Efficient Query-Based Attack against ML-Based Android Malware Detection under Zero Knowledge Setting

Ping He
Zhejiang University
gnip@zju.edu.cn

Yifan Xia
Zhejiang University
yfxia@zju.edu.cn

Xuhong Zhang
Zhejiang University
zhangxuhong@zju.edu.cn

Shouling Ji*
Zhejiang University
sji@zju.edu.cn

arXiv:2309.01866v2 [cs.CR] 6 Sep 2023

ABSTRACT

The widespread adoption of the Android operating system has made malicious Android applications an appealing target for attackers. Machine learning-based (ML-based) Android malware detection (AMD) methods are crucial in addressing this problem; however, their vulnerability to adversarial examples raises concerns. Current attacks against ML-based AMD methods demonstrate remarkable performance but rely on strong assumptions that may not be realistic in real-world scenarios, e.g., the knowledge requirements about feature space, model parameters, and training dataset. To address this limitation, we introduce AdvDroidZero, an efficient query-based attack framework against ML-based AMD methods that operates under the zero knowledge setting. Our extensive evaluation shows that AdvDroidZero is effective against various mainstream ML-based AMD methods, in particular, state-of-the-art such methods and real-world antivirus solutions.

CCS CONCEPTS

• **Security and privacy → Malware and its mitigation**; *Software security engineering*; • **Computing methodologies →** *Machine learning approaches*.

KEYWORDS

Malware; Machine Learning Security; Adversarial Android Malware

ACM Reference Format:

Ping He, Yifan Xia, Xuhong Zhang, and Shouling Ji. 2023. Efficient Query-Based Attack against ML-Based Android Malware Detection under Zero Knowledge Setting. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23), November 26–30, 2023, Copenhagen, Denmark.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3576915.3623117

1 INTRODUCTION

The Android operating system is prevalent in today's technologically advanced world, powering various mobile devices and applications [60]. However, its popularity has also led to a significant issue: Android malware. Recent security reports have identified over 33 million Android malware samples, posing a considerable threat to users' privacy and data integrity [16, 23, 57, 58, 61, 63]. To combat this, researchers have widely adopted machine learning-based (ML-based) Android malware detection (AMD) methods [13, 26, 29, 43, 45, 80]. These techniques efficiently identify and classify potential threats by analyzing Android application features, making ML-based AMD effective in protecting users and their devices.

Unfortunately, ML-based AMD techniques are vulnerable to attacks [20, 22, 24, 30, 64, 65, 71]. To evade them, adversaries can manipulate malicious applications in the problem space, causing the feature vector to cross the ML classifier's decision boundary. With such an attack, adversaries can effectively generate thousands of realistic, inconspicuous adversarial Android malware at scale, making "adversarial malware as a service" a real threat. Although various attacks [25, 32, 37, 53, 76, 81] against ML-based AMD methods have been proposed, they assume that the adversary has extensive knowledge of the target system, including access to the dataset, feature space, and model parameters. For instance, the HRAT attack [81] understands the feature space of the target system and consequently designs four types of function call manipulation to modify the function call graph. Besides, it additionally utilizes gradient information to choose the manipulation type and the manipulated function, necessitating knowledge of the dataset and model parameters. This assumption might be too strong and not always applicable in real-world situations. In practice, adversaries often have limited or no knowledge about the target system, making the system's internals, such as the dataset, feature space, and model parameters, unknown.

The efficient generation of adversarial Android malware under zero knowledge setting remains an open question. However, addressing this problem is far from straightforward, primarily due to two critical challenges: the **vast heterogeneous perturbation space** originating from the sample side and the **zero knowledge challenge** stemming from the detection side.

The first challenge is the numerous heterogeneous perturbations possible for the Android application package (APK) file, attributed to its complex structure. The APK file is a ZIP archive, typically containing a manifest file and Dalvik Executable (DEX) codes. The adversary's modification space for APK files is more diverse than that for images or text. For example, within an APK, an adversary could modify permissions in the manifest file or alter function call relations in DEX codes, while in images, they would only need to change pixel values. In addition, even when considering homogeneous perturbations, the perturbation space is still large and discrete. For instance, if the adversary wants to add a permission element in the manifest file, there are over 150 permissions with different protection levels in the Android system, each with varying evasion effectiveness.

Another challenge in this scenario is that the adversary has zero knowledge about the target ML-based AMD method. Different target methods may use different heterogeneous features, e.g., permission,

*Corresponding Author

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CCS '23, November 26–30, 2023, Copenhagen, Denmark.
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0050-7/23/11. . . $15.00
https://doi.org/10.1145/3576915.3623117

https://t.me/learningnets

activities, function call graphs, etc. As a result, carefully designed perturbations (e.g., modifying function call relations only) may not align with the target system's features (e.g., permissions only), making them ineffective. For instance, changing the function call relations is useless for Drebin [13] since Drebin does not use the function call-related feature and thus the elaborate perturbations in the problem space do not affect it at all. Consequently, employing transfer attacks [44, 50, 78] directly may prove to be suboptimal due to the feature space gap. Even if the adversary attempts to deduce the features of the target model, they can only rely on query feedback, i.e., labels and confidence, which is inadequate for efficiently identifying the target system's precise features. Furthermore, previous works [25, 37, 81] expect to acquire real-time fine-grained feedback from the target model, e.g., the gradients of the target model, to direct efficient and effective attack. Moreover, under the zero knowledge setting, the adversary can only get coarse-grained feedback, i.e., labels or confidence, which makes the attack process blind. Therefore, such an attack process is not applicable because the adversary only has zero knowledge in this scenario.

To address the aforementioned challenges, we propose a query-based **adv**ersarial An**droid** malware attack framework under **zero** knowledge setting, termed AdvDroidZero. At a high level, AdvDroidZero proposes a novel data structure termed perturbation selection tree to mitigate the challenge of vast heterogeneous perturbation space and a semantic-based adjustment policy to tackle the zero knowledge challenge. Our intuition is that perturbations in the APK file have varying levels of evasion effectiveness and exhibit semantic meaning. Perturbations with similar semantics are more likely to display comparable evasion effectiveness. For instance, adding a *uses-feature* element with *android.hardware.audio.output* value in the manifest file means the perturbation is related to hardware and audio. The perturbation will have similar evasion effectiveness with adding a *uses-feature* element with *android.hardware.audio.pro* value. Once a perturbation with positive evasion effectiveness is identified, we can infer other perturbations with positive evasion effectiveness based on semantic meaning, thereby facilitating the selection of optimal perturbations.

Based on these insights, AdvDroidZero designs a perturbation selection tree based on semantics at varying granularities, abstracting the perturbation selection process into path sampling in the tree. Similar to the decision tree models in machine learning that can handle heterogeneous data types, the perturbation selection tree can naturally manage heterogeneous perturbations. To address the vast perturbation space challenge, AdvDroidZero clusters semantically similar perturbations and executes them simultaneously. This approach reduces the perturbation search space, enhancing the efficiency of the generation process. To tackle the zero knowledge challenge, AdvDroidZero designs a semantic-based adjustment policy to change a batch of node probabilities based on their abstract semantic levels in cascade. The high-level semantics representing the perturbation type (e.g., perturbations in manifest or DEX codes) can be determined through the cascade process by a few initial queries. Meanwhile, the semantic-based adjustment policy can diffuse the query feedback information from the selected perturbation to its semantically similar perturbations through the tree structure. Hence, this policy can adjust semantically similar perturbations simultaneously via every single query. As a result, the semantic-based adjustment policy efficiently increases the probabilities of clusters with positive evasion effectiveness while reducing those with negative evasion effectiveness under the zero knowledge setting.

We evaluate AdvDroidZero using the large Android malware dataset containing 135,859 benign applications and 15,778 malware (151,637 total). This evaluation demonstrates the attack performance of AdvDroidZero against various mainstream ML-based AMD methods using static program analysis approaches, including Drebin [13], Drebin-DL [32], APIGraph [80] and MaMadroid [45]. AdvDroidZero outperforms the baseline methods [59] in terms of both attack effectiveness and runtime overheads. In particular, AdvDroidZero achieves about 90% attack success rate against these mainstream ML-based AMD methods. Additionally, AdvDroidZero requires only around 15 queries to generate most of (80%) adversarial Android malware and completes the process within 10 minutes. Furthermore, AdvDroidZero also demonstrates effectiveness against real-world antivirus solutions (AVs), sandboxes in VirusTotal, and the state-of-the-art (SOTA) robust AMD method FD-VAE [40]. Specifically, AdvDroidZero reduces the number of detected antivirus engines for approximately 92% of malware samples within 10 queries against VirusTotal and decreases the number of detected antivirus engines by 47.98% per successfully attacked malware. Moreover, AdvDroidZero also achieves about 89% attack success rate within 10 queries against the VirusTotal on the recent malware samples from 2022. Even when targeting the sandboxes in VirusTotal and the state-of-the-art robust AMD methods, AdvDroidZero still maintains the attack success rate of 37% under 10 query budgets and 77% under 30 query budgets, respectively.

The key contributions of this paper are summarized as follows. We propose AdvDroidZero, which is an efficient query-based attack framework for generating adversarial Android malware under zero knowledge setting. In AdvDroidZero, a novel data structure called perturbation selection tree is proposed to address the vast heterogeneous search space challenge, and a semantic-based adjustment policy is proposed to choose the corresponding perturbations efficiently.

2 BACKGROUND

2.1 Android Application Package

APK is the file format used for distributing and installing applications on the Android operating system. The APK file structure aims to encompass all the necessary aspects of an application, from its core code and resources to its metadata. It typically consists of several key components: *AndroidManifest.xml*, *classes.dex*, *resources.arsc*, *res/*, *assets/*, *META-INF/*, *lib/*. Among these, *AndroidManifest.xml* and *classes.dex* play pivotal roles within the APK. The *AndroidManifest.xml* file carries comprehensive configuration information about the APK, while the *classes.dex* file holds the program semantics of the application. Additional details regarding these components can be found in Appendix A.

The mainstream ML-based AMD methods [13, 45, 80] typically extract features from the manifest file and DEX codes (i.e., *AndroidManifest.xml* and *classes.dex*). As an informative configuration file, *AndroidManifest.xml* specifies Android application requirements, e.g., permission to access the Internet. The syntax features (static features) stemming from the manifest file are usually represented

https://t.me/learningnets

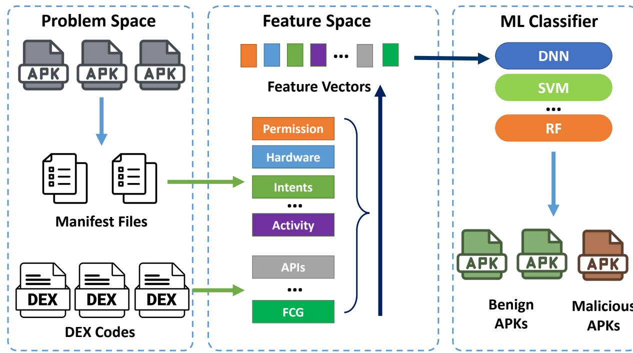

Figure 1: ML-based Android malware detection framework.

as binary features, where the value indicates whether a certain element is declared or not. *classes.dex* assembles the source codes of an Android application. The semantic features (dynamic features) extracted from *classes.dex* contain rich program semantics, e.g., the function call graph captures the function relation semantics.

2.2 ML-based AMD

ML-based AMD methods are widely employed in real-world applications due to their efficiency and accuracy. Typically, these methods contain two key stages: feature extraction and model prediction, as illustrated in Figure 1. The ML-based AMD methods first utilize program analysis tools [67] to analyze the configuration information and the behavior of the applications. The obtained analysis results are then used to extract specific features for the machine learning process. The choice of features can vary significantly depending on the specific ML-based AMD method in use. For instance, Drebin [13] primarily considers the syntax features (static features), e.g., permissions, while MaMadroid [45] only considers the semantic features (dynamic features), e.g., function call. Once the features are extracted, they are organized into a feature vector. This organized data is then used in the model prediction stage. In this phase, ML-based AMD methods train machine learning classifiers to differentiate between benign and malicious applications. Different ML-based AMD methods may employ different ML classifiers, e.g., Drebin [13] utilize the Support Vector Machine (SVM) classifier and the MaMadroid [45] can utilize the Random Forest (RF) classifier.

We utilize four SOTA and representative ML-based AMD methods in our evaluation. They are Drebin [13], Drebin-DL [32], MaMadroid [45], and APIGraph [80]. The details about the four ML-based AMD methods can be found in Appendix B. The rationale behind selecting the four methods can be attributed to their well-acknowledged performance in malware detection and popularity in research [25, 38, 53, 81]. For instance, Drebin [13] achieved an AUROC score of 0.96 in our evaluation (Table 2). Additionally, these ML-based AMD methods have been the target models for previous attack studies [25, 38, 53, 81]. Moreover, the diversity of their feature spaces and classifiers enhances the comprehensiveness of our evaluation.

Despite the remarkable achievements of the aforementioned ML-based AMD methods, they remain vulnerable to adversarial examples. Existing attack works demonstrate these vulnerabilities [19, 25, 37, 38, 81] but have their own limitations, e.g., require the knowledge about the target AMD methods, be limited to specific ML-based AMD methods, need a substantial number of target model queries, and/or not be robust to pre-processing. More details can be found in Section 6. The vulnerabilities of ML-based AMD methods stem from two areas: feature representation and bias in real-world data, which create evasion opportunities. Even though these methods leverage a wide spectrum of features to model the malicious behavior of malware, the features captured may not precisely portray all nuances of such behavior. This limitation provides a loophole for adversarial malware to evade detection. On the other hand, in real-world deployments, machine learning classifiers may encounter biases stemming from spatial and temporal factors [12, 17, 34, 52, 75]. These biases have the potential to negatively influence the decision boundaries of classification models, which in turn render these models suboptimal [22, 24, 33, 36, 42, 66, 77]. Consequently, existing ML-based AMD methods face considerable threats from attacks, which exploit these vulnerabilities to undermine the accuracy and effectiveness of ML-based AMD methods.

3 METHODOLOGY

In this section, we first describe the threat model associated with AdvDroidZero. Next, we provide an overview of the attack framework, which contains three stages from a practical perspective. Finally, we elucidate the detailed methodology of AdvDroidZero by the three stages.

3.1 Threat Model

We present the threat model by delineating three key components: adversary goals, knowledge and capabilities, building upon previous works [11, 18, 21, 53, 62].

Adversary Goals. The adversary aims to execute misclassification attacks against a given ML-based AMD method. This entails introducing a sequence of perturbations $\mathbf{P}^*$ within the perturbation space $\mathbb{P}$ to a malware sample in the problem space, causing the target model g to misclassify it as benign (b). The adversary focuses solely on the misclassification of the malicious samples. Consequently, the adversary goals can be formulated as:

$$\mathbf{P}^* = \arg\min_{\mathbf{P}\in\mathbb{P}} cost(\mathbf{P}), \tag{1}$$

$$\text{s.t. } g(\phi(\mathbf{X}+\mathbf{P}^*)) = b,\ malicious(\mathbf{X}) = malicious(\mathbf{X}+\mathbf{P}^*),$$

where $cost()$ denotes the metric function employed to assess the cost of generating perturbations (e.g., human efforts, runtime overheads), $\phi()$ represents the feature extraction function, $malicious()$ signifies the malicious functionality verification function. The adversary wants to find a sequence of perturbations with minimal cost. The sequence of perturbations can make the target model misclassify the malware ($g(\phi(\mathbf{X}+\mathbf{P}^*)) = b$). However, the sequence of perturbations can not influence the malicious functionality of the malware ($malicious(\mathbf{X}) = malicious(\mathbf{X}+\mathbf{P}^*)$).

Adversary Knowledge. Drawing from the zero knowledge setting defined in previous work [53], the adversary lacks information about the target system, being only aware that static analysis is employed on the programs. The adversary remains uninformed about feature spaces, model parameters, and the training dataset. However, the

https://t.me/learningnets

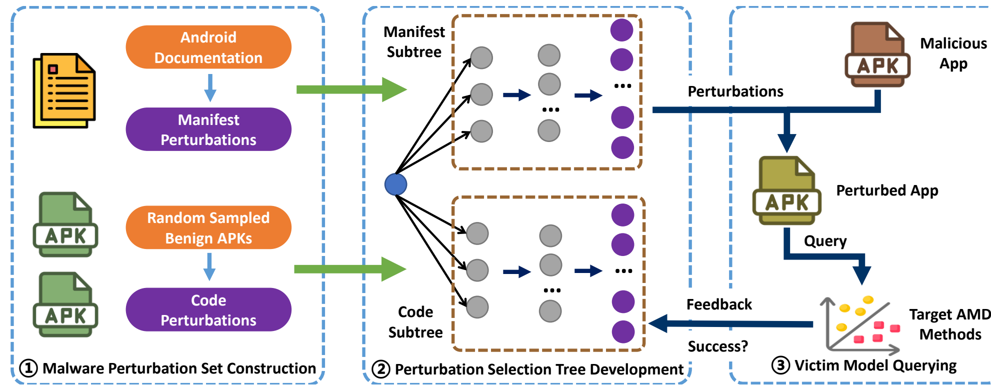

Figure 2: The overview of AdvDroidZero. It operates in three stages: malware perturbation set construction, perturbation selection tree development and victim model querying.

adversary can acquire certain open-source knowledge, such as the Android documentation [3], and access publicly available benign applications from app markets [31] or app dataset [1].

Adversary Capabilities. As more queries imply higher financial costs and an increased probability of detection, we assert that the adversary's ability to query the target system is limited. The allocation of query budgets reflects the ability of the attacker, as the higher query budgets mean the stronger attacker. We explore different query budgets in our experiments representing the attackers with different capabilities. Additionally, we assume that the adversary can access the confidence provided by the target system. This assumption is based on the fact that outputs from some real-world antivirus systems (e.g., VirusTotal) can be viewed as confidence, exemplified by the percentage of detected antivirus engines. We acknowledge that in certain scenarios, the attacker may not have the capacity to inspect the output of some ML-based AMD methods. We delve into a more detailed discussion about it in Section 5.3.

3.2 Framework Overview

At a high level, AdvDroidZero is a query-based attack framework to generate adversarial Android malware against ML-based AMD methods. It operates through three stages: (1) malware perturbation set construction, (2) perturbation selection tree development, and (3) victim model querying, as shown in Figure 2. In the following, we illustrate every stage briefly, and the algorithmic descriptions for AdvDroidZero are provided in Appendix C.

Malware Perturbation Set Construction. Without knowledge of the target ML-based AMD method, AdvDroidZero incorporates two kinds of malware perturbation from open-source information: manifest perturbations and code perturbations. The manifest perturbations targeting the syntax features (static features), e.g., permission, derive from Android documentation. The code perturbations targeting the semantic features (dynamic features), e.g., function call, derived from 100 benign applications selected uniformly at random, absent from the training set of target ML-based AMD methods.

Perturbation Selection Tree Development. AdvDroidZero builds the perturbation selection tree to guide the perturbation selection based on the semantic meanings inherent within the malware perturbations. The tree's leaf nodes represent specific malware perturbations, while internal nodes symbolize shared semantics of the leaf nodes within the associated subtree.

Victim Model Querying. AdvDroidZero selects malware perturbations iteratively using the perturbation selection tree, injects the selected perturbations, queries the target model with the perturbed application, and updates the perturbation selection tree based on the received feedback through a semantic-based adjustment policy. This iterative process continues until the framework achieves success or exhausts the query budget.

3.2.1 Key Intuition. Perturbations within an APK exhibit varying levels of evasion effectiveness. Some perturbations have positive attack effectiveness by reducing the malware label's confidence, while others can impede such attacks by increasing it. Furthermore, these perturbations exhibit significant semantic depth. For example, the perturbations in the manifest file involve setting a target element with a specific value. Adding a *uses-feature* element with an *android.hardware.audio.output* value relates to the declared hardware and software features required by the application. In this case, the *android.hardware.audio.output* value corresponds to hardware features, specifically audio requirements.

Upon understanding the semantics of perturbations, we posit that those with similar semantics are more likely to display comparable evasion effectiveness. This is based on the observation that benign Android applications employ semantically similar elements in the manifest file to achieve specific functionalities. We discuss the validation of the assumption in Appendix D. Consequently, once a perturbation is identified with positive attack effectiveness, we can infer other perturbations with positive attack effectiveness based on semantic meaning, thereby facilitating the selection of optimal perturbations.

Based on these insights, AdvDroidZero organizes perturbations in a tree structure according to their semantic depth, accommodating the heterogeneous perturbation space. To reduce this space, AdvDroidZero groups semantically related perturbations and executes them simultaneously. After receiving model feedback, AdvDroidZero adaptively adjusts the selection probabilities within the

https://t.me/learningnets

perturbation selection tree in a cascading manner, enhancing query efficiency and improving understanding of the target model.

3.3 Malware Perturbation Set Construction

As demonstrated in Section 3.1, the adversary lacks knowledge of the feature spaces, model parameters, and training dataset of the target model. Nevertheless, the adversary can obtain open-source materials such as Android documentation and benign applications to derive the malware perturbation set. Moreover, malware manipulation has three requirements stated by previous works [38]: all-feature influence, functional consistency, and robustness to pre-processing. We discuss the detailed requirements and existing Android malware manipulation methods in Appendix E.

To date, imposing all-feature influence under a zero knowledge setting remains inadequately explored. To address this issue, the perturbation set is proposed encompassing both manipulations on *AndroidManifest.xml* and manipulations on DEX code. By employing these two types of perturbations, AdvDroidZero can mislead mainstream ML-based AMD methods (Section 2.2), regardless of whether they extract features, e.g., syntax features, semantic features, from *AndroidManifest.xml* or DEX codes. It is worth noting that if the feature knowledge of the targeted ML-based AMD methods is available, the malware perturbation set can be limited in order to perform more precise attacks. For example, the malware perturbation set can only contain the code perturbations when targeting the function call graph-based AMD methods in practice. However, since there is no constraint on the perturbation space, we consider broader perturbations to enhance the overall adaptability.

In accordance with the functional consistency requirement, any perturbation should not remove existing items from the *AndroidManifest.xml* or DEX codes. However, it is feasible to add new items or codes which do not interfere with the original functionalities of the manipulated application. Specifically, to manipulate *AndroidManifest.xml* [4], we consider adding the *uses-feature* element, the *use-permission* element, and the *action* element or *category* element. For the *uses-feature* element, we take all referenced information about hardware features and software features into account [9]. Regarding the *uses-permission* element, the permission system is designed to protect security and privacy, which is a critical feature for identifying malware [15]. Every permission has a protection level that characterizes the potential risk implied by the permission [5]. We argue that if a malicious application possesses permissions at a dangerous level, it is more likely to be detected. Consequently, we only add permissions with normal or signature protection levels. Concerning *action* and *category* elements, these elements, encapsulated in the *intent-filter* element, provide information about the intents of the application. However, adding them directly to existing activities or services might interfere with the original functionality. Therefore, we create new *activity* elements and *receiver* elements for these elements. To preserve functionality, the newly added *activity* elements and *receiver* elements should not be launched during the application's lifetime. Their names are randomly generated to prevent explicit intents from the original code from launching them. Furthermore, to avoid being launched by implicit intents, the *action* and *data* elements with the random URI are added to the *activity* and *receiver*. In terms of added values, we consider the standard activity actions,

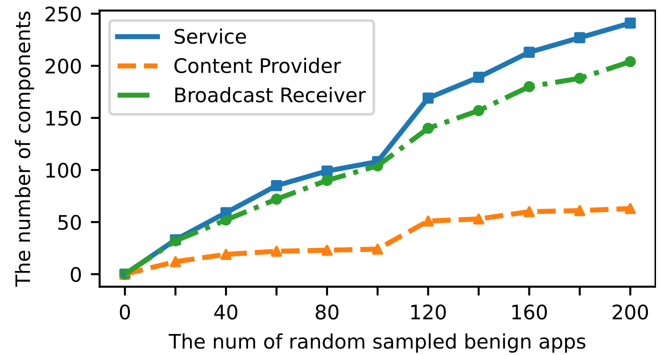

Figure 3: The number of services, content providers and broadcast receivers with the number of randomly sampled benign applications.

standard broadcast actions, and standard categories in Android documentation [7] because they are established norms. Specific examples are provided in Appendix F.

To manipulate the DEX codes, AdvDroidZero adds three types of Android app components from benign applications, i.e., service, broadcast receiver, and content provider [6]. These components are selected for the following reasons. On the one hand, these components can be easily obtained from benign applications. We randomly sample benign applications from AndroZoo, then tally the numbers of services, broadcast receivers, and content providers according to their *AndroidManifest.xml*. As depicted in Figure 3, only 100 sampled benign applications yield 108 different services, 24 different content providers, and 104 different broadcast receivers. This indicates that these components are abundant in benign applications. On the other hand, these components possess rich semantics, which is beneficial for reducing queries. We enumerate classes and functions for our randomly chosen components from 100 benign applications. The results show approximately 175 classes and 873 functions per service, 136 classes and 703 functions per broadcast receiver, and 417 classes and 2,044 functions per content provider. With such abundant code information, adding components to malware significantly alters its semantic features, making it more likely to deceive the classifier. Therefore, the iterative addition of components can quickly change the features of malware, leading to lower queries.

To maintain the functional consistency requirement, there are two cases for added components. On the one hand, the added components should not be launched during APK execution. Here, since no extra code is executed, the original functionality remains unaffected. On the other hand, if the added components are launched, their execution should not affect the original part of the malware code. Even though these components execute, they act as independent entities without impacting the original functionality.

To satisfy the first case, AdvDroidZero does not inject explicit invokes in the original part of the code as done in previous work [53]. However, simply adding components to malware is easily identifiable, as the added code can be viewed as an isolated part of the code, which static analysis tools can effortlessly remove. This violates the requirement for robustness to pre-processing. To achieve robustness in pre-processing, we leverage Android's intent mechanism.

https://t.me/learningnets

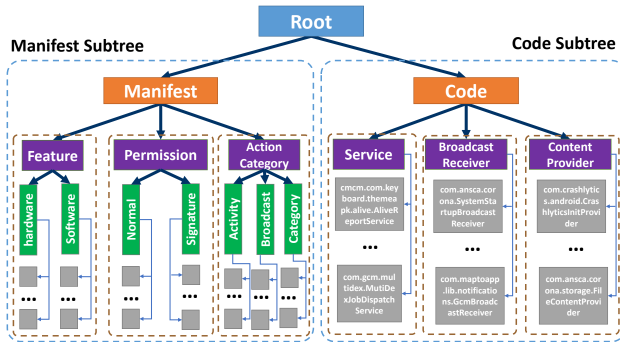

Figure 4: The structure of perturbation selection tree.

Specifically, when registering these components in *AndroidManifest.xml*, AdvDroidZero sets the attributes of *android:exported* and *android:enabled* to *true*. By setting *android:enabled* to true, the component is enabled and can be started and respond to intents, indicating that it is not dead code. By setting *android:exported* to true, the component can be accessed or invoked by other applications on the device, not just within the application that defines it. This allows external applications to interact with the component, provided they have the necessary permissions, ensuring that the added components cannot be considered dead code.

To satisfy the second case, we ensure the original malware functionality remains unaffected even if other applications invoke the added components. This is achieved using Android's subprocess mechanism [8]. By setting the *android:process* attribute with a random string, the added components run in another process, isolated at the system level from the original part of the malware. Furthermore, the added components are randomly chosen from benign applications and have no inter-process communication with the original part of the malware. Therefore, even if the added components are launched, the functionality of the malware is not influenced. Activities can also be executed in the subprocess. However, when an activity is invoked, it will be displayed on top of the current activity. As a result, the user will see the new activity on the screen, and it influences the original functionality of the malware. Hence, we exclude activities in our approach. We provide examples in Appendix F.

3.4 Perturbation Selection Tree Development

Upon acquiring the malware perturbation selection set, it is crucial to select specific perturbations from the set. However, this set contains numerous heterogeneous perturbations. With 492 specific perturbations in our set, it becomes challenging to select an optimal perturbation sequence. Moreover, the heterogeneous nature of the perturbation set makes it unsuitable for using an optimization algorithm to obtain the best solution [81]. To address these challenges, AdvDroidZero employs a novel data structure called the perturbation selection tree, designed to select optimal perturbations. The perturbation selection tree is designed with the following considerations in mind. Firstly, akin to the decision tree algorithms [49, 55] in machine learning that can process different feature types, such as numerical and categorical features, the perturbation selection tree model can handle heterogeneous perturbations using its tree structure. Secondly, the semantic depth of perturbations is helpful for designing the tree structure. By leveraging the unique design of the perturbation selection tree, AdvDroidZero can uncover hidden relationships between perturbations. This enables it to extract more information from a single query, thereby reducing the number of queries required.

At a high level, the perturbation selection tree is a hierarchical tree structure in which leaf nodes represent specific perturbations. During each selection procedure, AdvDroidZero samples a path from the root node to a leaf node, corresponding to the particular perturbations. Internal nodes represent the specific semantics shared by the leaf nodes belonging to the subtree with the internal node as its root. Figure 4 illustrates the structure of the perturbation selection tree, which consists of two nodes at the first layer: the manifest node and the code node.

The manifest subtree represents the subtree organizing the perturbations related to modifications in the *AndroidManifest.xml*. AdvDroidZero firstly considers the element-level semantic. Specifically, AdvDroidZero separates the perturbations into use-feature subtree, permission subtree, and action&category subtree. The use-feature subtree encompasses perturbations involving the addition of the *uses-feature* element in the *AndroidManifest.xml*, the permission subtree collects perturbations related to adding the *permission* element, and the action&category subtree comprises perturbations about adding the *action* or *category* elements. AdvDroidZero then examines the semantics at the name level. Within the use-feature subtree, element names follow the format *android.hardware.xxx* or *android.software.xxx*, where *xxx* denotes a specific feature name, such as *microphone* and *bluetooth*. Based on this observation, AdvDroidZero organizes the hardware and software subtree within the use-feature subtree. In the permission subtree, permissions have different protection levels, prompting AdvDroidZero to categorize perturbations into normal and signature level groups. Within the action&category subtree, standard activity actions, standard broadcast actions, and standard category are considered, resulting in the formation of activity action, broadcast, and category subtree, respectively.

In the leaf node layer of the manifest subtree, specific perturbations still exhibit rich semantics derived from their text representation. Consequently, AdvDroidZero clusters semantically similar perturbations into a single leaf node in the perturbation selection tree. This approach enables AdvDroidZero to sample multiple perturbations simultaneously and execute them collectively, effectively reducing query costs. To cluster perturbations in the uses-feature subtree is relatively straightforward due to the simple and regular text representations of the *uses-feature* element names. For instance, *android.hardware.audio.output* and *android.hardware.audio.pro* exhibit similar semantics, as they share the keyword *audio* in the third position from left to right. Therefore, AdvDroidZero clusters these perturbations based on the keyword. However, for perturbations in the permission subtree and action&category subtree, no explicit keywords are present in their name representations, rendering the clustering algorithm used in the uses-feature subtree unsuitable. Instead, AdvDroidZero devises a customized clustering algorithm to cluster perturbations in these two subtrees based on their text representation. For example, in the permission perturbation *android.permission.ACCESS_WIFI_STATE*, three keywords

https://t.me/learningnets

(i.e., *ACCESS*, *WIFI*, and *STATE*) can be extracted. The algorithm iteratively clusters perturbation groups based on the similarity of the keywords. The detailed clustering algorithm can be found in Appendix G.

The code subtree represents the subtree in which the root node organizes perturbations related to modifying DEX codes. As demonstrated in Section 3.3, AdvDroidZero injects three types of Android app components from benign applications into malware. These component types serve as explicit semantics for the perturbations. Consequently, AdvDroidZero constructs the service subtree, broadcast receiver subtree, and content provider subtree to accommodate these types. Although various kinds of program semantics exist, such as function call graphs [48, 56], Android app components are relatively independent and capable of representing specific functions. As a result, AdvDroidZero does not extract fine-grained semantics or cluster components together. Instead, each leaf node in the code subtree represents a single Android app component, preserving the independence and functionality representation of these elements.

Upon constructing the structure of the perturbation selection tree, AdvDroidZero initializes the probabilities of each node in the perturbation selection tree in a bottom-up manner. For leaf nodes in the manifest subtree, they comprise perturbation groups that combine multiple specific perturbations. Consequently, larger perturbation groups are more likely to cause large changes in model confidence. However, it is uncertain whether these confidence changes will benefit the attack (decreasing the model's confidence in the malware label) or harm it. As such, these perturbations are risky; if a perturbation is detrimental to the attack, it may cause significant damage. Conversely, smaller perturbation groups will have minimal impact on model confidence. Considering these intuitions, it is advantageous to select perturbation groups with median sizes. As a result, AdvDroidZero leverages a normal distribution to fit the sizes of perturbation groups and allocates initial probabilities based on the normal distribution. For leaf nodes in the code subtree, AdvDroidZero assigns equal probabilities to them due to their independent properties. For all internal nodes except for the manifest node and code node, AdvDroidZero aims to sample each leaf node uniformly. Thus, AdvDroidZero assigns node probabilities based on the number of leaf nodes that are descendants of the node, with a larger number of leaf nodes corresponding to smaller selection probabilities. The more number of leaf nodes, the smaller chosen probability. Regarding the probabilities of the manifest node and code node, the default value is set to 0.5, as the target model is unknown. However, if the attacker possesses prior knowledge about the feature type of the target classifier, they can adjust these values to expedite the attack process.

Lastly, in the perturbation selection process, AdvDroidZero samples a path from the root node to the leaf node. Specifically, AdvDroidZero iteratively selects nodes at random based on their selection probabilities until a leaf node is chosen.

3.5 Victim Model Querying

After developing the perturbation selection tree, AdvDroidZero then proceeds with a four-step iterative process: selecting the malware perturbation, perturbing the malicious applications, obtaining the model feedback, and updating the perturbation selection tree. Initially,

Algorithm 1 Semantic-based Adjustment Policy

Input: Perturbation selection tree $\mathbb{T}$; Perturbation node P; Previous malicious confidence y; Malicious confidence y'.
Output: Adjusted perturbation selection tree $\mathbb{T}'$.

```
1: while True do                          ▷ Recursively delete the perturbation node P.
2:     P' ← GetParentNode(P)              ▷ Get the parent perturbation node P'.
3:     if P' has other children then
4:         TransferProba(P, P')           ▷ Transfer probability of P to other children.
5:         break
6:     else
7:         P ← P'
8:     end if
9: end while
10: if y' ≥ y then                        ▷ Adjust the internal node probability.
11:     P ← P'
12:     P' ← GetParentNode(P')
13:     while P' is not root do
14:         ReInitProb(P')                ▷ Reinitialize the internal node probability.
15:         if y' = y then                ▷ Add penalty when no effectiveness.
16:             AddPenalty(P, P')
17:         end if
18:         P ← P'
19:         P' ← GetParentNode(P)
20:     end while
21: end if
22: if y' ≥ y then                        ▷ Adjust the first layer probability.
23:     AdjustFirstLayer(T)
24: end if
25: return T'
```

AdvDroidZero selects the malware perturbation from the tree by sampling a path from the root to a leaf node. Then, AdvDroidZero uses program analysis tools, such as FlowDroid [14], to implement the chosen perturbations to the malicious application. After this, the perturbed application is submitted to the target model to garner feedback. Finally, AdvDroidZero refines the perturbation selection by using a semantic-based adjustment policy leveraging the received model feedback. The iterative process continues until either success is achieved or the query budget depletes.

In the victim model querying step, the semantic-based adjustment policy is a crucial element, as it allows the perturbation selection tree to learn from the target model adaptively. The underlying intuition for this semantic-based adjustment policy is that if perturbations positively impact malware evasion (decreasing the model's confidence in the malware label), the probability of semantically related perturbations should be increased. Conversely, if perturbations negatively impact malware evasion, the probability of semantically related perturbations should be decreased. If perturbations have no effect on malware evasion (no changes in the model's confidence in the malware label), it means that the target model may not use the features altered by the perturbation. Then the probability of semantically related perturbations should be decreased with a penalty. Consequently, AdvDroidZero imposes an additional penalty to reduce the probability of all nodes that are ancestors of the perturbation node.

The details of the adjustment policy algorithm can be found in Algorithm 1. Upon obtaining the selected perturbation node, the adjustment policy removes it and equally distributes its probability among its sibling nodes (lines 1-9). After this deletion, the internal nodes with the selected perturbation node as a descendant should have their probability decreased. However, if the selected perturbation node has a positive impact, the probability of these nodes should increase, which is achieved by not decreasing the probability. Otherwise, the

https://t.me/learningnets

probability is reduced by computing it based on the leaf node count used during the initialization phase (lines 10-21). If the selected perturbation node has no impact, an extra penalty is applied to its ancestor nodes (lines 15-17). The penalty percentage is calculated as the node depth multiplied by a constant (0.1 in our experiments), resulting in deeper nodes receiving larger penalties. Lastly, the probability of the first-layer nodes is adjusted independently (lines 22-24) to identify the target model's feature type quickly. Consequently, the corresponding manifest node or code node probability is halved if the selected perturbation node does not positively affect evasion.

4 EVALUATION

In this section, we conduct comprehensive experiments to evaluate the performance of AdvDroidZero. We first present the experimental settings, including the datasets, target AMD methods, and evaluation metrics. We then assess the effectiveness and cost of AdvDroidZero. Subsequently, we examine its performance in the context of real-world antivirus solutions. Furthermore, we explore the effectiveness of AdvDroidZero against the dynamic analysis-based defense. Finally, we evaluate the functionality consistency requirement by both static and dynamic analysis.

4.1 Experimental Setup

Implementation Details. AdvDroidZero is an automatic attack framework with three stages to generate adversarial Android malware. Our implementation of the prototype of AdvDroidZero utilizes a hybrid approach combining both Python and Java. We use Python to handle the program logic of the attack process, encompassing the data structure of the malware perturbation set, the development of the perturbation selection tree and the semantic-based adjustment policy, as well as managing the iterative query process during the victim model querying stage. On the other hand, Java is utilized for the extraction and implementation of perturbations. In order to slice Android app components to construct the malware perturbation set and perform program manipulation in the stage of victim model querying, the framework integrates FlowDroid [14], which is built upon Soot [67].

In practice, our implementation automatically selects the malware perturbations (e.g., permissions) from the perturbation selection tree. It then automatically employs FlowDroid to modify the application by incorporating these chosen perturbations. It continues by autonomously querying the targeted ML-based AMD method with the modified application. The feedback from this query helps refine the perturbation selection tree. This iterative process continues until either success is achieved or the query budget depletes. To facilitate further research, the code and data of AdvDroidZero are responsibly shared with other researchers upon request[1] following previous works [53, 81] due to potential ethical concerns (Section 5.4).

Dataset. Our primary dataset comprises 135,859 benign applications and 15,778 malware samples, totaling 151,637 applications. This dataset is derived from the previous work [53]. We download the APKs from AndroZoo [1] based on the SHA-256 value provided in the aforementioned study [53]. As a result, we successfully obtain 151,637 applications dated between January 2016 and December 2018. The dataset has already been processed by Pierazzi *et al*. [53], adhering to the labeling criteria outlined in Tesseract [52].

However, we employ a time-aware split [12, 52] for our dataset, which differs from the approach taken in the previous work [53]. Our work aims to generate adversarial Android malware under the zero knowledge setting, a highly practical scenario. In contrast, Pierazzi *et al*. sought to reveal the weaknesses of machine learning classifiers in the perfect knowledge context. In practical applications, the concept drift problem naturally exists and must be considered. Therefore, we perform a time-aware split of the dataset to simulate a *real malware classifier*. Specifically, we utilize applications dated between January 2016 and December 2017 as the training set and applications dated between January 2018 and December 2018 as the test set.

In order to incorporate more recent malware samples, we also employed a supplementary dataset provided by VirusShare [68], consisting of 20,206 malware samples from 2022. The rapidly evolving landscape of Android APKs and the fact that the most recent samples in our primary dataset are only from 2018 make this supplementary dataset crucial for assessing the effectiveness of AdvDroidZero in a temporal context. Specifically, we evaluate AdvDroidZero against VirusTotal using the supplementary dataset (Section 4.3.1).

Target Model. We select four SOTA ML-based AMD methods, namely Drebin [13], Drebin-DL [32], MaMadroid [45], and API-Graph [80], as our target ML-based AMD methods. To ensure fidelity to their original implementations, we strictly adhere to the descriptions and configurations provided in their respective publications. Consequently, for Drebin and APIGraph, we employ SVM with the linear kernel as the target classifier, while for Drebin-DL, we utilize a two-layer multi layer perceptron (MLP) as the target classifier. For MaMadroid, we use RF and 3-Nearest Neighbor (3-NN) as target classifiers. These target models incorporate various feature types and machine learning classifiers, thereby offering a diverse representation of ML-based AMD methods. Further details regarding the implementation and the detection performance of these target models can be found in Appendix H.

Metric. In our experimental evaluation, we conduct attacks using true positive malware from the test set in the primary dataset, meaning that we target malware samples that have been accurately classified as malware. Intuitively, there is no need to generate adversarial examples for malware applications already misclassified as benign since they do not require any queries. Consequently, we employ the attack success rate (ASR) as the evaluation metric for the effectiveness of AdvDroidZero. ASR represents the proportion of successfully generated adversarial Android malware instances (denoted by N_s) to the total number of malware samples utilized for the attack (denoted by N_t), i.e., ASR = N_s/N_t.

Additionally, in order to measure the attack cost associated with AdvDroidZero, we take into account both human factors and runtime overheads, as recommended by Apruzzese *et al*. [11]. From the perspective of human factors, we employ the duration incorporating the attack preparation and design phases, which includes tasks such as source code writing to measure the attack cost. To measure the attack cost from the perspective of runtime overheads, we assess the program execution time and the number of query times (QT). The program execution time refers to the time from the query beginning to

[1]The instructions regarding access requests can be found at: https://github.com/gnipping/AdvDroidZero-Access-Instructions.

https://t.me/learningnets

Table 1: Attack performance of AdvDroidZero and baseline method measured by ASR. It can be seen that AdvDroidZero outperforms all baseline methods under most settings of query budget, target AMD methods and ML classifiers.

Attack Methods	Query Budgets	Drebin	Drebin-DL	APIGraph	MaMadroid	
		SVM	MLP	SVM	RF	3-NN
AdvDroidZero	10	57%	67%	50%	92%	83%
	20	80%	80%	70%	98%	93%
	30	87%	85%	94%	100%	100%
	40	100%	90%	94%	100%	99%
MAB	10	48%	41%	31%	91%	84%
	20	63%	78%	60%	98%	97%
	30	78%	84%	77%	100%	98%
	40	98%	91%	83%	100%	100%
RA	10	39%	44%	35%	93%	92%
	20	62%	77%	58%	98%	90%
	30	85%	80%	76%	100%	86%
	40	90%	88%	87%	100%	98%

the generation of adversarial Android malware serving as a reflection of the CPU cost to some extent, whereby a shorter duration indicates a more scalable attack. Meanwhile, QT signifies the number of queries executed in the process of generating a single adversarial Android malware instance. It provides insights into the potential financial cost and detection likelihood, considering the pricing strategies of commercial machine learning models and the established correlation between increased queries and the detection probability [39].

Experimental Environment. We run all experiments on a Ubuntu 20.04 server with 251G memory and 39G swap memory, 2 Intel(R) Xeon(R) Gold 6346 CPUs and one NVIDIA RTX 3090 GPU.

4.2 Attack Performance

To assess the performance of AdvDroidZero, we apply it to attack the aforementioned mainstream target models. The evaluation of the attack performance encompasses two dimensions: attack effectiveness and attack cost. Moreover, we compare AdvDroidZero with two attack methods, namely MAB [59] and Random Attack (RA). Detailed information regarding the baseline methods can be found in Appendix I.

4.2.1 Attack Effectiveness. To evaluate attack effectiveness, we establish different query budgets for generating adversarial Android malware. We consider the generation of adversarial Android malware successful if the attack algorithm can generate it within the query budget. To be specific, we randomly select 100 true positive malware samples from the test set within our primary dataset to carry out the attack, implying that the target malware samples have been correctly classified as malware. Subsequently, we compute the ASR in every case, as shown in Table 1. We also report the actual number of successfully generated adversarial Android malware instances in every case, which can be found in Appendix J.

Table 1 presents the attack effectiveness of AdvDroidZero and baseline methods against mainstream ML-based AMD methods using static program analysis approaches under varying query budgets. AdvDroidZero outperforms the baseline methods in most settings of query budget, target AMD methods, and ML classifiers. Additionally, AdvDroidZero achieves nearly 100% ASR against all AMD methods, regardless of feature type or ML classifier, when the query budget is set to 40. This demonstrates that AdvDroidZero exhibits strong attack effectiveness against ML-based AMD methods using static program analysis under zero knowledge settings.

AdvDroidZero performs better when the target model only considers graph-based features. For example, as shown in Table 1, the ASR of AdvDroidZero is 92% for the MaMadroid method with the RF classifier when the query budget is set to 10. However, the ASR of AdvDroidZero drops to 57% when the target AMD method is Drebin with the SVM classifier under the same query budget. This can be attributed to two factors. First, the adjustment policy design in AdvDroidZero allows for quickly learning the feature type of the target AMD methods. The adjustment policy modifies the probability of manifest nodes and code nodes exponentially, enabling AdvDroidZero to choose effective perturbations with just a few queries. Second, the Android app component-level perturbations are effective due to the rich program semantics within these components. The rich program semantics can significantly change the function call graph, resulting in substantial changes to the malware's feature value. Furthermore, since these Android app components are sourced from benign applications, they intuitively have a positive impact on the attack.

Another observation is that the APIGraph-enhanced Drebin method is more robust than the original Drebin method. For instance, under the 20 query budget setting, AdvDroidZero achieves a 70% ASR on the APIGraph-enhanced Drebin method, while the ASR is 80% for the original Drebin method. This can be explained by the fact that APIGraph replaces individual functions with their corresponding clusters, providing a higher level of abstraction and reducing the impact of minor variations. Consequently, during the ML model training process, ML models more readily learn robust features from APIGraph-enhanced features.

We also find that the 3-NN classifier is more robust than the RF classifier. For example, under the 10 query budget setting, AdvDroidZero achieves a 83% ASR for the MaMadroid method with the RF classifier, whereas the ASR is 92% for the 3-NN classifier.

https://t.me/learningnets

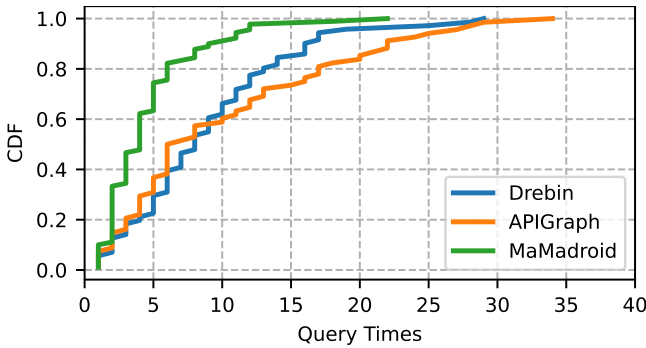

Figure 5: The CDF of query times in successfully evaded malware.

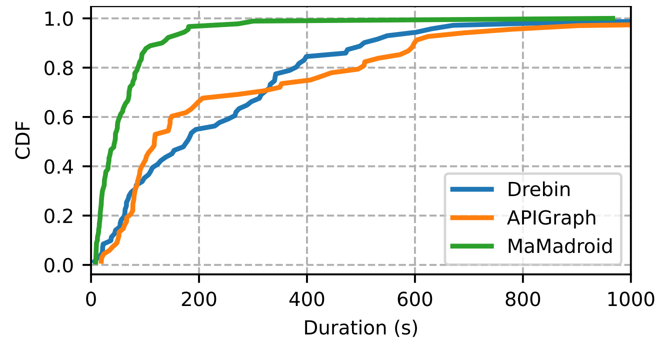

Figure 6: The CDF of the program execution time in successfully evaded malware.

This conclusion holds for all baseline methods. As noted by Li *et al.* [38], this is because the 3-NN classifier takes into account the data samples in the training set when classifying a sample. This process typically considers more features, making it easier to distinguish between benign applications and malware.

The performance of AdvDroidZero increases as the number of queries increases, as shown in Table 1. For example, AdvDroidZero achieves 50% ASR against APIGraph with 10 query budgets while 94% ASR with 30 query budgets. However, with more queries, the potential financial cost and the detection probability increase as well. Thus, we find that the 30-query budget is the optimal value to trade off the cost and utility because it is the smallest value to achieve 100% ASR against the MaMadroid methods. It is worth noting that AdvDroidZero can employ any query budget to conduct the attack depending on the practical capability of the attacker.

Compared to AdvDroidZero, all baseline methods are less effective against ML-based AMD methods, especially when the query budget is low. For instance, AdvDroidZero achieves a 94% ASR against APIGraph within 30 queries, a result unattainable by MAB and RA even with 40 queries. The reason behind this observation is as follows: MAB does not concern the context of the perturbations and only accounts for the perturbation type. Consequently, when MAB faces an AMD method like Drebin, it is less effective because the context information is crucial. For FCG-based methods like MaMadroid, due to the effectiveness of Android app component-level perturbation, context information is less important, allowing it to achieve comparable attack effectiveness with AdvDroidZero. RA does not consider any information in perturbation; therefore, its effectiveness depends on the perturbation set. The effectiveness of RA can be viewed as a result of our perturbation set's contribution.

In summary, AdvDroidZero demonstrates strong attack effectiveness against ML-based AMD methods using static program analysis approaches under zero knowledge setting. The results highlight the advantages of the semantic-based adjustment policy design and Android app component-level perturbations. Additionally, the comparisons with baseline methods, such as MAB and RA, further emphasize the superiority of AdvDroidZero in handling a variety of target models with different feature types and ML classifiers.

4.2.2 Attack Cost. To evaluate the attack cost of AdvDroidZero, we examine two aspects: human factors and runtime overheads. The human factors mainly involve the stages of malware perturbation set construction and perturbation selection tree development. The two stages require extensive domain expertise and OSINT operations, including detailed analysis of Android documentation. Based on our code update records, the two stages collectively take us approximately three months, highlighting their importance to our contribution. Breaking it down further, we spent about a week identifying the malware perturbation set and preparing all malware perturbations. The remainder of the time is devoted to designing the perturbation selection tree and the semantic-based adjustment policy. However, other potential attackers who might use AdvDroidZero may only need to occasionally revisit the malware perturbation set construction stage, which requires human involvement, such as analyzing the latest Android documentation. This stage requires considerably less time (one week for us) compared to designing the entire attack framework.

Once the initial setup is complete, the emphasis then shifts to the query process. At this stage, the QT and the program execution time become significant metrics. QT is important as it reflects the financial implications and detection likelihood of the attack, while the program execution time signifies the scalability of the attack. We apply AdvDroidZero to attack against Drebin, APIGraph, and MaMadroid using RF, setting the query budget to 40 in 100 random sampled malware, which can be identified by these methods in the test set of the primary dataset. In total, the numbers of successfully evaded malware are 74, 71, and 98, respectively. We then record the QT and program execution time for all successfully evaded malware samples and plot the cumulative distribution function (CDF) for both QT and program execution time in the successfully evaded malware samples. Figure 5 depicts the CDF of the QT, and Figure 6 illustrates the CDF of the program execution time.

In Figure 5, we observe that most successfully evaded malware samples (80%) against the MaMadroid classifier require only about 6 queries. Although attacking Drebin and APIGraph necessitates more QT, most successfully evaded malware instances need only about 15 queries. These results demonstrate that AdvDroidZero is able to perform effectively with a small number of queries. In Figure 6, we observe that most successfully evaded malware samples (80%) against the MaMadroid classifier require under 100 seconds of program execution time. While attacking Drebin and APIGraph demands more program execution time, most successfully evaded malware instances need only about 500 seconds of program execution time. To establish the baseline comparison, we apply the MAB and RA against the

https://t.me/learningnets

Drebin, keeping the settings identical to those in the AdvDroidZero. The results yield 53 and 54 successfully evaded malware instances, respectively. Subsequently, we delve into a comparative analysis in terms of the average QT and the average program execution time. Our results reveal that the average QT for AdvDroidZero, MAB, and RA are 9.04, 10.64, and 9.86, respectively. In terms of the average program execution time, AdvDroidZero, MAB, and RA are clocked at 250.53 seconds, 258.33 seconds, and 305.76 seconds, respectively. The superior performance of AdvDroidZero can be attributed to the application of the perturbation selection tree and the adjustment policy to some extent. It is supported by the fact that the sole point of divergence between AdvDroidZero and the baseline methods is their strategy for selecting perturbations. This finding suggests that AdvDroidZero can efficiently generate adversarial Android malware, indicating high scalability. Furthermore, AdvDroidZero does not depend on the information shared between malware samples, allowing for parallel implementation, which is another advantage in terms of scalability.

Takeaway. The designs of AdvDroidZero are effective as the attack performance outperforms the baseline methods in terms of the attack success rate. The QT and the program execution time indicate the low runtime overheads of AdvDroidZero.

4.3 Real-World AVs

To assess the performance of AdvDroidZero on real-world AVs, we evaluate it on VirusTotal, which hosts over 70 AVs whose specifics remain undisclosed to us, including large companies such as McAfee, Symantec, and Microsoft.

Specifically, we employ the VirusTotal API [69] to upload applications to VirusTotal and obtain query feedback. As described in Section 3.1, we interpret the percentage of detected antivirus engines as malware confidence. Due to VirusTotal's query limitations and the approximately one-minute analysis time for each uploaded APK, we randomly sample 100 malware samples from our test set for evaluation. We set 10 query budgets for each malware sample. After uploading malware to VirusTotal, we check the analysis results once per minute until the results are available or the duration exceeds five minutes.

In total, we successfully generate 71 adversarial Android malware samples that reduce the number of detected antivirus engines. Additionally, 23 malware samples reach the query exception, meaning it has hit the time limit for obtaining the analysis result, and thus, we discontinue further attempts at analyzing it. Consequently, AdvDroidZero reduces the number of detected antivirus engines for approximately 92% (71 / (100 - 23)) of the malware samples. To further analyze attack effectiveness, we compare the original and adversarial distribution of detected engines for the 71 malware samples. Figure 7 (a) shows that the average detection rate is approximately 18.02%, corresponding to 13.32 antivirus engines. In contrast, an average of only 9.84% (7.28 engines) of antivirus engines flag the AdvDroidZero generated adversarial Android malware as malicious. Overall, AdvDroidZero decreases the number of detected antivirus engines by 47.98%, indicating its effectiveness against VirusTotal.

For a more in-depth analysis of AdvDroidZero's attack effectiveness, we examine its performance against five specific antivirus

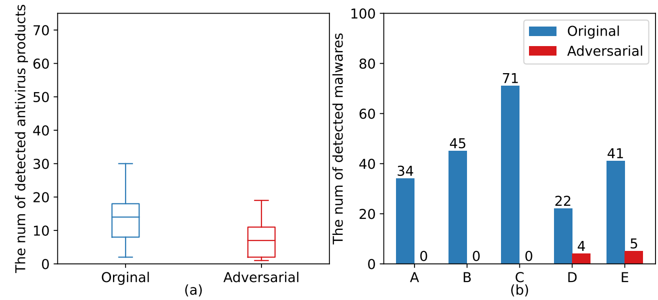

Figure 7: The attack effectiveness of AdvDroidZero against VirusTotal on the primary dataset. (a) The original and adversarial distribution of the number of detected engines. (b) The number of detected malware on 5 antivirus products, where A denotes McAfee, B denotes McAfee-GW-Edition, C denotes SymantecMobileInsight, D denotes Avira, and E denotes Microsoft. The result shows that AdvDroidZero can achieve attack effectiveness against VirusTotal.

products on VirusTotal provided by the large companies: **McAfee**, **McAfee-GW-Edition**, **SymantecMobileInsight**, **Avira**, and **Microsoft**. We report the number of detected original malware and adversarial malware samples for these five antivirus products. Figure 7 (b) reveals that **McAfee**, **McAfee-GW-Edition**, and **SymantecMobileInsight** fail to detect any adversarial Android malware generated by AdvDroidZero, indicating a 100% ASR against these products. For **Avira**, AdvDroidZero reduces the detection ratio from 30.98% to 5.63%. For **Microsoft**, AdvDroidZero decreases the detection ratio from 57.74% to 7.04%. These results demonstrate that AdvDroidZero can effectively compromise the products of large companies.

4.3.1 *Temporal Effectiveness.*

To assess the effectiveness of AdvDroidZero over time, we carry out evaluations against VirusTotal using the supplementary dataset. We randomly select 500 malware samples from the supplementary dataset, allocating a query budget of 10 for each malware sample. The strategy we employ to obtain analysis results for these samples is identical to that used for the malware samples in the primary dataset.

In total, we successfully generate 349 adversarial Android malware samples that reduce the number of antivirus engines detecting them. Furthermore, 109 malware samples reach the query limit, signifying that they have exhausted the time allotment for analysis. Thus, we terminate further analysis attempts for these samples. As a result, AdvDroidZero decreases the number of antivirus engines that detect about 89% (349 / (500 - 109)) of the malware samples. To delve deeper into the effectiveness of the attack, we compare the original and adversarial distribution of the detected engines for the 349 malware samples. Figure 8 (a) indicates that the average detection rate is approximately 22.32% (16.73 antivirus engines). Conversely, only an average of 11.37% (8.52 engines) of antivirus engines flag the adversarial Android malware produced by AdvDroidZero as malicious. In summary, AdvDroidZero reduces the percentage of antivirus engines detecting malware by 56.57%, underlining its effectiveness over time.

https://t.me/learningnets

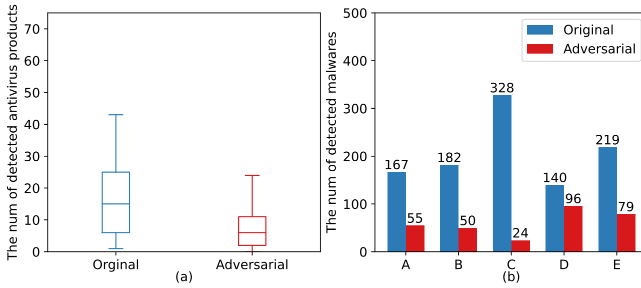

Figure 8: The attack effectiveness of AdvDroidZero against VirusTotal on the supplementary dataset. (a) The original and adversarial distribution of the number of detected engines. (b) The number of detected malware on 5 antivirus products, where A denotes McAfee, B denotes McAfee-GW-Edition, C denotes SymantecMobileInsight, D denotes Avira, and E denotes Microsoft.

For a more comprehensive analysis of the effectiveness, we assess its performance against the previously mentioned five specific antivirus products. Figure 8 (b) proves that AdvDroidZero continues to achieve satisfactory attack effectiveness in most cases against products from large companies. For example, AdvDroidZero diminishes the detection ratio from 65.60% to 4.80% when pitted against **SymantecMobileInsight**.

4.4 Dynamic Analysis-based Defense

In our evaluation of AdvDroidZero, we primarily focus on static analysis-based methods. This is because a majority (73% according to a recent survey [43]) of ML-based AMD methods rely on static analysis for feature extraction. Additionally, static analysis-based methods are likely employed in real-world antivirus systems (Section 4.3). The dynamic analysis-based defense is also deployed in the real world. Therefore, we also evaluate AdvDroidZero against the dynamic analysis-based defense.

To explore the attack effectiveness of AdvDroidZero against the dynamic-based analysis defenses, e.g., sandboxes. We apply AdvDroidZero to attack the sandboxes in VirusTotal [70], which are the SOTA dynamic analysis-based defenses. Specifically, we randomly select 30 malicious applications that can be detected by the sandboxes in VirusTotal. We then use AdvDroidZero with a query budget of 10 against VirusTotal to generate the perturbed applications. Among all the generated perturbed applications, 11 applications are reported by the VirusTotal sandboxes to have no potential malicious behaviors, suggesting that AdvDroidZero achieves an ASR of 37% against VirusTotal sandboxes within 10 queries.

The results indicate that AdvDroidZero can still have attack effectiveness against dynamic analysis-based defense. This can be attributed to Android app component perturbations (code perturbations) in the malware perturbation set. The components in Android applications typically implement independent functions and have rich semantics, making them likely to be triggered by the sandboxes. Furthermore, since the components are derived from benign applications, they inherently lack malicious behavior. This may lead the sandbox to spend an excessive amount of time analyzing them, potentially overshadowing the behavior of the original malware.

4.5 Functionality

In accordance with prior work [38, 80], we employ both static and dynamic analysis to assess the functionality consistency of AdvDroidZero.

For evaluating functionality consistency through static analysis, we utilize Apktool [10] to decompile the generated adversarial Android malware. More explicitly, we randomly sample 20 perturbed malware samples generated in the process against the VirusTotal. Then, we record every applied perturbation in their generation process and decompile these APKs. Subsequently, we manually review the decompiled source code of the adversarial APKs. Our primary objective here is to verify two key points. First, whether the perturbations have been correctly injected into the malware code. Second, to establish if the injected code exhibits any relation (i.e., function calls or class relationships) to the original code. Our results confirm that the decompile process is successful for all perturbed malware samples and that the perturbations are inserted appropriately. Moreover, we observe no explicit connections between the injected code and the original code. These observations collectively indicate that the functionality of the perturbed malware remains consistent despite the perturbations.

Regarding dynamic analysis, we install and execute these randomly selected 20 pairs of original and perturbed malware samples from the process against the VirusTotal in an Android Virtual Device (AVD) provided by Android Studio. The AVD is configured with API level 30 Google APIs and simulates a Nexus 5 device. Our observations reveal that each malware pair exhibits the same performance and runtime UI. This outcome suggests that all modified malware samples operate correctly, the inserted functions are not invoked, and thus, the malware's functionality remains unaffected.

5 DISCUSSION

5.1 Possible Defense

Li *et al.* [40] propose a robust ML-based AMD method called the FD-VAE model, designed to defend against the adversarial Android malware. The method employs a Variational Autoencoder (VAE) to disentangle features of different classes, thereby enhancing detection performance and robustness. We re-implement the FD-VAE algorithm using PyTorch [51], following the descriptions and configurations provided in the original paper and their open-source code. Subsequently, we randomly sample 100 malware samples from the test set in the primary dataset that the FD-VAE can detect for attack evaluation.

The results indicate that AdvDroidZero achieves an ASR of 77% under 30 query budgets, 50% under 20 query budgets, and 26% under 10 query budgets. These findings demonstrate that AdvDroidZero exhibits strong attack effectiveness when the query budget is relatively large, while the ASR is comparatively lower when the query budget is insufficient. This can be attributed to the FD-VAE model's use of a rejection mechanism within the VAE model. The model outputs a malware label when the loss of the VAE surpasses a predefined threshold, which may mislead our algorithm when the query budget is inadequate.

https://t.me/learningnets

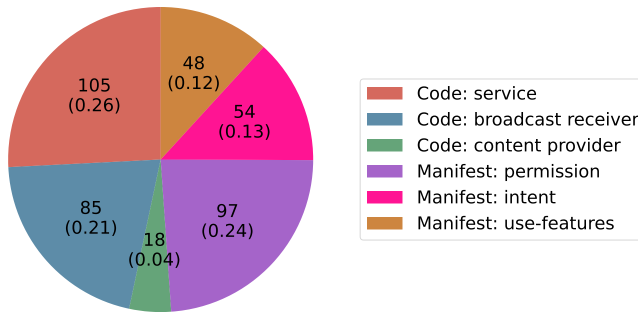

Figure 9: The distribution of the added perturbation type in our generated adversarial Android malware.

Although AdvDroidZero does not perform optimally in cases with small query budgets, it can still achieve a satisfactory ASR when the query budget is within an acceptable threshold (30 query budgets).

5.2 Perturbation Types

To gain insight into the types of perturbations added to our adversarial Android malware, we analyze the distribution of these perturbations in the adversarial Android malware samples we generated. Specifically, we randomly select 100 malicious applications that can be detected by VirusTotal. We then apply AdvDroidZero to these applications, attacking VirusTotal with a query budget of 10 for every sample. Consequently, we successfully generate 61 adversarial Android malware samples that manage to reduce the number of antivirus engines that detect them on VirusTotal.

As illustrated in Figure 9, the most common perturbations are primarily service, permission, and broadcast receivers, with respective occurrence ratios of 26%, 24%, and 21%. This can potentially be attributed to the fact that services and broadcast receivers are widely utilized in benign applications (as depicted in Figure 3), causing these features to be identified by ML-based AMD methods. As a result, they aid malicious applications in evading detection. Permissions have been extensively studied and identified as a critical feature in malware detection [15]. In designing AdvDroidZero, we consciously decide to avoid integrating permissions that are considered high-risk, opting instead to focus on the permissions that have normal or signature protection levels. This approach confers a seemingly benign appearance upon the perturbed malicious applications, assisting them in avoiding detection.

5.3 Limitations & Future Work

In this study, we introduce an efficient query-based attack framework, AdvDroidZero, targeting ML-based AMD methods under zero knowledge setting. Our work aims to offer valuable insights into the field of AMD and to raise awareness regarding the potential threats posed by attacks. Furthermore, our method can be employed to assess the robustness of existing ML-based AMD methods under zero knowledge setting. In the following, we discuss some limitations of our approach and outline potential future research directions.

Obscured Model Output. The AdvDroidZero framework needs the confidence of the target model provided by ML-based AMD methods [13, 73]. However, in some cases, the model output might be restricted or obscured, which can limit the effectiveness of AdvDroidZero. To adapt AdvDroidZero to these limited output scenarios, some strategies can be implemented, such as approximating the confidence level by the percentage of detected antivirus engines provided in VirusTotal. This technique can offer a reasonable estimate when the confidence of the ML-based AMD method is unavailable. Nevertheless, we acknowledge that there might be certain ML-based AMD methods that are not susceptible to the AdvDroidZero framework due to the constraints in inspecting output. In future research, we plan to tackle these specific cases.

Hybrid Analysis-based Defense. Our primary goal is to design an effective and efficient attack framework for generating adversarial Android malware in a practical setting. ML-based AMD methods that employ hybrid program analysis are a promising avenue for research. These methods have the potential to combine the strengths of both static and dynamic program analysis. However, according to a recent survey [43], these methods have yet to be thoroughly explored and have many unresolved challenges, such as high computational demands, etc. Thus, the application of them remains limited. We plan to explore the impact of hybrid analysis-based defenses in future research.

Malware Perturbation Set Limitation. The AdvDroidZero framework relies on the malware perturbation set. If none of the perturbations affect the feature values of the target AMD methods, the efficacy of AdvDroidZero would be compromised. However, we contend that such a scenario is unlikely to occur in real-world settings for several reasons. Firstly, our perturbation set is relatively general and has demonstrated its effectiveness in influencing real-world antivirus solutions. Moreover, extending the perturbation selection tree is straightforward, as it merely requires the addition of a subtree to accommodate new perturbations. This inherent flexibility and adaptability of AdvDroidZero make it a robust framework to attack against the ML-based AMD methods.

5.4 Potential Ethical Concerns

The primary objective of our study is to assess the robustness of ML-based AMD methods, a topic with established precedence in earlier works [38, 53, 81]. This is driven by the potential for adversaries to engineer malware applications that can evade detection, thus underscoring the necessity for robust ML-based AMD methods. Even though the intent is strict about evaluating the robustness of ML-based AMD methods, potential ethical concerns are associated with our research. Therefore, we will limit code sharing upon request to verified academic researchers only, following a precedent established by previous studies [53, 81]. Besides, in VirusTotal experiments, we utilized VirusTotal in the same capacity as an ordinary user would—submitting applications. It aids the security community by providing samples of this type of malicious application.

6 RELATED WORK

Adversarial examples have recently garnered significant attention in various domains, such as text classification [28, 41], reinforcement learning [74], and explainability [79]. In the malware domain, numerous attack algorithms [19, 25, 37, 38, 53, 59, 72, 81] have been developed to attack malware detection methods. For instance, Sun *et*

https://t.me/learningnets

al. [19] employ machine learning explainability techniques to select benign features for modification, while Song *et al*. [59] approach the attack problem as a multi-armed bandit problem, creating a generic machine and a specific machine to profile perturbations using Thompson sampling to cope with the delayed feedback.

In the Android malware domain, Chen *et al*. [25] utilize optimization algorithms to generate adversarial perturbations in the feature space and introduce a method for applying optimal perturbations to APKs. Pierazzi *et al*. [53] propose the APG (adversarial problem generation) framework, which generates adversarial malware from feature space to problem space. Zhao *et al*. [81] present a structural attack that employs reinforcement learning to target FCG-based AMD methods. Li *et al*. [37] explore ensemble learning algorithms for adversarial Android malware. Recently, Li *et al*. [38] developed the BagAmmo algorithm to create adversarial Android malware against FCG-based AMD methods in the limited knowledge setting. Bostani *et al*. [19] introduce EvadeDroid, which uses a random search algorithm to inject instruction-level gadgets into malware.

However, these methods do not fully address the practical requirements for generating adversarial Android malware. For instance, they may necessitate some knowledge (feature space, model parameters, training dataset) about the target AMD method [25, 37, 53, 81], or be limited to specific AMD models [38, 81]. Additionally, they may require a large number of queries for the target model [38], and/or their modifications in the problem space may not be robust to pre-processing [19].

7 CONCLUSION

This paper introduces AdvDroidZero, an efficient query-based attack framework designed to generate adversarial Android malware under zero knowledge setting. To address the vast and heterogeneous search space, AdvDroidZero employs an innovative data structure termed as the perturbation selection tree and proposes an adjustment policy for efficiently choosing the appropriate perturbations. Our experimental results, conducted on the large-scale real-world datasets, demonstrate that AdvDroidZero is effective against a wide variety of ML-based AMD methods using static program analysis. Furthermore, we evaluate AdvDroidZero's performance against real-world antivirus solutions, sandboxes and robust AMD methods, finding that it can bypass these defenses. The findings underscore the potential of AdvDroidZero as a valuable framework in the field of attacks on AMD methods.

ACKNOWLEDGMENTS

We sincerely appreciate our shepherd and the anonymous reviewers for their insightful comments. We would like to extend our gratitude to Chenghui Shi, Changjiang Li, Qinying Wang, and Yuhao Mao for their thoughtful feedback. We thank the support from the Zhejiang University College of Computer Science and Technology and the Zhejiang University NGICS Platform. This work was partly supported by NSFC under No. U1936215 and the Fundamental Research Funds for the Central Universities (Zhejiang University NGICS Platform).

REFERENCES

[1] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. AndroZoo: collecting millions of Android apps for the research community. In *MSR*.
[2] Androguard. 2023. Androguard. https://github.com/androguard/androguard. [Accessed on Apr. 21, 2023].
[3] Android. 2023. Android Documentation. [Accessed on Apr. 13, 2023].
[4] Android. 2023. Android Manifest Documentation. https://developer.android.com/guide/topics/manifest/manifest-intro. [Accessed on Apr. 21, 2023].
[5] Android. 2023. Android Permission. https://developer.android.com/reference/android/Manifest.permission. [Accessed on Apr. 21, 2023].
[6] Android. 2023. App Components. https://developer.android.com/guide/components/fundamentals. [Accessed on Apr. 21, 2023].
[7] Android. 2023. App Intent. https://developer.android.com/reference/android/content/Intent. [Accessed on Apr. 21, 2023].
[8] Android. 2023. App Process. https://developer.android.com/guide/components/processes-and-threads. [Accessed on Apr. 21, 2023].
[9] Android. 2023. Use-Feature reference. https://developer.android.com/guide/topics/manifest/uses-feature-element. [Accessed on Apr. 21, 2023].
[10] Apktool. 2023. Apktool. https://ibotpeaches.github.io/Apktool/. [Accessed on Apr. 21, 2023].
[11] Giovanni Apruzzese, Hyrum S. Anderson, Savino Dambra, David Freeman, Fabio Pierazzi, and Kevin A. Roundy. 2022. "Real Attackers Don't Compute Gradients": Bridging the Gap Between Adversarial ML Research and Practice. *CoRR* (2022).
[12] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. 2022. Dos and Don'ts of Machine Learning in Computer Security. In *USENIX Security Symposium*.
[13] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. 2014. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In *NDSS*.
[14] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *PLDI*.
[15] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 2012. PScout: analyzing the Android permission specification. In *ACM CCS*.
[16] AV-ATLAS. 2023. Total Amount of Android Malware. https://portal.av-atlas.org/malware/statistics. [Accessed on Apr. 13, 2023].
[17] Federico Barbero, Feargus Pendlebury, Fabio Pierazzi, and Lorenzo Cavallaro. 2022. Transcending TRANSCEND: Revisiting Malware Classification in the Presence of Concept Drift. In *IEEE S&P*.
[18] Battista Biggio and Fabio Roli. 2018. Wild patterns: Ten years after the rise of adversarial machine learning. *Pattern Recognit.* (2018).
[19] Hamid Bostani and Veelasha Moonsamy. 2021. EvadeDroid: A Practical Evasion Attack on Machine Learning for Black-box Android Malware Detection. *CoRR* (2021).
[20] Wieland Brendel, Jonas Rauber, and Matthias Bethge. 2018. Decision-Based Adversarial Attacks: Reliable Attacks Against Black-Box Machine Learning Models. In *ICLR*.
[21] Nicholas Carlini, Anish Athalye, Nicolas Papernot, Wieland Brendel, Jonas Rauber, Dimitris Tsipras, Ian J. Goodfellow, Aleksander Madry, and Alexey Kurakin. 2019. On Evaluating Adversarial Robustness. *CoRR* (2019).
[22] Nicholas Carlini and David A. Wagner. 2017. Towards Evaluating the Robustness of Neural Networks. In *IEEE S&P*.
[23] Rahul Chatterjee, Periwinkle Doerfler, Hadas Orgad, Sam Havron, Jackeline Palmer, Diana Freed, Karen Levy, Nicola Dell, Damon McCoy, and Thomas Ristenpart. 2018. The Spyware Used in Intimate Partner Violence. In *IEEE Symposium on Security and Privacy*.
[24] Jianbo Chen, Michael I. Jordan, and Martin J. Wainwright. 2020. HopSkipJumpAttack: A Query-Efficient Decision-Based Attack. In *IEEE S&P*.
[25] Xiao Chen, Chaoran Li, Derui Wang, Sheng Wen, Jun Zhang, Surya Nepal, Yang Xiang, and Kui Ren. 2020. Android HIV: A Study of Repackaging Malware for Evading Machine-Learning Detection. *IEEE Trans. Inf. Forensics Secur.* (2020).
[26] Nadia Daoudi, Kevin Allix, Tegawendé François D. Assise Bissyandé, and Jacques Klein. 2022. A Deep Dive Inside DREBIN: An Explorative Analysis beyond Android Malware Detection Scores. *ACM Trans. Priv. Secur.* (2022).
[27] Ambra Demontis, Marco Melis, Battista Biggio, Davide Maiorca, Daniel Arp, Konrad Rieck, Igino Corona, Giorgio Giacinto, and Fabio Roli. 2019. Yes, Machine Learning Can Be More Secure! A Case Study on Android Malware Detection. *IEEE Trans. Dependable Secur. Comput.* (2019).
[28] Tianyu Du, Shouling Ji, Lujia Shen, Yao Zhang, Jinfeng Li, Jie Shi, Chengfang Fang, Jianwei Yin, Raheem Beyah, and Ting Wang. 2021. Cert-RNN: Towards Certifying the Robustness of Recurrent Neural Networks. In *ACM CCS*.
[29] Parvez Faruki, Ammar Bharmal, Vijay Laxmi, Vijay Ganmoor, Manoj Singh Gaur, Mauro Conti, and Muttukrishnan Rajarajan. 2015. Android Security: A Survey of Issues, Malware Penetration, and Defenses. *IEEE Commun. Surv. Tutorials* (2015).
[30] Qi-An Fu, Yinpeng Dong, Hang Su, Jun Zhu, and Chao Zhang. 2022. AutoDA: Automated Decision-based Iterative Adversarial Attacks. In *USENIX Security Symposium*.
[31] Google Play. 2023. Google Play. [Accessed on Apr. 13, 2023].

https://t.me/learningnets

[32] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick D. McDaniel. 2017. Adversarial Examples for Malware Detection. In *ESORICS*.
[33] Wenbo Guo, Dongliang Mu, Jun Xu, Purui Su, Gang Wang, and Xinyu Xing. 2018. LEMNA: Explaining Deep Learning based Security Applications. In *ACM CCS*.
[34] Roberto Jordaney, Kumar Sharad, Santanu Kumar Dash, Zhi Wang, Davide Papini, Ilia Nouretdinov, and Lorenzo Cavallaro. 2017. Transcend: Detecting Concept Drift in Malware Classification Models. In *USENIX Security Symposium*.
[35] Hiral H. Karer and Purvi B. Soni. 2015. Dead code elimination technique in eclipse compiler for Java. In *ICCICCT*.
[36] Changjiang Li, Shouling Ji, Haiqin Weng, Bo Li, Jie Shi, Raheem Beyah, Shanqing Guo, Zonghui Wang, and Ting Wang. 2022. Towards Certifying the Asymmetric Robustness for Neural Networks: Quantification and Applications. *IEEE Trans. Dependable Secur. Comput.* (2022).
[37] Deqiang Li and Qianmu Li. 2020. Adversarial Deep Ensemble: Evasion Attacks and Defenses for Malware Detection. *IEEE Trans. Inf. Forensics Secur.* (2020).
[38] Heng Li, Zhang Cheng, Bang Wu, Liheng Yuan, Cuiying Gao, Wei Yuan, and Xiapu Luo. 2023. Black-box Adversarial Example Attack towards FCG Based Android Malware Detection under Incomplete Feature Information. *CoRR* (2023).
[39] Huiying Li, Shawn Shan, Emily Wenger, Jiayun Zhang, Haitao Zheng, and Ben Y. Zhao. 2022. Blacklight: Scalable Defense for Neural Networks against Query-Based Black-Box Attacks. In *USENIX Security Symposium*.
[40] Heng Li, ShiYao Zhou, Wei Yuan, Xiapu Luo, Cuiying Gao, and Shuiyan Chen. 2021. Robust Android Malware Detection against Adversarial Example Attacks. In *WWW*.
[41] Jinfeng Li, Shouling Ji, Tianyu Du, Bo Li, and Ting Wang. 2019. TextBugger: Generating Adversarial Text Against Real-world Applications. In *NDSS*.
[42] Linyi Li, Tao Xie, and Bo Li. 2023. SoK: Certified Robustness for Deep Neural Networks. In *IEEE S&P*.
[43] Yue Liu, Chakkrit Tantithamthavorn, Li Li, and Yepang Liu. 2023. Deep Learning for Android Malware Defenses: A Systematic Literature Review. *ACM Comput. Surv.* (2023).
[44] Yuhao Mao, Chong Fu, Saizhuo Wang, Shouling Ji, Xuhong Zhang, Zhenguang Liu, Jun Zhou, Alex X. Liu, Raheem Beyah, and Ting Wang. 2022. Transfer Attacks Revisited: A Large-Scale Empirical Study in Real Computer Vision Settings. In *IEEE S&P*.
[45] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon J. Ross, and Gianluca Stringhini. 2017. MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models. In *NDSS*.
[46] Jiang Ming, Dongpeng Xu, Li Wang, and Dinghao Wu. 2015. LOOP: Logic-Oriented Opaque Predicate Detection in Obfuscated Binary Code. In *ACM CCS*.
[47] Andreas Moser, Christopher Kruegel, and Engin Kirda. 2007. Limits of Static Analysis for Malware Detection. In *ACSAC*.
[48] Gail C. Murphy, David Notkin, William G. Griswold, and Erica S.-C. Lan. 1998. An Empirical Study of Static Call Graph Extractors. *ACM Trans. Softw. Eng. Methodol.* (1998).
[49] Anthony J Myles, Robert N Feudale, Yang Liu, Nathaniel A Woody, and Steven D Brown. 2004. An introduction to decision tree modeling. *Journal of Chemometrics: A Journal of the Chemometrics Society* (2004).
[50] Nicolas Papernot, Patrick D. McDaniel, Ian J. Goodfellow, Somesh Jha, Z. Berkay Celik, and Ananthram Swami. 2017. Practical Black-Box Attacks against Machine Learning. In *ACM AsiaCCS*.
[51] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *NeurIPS*.
[52] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder, and Lorenzo Cavallaro. 2019. TESSERACT: Eliminating Experimental Bias in Malware Classification across Space and Time. In *USENIX Security Symposium*.
[53] Fabio Pierazzi, Feargus Pendlebury, Jacopo Cortellazzi, and Lorenzo Cavallaro. 2020. Intriguing Properties of Adversarial ML Attacks in the Problem Space. In *IEEE Symposium on S&P*.
[54] Mila Dalla Preda, Matias Madou, Koen De Bosschere, and Roberto Giacobazzi. 2006. Opaque Predicates Detection by Abstract Interpretation. In *AMAST*.
[55] J. Ross Quinlan. 1996. Learning Decision Tree Classifiers. *ACM Comput. Surv.* (1996).
[56] Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. 2021. PyCG: Practical Call Graph Generation in Python. In *ICSE*.
[57] Yun Shen, Pierre-Antoine Vervier, and Gianluca Stringhini. 2021. Understanding Worldwide Private Information Collection on Android. In *NDSS*.
[58] Yun Shen, Pierre Antoine Vervier, and Gianluca Stringhini. 2022. A Large-scale Temporal Measurement of Android Malicious Apps: Persistence, Migration, and Lessons Learned. In *USENIX Security Symposium*.
[59] Wei Song, Xuezixiang Li, Sadia Afroz, Deepali Garg, Dmitry Kuznetsov, and Heng Yin. 2022. MAB-Malware: A Reinforcement Learning Framework for Blackbox Generation of Adversarial Malware. In *ACM AsiaCCS*.
[60] Statista. 2023. Mobile Operating Systems' Market. https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/. [Accessed on Apr. 13, 2023].
[61] Guillermo Suarez-Tangil and Gianluca Stringhini. 2022. Eight Years of Rider Measurement in the Android Malware Ecosystem. *IEEE Trans. Dependable Secur. Comput.* (2022).
[62] Octavian Suciu, Radu Marginean, Yigitcan Kaya, Hal Daumé III, and Tudor Dumitras. 2018. When Does Machine Learning FAIL? Generalized Transferability for Evasion and Poisoning Attacks. In *USENIX Security Symposium*.
[63] Zhichuang Sun, Ruimin Sun, Long Lu, and Alan Mislove. 2021. Mind Your Weight(s): A Large-scale Study on Insufficient Machine Learning Model Protection in Mobile Apps. In *USENIX Security Symposium*.
[64] Fnu Suya, Jianfeng Chi, David Evans, and Yuan Tian. 2020. Hybrid Batch Attacks: Finding Black-box Adversarial Examples with Limited Queries. In *USENIX Security Symposium*.
[65] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. 2014. Intriguing properties of neural networks. In *ICLR*.
[66] Thomas Tanay and Lewis D. Griffin. 2016. A Boundary Tilting Persepective on the Phenomenon of Adversarial Examples. *CoRR* (2016).
[67] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java bytecode optimization framework. In *CASCON*.
[68] VirusShare. 2023. VirusShare Dataset. https://virusshare.com/. [Accessed on July 17, 2023].
[69] VirusTotal. 2023. VirusTotal API Documentation. [Accessed on Apr. 13, 2023].
[70] VirusTotal. 2023. VirusTotal Sandboxes. https://support.virustotal.com/hc/en-us/articles/6253253596957. [Accessed on July 17, 2023].
[71] Viet Quoc Vo, Ehsan Abbasnejad, and Damith C. Ranasinghe. 2022. RamBoAttack: A Robust and Query Efficient Deep Neural Network Decision Exploit. In *NDSS*.
[72] Wei Wang, Ruoxi Sun, Tian Dong, Shaofeng Li, Minhui Xue, Gareth Tyson, and Haojin Zhu. 2021. Exposing Weaknesses of Malware Detectors with Explainability-Guided Evasion Attacks. *CoRR* (2021).
[73] Bozhi Wu, Sen Chen, Cuiyun Gao, Lingling Fan, Yang Liu, Weiping Wen, and Michael R. Lyu. 2021. Why an Android App Is Classified as Malware: Toward Malware Classification Interpretation. *ACM Trans. Softw. Eng. Methodol.* (2021).
[74] Xian Wu, Wenbo Guo, Hua Wei, and Xinyu Xing. 2021. Adversarial Policy Training against Deep Reinforcement Learning. In *USENIX Security Symposium*.
[75] Limin Yang, Wenbo Guo, Qingying Hao, Arridhana Ciptadi, Ali Ahmadzadeh, Xinyu Xing, and Gang Wang. 2021. CADE: Detecting and Explaining Concept Drift Samples for Security Applications. In *USENIX Security Symposium*.
[76] Wei Yang, Deguang Kong, Tao Xie, and Carl A. Gunter. 2017. Malware Detection in Adversarial Settings: Exploiting Feature Evolutions and Confusions in Android Apps. In *ACSAC*.
[77] Zhuolin Yang, Zhikuan Zhao, Boxin Wang, Jiawei Zhang, Linyi Li, Hengzhi Pei, Bojan Karlaš, Ji Liu, Heng Guo, Ce Zhang, and Bo Li. 2022. Improving Certified Robustness via Statistical Learning with Logical Reasoning. In *NeurIPS*.
[78] Jin Zhang, Chennan Zhang, Xiangyu Liu, Yuncheng Wang, Wenrui Diao, and Shanqing Guo. 2021. ShadowDroid: Practical Black-box Attack against ML-based Android Malware Detection. In *ICPADS*.
[79] Xinyang Zhang, Ningfei Wang, Hua Shen, Shouling Ji, Xiapu Luo, and Ting Wang. 2020. Interpretable Deep Learning under Fire. In *USENIX Security Symposium*.
[80] Xiaohan Zhang, Yuan Zhang, Ming Zhong, Daizong Ding, Yinzhi Cao, Yukun Zhang, Mi Zhang, and Min Yang. 2020. Enhancing State-of-the-art Classifiers with API Semantics to Detect Evolved Android Malware. In *ACM CCS*.
[81] Kaifa Zhao, Hao Zhou, Yulin Zhu, Xian Zhan, Kai Zhou, Jianfeng Li, Le Yu, Wei Yuan, and Xiapu Luo. 2021. Structural Attack against Graph Based Android Malware Detection. In *ACM CCS*.

https://t.me/learningnets

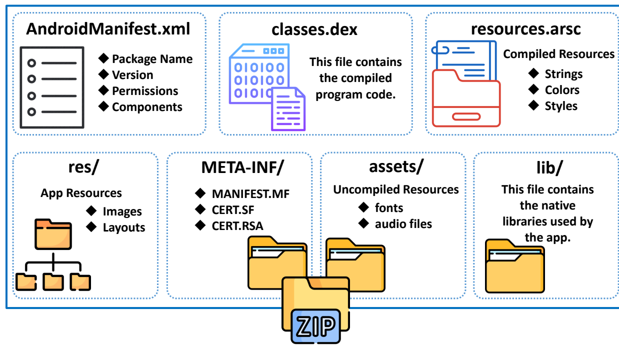

Figure 10: The APK file structure is a zip archive containing all the resources, assets, and compiled code needed for an application to run on an Android device.

A APK FILE STRUCTURE

As shown in Figure 10, the APK file structure is designed to encapsulate all the necessary components of an application, including code, resources, and metadata. It usually contains the following key components: *AndroidManifest.xml* is a crucial XML file of an APK file. as it contains essential information about the application, such as its package name, version, permissions, and list of components (e.g., activities, services, broadcast receivers, and content providers). The Android system uses the file to manage and execute the application correctly. *classes.dex* is the DEX file containing the compiled bytecode of the application. It is executed by the Android Runtime (ART) or the Dalvik Virtual Machine (DVM). *resources.arsc* contains compiled resources, such as strings, colors, and styles, used by the application. *res/* is the resources directory containing various files needed for the application's user interface, such as images, XML layouts, and string values. *assets/* contains any additional files the application needs, such as fonts, audio files, or other binary data *META-INF/* contains metadata about the APK file, such as digital signatures and certificate information. *lib/* contains the compiled libraries if an application uses native code written in C or C++. The APK file is essentially a compressed ZIP archive, which bundles all the above components together.

B FOUR ML-BASED AMD METHODS

Drebin. Drebin [13] takes into account a broad spectrum of features derived from both the manifest file and DEX codes. From the manifest file, it extracts four sets of string features, including hardware components, requested permissions, app components, and filtered intents. For the DEX codes, Drebin disassembles the *classes.dex* file into smali files and extracts an additional four sets of string features: restricted API calls, permissions, suspicious API calls, and network addresses. Subsequently, it obtains these eight feature sets from the training dataset and merges them into a binary feature vector. Drebin employs the Linear SVM classifier to discern patterns between benign applications and malware. Moreover, Drebin offers explanations for its decisions by utilizing the weights and detection scores of the Linear SVM classifier.

Drebin-DL. Drebin-DL [32] employs the same feature set as Drebin, but utilizes MLP as the classifier. Despite sharing the same feature space, Drebin-DL achieves superior Android malware detection performance compared to Drebin.[2]

MaMadroid. MaMadroid [45] focuses on the function call graph of Android applications to detect malware. By extracting the FCG, It is able to analyze the behavioral patterns of applications, thereby providing valuable insights for malware detection. To enhance robustness against API changes in the Android framework, it abstracts functions into different states based on package or family names, creating a higher level of abstraction less prone to minor changes or variations. Subsequently, it constructs a Markov chain model that captures the transition probabilities between states, i.e., families or packages of target functions. This model helps represent the dynamic behavior of applications, thus allowing the method to identify distinctive patterns exhibited by malicious applications. Finally, it trains a machine learning classifier, such as RF, SVM, or kNN, to detect malware.

APIGraph. Distinct from the aforementioned AMD methods, APIGraph [80] is a general framework designed to enhance AMD methods that use the function call-related features. Initially, APIGraph constructs a comprehensive knowledge graph about APIs by extracting API relationships from the Android API documentation. This knowledge graph provides an organized and structured representation of the API ecosystem, enabling a deeper understanding of the connections between different APIs. To enhance the representation ability of function call-related features and better capture the underlying semantics, APIGraph employs clustering algorithms to aggregate functions based on their relationships within the knowledge graph. When applied to specific AMD methods, APIGraph replaces individual functions with the corresponding cluster to which the function belongs. This replacement provides a higher level of abstraction, reducing the impact of minor variations and potential obfuscation techniques in the analyzed applications. For example, to augment Drebin, APIGraph substitutes the binary vector of API occurrence with the one representing API cluster occurrence.

C FRAMEWORK ALGORITHM

The primary procedure of AdvDroidZero is shown in Algorithm 2. Within this algorithm, g represents the target classifier, N stands for the query budget, $\mathcal{D}$ symbolizes the Android documentation, $\mathbb{B}$ signifies the set of randomly sampled benign applications, and $\mathbb{P}$ denotes the malware perturbation set.

AdvDroidZero first builds the malware perturbation set from the Android documentation and the randomly sampled benign applications (lines 1). Then AdvDroidZero organizes the perturbations within the perturbation set into the perturbation selection tree (line 2). For a malicious application, AdvDroidZero initially obtains the initial malicious confidence (line 4). Throughout each iteration, AdvDroidZero samples a perturbation from the perturbation selection tree (line 6) and implements the perturbation within the APK (line 7). Subsequently, the perturbed malware is submitted to the target classifier (line 8). In the event that the target classifier is caused

[2]Though the primary focus of Drebin-DL [32] is on model robustness, it also introduces an improved AMD method.

https://t.me/learningnets

Algorithm 2 AdvDroidZero

Input: Target classifier g; Target malicious application z; Query budget N; Android documentation $\mathcal{D}$; Benign applications $\mathbb{B}$.
Output: Adversarial application z'.

1: $\mathbb{P} \leftarrow$ BuildPerturbation($\mathcal{D}$, $\mathbb{B}$) ▷ Build the malware perturbation set.
2: $\mathbb{T} \leftarrow$ InitTree($\mathbb{P}$) ▷ Develop the perturbation selection tree
3: $q \leftarrow 0$ ▷ q represents the query times.
4: $y \leftarrow g(z)$ ▷ Obtain the initial malicious confidence.
5: **while** $q \leq N$ **do**
6: $P \leftarrow$ SelectPerturbation($\mathbb{T}$) ▷ Sample the perturbation node P.
7: $z' \leftarrow$ Implement(P, z) ▷ Add the perturbation in problem space.
8: $y' \leftarrow g(z')$ ▷ Obtain the malicious confidence.
9: **if** y' is benign **then**
10: **return** z'; ▷ Attack successful.
11: **else**
12: $\mathbb{T} \leftarrow$ AdjustTree($\mathbb{T}$, P, y, y') ▷ Apply the adjustment policy.
13: **end if**
14: **if** $y' \leq y$ **then** ▷ Apply the evasive perturbation.
15: $z \leftarrow z'$
16: $y \leftarrow y'$
17: **end if**
18: $q \leftarrow q + 1$
19: **end while**
20: **return** Failure

to misclassify, the attack is deemed successful; otherwise, AdvDroidZero proceeds to adjust the perturbation selection tree (lines 9-13). Perturbations that result in increasing malware confidence are not recorded by the framework (lines 14-17).

D INTUITION VALIDATION

Our assumption that perturbations sharing semantic similarity are more likely to demonstrate equivalent evasion effectiveness is rooted in observations of benign Android applications. Such applications often utilize semantically similar manifest elements to facilitate specific functionalities. For example, a music-related application may necessitate audio capabilities for a device and declare both a *uses-feature* element with *android.hardware.audio.output* value and a *uses-feature* element with *android.hardware.audio.pro* value. Consequently, models trained on such data will recognize these patterns suggesting that both values contribute to the classification of the application as benign. Moreover, the only difference between AdvDroidZero and the baseline methods, i.e., MAB and RA, resides in the perturbation selection strategy, while the malware perturbation set remains the same across all methods. Therefore, the enhanced performance of AdvDroidZero can be attributed to this assumption to some extent.

To provide empirical validation of our assumption, we evaluate the effectiveness of the semantically related perturbations in our implemented ML-based AMD methods. To be specific, we randomly sample 100 malicious samples detected by the ML-based AMD methods in the test set in the primary dataset. Then, we randomly sample 100 pairs of semantically related perturbations from the malware perturbation set. For every pair of semantically related perturbation pair, we individually implement each perturbation in a pair into the malware sample and observe if they yield similar results, e.g., both of them decrease the confidence of the malware label. Through this analysis, we find that 64.18% of semantically related perturbation pairs against Drebin and 67.69% against APIGraph yield similar rewards. These results suggest that semantically related perturbations likely possess similar evasion effectiveness, thereby supporting our assumption.

E MALWARE MANIPULATION REQUIREMENTS

In the context of malware, manipulations must adhere to the following requirements.

R1: All-features Influence. Given that the target malware detectors are entirely unknown, manipulations should be capable of affecting different feature categories (e.g., manifest-related feature and code-related feature).

R2: Functional Consistency. The functionality of the manipulated malware should maintain its malicious functional consistency before and after the manipulation.

R3: Robust to Pre-Processing. Manipulations in the problem space should exhibit robustness against pre-processing techniques. This means they should not be eliminated by static analysis inspection [27, 47].

Existing manipulation methods are summarized below.

Inserting Dead Codes. To preserve functional consistency, Chen *et al.* [25] insert dead codes (e.g., no-op calls) into smali files, while Bostani *et al.* [19] insert code gadgets into unreachable program positions (e.g., after return statements). However, these codes could be easily detected and eliminated, violating **R3**. For instance, Karer *et al.* [35] propose an SSA-based static analysis method for dead code elimination.

Opaque Predicates. Pierazzi *et al.* [53] devise a method called opaque predicates to insert new API calls. Specifically, opaque predicates are conditional statements that are practically impossible to be true but difficult to determine for static analysis. However, it is straightforward to approximate the probability of the truth of these conditional statements. By doing so, static analysis tools may remove the opaque predicates [46, 54], causing them to violate **R3**.

Try-Catch Wrappers. Li *et al.* [38] employ try-catch blocks to wrap new function calls to evade malware detection. To maintain **R2**, this method inserts try-catch blocks with statements that always trigger exceptions. New function calls after the exception-triggering statements will never be executed due to the early exceptions. Hence, it is possible for static analysis to detect such dead code by identifying the exceptions. Moreover, this method may introduce extra features (e.g., the abuse of try-catch statements), which could be used to reveal malicious APKs. Therefore, this method violates **R3**.

F MANIPULATION EXAMPLES

We employ adversarial Android malware generated by AdvDroidZero against the VirusTotal to illustrate the manipulations performed.[3] As a result, there are 15 antivirus engines that detect the original APK, but only 7 antivirus engines can detect the adversarial APK.

We utilize Apktool to extract the *AndroidManifest.xml* of the two APKs. Listing 1 presents the *AndroidManifest.xml* of the original

[3]The VirusTotal analysis link of the original APK at https://www.virustotal.com/gui/file/094282e6c9a78395787c50f0880759c44bb16a7394e852a24b7b4ad63d9b5fb7. The VirusTotal analysis link of the adversarial APK at https://www.virustotal.com/gui/file/5558470e0bcfd3c137921aa49f3d5a92c7f513ef4aa418c96237544f90841b82.

https://t.me/learningnets

```xml
<?xml version="1.0" encoding="utf-8" standalone="no"?><manifest xmlns:android="http://schemas.android.com/apk/res/android" package="com.lifeapps.modernwindowdesign" platformBuildVersionCode="22" platformBuildVersionName="5.1.1-1819727">
  <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
  <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
  <uses-permission android:name="android.permission.CAMERA"/>
  <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
  <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
  <uses-permission android:name="android.permission.INTERNET"/>
  <application android:allowBackup="true" android:icon="@drawable/ic_launcher" android:label="@string/app_name" android:theme="@style/AppTheme">
    <activity android:configChanges="keyboardHidden|orientation|screenSize" android:hardwareAccelerated="true" android:label="@string/app_name" android:name="com.lifeapps.modernwindowdesign.MainActivity" android:screenOrientation="user">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
    <activity android:configChanges="keyboard|keyboardHidden|orientation|screenLayout|screenSize|smallestScreenSize|uiMode" android:name="com.google.android.gms.ads.AdActivity"/>
    <meta-data android:name="com.google.android.gms.version" android:value="@integer/google_play_services_version"/>
  </application>
</manifest>
```

Listing 1: Orginal AndroidManifest.xml of malicious APK

```xml
<?xml version="1.0" encoding="utf-8" standalone="no"?><manifest xmlns:android="http://schemas.android.com/apk/res/android" package="com.lifeapps.modernwindowdesign" platformBuildVersionCode="22" platformBuildVersionName="5.1.1-1819727">
  <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
  <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
  <uses-permission android:name="android.permission.CAMERA"/>
  <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
  <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
  <uses-permission android:name="android.permission.INTERNET"/>
  <application android:allowBackup="true" android:icon="@drawable/ic_launcher" android:label="@string/app_name" android:theme="@style/AppTheme">
    <activity android:configChanges="keyboardHidden|orientation|screenSize" android:hardwareAccelerated="true" android:label="@string/app_name" android:name="com.lifeapps.modernwindowdesign.MainActivity" android:screenOrientation="user">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
    <activity android:configChanges="keyboard|keyboardHidden|orientation|screenLayout|screenSize|smallestScreenSize|uiMode" android:name="com.google.android.gms.ads.AdActivity"/>
    <meta-data android:name="com.google.android.gms.version" android:value="@integer/google_play_services_version"/>
    <receiver android:exported="true" android:name="com.seattleclouds.scm.PushManagerReceiver" android:process=":sojVFdeGbnNtwndwghJH">
      <intent-filter>
        <action android:name="com.gnip.RRpEgRJXMCYKeff"/>
      </intent-filter>
    </receiver>
    <provider android:authorities="com.gnip.iGAMPzaW" android:enabled="true" android:exported="true" android:name="com.seattleclouds.util.InternalFileContentProvider" android:process=":LOPuTPUUlLRagJmqNRnj"/>
    <activity android:exported="true" android:name="coma.sbda.uygi.Vctq">
      <intent-filter>
        <action android:name="JcqPliYQ"/>
        <category android:name="android.intent.category.INFO"/>
      </intent-filter>
    </activity>
  </application>
  <uses-feature android:name="android.software.managed_users"/>
  <uses-permission android:name="android.permission.USE_ICC_AUTH_WITH_DEVICE_IDENTIFIER"/>
</manifest>
```

Listing 2: Perturbed AndroidManifest.xml of malicious APK

APK, and Listing 2 depicts the adversarial APK. Notably, the code after line 17 in Listing 2 has been added by AdvDroidZero. The modifications introduced by AdvDroidZero include the injection of a *uses-feature* element, a *uses-permission* element, and a *category* element. Additionally, AdvDroidZero injects a broadcast receiver component and a content provider component.

G PERTURBATION CLUSTERING

As discussed in Section 3.2, semantically related perturbations are more likely to yield similar rewards for an attack. To capitalize on this characteristic, AdvDroidZero develops a customized clustering algorithm to group semantically related perturbations. Based on the central insight that semantically related permissions or actions share similar text representations, the algorithm continuously merges the two most similar clusters according to their keyword similarity K_s until no K_s of existing cluster pairs surpasses the pre-defined threshold T_s.

As illustrated in Equation 2, the keyword similarity of a cluster pair c_i, c_j is calculated, where $Overlapping(c_i, c_j)$ counts the number of identical keywords in the two clusters.

$$K_s = \frac{Overlapping(c_i, c_j)}{\min(|c_i|, |c_j|)}. \quad (2)$$

As depicted in Algorithm 3, AdvDroidZero initializes every perturbation in the pool P as a cluster. Subsequently, AdvDroidZero computes the K_s for every cluster pair (lines 4-5). Then, the cluster pair with K_s exceeding T_s is merged (lines 6-9). The process terminates when the perturbation groups no longer change (lines 11-13).

Algorithm 3 Perturbation Clustering

Input: Perturbation set $\mathbb{P}$; Keyword similarity threshold T_s
Output: Perturbation groups C'.

1: $C \leftarrow$ Init($\mathbb{P}$) ▷ Initialize the clustering group.
2: $C' \leftarrow C$
3: **while** True **do**
4:   **for** $c_i, c_j \leftarrow$ Combinations(C, 2) **do**
5:     K_s = KeywordSim(c_i, c_j) ▷ Compute the keyword-based similarity.
6:     **if** $K_s > T_s$ **then**
7:       $C' \leftarrow$ MergeGroup(C, c_i, c_j) ▷ Merge the similar groups.
8:       **break**
9:     **end if**
10:   **end for**
11:   **if** $C = C'$ **then**
12:     **break**
13:   **end if**
14: **end while**
15: **return** C'

Table 2: Detection performance of the four ML-based AMD methods.

AMD Methods	Classifier	Precision	Recall	F1	AUROC
Drebin	SVM	0.73	0.88	0.80	0.96
Drebin-DL	MLP	0.73	0.89	0.80	0.96
APIGraph	SVM	0.73	0.88	0.80	0.96
MaMadroid	RF	0.76	0.80	0.78	0.94
	3-NN	0.65	0.72	0.69	0.90

H DETAILS OF TARGET MODEL

Implementation. For Drebin, we utilize the same implementation provided by Pierazzi *et al.* [53]. For Drebin-DL, we employ the same feature extractor as Drebin and implement an MLP using PyTorch, following the descriptions in Grosse *et al.* [32]. As for APIGraph, we make use of its official code available at https://github.com/seclab-fudan/APIGraph. Regarding MaMadroid, we re-implement it using Androguard [2], adhering to the descriptions from the original paper and the configurations from the official code, which can be accessed at https://bitbucket.org/gianluca_students/mamadroid_code/src/master/.

Detection Performance. The TPR values for Drebin, Drebin-DL, APIGraph, MaMadroid with RF, and MaMadroid with 3-NN are 0.88,

https://t.me/learningnets

Algorithm 4 MAB

Input: Target classifier g; Target malicious application z; Query budget N; Malware perturbation set $\mathbb{P}$.
Output: Adversarial application z'.

```
 1: B ← BuildBandit(P)                         ▹ Build the multi-arm bandit machine P.
 2: q ← 0                                      ▹ q represents the query times.
 3: while q ≤ N do
 4:     P ← SelectPerturbation(B)              ▹ Obtain the perturbation P.
 5:     z′ ← Implement(P, z)                   ▹ Add the perturbation in problem space.
 6:     y ← g(z′)                              ▹ Obtain the model feedback.
 7:     if y is benign then
 8:         return z′;                         ▹ Attack successful.
 9:     else
10:         B ← UpdateBandit(B, P, y)          ▹ Update the bandit machine.
11:     end if
12:     q ← q + 1
13: end while
14: return Failure
```

0.89, 0.88, 0.80, and 0.72, respectively. Supplementary measures such as precision, recall, f1 score, and AUROC for the four ML-based AMD techniques are detailed in Table 2.

I BASELINE METHODS

MAB. In the original implementation, the MAB algorithm is designed specifically for Windows PE malware (source code available at: https://github.com/weisong-ucr/MAB-malware). To adapt this baseline MAB algorithm for generating adversarial Android malware, we follow the descriptions and code configurations provided in the relevant literature. Specifically, we treat the second layer in the perturbation selection tree as the arms of the multi-armed bandit machine. We consider all leaf nodes that belong to the descendants of the corresponding arm as the contextual information for the bandit machine. The detailed algorithm can be found in Algorithm 4.

RA. The random attack is to select perturbation from the perturbation space uniformly at random. The detailed procedure is shown in Algorithm 5. RA uniformly samples the malware perturbation in the malware perturbation set iteratively, injects the selected perturbations, and queries the target model with the perturbed application until success is achieved or the query budget depletes.

To enable fair comparison, we utilize the same malware perturbation set between AdvDroidZero and the baseline methods. The only difference between AdvDroidZero and the baseline methods, i.e., MAB and RA, lies in their approach to perturbation selection. AdvDroidZero employs a perturbation selection tree, MAB utilizes the multi-armed bandit algorithm, and RA makes random, indiscriminate perturbation selections. Consequently, the enhanced performance of AdvDroidZero can be ascribed to our strategic perturbation selection, while results of RA serve as a testament to our perturbation set.

J ACTUAL NUMBER OF SUCCESSFULLY GENERATED ADVERSARIAL ANDROID MALWARE

Table 3 provides the actual number of adversarial applications successfully generated by AdvDroidZero that can evade the detection by the targeted ML-based AMD method. Recognizing the bugs and corner cases present in the FlowDroid research prototype, as discussed by Pierazzi *et al.* [53], we encounter crashes during the modification of APKs with FlowDroid. Following the previous work [53], these crashes are not indicative of limitations in our method. Hence such instances are excluded from our ASR computation.

Algorithm 5 Random Attack

Input: Target classifier g; Target malicious application z; Query budget N; Malware perturbation set $\mathbb{P}$.
Output: Adversarial application z'.

```
 1: q ← 0                                      ▹ q represents the query times.
 2: while q ≤ N do
 3:     P ← RandomSample(P)                    ▹ Random sample uniformly the perturbation P.
 4:     z′ ← Implement(P, z)                   ▹ Add the perturbation in problem space.
 5:     y ← g(z′)                              ▹ Obtain the model feedback.
 6:     if y is benign then
 7:         return z′;                         ▹ Attack successful.
 8:     end if
 9:     q ← q + 1
10: end while
11: return Failure
```

https://t.me/learningnets

Table 3: The number of adversarial Android malware successfully generated by AdvDroidZero, capable of evading the ML-based AMD method, selected from a random sample of 100 malware.

Attack Methods	Query Budgets	Target AMD Methods				
		Drebin	Drebin-DL	APIGraph	MaMadroid	
		SVM	MLP	SVM	RF	3-NN
AdvDroidZero	10	45	57	43	92	74
	20	60	56	56	88	84
	30	71	63	65	95	88
	40	74	58	71	98	85
MAB	10	37	38	24	85	81
	20	39	55	38	91	86
	30	44	54	41	93	82
	40	53	54	41	94	82
RA	10	28	36	24	85	79
	20	44	53	39	90	82
	30	53	49	44	94	96
	40	54	56	40	90	93

https://t.me/learningnets