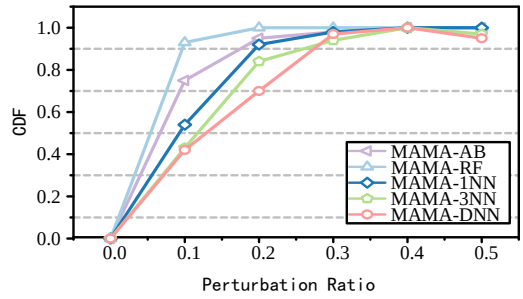

Figure 14: CDF of ASR under different perturbation ratios.

6.5 RQ5: Resilience

Experimental Setup. Concept drift [63] is often observed in the realistic applications of Android malware detection. Concept drift undermines existing AE generation methods that insert APIs selected from a pre-determined white list, if the white list is not updated accordingly. Hence we want to know whether BagAmmo is also susceptible to concept drift. To this end, we use newly-emerged malware samples to generate AEs and attack the classifiers trained over old data. We divide the dataset into training sets and testing sets according to the years that Android app emerge. We construct four new datasets to evaluate BagAmmo under concept drift, as depicted in Table 2. The first row of Table 2 is the year of training samples used for training target classifiers. The second row is the year of testing samples used for generating AEs. The third row is the accuracy of the classifier.

Table 2: The attack performance under concept drift.

Training set (year)	2016	2016-2017	2016-2018	2016-2019
Testing set (year)	2017	2018	2019	2020
ACC	92.92%	94.08%	94.58%	95.47%
ASR	100%	100%	100%	100%

Table 3: The APR on balanced and imbalanced data.

Level	Case	AB	RF	1NN	3NN	Average
Family	Balance	0.066	0.021	0.031	0.037	0.039
	Imbalance	0.050	0.024	0.021	0.021	0.029
Package	Balance	0.072	0.049	0.109	0.142	0.093
	Imbalance	0.041	0.040	0.094	0.105	0.070

Data imbalance is another practical problem worthy of consideration [2, 16]. Since malicious samples are more difficult to collect than benign samples, malware detection models are usually trained over imbalanced data. We want to know whether data imbalance negatively impacts the attack performance of BagAmmo. Hence we evaluate BagAmmo on the target model trained with imbalanced data (the benign-malicious ratio is 10:1).

Result & Analysis. In the experiments on concept drift, we use BagAmmo to manipulate the test samples, and the results are used to attack the MaMaDroid model with the family-level feature and the classifier of RF. The ASR of BagAmmo in every scenario is presented in the last row of Table 2. First, this table indicates that with more training samples, the accuracy of the target classifier becomes higher. No matter how high the accuracy is, however, BagAmmo always achieves a perfect ASR of 100%. This shows that BagAmmo performs well under concept drift and is still efficient when the malware detection models learn more with the new data. Note that BagAmmo uses the functions coming from the malware itself (instead of a static function set). BagAmmo reduces the risk of using functions that become outdated due to concept drift. As a result, BagAmmo poses a persistent threat to malware detectors. Finally, we also discuss how BagAmmo performs

when the defender has the knowledge of adversarial example in Appendix 10.5

Table 3 shows the experimental results of BagAmmo in the cases of balanced and imbalanced data. Our experiments demonstrate that the DNN model performs very poorly when trained with imbalanced data. Therefore, we do not choose DNN as our target model. In both cases (i.e., balanced dataset and imbalance dataset), BagAmmo achieves an attack success rate (i.e., ASR) of 100%. Here we only show the values of APR in Table 3. A higher APR means a more difficult attack task. It can be seen that in the vast majority of cases, BagAmmo needs fewer perturbations (i.e., has a lower APR) to attack the target model trained with imbalanced data. That is, data imbalance does not bring troubles to BagAmmo. This is because that training with imbalanced data makes the target model more likely to classify malware as benign apps. Accordingly, this reduces the degree of difficulty in generating AEs.

6.6 RQ6: Functionality

Experimental Setup. In this section, we first use static analysis to verify whether the perturbations generated by BagAmmo are successfully imposed on malware. We then employ dynamic analysis to check whether the perturbation changes the functionality of the malware.

Result & Analysis. To know whether our perturbations are injected, we add a unique log statement when a perturbation (i.e., a try-catch trap) is injected. This log statement helps us to find the perturbation in the smali file. We then check if the found function calls in the smali file coincide with the perturbations generated by BagAmmo. In our experiments, we evaluate 50 APK files, and we realize that all the generated perturbations are correctly injected into the smali file.

In our experiments of dynamic analysis, we first install and run 50 pairs of original and perturbed malware samples in Android Virtual Device (AVD). It is observed that every malware pair performs the same and has the same run-time UI. For further analysis, we insert three log statements, denoted by LOG1, LOG2 and LOG3, into every try-catch block to record execution information. LOG1 is in front of the run-time exception, LOG2 is in front of the inserted function, and LOG3 is at the beginning of the catch block. We analyze 50 APK files aided by Android Studio's log analysis tool (i.e., LogCat). We realize that either LOG1 or LOG3 of every APK file is normally executed, but no LOG2 is executed. This phenomenon means that all manipulated malware samples run properly, and the inserted functions are not invoked, hence posing no impact on the malware functionality.

7 Related Work

Recently, adversarial attacks have been widely used in various fields, i.e., image classification [60, 61], traffic analysis [43, 46], autonomous driving [28, 29] and object detection [39]. As for Android malware detection, there have been many studies [21, 24, 27, 33, 34] on syntax features oriented AE generation. Huang *et al.* [27] use the saddle-point optimization formulation to generate adversarial examples in the discrete (e.g., binary) domain for malware detection. Grosse *et al.* [21] expand existing AE generation algorithms to construct a highly effective attack against malware detection models. In [24, 34], Hu *et al.* utilize a GAN to generate adversarial examples in black-box mode for malware detection. Li *et al.* [33] propose an ensemble approach that allows attackers to perturb a malware example via multiple attack methods and multiple manipulation sets.

To achieve higher detection accuracy, more and more Android malware detection methods [41, 58, 67] focus on semantic features. Chen *et al.* [10] introduce two AE generations methods in image classification to detect Android malware, and propose a method applying optimal perturbations onto Android APKs. Their method directly perturbs features in feature space. Pierazzi *et al.* [45] extract slices of bytecode (i.e., gadgets) from benign APKs and inject them into a malicious APK to generate adversarial malware. Zhang *et al.* [65] propose a reinforcement learning based attack to deceive graph feature based malware detection models. Recently, Bostani *et al.* [6] propose an interesting black-box attack EvadeDroid without requiring the knowledge about feature space. Different from BagAmmo, EvadeDroid employs random search to find the desired perturbation from the code of benign apps.

8 Limitations and Discussion

In this paper, we propose a black-box AE attack BagAmmo towards the FCG based Android malware detection. We hope that our work has reference value for the study of Android malware detection, and raises the concern for the threats posed by AE attacks. Moreover, our method can be used to evaluate the robustness of existing Android malware detection methods. Below we discuss some limitations and future works.

Dynamic analysis based defense. Our method targets the static analyse methods. It relies on inserting function calls to change FCG. But it does not change the information flow of malware. Therefore, it does not negatively impact dynamic analysis [31]. We will explore how to construct adversarial examples against dynamic analysis based Android malware detection methods in future work.

Transfer to other domains. The idea and the framework of BagAmmo are transferable to a certain degree, since many domains use semantic features and graph structured data (e.g., intrusion detection system [69] and trajectory prediction system [56, 59]).

Try/catch detection based defense. Another concern is that whether a defender can detect the AEs generated by BagAmmo by counting the number of try/catch blocks. This defense method requires a detection threshold for the number

of try-catch blocks. Through comparing the try-catch block number of original malicious APKs and that of adversarially perturbed APKs, however, we find that the number of try-catch blocks added by our method is relatively small. So it is difficult to find an appropriate threshold for all APKs. Without such a threshold, this defense method may cause a high false positive or false negative rate.[11]

9 Acknowledgments

This work was supported partially by the Hong Kong RGC Project (No. PolyU15219319), HKPolyU Grant No.ZVG0, Fundamental Research Funds for the Central Universities (HUST: Grant No. YCJJ202202016 and 2022JYCXJJ035) .

References

[1] Maksym Andriushchenko, Francesco Croce, Nicolas Flammarion, and Matthias Hein. Square attack: A query-efficient black-box adversarial attack via random search. In *Proc. ECCV*, 2020.

[2] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. Dos and don'ts of machine learning in computer security. In *Proc. Security*, 2022.

[3] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. DREBIN: effective and explainable detection of android malware in your pocket. In *Proc. NDSS*, 2014.

[4] Yude Bai, Zhenchang Xing, Xiaohong Li, Zhiyong Feng, and Duoyuan Ma. Unsuccessful story about few shot malware family classification and siamese network to the rescue. In *Proc. ICSE*, 2020.

[5] Aleksandar Bojchevski and Stephan Günnemann. Adversarial attacks on node embeddings via graph poisoning. In *Proc. ICML*, 2019.

[6] Hamid Bostani and Veelasha Moonsamy. Evadedroid: A practical evasion attack on machine learning for black-box android malware detection. *CoRR*, abs/2110.03301, 2021.

[7] Breiman. Random forests. *MACH LEARN*, 2001,45(1)(-):5–32, 2001.

[8] Minghui Cai, Yuan Jiang, Cuiying Gao, Heng Li, and Wei Yuan. Learning features from enhanced function call graphs for android malware detection. *Neurocomputing*, 423:301–307, 2021.

[11] More experimental results can be found in Appendix 10.7

[9] Nicholas Carlini and David A. Wagner. Towards evaluating the robustness of neural networks. In *Proc. S&P*, 2017.

[10] Xiao Chen, Chaoran Li, Derui Wang, Sheng Wen, Jun Zhang, Surya Nepal, Yang Xiang, and Kui Ren. Android hiv: A study of repackaging malware for evading machine-learning detection. *IEEE Trans. Inf. Forensics Secur.*, 15:987–1001, 2020.

[11] M. Collins, R. E. Schapire, and Y. Singer. Logistic regression, adaboost and bregman distances. *Machine Learning*, 48(1/2/3):253–285, 2002.

[12] Hanjun Dai, Hui Li, Tian Tian, Xin Huang, Lin Wang, Jun Zhu, and Le Song. Adversarial attack on graph structured data. In *Proc. ICML*, 2018.

[13] Ambra Demontis, Marco Melis, Battista Biggio, Davide Maiorca, Daniel Arp, Konrad Rieck, Igino Corona, Giorgio Giacinto, and Fabio Roli. Yes, machine learning can be more secure! A case study on android malware detection. *IEEE Trans. Dependable Secur. Comput.*, 16(4):711–724, 2019.

[14] William Enck, Machigar Ongtang, and Patrick D. McDaniel. On lightweight mobile phone application certification. In *Proc. CCS*, pages 235–245, 2009.

[15] Ming Fan, Jun Liu, Xiapu Luo, Kai Chen, Zhenzhou Tian, Qinghua Zheng, and Ting Liu. Android malware familial classification and representative sample selection via frequent subgraph analysis. *IEEE Trans. Inf. Forensics Secur.*, 13(8):1890–1905, 2018.

[16] Feargus Pendlebury and Fabio Pierazzi and Roberto Jordaney and Johannes Kinder and Lorenzo Cavallaro. TESSERACT: Eliminating Experimental Bias in Malware Classification across Space and Time. In *Proc. USENIX Security*, 2019.

[17] Yu Feng, Osbert Bastani, Ruben Martins, Isil Dillig, and Saswat Anand. Automated synthesis of semantic malware signatures using maximum satisfiability. In *Proc. NDSS*, 2017.

[18] Hossein Fereidooni, Mauro Conti, Danfeng Yao, and Alessandro Sperduti. ANASTASIA: android malware detection using static analysis of applications. In *Proc. NTMS*, 2016.

[19] Evelyn Fix and J. L Hodges, Jr. Discriminatory analysis - nonparametric discrimination: Small sample performance. 1952.

[20] Gata. Mobile malware report - no let-up with android malware. https://www.gdatasoftware.com, 2019. Accessed April 4, 2010.

[21] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, M Backes, and Patrick Mcdaniel. Adversarial examples for malware detection. In *Proc. ESORICS*, 2017.

[22] guardsquare. The industry-leading java optimizer for android apps. https://www.guardsquare.com/proguard.

[23] Shifu Hou, Yanfang Ye, Yangqiu Song, and Melih Abdulhayoglu. Hindroid: An intelligent android malware detection system based on structured heterogeneous information network. In *Proc. SIGKDD*, 2017.

[24] Weiwei Hu and Ying Tan. Generating adversarial malware examples for black-box attacks based on gan. *arXiv: 1702.05983*, 2017.

[25] Wenjun Hu, Jing Tao, Xiaobo Ma, Wenyu Zhou, Shuang Zhao, and Ting Han. Migdroid: Detecting app-repackaging android malware via method invocation graph. In *Proc. ICCCN*, 2014.

[26] Zichao Hu, Heng Li, Liheng Yuan, Zhang Cheng, Wei Yuan, and Ming Zhu. Model scheduling and sample selection for ensemble adversarial example attacks. *Pattern Recognition*, 130:108824, 2022.

[27] Alex Huang, Abdullah Aldujaili, Erik Hemberg, and Unamay Oreilly. Adversarial deep learning for robust detection of binary encoded malware. In *Proc. IEEE S&P Workshops*, 2018.

[28] Wei Jia, Zhaojun Lu, Haichun Zhang, Zhenglin Liu, Jie Wang, and Gang Qu. Fooling the eyes of autonomous vehicles: Robust physical adversarial examples against traffic sign recognition systems. *CoRR*, abs/2201.06192, 2022.

[29] Pengfei Jing, Qiyi Tang, Yuefeng Du, Lei Xue, Xiapu Luo, Ting Wang, Sen Nie, and Shi Wu. Too good to be safe: Tricking lane detection in autonomous driving with crafted perturbations. In *Proc. USENIX Security*, 2021.

[30] Hyung-Jong Kim and Hae Young Lee. A study on the privacy protection layer for android iot services (lightning talk). In *Proc. ICSSA*, 2018.

[31] X. Lei, Y. Zhou, T. Chen, X. Luo, and G. Gu. Malton: Towards on-device non-invasive mobile malware analysis for art. In *Proc. USENIX Security*, 2017.

[32] Y. Leung and G. Yong. Degree of population diversity - a perspective on premature convergence in genetic algorithms and its markov chain analysis. *IEEE Transactions on Neural Networks*, 8(5):1165–1176, 1997.

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

[34] Heng Li, ShiYao Zhou, Wei Yuan, and Henry Leung. Adversarial-example attacks towards android malware detection system. *IEEE Systems Journal*, 2019.

[35] Jin Li, Lichao Sun, Qiben Yan, Zhiqiang Li, Witawas Srisa-an, and Heng Ye. Significant permission identification for machine-learning-based android malware detection. *IEEE Trans. Ind. Informatics*, 14(7):3216–3225, 2018.

[36] Pengcheng Li, Jinfeng Yi, and Lijun Zhang. Query-efficient black-box attack by active learning. In *Proc. ICDM*, 2018.

[37] Xing Liu, Jiqiang Liu, Sencun Zhu, Wei Wang, and Xiangliang Zhang. Privacy risk analysis and mitigation of analytics libraries in the android ecosystem. *IEEE Trans. Mob. Comput.*, 19(5):1184–1199, 2020.

[38] Yanpei Liu, Xinyun Chen, Chang Liu, and Dawn Song. Delving into transferable adversarial examples and black-box attacks. In *Proc. ICLR*, 2017.

[39] Giulio Lovisotto, Henry Turner, Ivo Sluganovic, Martin Strohmeier, and Ivan Martinovic. SLAP: improving physical adversarial examples with short-lived adversarial perturbations. In *Proc. USENIX Security*, 2021.

[40] Yao Ma, Suhang Wang, Tyler Derr, Lingfei Wu, and Jiliang Tang. Graph adversarial attack via rewiring. In *Proc. KDD*, 2021.

[41] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon J. Ross, and Gianluca Stringhini. Mamadroid: Detecting android malware by building markov chains of behavioral models. In *Proc. NDSS*, 2017.

[42] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *Proc. ACSAC*, 2007.

[43] Milad Nasr, Alireza Bahramali, and Amir Houmansadr. Defeating dnn-based traffic analysis systems in real-time with blind adversarial perturbations. In *Proc. USENIX Security*, 2021.

[44] Damien Octeau, Patrick D. McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in android: An essential step towards holistic security analysis. In *Proc. USENIX Security*, 2013.

[45] Fabio Pierazzi, Feargus Pendlebury, Jacopo Cortellazzi, and Lorenzo Cavallaro. Intriguing properties of adversarial ML attacks in the problem space. In *Proc. S&P*, 2020.

[46] Mohammad Saidur Rahman, Mohsen Imani, Nate Mathews, and Matthew Wright. Mockingbird: Defending against deep-learning-based website fingerprinting attacks with adversarial traces. *IEEE Trans. Inf. Forensics Secur.*, 16:1594–1609, 2021.

[47] Seung-Hyun Seo, Aditi Gupta, Asmaa Mohamed Sallam, Elisa Bertino, and Kangbin Yim. Detecting mobile malware threats to homeland security through static analysis. *J. Netw. Comput. Appl.*, 38:43–53, 2014.

[48] Yiwei Sun, Suhang Wang, Xianfeng Tang, Tsung-Yu Hsieh, and Vasant G. Honavar. Adversarial attacks on graph neural networks via node injections: A hierarchical reinforcement learning approach. In *Proc. WWW*, 2020.

[49] Zhichuang Sun, Ruimin Sun, Long Lu, and Alan Mislove. Mind your weight(s): A large-scale study on insufficient machine learning model protection in mobile apps. In *Proc. USENIX Security*, 2021.

[50] Fnu Suya, Jianfeng Chi, David Evans, and Yuan Tian. Hybrid batch attacks: Finding black-box adversarial examples with limited queries. In *Proc. USENIX Security*, 2020.

[51] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In *Proc. ICLR*, 2014.

[52] Thomas Tanay and Lewis D. Griffin. A boundary tilting persepective on the phenomenon of adversarial examples. *CoRR*, abs/1608.07690, 2016.

[53] Anita Thengade and Rucha Dondal. Genetic algorithm – survey paper. *Foundation of Computer Science (FCS)*, 2012.

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

[55] Wei Wang, Xing Wang, Dawei Feng, Jiqiang Liu, Zhen Han, and Xiangliang Zhang. Exploring permission-induced risk in android applications for malicious application detection. *IEEE Trans. Inf. Forensics Secur.*, 9(11):1869–1882, 2014.

[56] Conghao Wong, Beihao Xia, Ziming Hong, Qinmu Peng, Wei Yuan, Qiong Cao, Yibo Yang, and Xinge You. View vertically: A hierarchical network for trajectory prediction via fourier spectrums. In *Proc. ECCV*, 2022.

[57] Huijun Wu, Chen Wang, Yuriy Tyshetskiy, Andrew Docherty, Kai Lu, and Liming Zhu. Adversarial examples for graph data: Deep insights into attack and defense. In *Proc. IJCAI*, 2019.

[58] Yueming Wu, Xiaodi Li, Deqing Zou, Wei Yang, Xin Zhang, and Hai Jin. Malscan: Fast market-wide mobile malware scanning by social-network centrality analysis. In *Proc. ASE*, pages 139–150, 2019.

[59] Beihao Xia, Conghao Wong, Qinmu Peng, Wei Yuan, and Xinge You. Cscnet: Contextual semantic consistency network for trajectory prediction in crowded spaces. *Pattern Recognition*, 126:108552, 2022.

[60] Pengfei Xia, Ziqiang Li, Wei Zhang, and Bin Li. Data-efficient backdoor attacks. In *Proc. IJCAI*, 2022.

[61] Pengfei Xia, Hongjing Niu, Ziqiang Li, and Bin Li. Enhancing backdoor attacks with multi-level mmd regularization. *IEEE Trans. Dependable Secur. Comput.*, 2022.

[62] Kaidi Xu, Hongge Chen, Sijia Liu, Pin-Yu Chen, Tsui-Wei Weng, Mingyi Hong, and Xue Lin. Topology attack and defense for graph neural networks: An optimization perspective. In *Proc. IJCAI*, 2019.

[63] Limin Yang, Wenbo Guo, Qingying Hao, Arridhana Ciptadi, Ali Ahmadzadeh, Xinyu Xing, and Gang Wang. CADE: detecting and explaining concept drift samples for security applications. In *Proc. USENIX Security*, 2021.

[64] Wei Yuan, Yuan Jiang, Heng Li, and Minghui Cai. A lightweight on-device detection method for android malware. *IEEE Trans. Syst. Man Cybern. Syst.*, 51(9):5600–5611, 2021.

[65] Lan Zhang, Peng Liu, Yoonho Choi, and Ping Chen. Semantics-preserving reinforcement learning attack against graph neural networks for malware detection. *IEEE Trans Dependable Secure Comput*, pages 1–1, 2022.

[66] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. Semantics-aware android malware classification using weighted contextual API dependency graphs. In *Proc. SIGSAC*, 2014.

[67] Xiaohan Zhang, Yuan Zhang, Ming Zhong, Daizong Ding, Yinzhi Cao, Yukun Zhang, Mi Zhang, and Min Yang. Enhancing state-of-the-art classifiers with API semantics to detect evolved android malware. In *Proc. CCS*, 2020.

[68] Zhengli Zhao, Dheeru Dua, and Sameer Singh. Generating natural adversarial examples. In *Proc. ICLR*, 2018.

[69] Xiaokang Zhou, Wei Liang, Weimin Li, Ke Yan, Shohei Shimizu, and Kevin I-Kai Wang. Hierarchical adversarial attacks against graph-neural-network-based iot network intrusion detection system. *IEEE Internet Things J.*, 9(12):9310–9319, 2022.

[70] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *Proc. NDSS*, 2012.

[71] Daniel Zügner and Stephan Günnemann. Adversarial attacks on graph neural networks via meta learning. In *Proc. ICLR*, 2019.

10 Appendix

10.1 The limitations in callee selection

As shown in Section 4.2, we choose the leaf nodes as candidate callees. However, not all leaf nodes can be chosen as callees. There exist two limitations:

- *Access modifier*. Some leaf-node functions are not allowed to be invoked at all. Therefore, we only consider those leaf-node functions whose access modifier is *public*.
- *Parameter type*. The arguments of some leaf-node functions are the instances of classes. Under this situation, invoking these functions will incur instantiating a class, hence generating an unintended edge. To avoid this problem, we propose to choose the leaf-node functions whose arguments are void or belong to the category of primitive data types (e.g., int and short) and the String class.

10.2 Theoretical Analyses for our method

Our method BagAmmo utilizes the algorithm Apoem to find the desired perturbation for a given malware sample. Since Apoem is an evolutionary algorithm, how to mitigate premature convergence is an important issue. Here, premature convergence or prematurity is a common phenomenon that leads an evolutionary algorithm to converge quickly to a local optimum. For evolutionary algorithms, prematurity is often caused by the lack of gene diversity.

In the following, we analyze how multiple populations introduced in Apoem mitigate the problem of premature convergence.

Due to the introduction of multiple populations, there exist a local optimal solution in each population. We define this locally optimal solution as x_p^*, where $p = 1, 2, ..., l$ is the index of the population.

Then, the individuals that can achieve the local optimal solutions with the Apoem G are termed as:

$$A_p^* = \left\{ x \in A : G(x) = x_p^* \right\} \tag{3}$$

where A is the solution space.

Then, the probability that an individual $x \in A$ belongs to set A_p^* can be represented as $\theta_p = P\left(A_p^*\right)$. It is clear that $\theta_p > 0$ for $p = 1, \ldots, l$ and $\sum_{p=1}^{l} \theta_p = 1$.

The size of the set A_p^* can be termed as n_p. According to the definition, we have $n_p \geq 0$ $(p = 1, \ldots, l)$, the random vector $(N_1, \ldots, N_l)$ follows the multinomial distribution and $\sum_{p=1}^{l} N_p = N$.

$$\Pr\{n_1 = N_1, \ldots, n_l = N_l\} = \binom{N}{N_1, \ldots, N_l} \theta_1^{N_1} \ldots \theta_l^{N_l} \tag{4}$$

where

$$\binom{N}{N_1, \ldots, N_l} = \frac{N!}{N_1! \ldots N_l!}, \quad N_p \geq 0 \quad (p = 1, \ldots, l) \tag{5}$$

We define W as the number of locally optimal solutions found by Apoem. Then the probability of l locally optimal solutions being found can be termed as

$$\Pr\{W = l \mid \theta\} = \sum_{N_1 + \cdots + N_l = N} \binom{N}{N_1, \ldots, N_l} \theta_1^{N_1} \ldots \theta_l^{N_l} \tag{6}$$

where

$$\theta = (\theta_1, \ldots, \theta_l). \tag{7}$$

For the sake of analyzing the limit, we define

$$\delta = \min\{\theta_1, \ldots, \theta_l\} \leq 1/l \tag{8}$$

Then we have

$$\begin{aligned} \Pr\{W = l \mid \theta\} &\geq \sum_{N_1 + \ldots + N_l = N} \binom{N}{N_1, \ldots, N_l} \delta^N \\ &= (\delta l)^N \Pr\left\{ W = l \mid \left(\frac{1}{l}, \ldots, \frac{1}{l} \right) \right\} \end{aligned} \tag{9}$$

For any l and θ, we can find the least evaluation number n^* such that for any given $\gamma \in (0,1)$, we will have $\Pr\{W = l \mid \theta\} \geq \gamma$ for all $n \geq n^*$. Finding $n^* = n^*(\gamma, \theta)$ is the problem of finding the (minimal) number of points in A such that the probability that all local minimizers will be found is at least γ.

We analyze the extreme cases that $\theta^* = \left(l^{-1}, \ldots, l^{-1}\right)$. Hence the problem of finding $n^*(\gamma, \theta)$ is reduced to that of finding $n^*(\gamma, \theta^*)$. For a large N, $n^*(\gamma, \theta)$ can be approximated as

$$\begin{aligned} \Pr\{W = l \mid \theta^*\} &= l^{-N} \sum_{N_1 + \cdots + N_l = N} \binom{N}{N_1, \ldots, N_l} \\ &= \sum_{p=0}^{l} (-1)^p \binom{l}{p} (1 - p/l)^N \\ &\sim \exp\{-l \exp\{-N/l\}\}, \quad N \to \infty \end{aligned} \tag{10}$$

By solving the equation $\exp(-l\exp(-N/l)) = \gamma$ with respect to N, we obtain the approximation

$$n^*(\gamma, \theta^*) \simeq l\ln l + l\ln(-\ln\gamma) \tag{11}$$

With Eq. (11), we analyze the relationship between the number of required queries and the population number as follows. We can see that multiple populations (i.e., $l > 1$) help to slow down the convergence rate of the algorithm. As we all know, prematurity is a common phenomenon in which an evolutionary algorithm early converges to a poor local optimum. However, Apoem begins its search in multi start l which makes the algorithm can find a better solution with a higher probability. Our algorithm effectively relieves this problem by introducing multiple populations, and prevents the algorithm from wasting many efforts on repeatedly finding the same local optimum.

10.3 Implementation details and an instance of the smali code

invoke-direct

```
new-instance p2, Ljava/lang/StringBuilder;
invoke-direct {p2}, Ljava/lang/StringBuilder;-><init>()V
```

invoke-virtual

```
new-instance p2, Ljava/lang/StringBuilder;
invoke-direct {p2},
    Ljava/lang/StringBuilder;-><init>()V
invoke-virtual {p2, p0},
    Ljava/lang/StringBuilder;->append(I;)Ljava/lang/StringBuilder;
```

invoke-static

```
invoke-static {p2},
    Ljava/lang/Class;->forName(Ljava/lang/String;)Ljava/lang/Class;
```

invoke-super

```
invoke-super {p0},
    Ljava/lang/Object;->toString()Ljava/lang/String;
```

invoke-interface

```
invoke-interface {p0},
    Ljava/lang/Runnable;->run()V
```

Figure 15: The examples of smali code with different invocation types.

In this section, we first provide the implementation details of the transformation from the generator's output to the perturbation on the malware samples. Then we give an instance of the samli code. The output of the generator is pairs of caller-callee functions. There are three steps to implement output-to-perturbation transformation. First, for every function pair, we find the smali file related to the selected caller, according to the latter's full name. Second, we insert statements into the smali file to implement a try-catch trap. Here we can use five types of function invocation, including invoke-direct, invoke-virtual, invoke-static, invoke-super and invoke-interface. Different invocation types require different smali manipulation. Fig. 15 shows an example for every invocation type. Third, we use *Apktool* to rebuild the modified smali files to APK file. The above operations are automatically conducted by a Python script.

```
.class public Lcom/example/MainActivity;                          0
    ..........                                                    1
.method public static callerEx(ICLjava/lang/String;F)V            2
  .local 3                                                        3
  :try_start_0                                                    4
                                                                  5
  const/4 v0, 0x3                                                 6
  new-array v1, v0, [I                                            7
  const/4 v2, 0x1          compile   int myAttack[] = new int[3];  8
  aput v0, v1, v2                    myAttack[1] = 3;             9
  aget v2, v1, v2                    myAttack[myAttack[1]] = 3;   10
  aput v0, v1, v2                                                 11
                                                                  12
  new-instance p0,                                                13
        Ljava/lang/StringBuilder;                                 14
  invoke-direct {p0},                                             15
        Ljava/lang/StringBuilder;-><init>()V                      16
                                                                  17
  :try_end_0                                                      18
  .catch Ljava/lang/Exception; {:try_start_0 .. :try_end_0} :catch_0  19
  :catch_0                                                        20
                                                                  21
  # Original code                                                 22
.end method                                                       23
```

Figure 16: An instance of the smali code.

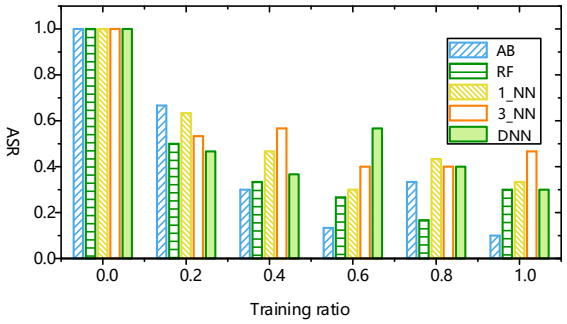

Figure 17: Attack success rate after retraining.

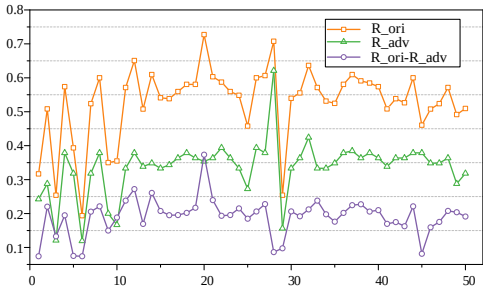

Figure 18: The detection success ratio on VirusTotal.

To show how BagAmmo manipulates the smali code, we supply a practical manipulation instance in Fig. 16. From line 6 to line 11, we can find a runtime exception. To be specific, we initialize an array with length 3 and employ an opaque method to visit the 4-th element of this array. Then it will throw an exception

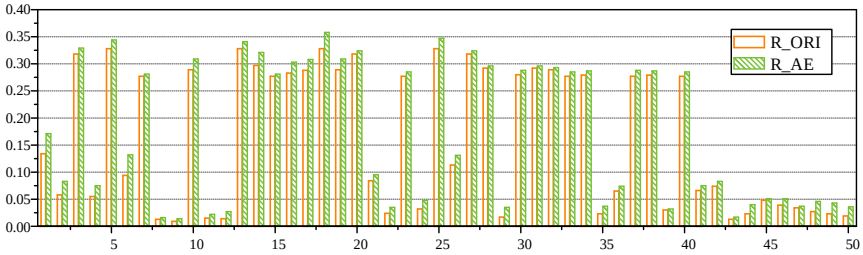

Figure 19: The number of try-catch blocks before and after BagAmmo attack.

java.lang.ArrayIndexOutOfBoundsException and skip the inserted callee functions. In this way, our method can effectively insert calls and preserve the malware's original functionality.

It is worth noting that the statements added into a try block are not fixed. Hence BagAmmo can resist the whitelist-based defense. For example, suppose we want to trigger the exception of *IndexOutOfBoundsException* by inducing array access violation. For this purpose, we access the array index that exceeds the array length. BagAmmo can generate countless variable names and variable values for such an array index. Therefore, it is impossible to build a white list to rule out the statements added by BagAmmo.

10.4 Dataset in our experiments

Our dataset includes 44375 Android APKs released from 2010 to 2020, which are collected from AndroZoo, FalDroid,and Drebin. Table 4 gives the source, count,and years of APKs in our dataset.

Table 4: Dataset used in our experiments.

Source	Label	Years	Count
Androdzoo	Benign	2010-2020	21399
	Malicious	2015-2020	9668
FalDroid	Malicious	2013-2014	8407
Drebin	Malicious	2010-2012	4900
Total	-	2010-2020	44374

10.5 Resistance to adversarial retraining.

Adversarial retraining is regarded as the most effective defense method against AE attacks. In this section, we test BagAmmo with adversarial retraining. We randomly select 100 adversarial examples that are generated by BagAmmo and can deceive target systems. We divide these adversarial examples into a training and a test set. Under various training sample proportions, we retrain the target classifier in order to evaluate ASR on the test set.

Our results are given in Fig. 17, whose vertical axis is the ASR of BagAmmo and the horizontal axis is the proportion of AEs used in adversarial retraining. Not surprisingly, the ASR decreases with the increase of the AEs adopted by adversarial retraining. When the ratio exceeds 40%, adversarial retraining becomes effective in resisting BagAmmo. In practice, however, it is extremely difficult to collect sufficient adversarial examples for adversarial retraining. On the other hand, it is also noted that aided by BagAmmo, model owners can improve their models' defense capability with adversarial retraining.

10.6 Attack performance on VirusTotal

We evaluate the performance of BagAmmo on VirusTotal. To be specific, we use BagAmmo to generate AEs (adversarial examples) through querying the MaMadroid detector, and upload them to VirusTotal for malware detection. VirusTotal uses about 60 malware detection methods unknown to us. We then record the ratio of the successful detection methods to all the methods, denoted by R_adv. For comparison, we also conduct the same setting for the original sample, and the corresponding ratio is termed as R_ori. What's more, we calculate the difference between the R_adv and R_ori, which termed as R_ori - R_adv. The results are shown in Fig. 18. The horizontal axis of this figure shows different APKs, and the vertical axis gives the ratios R_adv (denoted by the red line) and R_ori (denoted by the blue line). The yellow line shows the decreasing ratio of the successful detection methods. It can be seen that BagAmmo can effectively reduce the probability of malware being detected, owing to the transferability of AEs [38]. It is worth noting that this attack effect is achieved under the scenario where no queries are conducted and no prior knowledge about detection methods can be obtained.

10.7 The number of added try-catch blocks

Since BagAmmo inserts try-catch blocks into malware code, a defender may choose to detect it through judging whether the number of try-catch blocks exceeds a predetermined threshold. However, it is difficult to find an appropriate threshold for all APKs. Without such a threshold, this defense method may cause a high false positive or false negative rate.

To verify it, we record the ratio of try-catch block number to the function-calls number in 50 malicious APKs and the corresponding adversarially perturbed APKs, termed as R_ORI and R_AE, respectively. The results are shown in Fig.

19. The horizontal axis of this figure shows the IDs of these APKs, and the vertical axis gives the count ratio of try-catch blocks. The orange and green bars mean the original APK and the corresponding modified APK, respectively. We can draw two conclusions from this figure. First, the number of try-catch blocks added by our method is relatively small compared to that of existing try-catch blocks. Therefore it is hard to find a threshold to clearly distinguish the original APK and the perturbed APK. Second, the number of try-catch blocks drastically fluctuates among various APKs. Thus, it is also difficult to set a fixed threshold for all APKs.