Fighting Fire with Fire: Continuous Attack for Adversarial Android Malware Detection

Yinyuan Zhang, *School of Computer Science, Peking University; Key Laboratory of High Confidence Software Technologys (Peking University), Ministry of Education;* Cuiying Gao, *Huazhong University of Science and Technology; JD.com;* Yueming Wu, *Nanyang Technological University;* Shihan Dou, *Fudan University;* Cong Wu, *Nanyang Technological University;* Ying Zhang, *Key Laboratory of High Confidence Software Technologys (Peking University), Ministry of Education; National Engineering Research Center of Software Engineering, Peking University;* Wei Yuan, *Huazhong University of Science and Technology;* Yang Liu, *Nanyang Technological University*

https://www.usenix.org/conference/usenixsecurity25/presentation/zhang-yinyuan

This paper is included in the Proceedings of the 34th USENIX Security Symposium.

August 13–15, 2025 • Seattle, WA, USA

978-1-939133-52-6

Fighting Fire with Fire: Continuous Attack for Adversarial Android Malware Detection

Yinyuan Zhang[1,5#], Cuiying Gao[2,7#], Yueming Wu[3*], Shihan Dou[4], Cong Wu[3], Ying Zhang[5,6], Wei Yuan[2], Yang Liu[3]

[1]*School of Computer Science, Peking University, Beijing, China*
[2]*Huazhong University of Science and Technology, China*
[3]*Nanyang Technological University, Singapore*
[4]*Fudan University, China*
[5]*Key Laboratory of High Confidence Software Technologys (Peking University), Ministry of Education, China*
[6]*National Engineering Research Center of Software Engineering, Peking University, China*
[7]*JD.com, China*

Abstract

The pervasive adoption of Android as the leading operating system, due to its open-source nature, has simultaneously rendered it a prime target for malicious software attacks. In response, various learning-based Android malware detectors (AMDs) have been developed, achieving notable success in malware identification. However, these detectors are increasingly compromised by adversarial examples (AEs), which are subtly modified inputs designed to evade detection while maintaining malicious functionality. Recently, advanced adversarial example generation tools have been introduced that can reduce the efficacy of popular detectors to 1%. In this background, to address the critical need for more resilient AMDs, we propose a novel defense mechanism, Harnessing Attack Generativity for Defense Enhancement, i.e., HagDe. HagDe involves applying iterative perturbations in the direction of gradient ascent to all samples, aiming to exploit the high sensitivity of AEs to perturbations. This method enables the detection of adversarial samples by observing the disproportionate increase in the loss function following minor perturbations, distinguishing them from regular samples. To evaluate HagDe, we conduct an extensive evaluation on 15,000 samples and 15 different attack patterns. Results show that HagDe can achieve a defense effectiveness of 88.5% on *AdvDroidZero* and 90.7% on *BagAmmo*, representing an increase of 32.45% and 11.28%, respectively, compared to the latest defense method *KD_BU* and *LID*.

1 Introduction

Due to its open-source nature, Android has become the most popular operating system, yet it is also a prime target for malicious software attacks [14]. Currently, a variety of learning-based Android malware detectors (AMDs) have emerged [18] [34], achieving relatively good results in detecting Android malware. For example, MaMaDroid [34], Malscan [45] and Drebin [3] have all achieved high detection effectiveness in the ideal setting [17]. However, existing research reveals that learning-based malware detectors are vulnerable to attacks by adversarial examples (AEs) [22, 28]. As shown in Figure 1, AEs are inputs to learning-based models that have been intentionally designed or modified to cause the model to make a mistake. In the field of computer vision, AEs are typically designed to be imperceptible to the human eye [50], while in the domain of malware, AEs are crafted to maintain their functionality [28].

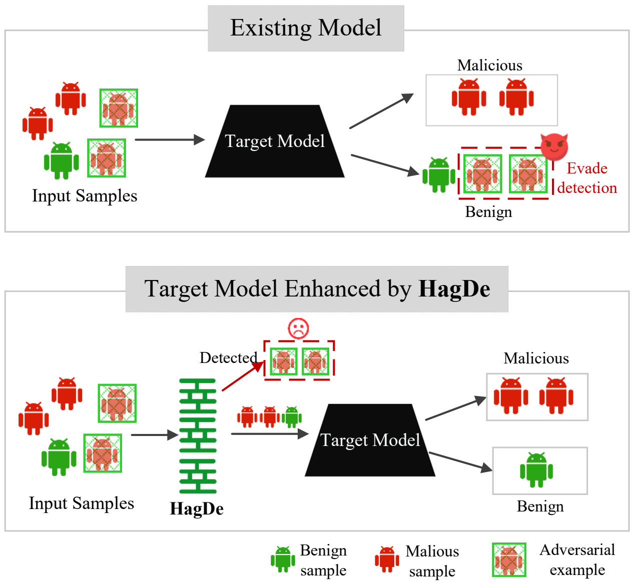

Figure 1: How HagDe Enhances AMD Target Model

Currently, adversarial example tools targeting AMDs have been proposed and possess significant disruptive potential against AMDs. For example, BagAmmo [30] introduces an adversarial multi-population co-evolution algorithm to attack AMD methods based on Function Call Graphs (FCG). It demonstrates an average attack success rate of over 99.9% on detection methods including MaMaDroid. Moreover, AdvDroidZero [22] introduces a query-based adversarial attack framework specifically designed for Android malware, operating under a zero-knowledge setting where the adversary lacks

*The first two authors contributed equally. Yueming Wu is the corresponding author.

information about the feature space, model parameters, and training dataset. AdvDroidZero demonstrates a success rate of approximately 90% against prominent learning-based Android malware detection methods, including Drebin, Drebin-DL, APIGraph, and MaMaDroid.

In this context, it is crucial to consider how to construct robust AMDs that can withstand attacks from AEs. Currently, various methods are proposed to enhance learning-based malware classifiers against adversarial attacks. In summary, these methods can be categorized into two types based on the target application of methods. The first way directly intervenes at various stages of the malware classifier [28], including Data Preprocessing, Feature Extraction, Classification, and Decision, to enhance the robustness of the malware classifier. For example, Li et al. [30] propose a robust Android malware detection framework, combining the Variational Autoencoder (VAE) and the Multi-Layer Perceptron (MLP) to classify malware examples during the classification phase. The second way involves enhancing the robustness of machine learning models through adversarial example detection [35, 50], adversarial training [33], and randomization-based defense [49], thereby aiding malware detectors in resisting adversarial attacks. For example, Liu and Hsieh [32] developed the Rob-GAN framework, which enhances discriminator robustness by training the discriminator to distinguish between clean, fake, and PGD-generated adversarial examples. In this paper, we specifically focus on augmenting the resilience of malware detectors against adversarial attacks via adversarial example detection. As shown in Figure 1, in this paper, we propose a novel defense mechanism, harnessing attack generativity for defense enhancement, i.e., HagDe. For input samples, HagDe preprocesses by filtering to identify adversarial examples. Only the filtered samples are then fed into the AMD for classification.

Currently, efforts to detect adversarial examples primarily focus on implementing a variety of proactive strategies, including network input regularization [37], output regularization [23] and k-NN search [13], among others. However, these defenses can still be circumvented by optimization-based attacks [8]. Beyond these methods, feature squeezing [50] is a classic approach that posits the input to deep neural networks inherently contains many "redundant" features, which facilitates the creation of adversarial examples by attackers. By comparing compressed and uncompressed inputs, adversarial examples can be detected. Moreover, more works have focus towards distinguishing features between adversarial examples and natural samples. Among these approaches, methods aim to distinguish adversarial images from natural images by leveraging features extracted from DNN layers or a learned encoder. Notable techniques include Kernel Density (KD) [15], Bayesian Uncertainty (BU) features [15], and Local Intrinsic Dimensionality (LID) [33], among others. However, these specific approaches, designed for image data, perform poorly in the domain of malware detection due to differences among features.

In practice, we find that in the process of generating adversarial examples for malware, attackers seek optimal perturbations to evade AMD detection more effectively. They typically attack along the gradient descent direction and cease perturbation once successful evasion is achieved. Given this principle, adversarial examples are closer to the classifier's decision boundary compared to regular samples, exhibiting higher sensitivity to perturbations, with their loss function increasing more rapidly.

Leveraging this characteristic, we propose a novel defense framework HagDe that proactively employs multi-stage iterative perturbations in the direction of gradient ascent where the attack likely generated from, to harvest multi-dimension loss features for classifier training, thereby enhancing detection performance. The crux of this method lies in detecting adversarial examples by increasing the sample's loss function. Specifically, minor perturbations are applied to each sample to observe how these perturbations affect the model's loss function. For adversarial examples, even very small perturbations can lead to a significant increase in the loss function since they are already near the decision boundary. By comparing the change in the loss function before and after perturbation, adversarial samples can be distinguished from regular samples. The advantage of this method is that it not only passively compares the differences between adversarial and regular samples on certain features but actively exploits the intrinsic characteristic of malware adversarial example generation their high sensitivity to perturbations. Such an approach may more effectively identify and defend against adversarial examples because it directly targets the key vulnerability in the adversarial example generation process.

Through extensive experiments on over 15,000 Android apps and 15 different attack patterns, HagDe has been proven to defend against state-of-the-art (SOTA) attack methods. Moreover, HagDe outperforms the SOTA adversarial detection method in the F1 score. Specifically, HagDe achieves defense effectiveness of 88.5% on AdvDroidZero and 90.7% on BagAmmo, representing an increase of 32.45% and 11.28%, respectively, compared to the latest defense method KD_BU [15] and LID [33].

In summary, our contributions are as follows:

- **Method**. We introduce a novel defense framework, HagDe, which leverages the generativity of attacks to enhance defense. This method preprocesses input samples to identify adversarial examples before they are classified by the detector.
- **Insight**. Our defense strategy capitalizes on a key characteristic of the adversarial malware sample generation process: high sensitivity to perturbations. By applying minor perturbations to samples and observing how these perturbations affect the model's loss function, we can distinguish between adversarial and regular samples. This method does not merely passively compare differences in certain fea-

tures between adversarial and regular samples but actively utilizes the intrinsic property of adversarial malware example generation—their high sensitivity to perturbations.

- **Effectiveness**. Through comprehensive experiments involving over 15,000 Android applications and 15 different attack combinations, HagDe has proven to be effective against SOTA attack methods. Specifically, HagDe achieves a defense effectiveness of 88.5% against AdvDroidZero and 90.7% against BagAmmo, marking an increase of 32.45% and 11.28%, respectively, compared to the latest defense methods KD_BU and LID.

The remainder of this paper is organized as follows. Section 2 presents a preliminary analysis of our background, threat model, and motivation. Section 3 describes our approach. Section 4 evaluates our tool by conducting detailed experiments. Section 5 discusses our work. Section 6 surveys our related work. Section 7 concludes the present paper.

2 Preliminary Analysis

2.1 Background

Learning-based Android Malware Detection. APK (Android Package Kit) files are used by the Android OS to distribute and install applications, containing essential elements like core code, resources, and metadata. Typically, an APK includes components such as *AndroidManifest.xml, classes.dex, resources.arsc*, and folders like *res/, assets/, META-INF/, and lib/*. Specifically, AndroidManifest.xml and *classes.dex* are crucial for the APK's functionality. The *AndroidManifest.xml* file contains detailed configuration details about the APK, including permissions and definitions for Activities, Services, and Broadcast Receivers. In contrast, *classes.dex* encapsulates the application's program semantics, storing the Dalvik bytecode to be executed on the Android Runtime environment.

In recent years, many representative learning-based methods for detecting Android malware have been proposed [4, 5, 7, 11, 17, 44, 46, 51, 54, 56]. The fundamental steps in constructing detectors include preprocessing, feature extraction, model training, and model testing. Methods constructing detection features based on static analysis have been widely advocated [3, 16, 34, 45] due to their ability to operate without running the app, examples of which include Drebin [3], MaMaDroid [34], APIGraph [53], and et al. The key steps in constructing detectors include feature extraction and model training. As depicted in Figure 2, program analysis tools [41] are initially leveraged to analyze the configuration information from *AndroidManifest.xml* and the behavior of applications from *classes.dex*. The analysis results are subsequently used to extract specific features for the machine learning phase. Different learning-based AMD methods select features in distinct ways. For instance, Drebin targets static features like permissions, while MaMaDroid focuses on dynamic features such as function calls. After feature extraction, these extracted features are then compiled into a feature vector. In the model

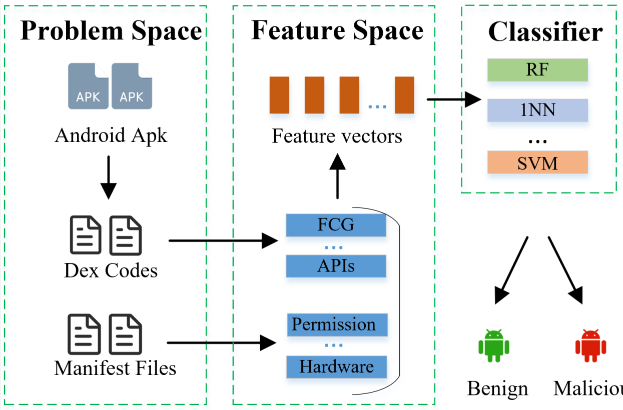

Figure 2: Learning-based Android Malware Detection

training phase, learning-based AMD methods use organized feature vectors to train classifiers that can differentiate between benign and malicious applications, enabling the trained model to identify malware. Recent studies indicate that under ideal settings, these detectors can achieve high detection efficacy [17]. Detailed learning-based AMD methods are in Appendix A.

Adversary Examples. Adversarial examples [30] refer to inputs subtly modified to cause erroneous outputs in machine learning models. In Android malware detection, these specifically involve meticulously crafted malware samples with minor alterations, designed to bypass AMD detection systems and remain undetected as malicious. Specifically, the adversary seeks to apply a sequence of perturbations P^* within the perturbation space $\mathcal{P}$ to a malware sample in the problem space, leading the target model $\mathcal{M}$ (learning-based AMD) to misclassify it as benign. More importantly, the functionality of the software remains unchanged. To minimize perturbation costs, finding an adversarial perturbation could be formulated as an optimization problem [22]:

$$\begin{gathered} P^* = \arg\min_{P \in \mathcal{P}} \text{cost}(P), \\ \text{s.t. } \mathcal{M}(\phi(X + P^*)) = b, \quad F(X) = F(X + P^*) \end{gathered} \tag{1}$$

where *cost*() represents the cost of generating perturbations (e.g., runtime overheads, efforts), $\phi()$ represents the feature extraction, $F()$ represents the malicious functionality verification.

2.2 Threat Model

An adversary aims to launch covert and targeted attacks to undermine the effectiveness of learning-based AMD against malware. As illustrated in Figure 3, the adversary seeks to perturb the malicious APK through modifications P^*, thereby influencing the feature vectors extrated by the machine learning-based AMD, and misleading it into classifying the APK as benign. The malicious APK is initially unpacked and decompiled into *smali codes*, *manifest files*, and other associated resources (e.g., *res/, assets/*). The adversary manipulates the *smali codes* or *manifest files* in accordance with the perturba-

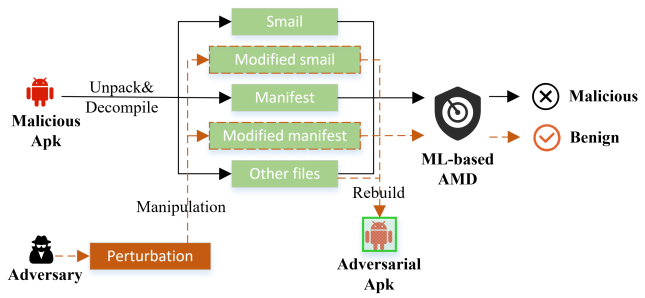

Figure 3: Threat Model of Adversarial Android Attack

Table 1: Adversarial Android Attack

Approach	Way of Generating Desired Perturbation
AdvDroidZero [22]	**Optimization-based:** employs a query-based perturbation selection tree.
BagAmmo [28]	**Optimization-based:** utilizes a multi-population co-evolution algorithm.
HIV-JSMA [9]	**Gradient-based:** utilizes the Jacobian-based Saliency Map Approach (JSMA).
HIV-CW [9]	**Gradient-based:** applies the Carlini-Wagner (CW) attack.

tion P^*. Only the modified *smali codes* or *manifest files* that successfully evades detection by learning-based AMD and can be rebuilt into a new APK file is considered a valid attack.

Drawing from the knowledge setting defined in previous work [22, 28], adversary lacks information about the target model parameters, and the training dataset. However, the adversary can query the target system and utilize the binary classification results (benign or malicious) to guide perturbation generation. This process is iteratively refined until an effectively evasive malware sample is produced. The generation of perturbation strategies is critical in the aforementioned attack process; however, current adversarial attack methods primarily differ in their approaches to generating these perturbation strategies, with each method employing distinct techniques.

As shown in Table 1, both AdvDroidZero and BagAmmo utilize optimization-based approaches for perturbation generation. AdvDroidZero introduces a perturbation tree, utilizing paths from the root to the leaf nodes as perturbation path ways. It continuously adjusts the weights of the nodes within the perturbation tree based on the target model's responses to search the optimal perturbation. In contrast, BagAmmo develops an evolutionary algorithm named Apoem, which incrementally discovers the desired perturbation through continuous refinement. For gradient-based methods, HIV employs the CW and JSMA algorithms, leveraging model gradients with respect to the input to generate an optimal perturbation that most significantly impacts the model's output features.

2.3 Motivation

Adversarial example attacks pose significant practical threats to AMD systems. Therefore, it is imperative to implement robust detection mechanisms to ensure the security and reliability of AMD systems. The essence of detecting

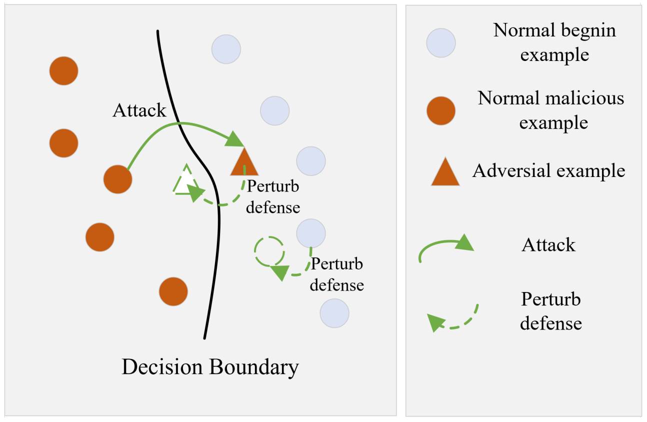

Figure 4: Attack and Defense of Android Malware

adversarial examples lies in distinguishing them from normal samples. As illustrated in Figure 4, the generation of adversarial examples involves shifting a *Normal malicious example* from one side of the decision boundary (of the AMD classifier) to the other. However, the implementation methods of different attacks vary significantly, primarily in the generation of the optimal perturbation, as discussed in Section 2.2. Therefore, identifying adversarial examples based on the generation of optimal perturbations might enable the design of detection methods tailored to specific attacks. However, achieving a general-purpose detection method remains challenging.

These attack methods share a common characteristic: seeking the optimal perturbation P^* while preserving the function's integrity. To achieve this optimal perturbation, adversaries iteratively query the target model using the repackaged APK's labels, adjusting the perturbation accordingly. To adhere to the specified constraints, two notable findings can be observed: (1) Adversaries cease further perturbation once an adversarial example is successfully generated, thereby minimizing computational overhead. Consequently, adversarial examples are closer to the decision boundary than regular samples. (2) Adversaries update the perturbation in the direction that most rapidly induces misclassification by the classifier, demonstrating greater sensitivity to perturbations near the decision boundary. To further elucidate finding 1, Figure 5 demonstrates that the predicted labels of adversarial samples are more readily flipped under the same perturbation level, whereas regular samples exhibit a lower flip rate. To validate finding 2, we use the model's loss function value as the evaluation metric. A loss below a certain threshold indicates correct classification, while a value above this threshold suggests misclassification. The higher the loss, the farther the sample is from the decision boundary. We iteratively apply perturbations of equal intensity to both normal and adversarial samples and observe the loss values. Figure 6 shows that with increasing perturbation steps, the loss for both sample types rises. Notably, the increase is significantly more pronounced for adversarial samples compared to normal samples. This suggests that by iteratively applying reverse perturbations and using the loss value as a metric for the position relative to

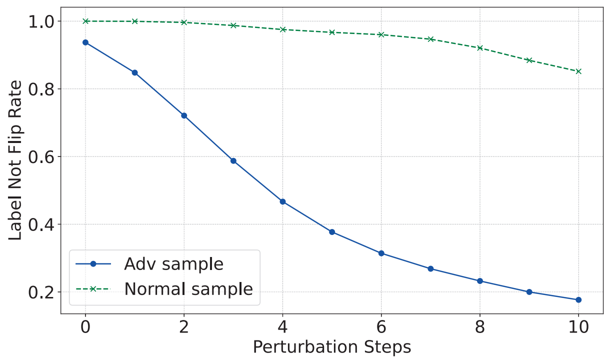

Figure 5: Label Not Flip Rate of Adversarial and Normal Examples

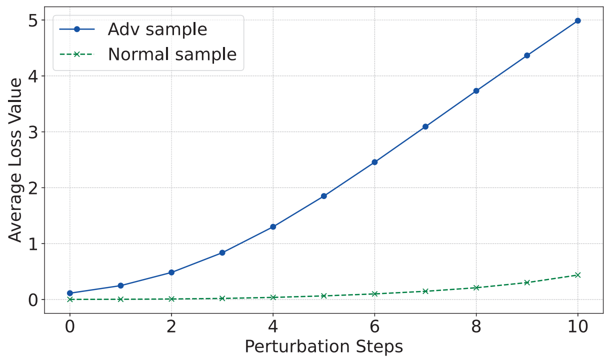

Figure 6: Loss Comparison: Adversarial Samples vs. Normal Samples with Increasing Perturbation Steps

the decision boundary, it is possible to distinguish between adversarial and normal samples.

As illustrated in Figure 4 and inspired by the aforementioned findings, an interesting and straightforward idea is to apply the same degree of perturbation to all samples (both adversarial and normal). By evaluating the position of samples relative to the decision boundary under varying perturbation intensities (using loss as the metric), we can distinguish between adversarial and normal samples.

3 Methodology

In this section, we first present an overview of the adversarial sample detection framework, HagDe, which comprises three stages. We then detail the methodology of HagDe across these stages.

3.1 Framework Overview

At a high-level overview, HagDe serves as a defense framework designed to enhance the resilience of AMD tools against adversarial sample attacks. The attack methods described in Section 2.2 involve APK-level perturbations, such as inserting code in APK, reverting to AMD-extracted features at feature-level and evading detection, while ensuring APK reconstruction. Conversely, defense only needs to determine if an APK is adversarial, allowing it to focus on feature-level detection with AMD-extracted features, avoiding the need for APK reconstruction. Shown as Figure 7, it operates through three stages: (1) train substitute models, (2) multi-stage perturbations, and (3) train classifier for detection. In the following, we outline each stage, starting with an analysis of the reasons for implementing this step and then detailing its implementation.

Train Substitute Model. AMD implementations currently rely on classifiers like SVM and Random Forest, which do not support gradient retrieval as neural network algorithms do. Therefore, training a substitute model aims to simulate the AMD classifier to obtain its gradients, enabling the calculation of loss for multi-stage perturbations.

Multi-Stage Perturbations. Motivated by Section 2.3, adversarial samples and normal samples differ in their position relative to the classifier's decision boundary and their sensitivity to perturbations. Therefore, HagDe applies multi-stage perturbations to APK feature iteratively, extracting features that indicate their proximity to the decision boundary to distinguish adversarial samples from regular ones.

Train Classifier for Detection. HagDe employs an effective classification algorithm to learn features related to multi-stage perturbations. For unknown Android software, the trained classification algorithm can directly predict whether it is an adversarial Android sample.

3.2 Stage1: Train Substitute Model

The substitute model aims to simulate the target model, and can evaluate an APK's position relative to the decision boundary. As illustrated in Figure 7, substitute model is trained using the APK feature and predicted labels (benign or malicious) generated by the target model (AMD).

To enable efficient training and inference of the substitute model, we designed a simple model architecture. In addition to standard fully connected layers and non-linear activation functions, we employ Dropout [39] during the training of substitute models to randomly deactivate neurons and enhance robust feature learning. This is because Dropout is a proven regularization technique in neural networks that effectively prevents overfitting and is widely used across various architectures. Detailed structure of the substitute model is depicted in Appendix C.

3.3 Stage2: Multi-Stage Perturbations

As discussed in Section 2.3, adversarial samples are closer to the decision boundary than normal samples, making them more sensitive to perturbations and more easily driven away from the decision boundary. However, two problems arise from utilizing the above findings to distinguish between adversarial and normal samples: (1) how to represent the proximity to the decision boundary, and (2) how to leverage the perturbation sensitivity of adversarial samples.

For problem 1, the loss value naturally measures the confidence in the model prediction y for the given original input x, which is closely related to the distance from the decision boundary. Consequently, we select the loss value as the key metric to measure the position of a sample relative to the

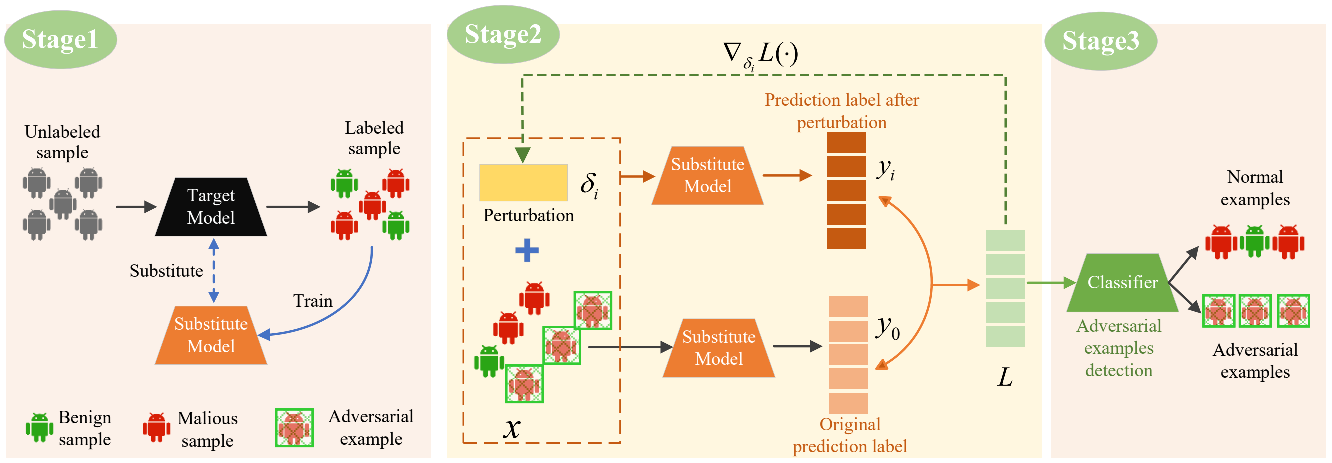

Figure 7: Framework of HagDe

decision boundary. When perturbations cause a sample to be misclassified, a higher loss value typically indicates that the sample is further from the decision boundary hence.

For problem 2, a straightforward and intuitive approach is to add perturbations to adversarial Android software due to its high sensitivity to such changes. However, the more challenging aspects include determining the direction of the perturbations, the perturbation way (i.e., the magnitude of the perturbations, and when to stop the perturbation process). Considering the aforementioned challenges, HagDe is designed to be a more targeted and proactive detection framework.

Perturbation Direction. Regarding the perturbation direction, our perturbations directly target the generation weaknesses of adversarial Android software. Since the generation of adversarial Android software typically aims to make the target model predict incorrectly (originally malware is incorrectly predicted as benign software), our perturbation direction is designed to move in the direction of the attack generation.

Given an input $\mathbf{x}$, a substitute model $\mathcal{M}$, and an initial predicted value y_0 from the model $\mathcal{M}$, we aim to perturb $\mathbf{x}$ such that the loss value for y_0 with respect to $\mathcal{M}$'s prediction increases along the direction of gradient ascent. Let $\mathcal{M}(\theta,\mathbf{x})$ represents the model prediction with model parameters θ and input $\mathbf{x}$, and $\mathcal{L}$ be the loss value $\mathcal{L}(\mathcal{M}(\theta,\mathbf{x}),y_0)$. The gradient of the loss function with respect to the input $\mathbf{x}$ is denoted as $\nabla_{\mathbf{x}}\mathcal{L}(\mathcal{M}(\theta,\mathbf{x}),y_0)$. Our goal is to find a perturbation δ that increases the loss value.

The perturbation δ can be defined as:

$$\delta = \varepsilon \cdot \text{sign}(\nabla_{\mathbf{x}}\mathcal{L}(\mathcal{M}(\theta,\mathbf{x}),y_0)), \tag{2}$$

where ε is a small positive scalar that controls the magnitude of the perturbation. The perturbed input $\mathbf{x}'$ is then given by:

$$\mathbf{x}' = \mathbf{x} + \delta. \tag{3}$$

This ensures that the model $\mathcal{M}$'s prediction on the perturbed input $\mathbf{x}'$ moves in the direction of gradient ascent with respect to the initial prediction y_0.

Perturbation Way. In terms of the perturbation way, the main influencing factors include the total number of perturbations, denoted by λ and the size of each perturbation, denoted by η. Unlike the conventional approach of applying a fixed perturbation once, we adopt an iterative approach to perform multiple perturbations. This is because the multi-stage perturbation loss values obtained through multiple iterations can better capture the varying rates of increase in response to perturbations between adversarial and normal samples. Consequently, it enhances robustness against various adversarial attacks for learning-based AMD.

Detailed, the perturbation $\boldsymbol{\delta}$ is initialized as:

$$\boldsymbol{\delta}^{(0)} \leftarrow \text{Uniform}\left(-\frac{\eta_0}{\sqrt{d}}, \frac{\eta_0}{\sqrt{d}}\right) \tag{4}$$

where η_0 is a fixed parameter representing the initial perturbation size, set to 0.005 here, d is the dim of input feature that are efficiently captured by AMD target model.

For the i-th iteration, where i represents the number of perturbation steps, the perturbation is updated and the loss is calculated as follows:

$$\boldsymbol{\delta}^{(i)} \leftarrow \boldsymbol{\delta}^{(i-1)} + \eta \cdot \frac{\nabla_{\boldsymbol{\delta}}\mathcal{L}(\mathcal{M}(\mathbf{x}+\boldsymbol{\delta}^{(i-1)}),y_0)}{\|\nabla_{\boldsymbol{\delta}}\mathcal{L}(\mathcal{M}(\mathbf{x}+\boldsymbol{\delta}^{(i-1)}),y_0)\|} \quad \text{for } i \geq 1 \tag{5}$$

$$\text{Loss}_i = \mathcal{L}(\mathcal{M}(\mathbf{x}+\boldsymbol{\delta}^{(i)}),y_0) \quad \text{for } i = 1,2,\ldots,\lambda \tag{6}$$

Thus, the sequence of loss values after each perturbation step is $\{\text{Loss}_0, \text{Loss}_1, \text{Loss}_2, \ldots, \text{Loss}_\lambda\}$.

As depicted in Fig 7, we perform multi-stage perturbation on all Android software, irrespective of whether the software are adversarial or normal. We iteratively update δ along the direction of gradient ascent. After each perturbation, the loss at that step is obtained, and then this step loss is further used to update δ for obtaining next step loss.

Algorithm 1 HagDe Algorithm Implementation

```
Input: Feature vector of training set $X_{\text{train}}$, Target classifiers predictions Label $Y_{\text{train}}$ (predict whether is benign or malicious), Normal and adversarial test set $X_{\text{test}} = \{X_{\text{normal\_test}}, X_{\text{adv\_test}}\}$.
Output: Predict whether it is an adversarial example $Y_{\text{test}}$
1  Stage 1: Train substitute model
   for Each sample $(x,y)$ in $X_{train}$ do
2      Use the following loss to train:
           $\text{Cross-Entropy Loss} = -\sum_{i=1}^{N}\sum_{k=1}^{2} y_{i2}\log(\hat{y}_{i2})$
3  end
4  return substitute model $\mathcal{M}$;
5  Stage 2: Multi-Stage Perturbations
   Initialize init_perturb_size $\eta_0$, perturb_size $\eta$,
   perturb_steps $\lambda$, and perturb_loss_list L
   for Each sample x in $X_{test}$ do
6      Initialize $l$
       $y_0, label_y \leftarrow$ getModelOutput($\mathcal{M}$, $x$)
       loss_feature $\leftarrow$ cross_entropy($y_0$, $label_y$)
       $l$.add(loss_feature)
       $\delta \leftarrow$ randomInitDelta($\eta_0$)
       for $i$ in $\lambda$ do
7          if $i == 0$ then
8              break
9          end
10         $pred \leftarrow$ getModelOutput($\mathcal{M}$, $x + \delta$)
            loss $\leftarrow$ cross_entropy($pred$, $label_y$)
            loss.backward()
            grad $\leftarrow$ calculateGradient($\delta$, loss)
            $\delta \leftarrow$ updateDelta($\delta$, grad, $\eta$)
            $y_i \leftarrow$ getModelOutput($\mathcal{M}$, $x + \delta$)
            loss_feature $\leftarrow$ cross_entropy($y_i$, $label_y$)
            $l$.add(loss_feature)
11     end
12     L.add($l$)
13 end
14 return L;
15 Stage 3: Train Classifier for Detection
   for Each sample feature x in L do
16     trainClassifier($x$, $y$) by 10-fold cross-validation
17 end
```

Specifically, the algorithm implementation is depicted as shown in Algorithm 1. The function getModelOutput($\mathcal{M}$, x) is utilized to obtain the model's prediction logits y_i and label $label_y$ for the sample x. The perturbation δ is initialized normal randomly with the size determined by the initial perturbation size η_0 through the function *randomInitDelta()*. For iterate λ times, obtaining the loss values at each step involves obtaining the prediction logits y_i for the perturbed sample $x+\delta$ using getModelOutput($\mathcal{M}$, $x+\delta$) . Subsequently, the cross-entropy loss between y_i and $label_y$ is computed. The gradient of the loss with respect to δ are used to update δ and calculate the next loss. For each Android software sample, a representative sequence of loss_feature values is obtained, calculated at each attack stage.

3.4 Stage3: Train Classifier for Detection

To mitigate the high computational demands and costs associated with training intricate neural networks, we employ machine learning models for the classification of adversarial Android malware. Machine learning models are used in our classification stage, employing a multi-dimensional loss feature for predicting the final adversarial detection. During training, multi-dimensional loss feature from the test set, which include both normal and adversarial Android software, serve as input. The machine learning algorithm is trained using these feature along with their corresponding labels, and the resulting trained model is stored for inference. In the inference phase, Android malware to be evaluated undergo the same process by feature extraction of AMD and first two stages by HagDe in order to generate feature vectors. These loss value vectors are fed into the model to predict outcomes, differentiating between normal (zero) and adversarial (one) software. We select machine learning models for their robust classification capabilities, leveraging algorithms such as 1-nearest neighbor (1NN), 3-nearest neighbor (3NN), random forest (RF), and support vector machines (SVM).

In summary, HagDe offers several notable advantages: Firstly, by leveraging the proximity of adversarial Android software to the decision boundary, in conjunction with their heightened sensitivity to perturbations, we are able to effectively quantify the differences between adversarial and benign software through the measurement of loss post-perturbation. Secondly, taking advantage of the inherent characteristics of the attack method, we mount counter-attacks oriented along the trajectory of the Attack to AMD, which thereby significantly accelerates convergence of feature differences. Finally, through the proactive implementation of multi-stage perturbations, we are able to capture a more comprehensive range of loss characteristics, and thus provide a more accurate reflection of the acceleration trends in perturbation response between adversarial and normal Android software.

4 Evaluation

4.1 Research Questions

We aim to answer the following research questions through experiments and analyses.

RQ1: How does the detection performance vary with different parameters, and different machine learning algorithms?

RQ2: Can HagDe outperform other state-of-the-art adversarial sample malware detectors?

RQ3: What is the runtime overhead during the process of adversial android malware detection by HagDe?

RQ4: How effective is HagDe in enhancing malware detection methods?

4.2 Experimental Settings

4.2.1 Attack Pattern

In the domain of attack on malware detection systems, the attacker collaborates with the targeted model (specifically, the AMD) to establish a distinct attack pattern.

AMD. In Section 2.1, we discuss existing Android malware detection systems. MaMaDroid, Drebin, and API-Graph are recognized as SOTA AMD detectors due to their high-performance detection capabilities, and they have been widely utilized. Furthermore, MaMaDroid can be further divided into two variants based on feature granularity: MaMaDroid-Family and MaMaDroid-Package. Due to the varying classification algorithms employed by each detector, this study focuses on selecting the classification algorithms that yield the best detection results [22, 34] for the AMD systems. Consequently, we choose the following SOTA AMD detection methods for this research: MaMaDroid-Family-RF (MaF), MaMaDroid-Family-APIGraph-RF (MaF-Api), MaMaDroid-Package-RF (MaP), MaMaDroid-Package-APIGraph-RF (MaP-Api), Drebin-SVM (Drebin), and Drebin-APIGraph-SVM (APIGraph).

Attacker. We discuss existing attack methods targeting AMD systems in Section 2.2. Currently, the mainstream attack strategies are primarily categorized into optimization-based attacks and gradient-based attacks. AdvDroidZero [22] and BagAmmo [28] are recognized as SOTA optimization-based attack methods due to their stealth and high success rates. These attacks are capable of generating real attacked APK files, with their real-world impact validated on the VirusTotal detection platform. Meanwhile, HIV-CW [9] and HIV-JSMA [9] are among the most established and currently SOTA gradient-based attack methods. Therefore, conducting experiments on these four representative and realistic attackers provides a more robust validation of adversarial sample detectors.

Combination of Attacker and AMD. An attacker that successfully circumvents a specific AMD system can be classified as an attack pattern. This study's attacker-AMD combinations draw on attack patterns from previous research. BagAmmo, HIV-CW, and HIV-JSMA have effectively countered MaMaDroid and APIGraph, leading us to integrate these attacks with the two AMD systems. Likewise, AdvDroidZero has targeted MaMaDroid-Family, Drebin, and Drebin-APIGraph in past studies, leading us to incorporate this attack design as well.

4.2.2 Baseline

We compare our proposed detectors HagDe based on continuous attack with several strong baselines in adversarial sample detection including FS [50], KD_BU [15], LID [33].

FS computes the feature squeezing of the input, extracts its prediction, and compares it to the original prediction. The further away they are, the more likely the input is adversarial.

KD_BU propose KD and BU for each class in the training data and then trained a binary classifier detector using densities and uncertainties features of clean, noisy, and AEs.

LID characterize the dimensional properties of the adversarial subspaces regions and proposed to use a property, called Local Intrinsic Dimentionaloty (LID). For natural or adversarial sample, this method calculated a LID score using extreme value theory at every DNN layer, and then classifier model was fitted on the LID features for adversarial detection.

4.2.3 Dataset and Metric

To assess adversarial sample detection in AMD, our experiments primarily utilized the AMD dataset, which consists of two parts: Data-MD and Data-AD.

Data-MD is a widely-used Malware detection set, collected from Androzoo [2] and labeled on VirusTotal [42], spanning the years 2016-2020. This dataset comprises 7,933 benign applications and 7,423 malware samples, totaling 15,356 applications. For Data-MD, we randomly selected 80% of samples from each category as the training set to train substitute models, and the remaining 20% as the testing set.

Data-AD is a adversial android malware sample made by us according the attack algorithms mentioned in the paper. Following the evaluation settings of BagAmmo and AdvDroidZero, which employ random sampling for evaluation, we thus conducted attacks on 400 randomly sampled Android malware instances that successfully evaded detection by AMD, and then the successfully attacked samples were then saved to form the final Data-AD dataset.

For the assessment of RQ1,RQ2, and RQ3, our evaluation dataset was composed of half the data from DATA-MD, complemented by an equal volume of normal samples randomly drawn from the test set of Data-AD. In the evaluation of RQ4, we selected a dataset of equivalent size from Data-MD and Data-AD, but distinct from that used in RQ2 to serve as the final detection sample.

To evaluate the performance of our Android adversarial sample detection, we employ standard metrics: Precision ($P = \frac{TP}{TP+FP}$), Recall ($R = \frac{TP}{TP+FN}$), and F1 score ($F1 = \frac{2PR}{P+R}$). Here, TP denotes true positives (correctly identified adversarial samples), FP denotes false positives (incorrectly identified adversarial samples), and FN denotes false negatives (adversarial samples missed by the detection).

4.3 RQ1: Parameters Selection

To illustrate the effectiveness of different classifier algorithms, and different parameters in detecting adversial sample, we set up comparison experiments in this subsection. In our approach, two pivotal parameters are considered within multi-stage perturbations, denoted as λ for the perturbation step count and η for the perturbation size. Both excessively low and high values of these parameters can compromise the algorithm's performance or execution efficiency. Additionally, the

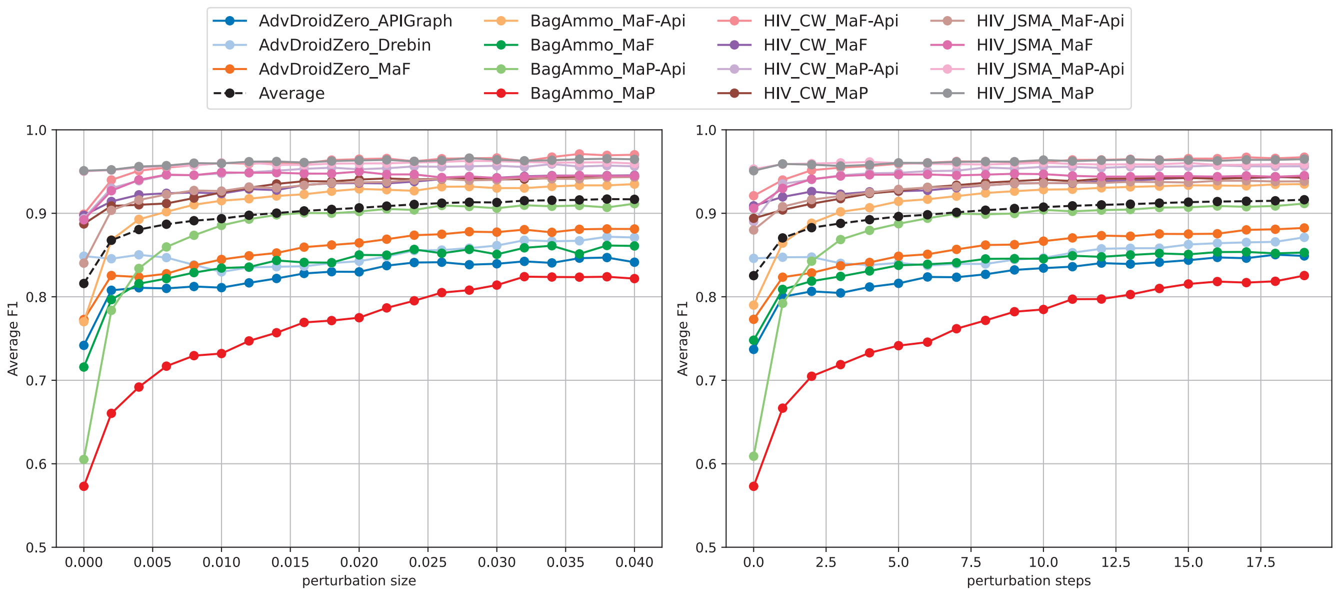

Figure 8: The Average F1 Score Assessment Across Parameters: Perturbation Step and Perturbation Size

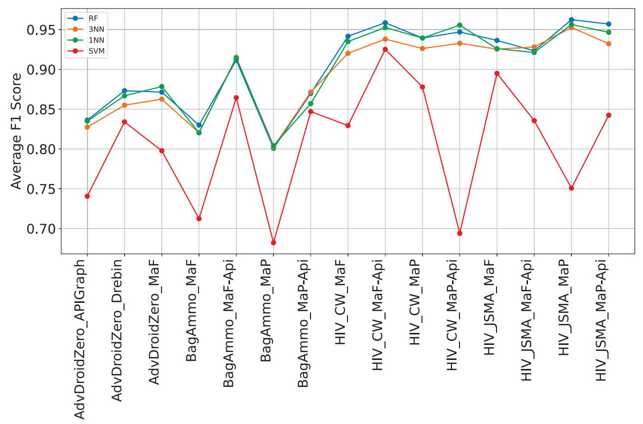

Figure 9: The Average F1 Score Assessment Across Attack Pattern

choice of classification algorithm impacts the utilization of loss features derived from the perturbed data. To investigate the impact of the aforementioned parameters and classifier algorithms on adversarial Android malware detection performance, we conducted extensive experiments across 15 types of attack patterns. The parameter λ was varied between 0 and 19, η was varied between 0 and 0.04 in increments of 0.002, and the classification algorithms was tested include 1NN, 3NN, RF, and SVM.

Selection of Classifier. Initially, we examined the classification algorithms, specifically 1NN, 3NN, RF, and SVM, under 15 distinct attack patterns. In various attack scenarios, across different perturbation frequencies and intensities, the average F1 scores for 1NN, 3NN, RF, and SVM classifiers were sequentially observed to be 0.901, 0.894, 0.904, 0.809, respectively. Additionally, as illustrated in Figure 9, RF classification algorithm exhibits the most optimal performance was observed with the highest frequency among the four, thus selected as our default classifier algorithm.

Selection of Param η. After determining the use of the RF classification algorithm, we conducted an in-depth investigation into the impact of varying parameter values on performance. As depicted in Figure 8, for parameter η, a discernible trend has been observed across all attack patterns, indicating that the detection performance consistently improves with the increase in perturbation size η. This consistent performance enhancement trend underscores the general applicability of HagDe in defending against various Android adversarial attacks. HagDe iteratively perturbs inputs to acquire multi-stage loss values for classifier training, culminating in an average F1 score surpassing 0.8 for detection performance. As indicated by the black line, the average F1 performance across the 15 attack mode exhibited a consistent upward trend with gradually plateauing as the perturbation size increased. When the perturbation size reached 0.032, the overall performance enhancement slowed significantly, and in some attack patterns, the detection F1 score even began to decline. Given that higher perturbation size also incur greater time overhead, we choose to fix this parameter at 0.032.

Selection of Param λ. For parameter λ, the detection performance of the algorithm demonstrates an enhancement and convergence trend as the number of perturbation steps increases, similar to the parameter η. When the perturbation steps reach 15, the overall performance enhancement also gradually decelerated. we fix this parameter at 15 under the same performance and time overhead considerations. When λ is set to 0, no multi-perturbations are applied, and detection relies solely on the init loss of adversarial and regular samples. In gradient-based attack patterns like *HIV_JSMA_MaP*, a λ

Table 2: Adversarial Android Malware Detection

Method	Metric	AdvDroidZero			BagAmmo				HIV-CW				HIV-JSMA				Avg.
		MaF	Drebin	Drebin-Api	MaF	MaF-Api	MaP	MaP-Api	MaF	MaF-Api	MaP	MaP-Api	MaF	MaF-Api	MaP	MaP-Api	
FS	Acc	57.1	55.6	56.7	65.1	72.8	58.6	50.6	60.9	79	53.3	62.3	69.5	54.8	27.2	50	58.2
	F1	69.1	65	62	72.4	72.5	40.3	65.6	70.6	81.5	66.1	71.3	67.6	49	5.8	0	57.3
	P	54	53.6	55.3	59.9	73.1	72.2	50.3	56.6	72.7	51.9	57.5	72.2	56.3	8.3	0	52.9
	R	96.2	82.4	70.6	91.5	72	28	94.3	93.9	92.7	91	93.5	63.5	43.4	4.5	0	67.8
KD_BU	Acc	76.7	76.8	79.7	65.9	85.2	43	55.2	92.9	75.9	51.9	61.6	88	89.5	49.5	58.5	70.0
	F1	75.1	74.9	78.7	68.5	84.4	39.1	53.9	92.7	73.3	32.5	62.2	87.1	89.5	32	61.5	67.0
	P	80	82.7	81.5	63.8	88.9	39.6	55	93.5	81.9	29.3	60.9	93.5	90	56.5	57.2	70.3
	R	71.2	70.3	77.6	74.9	81	43.9	58.9	92.3	66.9	39.1	65.1	82	89.3	23.5	67	66.9
LID	Acc	80.1	78.8	80.1	75.6	**96.5**	63.9	92.6	89.6	90.2	92.8	69.4	89.8	93.6	91	87.5	84.8
	F1	80.7	77	78.7	75.9	**96.6**	63.4	92.5	89.5	90.7	92.5	64	89.4	93.7	90.6	88.4	84.2
	P	79.4	85.6	84.8	74.9	**95.3**	65.7	93.9	90.1	89.3	94.1	74.1	92.1	92.2	94.3	83.7	86.0
	R	82.7	71.9	74.4	77.4	**98**	63.6	91.9	89.4	92.6	91.6	57.2	87	95.4	87.5	94	83.6
HagDe (Ours)	Acc	**89.4**	**90**	**86.5**	**86.9**	94.8	**88.2**	**92.6**	**95.2**	**96.9**	**95.2**	**96.5**	**94**	**95.4**	**96.2**	**96.2**	**92.9**
	F1	**89.8**	**89.6**	**86.1**	**86.9**	94.7	**88.5**	**92.8**	**95.3**	**96.9**	**95.1**	**96.4**	**94**	**95.2**	**96.2**	**96.2**	**92.9**
	P	**86.7**	**91.7**	**88.6**	**87.7**	95.2	**86.6**	**91.1**	**94.4**	**97.1**	**96.7**	**96.9**	**93.7**	**96.9**	**96.6**	**97.2**	**93.1**
	R	**93.5**	**88.8**	**84**	**86.4**	94.5	**91.2**	**95.6**	**96.4**	**96.9**	**93.9**	**96.2**	**94.5**	**93.8**	**96**	**95.5**	**93.1**

of 0 can achieve an F1 score over 0.95. However, multiple perturbations further boost detection. For optimization-based attacks, such as *BagAmmo_MaP*, the dependence on gradients is less pronounced, resulting in an even greater improvement from multiple perturbations.

Therefore, considering the increased computational cost with larger perturbation sizes and step counts, and the observed performance convergence, we determine λ = 15 and η = 0.032 are the optimal parameters.

Answer to RQ1*: The detection results based on the RF model demonstrated optimal average F1 scores across various attack patterns. The detection performance of the algorithm progressively improves with the increase in λ and η . On average, this enhancement gradually plateaued after λ reaches 15 and η reaches 0.032.*

4.4 RQ2: Effectiveness

To conduct a more comprehensive evaluation of HagDe's adversarial detection performance, we focus on two main aspects. On the attack side, we employ SOTA attack methods targeting target model, including AdvDroidZero, BagAmmo, HIV-CW, and HIV-JSMA. On the detection side, we compare HagDe with SOTA adversarial sample detection methods, which have demonstrated outstanding detection performance. Above these methods, HagDe, LID, and KD_BU respectively acquire the loss value feature, local intrinsic dimensionality feature, and kernel density combined with binary uncertainty features of the samples. These features are then utilized to train a classifier for the ultimate prediction. The distinguishing factor is that FS employs a threshold-based detection method by comparing the compressed features with the original features. Consequently, the results for FS are derived directly from the test set composed of normal and adversarial samples, whereas the outcomes for the other three methods are obtained through 10-fold cross-validation on same dataset.

Table 2 shows the detection results including accuracy, recall, precision and F1-score under various attack patterns. HagDe achieved the highest F1 score across 14 attack patterns, with the exception of the BagAmmo_MaF-Api attack pattern. The average F1 metric demonstrates that our method outperforms KD_BU and LID by 38.7% and 10.3%, respectively. Across 15 distinct attack patterns, our method consistently achieved F1 detection values exceeding 85%, indicative of its robustness against adversarial attacks.

For FS, its performance is relatively modest when confronted with adversarial attacks on Android software. The reason behind may be FS was originally proposed for detecting adversarial samples in the image domain, where features are continuous, unlike the discrete nature of Android malware features, harder to detect. Consequently, the feature compression-based method of FS proves ineffective under many attack patterns, especially the attack pattern of *HIV-JSMA_MaP-Api*. This finding highlights the discrete nature of feature information in Android malware detection, rendering many methods potentially ineffective initially working well in continuous spaces. Nonetheless, HagDe places greater emphasis on the calculation of loss values, demonstrating efficacy even within discrete data spaces. KD_BU exhibits inconsistent performance. It performs well with Family-level features (11×11), achieving an F1 score of 92.7% under the *HIV-CW_MaF* attack pattern. However, its effectiveness diminishes significantly with Package-level features (368×368) due to the finer granularity of features, where it frequently fails, resulting in an F1 score below 50%. Among the three baselines, LID demonstrates the most superior overall performance. Its relatively generic features enable it to achieve commendable detection results across most attack modalities, particularly excelling against *BagAmmo_MaF-Api*, where it attains the highest F1 score of 96.6. However, it exhibits suboptimal performance when confronted with data characterized by a higher dimensionality, yielding scores of 63.4 and 64 under the attack modalities of *BagAmmo-MaP* and *HIV-JSMA_MaP-Api*, respectively.

Overall, HagDe, KD_BU and LID show significantly better defense capabilities against HIV-CW and HIV-JSMA compared to AdvDroidZero and BagAmmo. This may attributed to the fact that HIV-CW and HIV-JSMA are designed to employ gradient-based attacks, rendering these features more susceptible to detection by the employed defense mechanisms. In contrast, AdvDroidZero employs a tree search algorithm to navigate and optimize perturbations in the problem space, ensuring that the perturbations are positioned to maximize their impact. Meanwhile, BagAmmo employs genetic algorithms to iteratively refine and select the most effective perturbations, utilizing principles of natural selection and evolution for optimal adversarial modifications. These perturbation-based attack methods pose a greater challenge to defense mechanisms. Consequently, HagDe exhibits robust detection capabilities across a diverse array of attack patterns. Detailed false positives and false negatives analysis of HagDe are depicted in Appendix D.

***Answer to RQ2**: The experimental results show that HagDe can achieve a defense effectiveness of 88.5% on AdvDroidZero and 90.7% on BagAmmo, representing an increase of 32.45% and 11.28%, respectively, compared to the latest defense method KD_BU and LID.*

4.5 RQ3: Efficiency

In this section, we pay attention on the runtime overhead of HagDe. For real-time detection systems, which encompass both AMD and adversarial malware detection. Existing AMD processes have implemented a relatively time-consuming feature extraction workflow from an Android APK (generating APK feature). Thus, this experiment primarily concerns the efficiency of adversarial malware detection based on APK feature. The detection efficiency of adversarial malware detection primarily depends on the size of the APK feature (e.g., 11x11 for MaF, 368x368 for MaP, and over 180,000 for Drebin). Consequently, this section mainly explores the efficiency of detection methods under these three different types of feature: MaF, MaP, and Drebin.

Training Time Cost. During the training phase, the time consumption for each method includes two components: training the substitute model, core process and training the classifier. As indicated by Table 3, the majority of training time is spent on the substitute model, which is crucial for all detection methods. However, the time required to train the substitute model varies significantly depending on the type of feature input. For example, training the substitute model for MaF takes approximately 17.92 seconds, whereas for MaP, it takes around 172.19 seconds. For the core process and classifier training component, FS is the least time-consuming across all data scales, due to the straightforwardness of feature compression and the absence of classifier training. KD_BU takes longer, mainly because Bayesian and Kernel Density Estimation features require calculation based on the data distribution, which can be time-consuming for high-dimensional data. LID typically requires less time, but for high-dimensional features like MaP and Drebin, its computational demands increase, resulting in longer processing times. For HagDe, when using CPU resources, the overall processing time is higher than that of LID and also increases as the dimensionality of the data features grows. When using GPU resources, the processing time for MaF features is slightly higher than with CPU due to the additional overhead from tensor transfers, given the small dimensionality of the MaF data. However, when the input features are MaP and Drebin, processing time significantly decreases with GPU usage compared to CPU. This is because HagDe primarily relies on gradient ascent perturbation, involving gradient backpropagation. GPUs are designed to accelerate neural network training, providing significant

Table 3: Training Time Cost

Data feature	Method	Train substitute model	Core process and train classifier
MaF	FS	17.92s	0.08s
	KD_BU		5.1s
	LID		1.52s
	HagDe		1.64s(CPU) ǀ 2.04s(GPU)
MaP	FS	172.19s	2.23s
	KD_BU		233.49s
	LID		9.92s
	HagDe		13.04s(CPU) ǀ 2.02s(GPU)
Drebin	FS	151.25s	1.7s
	KD_BU		279.48s
	LID		10.01s
	HagDe		12.96s(CPU) ǀ 2.06s(GPU)

Table 4: Inference Time Cost

Data feature	Method	10 samples	100 samples	1000 samples
MaF	FS	0.07s	0.07s	0.10s
	KD_BU	5.14s	7.33s	22.20s
	LID	0.13s	0.13s	0.33s
	HagDe(CPU)	0.12s	0.94s	9.12s
	HagDe(GPU)	0.24s	2.01s	20.37s
MaP	FS	0.21s	0.63s	5.23s
	KD_BU	236.78s	241.17s	332.67s
	LID	0.22s	2.95s	54.49s
	HagDe(CPU)	3.16s	34.33s	332.32s
	HagDe(GPU)	0.26s	2.34s	23.21s
Drebin	FS	0.23s	0.76s	6.32s
	KD_BU	274.52s	294.02s	360.82s
	LID	0.24s	3.88s	64.87s
	HagDe(CPU)	4.57s	46.87s	448.08s
	HagDe(GPU)	0.3s	2.41s	25.08s

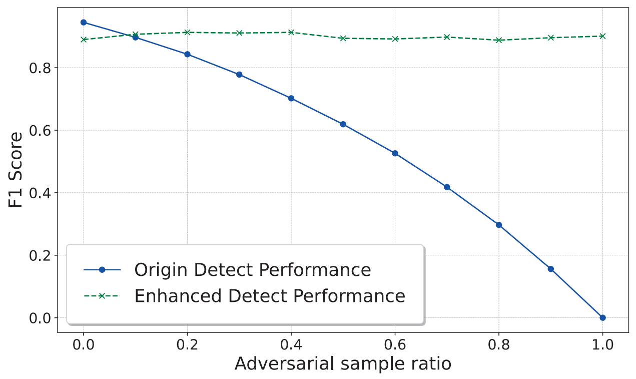

Figure 10: F1 Performance Enhancement Comparison with vs without HagDe (Under The Attack Pattern of *AdvDroidZero_MaF*)

speedup for this process. Additionally, the time taken is nearly the same as for MaF due to the GPU's strong concurrency support. In contrast, LID does not rely on computations involving neural networks or similar operations, making it less dependent on GPU resources.

Inference Time Cost. To clearly demonstrate the inference time of each method across different data scales, we conducted experiments with data sizes of 10, 100, and 1000. For a fair comparison, each method process data serially. Overall, similar trends to those in the training phase are observed from Table 4: the runtime for LID increases with the dimensionality of the input features, while HagDe relies heavily on the GPU to enhance efficiency. Additionally, the processing time for HagDe exhibits a linear relationship with increasing data size. This is because, theoretically, it operates with a complexity of O(λ), where λ represents the fixed number of perturbations, set at λ=15. All these methods can process large-scale data more quickly through multithreading, although this requires more computational resources. For example, increasing the number of process threads to 10 can theoretically enhance the inference speed nearly tenfold.

Answer to RQ3*: HagDe relies on the GPU and concurrency to accelerate both training and inference processes.*

4.6 RQ4: Enhanced Malware Detection

The primary aim of introducing our adversarial sample detection framework is to enhanced AMD methods against adversarial attacks. In this part, we primarily investigate the performance of existing malware detection methods with and without our adversarial sample detection framework to assess the extent of enhancement provided by our framework. Considering a more diverse and real-world scenarios, we randomly selected an equal number of normal and adversarial samples from a dataset different from RQ1 and RQ2, to constitute our test environment.

As clearly depicted in Figure 10, at a ratio of 0, i.e., in the absence of adversarial samples, the detection performance of the malware itself slightly surpasses that of our enhanced framework. This discrepancy arises from false positive predictions made by our adversarial sample detection framework in non-adversarial scenarios. However, as the number of adversarial samples continues to grow, we witness a sharp decline in the detection performance of the malware itself. Ultimately, when the ratio of adversarial to malware samples reaches 1, it is unable to correctly identify a single malware sample. Therefore, HagDe helps existing Android malware detection methods maintain stable performance when faced with adversarial samples in varying proportions. As the proportion of adversarial samples increases from 10% to 90%, HagDe demonstrates varying degrees of enhancement in the F1 score of AMD systems, with improvement ratios ranging from 0.79% to 474.4%.

Answer to RQ4*: HagDe can enhance AMD against adversarial attacks, with improvement ratios in the detection F1 score ranging from 0.79% to 474.4% as the proportion of adversarial samples increases from 10% to 90%.*

5 Discussion and Limitation

5.1 Adaptive Attack

To ensure the long-term reliability of detection systems, it is crucial to discuss the effectiveness of HagDe against adaptive adversaries. Previous works, such as AdvDroidZero and BagAmmo tend to stop after a successful attack to minimize exposure risk and reduce costs. In this context, we propose an adaptive attack scenario where the attacker does not adhere to the minimum attack cost but instead adopts a persistent attack strategy to generate adversarial samples. Specifically, we assume that after the initial successful attack, the attacker iteratively perturbs the APK multiple times to produce the final adversarial sample. However, despite employing adaptive attack strategies, attackers still face significant challenges in evading HagDe detection for the following reasons.

First, attackers encounter major difficulties in creating adversarial samples far from the decision boundary through persistent attacks. For attackers, seeking the optimal perturbation is not arbitrary; instead, it is systematically adjusted based on the response of AMD (e.g., MaMaDroid) regarding malware classification. For instance, AdvDroidZero continuously refines the perturbation search by adjusting the weights of perturbation nodes in the search tree based on the target model's feedback, enabling the selection of the optimal perturbation path. Consequently, once AdvDroidZero succeeds in the initial attack, it receives a response indicating a benign sample. Even if the attacker continues to apply perturbations, the subsequent responses are still likely to indicate a benign sample. At this point, the response becomes nearly ineffective, making it challenging for the attacker to identify the optimal perturbation based on the feedback. This near-ineffective response hinders the attacker from generating adversarial samples that are far from the decision boundary, thereby making it difficult to evade detection by HagDe.

Second, as attacks increase, the APK faces more perturbation, raising the likelihood of APK corruption. Zhao et al. [55] highlighted that feature-level attacks may not always result in successful APK-level reversion, as changes to the APK's *AndroidManifest.xml* or DEX code can cause reconstruction failures. AdvDroidZero also proposed that modifying APKs with FlowDroid can lead to crashes. Thus, if attackers continue to apply perturbations after a successful initial attack, they cannot avoid the issue of APK corruption during the rebuild process.

Third, while attackers cannot significantly enhance the success rate through the aforementioned adaptive attack strategies, they still also likely to expose themselves through easily detectable features. Notably, each attack attempt involves modifications to the *AndroidManifest.xml* or DEX code, while ensuring that the original malicious functionality remains unchanged. To achieve this, BagAmmo proposed inserting multiple try-catch blocks into the DEX code to preserve functionality. Consequently, continuous attacks would result in a significant increase in the number of try-catch blocks, allowing for the detection of such adversarial samples simply by counting these blocks. AdvDroidZero uses FlowDroid to perform code insertion by randomly generating extensive meaningless code with disordered variable and method names. Detecting these chaotic and meaningless features efficiently identifies these adversarial samples.

5.2 Concept Drift

Concept drift in Android malware detection is the challenge where models trained on historical data fail to recognize new malware over time, due to factors like software updates and evolving attacker techniques [17]. In this context, enhancing the model's detection performance is more urgent than defending against adversarial samples, as there is little incentive to attack a poorly performing detector. Existing methods to counter concept drift primarily include model retraining [10], continuous learning [10], and developing features [53] to resist it. For instance, APIgraph [53] employed semantically enriched graphs to enhance the detectors, e.g., Drebin, to address concept drift. These methods offer potential approaches for addressing the issue of concept drift.

5.3 Unbalanced Dataset

Building on previous research [17,19,43], we use a roughly balanced dataset for detection, achieving SOTA detection results. However, achieving low false positives is challenging with unbalanced datasets, as detectors need high precision and recall rates. The maintenance of a detection F1 score of approximately 0.9 on a balanced dataset inadequately addresses the issue of false positives prevalent in imbalanced datasets. This presents a significant challenge for the current detection system due to the complexity and threat posed by existing attacks. Only by further improving the effectiveness of existing detection systems can we fundamentally address the issue of detection effectiveness in imbalanced datasets. As mentioned above, ensemble-based approaches can enhance the detection performance of HagDe. In future work, we will enhance feature extraction and integration, and apply decision integration methods to minimize false positives in unbalanced datasets.

5.4 Mimicry Attack

In the field of malware detection, a distinct attack method exists that imitates benign samples instead of persistently optimizing perturbations based on the target model's response. For example, EvadeDroid [6] employs problem-space transformations from benign sources,which are then iteratively and incrementally applied to morph malware into benign forms. Overall, the effectiveness of HagDe in detecting mimicry attacks depends on the mimicry degree attack achieved. The less an attack sample mimics benign behavior, evidenced by a lower confidence score classifying it as benign in AMD, the closer it is to the decision boundary of AMD, making it easier for HagDe to detect. Conversely, if AMD classifies the sample with higher confidence as benign, it moves away from the decision boundary, increasing the chance of misclassification by HagDe. However, This method of enhancing mimicry degree by extensively transplanting benign code also has its limitations. To realize this higher degree of mimicry, it relies on specific semantic extraction techniques, leading to significant overhead and easily detectable features, such as an increased payload size [6]. We plan to enhance HagDe by integrating more semantic-related and finer-grained overhead features to better detect such attacks in the future.

6 Related Work

6.1 Attack to Android Malware Detection

Adversarial attack algorithms can be manipulated by malicious entities to generate deceptive malware instances, thereby bypass learning-based malware classifiers, that presents a threat to defenders. Consequently, numerous fields have begun to focus on adversarial example attacks, including image classification [47,48], text classification [12,31], autonomous driving [25]. In the context of Android malware detection, many studies [1,21,24,27,29] focused on syntax features oriented AE generation. Chen et al. [9] employ optimization techniques (i.e., C&W, JSAM) to craft adversarial perturbations within the feature space, and they propose a methodology for integrating these optimal perturbations into APK files. Zhao et al. [55] proposed a structural attack mapped from feature space to problem space based on reinforcement learning to attack FCG-based AMD methods. Li et al. [27] explored the application of ensemble learning algorithms to bolster the detection and mitigation strategies for adversarial Android malware. Subsequently, in a more recent study, Li et al. [28] introduced an optimization method leveraging Apoem to generate adversarial Android malware against FCG-based AMD methods without knowledge of the detection granularity. Recently, He et al. [22] proposed a perturbation tree, rooted in semantics at varying granularities for path planning to search for the optimal perturbation, thereby generating adversarial sample under zero knowledge set. These studies brought practical threats to AMD systems and thus prompted this paper to investigate adversarial sample detection.

6.2 Defense of AEs

Adversarial attacks have demonstrated high success rates in compromising learning-based classifiers, leading to their malfunction. To counter this, methods have been developed to strengthen classifiers, focusing either on enhancing learning-based malware classifiers or enhancing learning models.

6.2.1 Enhancing Learning-based Malware Classifiers

In general, learning-based malware classification can be divided into five phases, including data preprocessing, feature collection, feature extraction, classification, and decision making. Methods that improve various stages of malware classification, except for the feature collection phase, are known as enhancing learning-based malware classification. During data preprocessing, Smutz and Stavrou [38] presented an ensemble model that uses voting to integrate classifier predictions, effectively diminishing the success of gradient descent and KDE-based attacks. Zhang et al. [52] proposed a feature-based defense employing reduced feature sets to improve the robustness and generalization of learning-based malware classifiers. Li et al. [30] introduced a robust Android malware detection framework that employs a VAE and an MLP for effective classification and adversarial defense, eliminating the need for adversarial example knowledge or instances. Experiments demonstrate its improved adversarial robustness against both white-box and black-box attacks. These methods, aiming to enhance different phases of malware classifiers collectively underscore the diverse strategies employed to enhance the resilience of learning-based malware classifiers.

6.2.2 Enhancing Learning Models

Unlike the way of enhancing learning-based malware classifiers, numerous methods have been proposed to enhance the robustness of learning models, rather than directly augmenting the classifier's process. These methods include adversarial example detection, adversarial training, randomization-based defense. To detect adversarial examples, researchers have proposed a variety of proactive defense strategies. These include adversarial (re)training [20] [26] [40], distillation networks [36], feature squeezing [50], k-NN search [13], and network input regularization [37]. However, those defenses can be evaded by the optimization-based attack [8].

As a representative of feature compression techniques, the method proposed by Xu [50] et al. stands out in the field of adversarial example detection. This approach reduces the feature space of input examples by squeezing color depth and applying spatial smoothing. Adversarial examples can be detected by comparing the differences between compressed and uncompressed inputs. To identify distinguishing features between adversarial examples and natural samples, an increasing number of methods focus on distinguishing adversarial images from normal images based on features extracted from DNN layers or from a learned encoder. Among these, KD [15], BU features [15], and LID [33], are the SOTA detection methods proven in previous studies.

Despite these advancements, it is noted that these methods were primarily designed and applied in the image domain, and their applicability to Android malware attacks may be limited. The underlying reason is that, unlike the continuous feature space in the image domain, the features in the malware domain are discrete. Consequently, methods designed for images often fail in the realm of adversarial malware detection. In contrast, our approach, tailored to the nature of Android malware generation, exhibits distinct loss characteristics and enhanced interpretability by following the path of maximum loss and halting immediately after a successful attack, making it more effective in detecting adversarial Android malware.

7 Conclusion

In this paper, we introduce a novel defense mechanism, Harnessing Attack Generativity for Defense Enhancement (HagDe), aimed at addressing the challenge of malware attacks on the Android malware detection system. By applying iterative perturbations to all samples, HagDe exploits the high sensitivity of adversarial examples to disturbances, effectively distinguishing between malicious and benign inputs. Extensive evaluation on 15,000 samples and 15 different attack patterns demonstrated HagDe's superior defense effectiveness, with success rates of 88.5% against AdvDroidZero and 90.7% against BagAmmo, significantly outperforming existing SOTA defense methods. This achievement not only enhances the resilience of Android malware detectors but also lays a foundation for future research.

Acknowledgment

We thank the anonymous reviewers for their helpful and informative feedback. This material was supported in part by the China National Petroleum Corporation Science and Technology Project under grant No. 2024ZZ46-06; by the National Research Foundation, Singapore, and DSO National Laboratories under the AI Singapore Programme (AISG Award No: AISG2-GC-2023-008); by the National Research Foundation Singapore and the Cyber Security Agency under the National Cybersecurity R&D Programme (NCRP25-P04-TAICeN); and by the National Research Foundation, Prime Minister's Office, Singapore under the Campus for Research Excellence and Technological Enterprise (CREATE) programme. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore and Cyber Security Agency of Singapore.

Ethics Considerations

We have carefully considered the ethical implications of this study. We have ensured that all research activities were conducted in accordance with established ethical guidelines and principles. Specifically, we have considered the potential impacts of our findings on society, including the possible misuse of our methods and results. We have also taken steps to mitigate any negative consequences by designing our experiments and reporting our results in a way that prioritizes responsible use.

Open Science Policy

In accordance with the Open Science Policy, we commit to making all artifacts related to this research available by the camera-ready deadline. Our artifacts is accessible through a publicly available repository (i.e., https://zenodo.org/records/14713949), and we provide detailed documentation to facilitate their use by other researchers.

References

[1] Abdullah Al-Dujaili, Alex Huang, Erik Hemberg, and Una-May O'Reilly. Adversarial deep learning for robust detection of binary encoded malware. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 76–82. IEEE, 2018.

[2] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR)*, pages 468–471, New York, NY, USA, 2016.

[3] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *Proceedings of Symposium on Network and Distributed System Security (NDSS)*, volume 14, pages 23–26, 2014.

[4] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Mining apps for abnormal usage of sensitive data. In *Proceedings of the 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*, volume 1, pages 426–436, 2015.

[5] Yude Bai, Zhenchang Xing, Xiaohong Li, Zhiyong Feng, and Duoyuan Ma. Unsuccessful story about few shot malware family classification and siamese network to the rescue. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering(ICSE)*, pages 1560–1571. ACM/IEEE, 2020.

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

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

[8] Nicholas Carlini and David Wagner. Adversarial examples are not easily detected: Bypassing ten detection methods. In *Proceedings of the 10th ACM workshop on artificial intelligence and security*, pages 3–14, 2017.

[9] Xiao Chen, Chaoran Li, Derui Wang, Sheng Wen, Jun Zhang, Surya Nepal, Yang Xiang, and Kui Ren. Android hiv: A study of repackaging malware for evading machine-learning detection. *IEEE Transactions on Information Forensics and Security*, 15:987–1001, 2019.

[10] Yizheng Chen, Zhoujie Ding, and David Wagner. Continuous learning for android malware detection. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1127–1144, 2023.

[11] Gao Cuiying, Yueming Wu, Heng Li, Wei Yuan, Haoyu Jiang, Qidan He, and Yang Liu. Uncovering and mitigating the impact of code obfuscation on dataset annotation with antivirus engines. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 553–565, 2024.

[12] Tianyu Du, Shouling Ji, Lujia Shen, Yao Zhang, Jinfeng Li, Jie Shi, Chengfang Fang, Jianwei Yin, Raheem Beyah, and Ting Wang. Cert-rnn: Towards certifying the robustness of recurrent neural networks. *CCS*, 21(2021):15–19, 2021.

[13] Abhimanyu Dubey, Laurens van der Maaten, Zeki Yalniz, Yixuan Li, and Dhruv Mahajan. Defense against adversarial images using web-scale nearest-neighbor search. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 8767–8776, 2019.

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

[15] Reuben Feinman, Ryan R Curtin, Saurabh Shintre, and Andrew B Gardner. Detecting adversarial samples from artifacts. *arXiv preprint arXiv:1703.00410*, 2017.

[16] Cuiying Gao, Minghui Cai, Shuijun Yin, Gaozhun Huang, Heng Li, Wei Yuan, and Xiapu Luo. Obfuscation-resilient android malware analysis based on complementary features. *IEEE Transactions on Information Forensics and Security*, 18:5056–5068, 2023.

[17] Cuiying Gao, Gaozhun Huang, Heng Li, Bang Wu, Yueming Wu, and Wei Yuan. A comprehensive study of learning-based android malware detectors under challenging environments. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

[18] Song Gao, Ruxin Wang, Xiaoxuan Wang, Shui Yu, Yunyun Dong, Shaowen Yao, and Wei Zhou. Detecting adversarial examples on deep neural networks with mutual information neural estimation. *IEEE Transactions on Dependable and Secure Computing*, 2023.

[19] Joshua Garcia, Mahmoud Hammad, and Sam Malek. Lightweight, obfuscation-resilient detection and family identification of android malware. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 26(3):1–29, 2018.

[20] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[21] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. Adversarial examples for malware detection. In *Computer Security–ESORICS 2017: 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part II 22*, pages 62–79. Springer, 2017.

[22] Ping He, Yifan Xia, Xuhong Zhang, and Shouling Ji. Efficient query-based attack against ml-based android malware detection under zero knowledge setting. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 90–104, 2023.

[23] Matthias Hein and Maksym Andriushchenko. Formal guarantees on the robustness of a classifier against adversarial manipulation. *Advances in neural information processing systems*, 30, 2017.

[24] Weiwei Hu and Ying Tan. Generating adversarial malware examples for black-box attacks based on gan. In *International Conference on Data Mining and Big Data*, pages 409–423. Springer, 2022.

[25] Pengfei Jing, Qiyi Tang, Yuefeng Du, Lei Xue, Xiapu Luo, Ting Wang, Sen Nie, and Shi Wu. Too good to be safe: Tricking lane detection in autonomous driving with crafted perturbations. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3237–3254, 2021.

[26] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial machine learning at scale. *arXiv preprint arXiv:1611.01236*, 2016.

[27] Deqiang Li and Qianmu Li. Adversarial deep ensemble: Evasion attacks and defenses for malware detection. *IEEE Transactions on Information Forensics and Security*, 15:3886–3900, 2020.

[28] Heng Li, Zhang Cheng, Bang Wu, Liheng Yuan, Cuiying Gao, Wei Yuan, and Xiapu Luo. Black-box adversarial example attack towards {FCG} based android malware detection under incomplete feature information. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1181–1198, 2023.

[29] Heng Li, ShiYao Zhou, Wei Yuan, Jiahuan Li, and Henry Leung. Adversarial-example attacks toward android malware detection system. *IEEE Systems Journal*, 14(1):653–656, 2019.

[30] Heng Li, Shiyao Zhou, Wei Yuan, Xiapu Luo, Cuiying Gao, and Shuiyan Chen. Robust android malware detection against adversarial example attacks. In *Proceedings of the Web Conference 2021*, pages 3603–3612, 2021.

[31] Jinfeng Li, Shouling Ji, Tianyu Du, Bo Li, and Ting Wang. Textbugger: Generating adversarial text against real-world applications. *arXiv preprint arXiv:1812.05271*, 2018.

[32] Xuanqing Liu and Cho-Jui Hsieh. Rob-gan: Generator, discriminator, and adversarial attacker. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 11234–11243, 2019.

[33] Xingjun Ma, Bo Li, Yisen Wang, Sarah M Erfani, Sudanthi Wijewickrema, Grant Schoenebeck, Dawn Song, Michael E Houle, and James Bailey. Characterizing adversarial subspaces using local intrinsic dimensionality. *arXiv preprint arXiv:1801.02613*, 2018.

[34] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon J. Ross, and Gianluca Stringhini. Mamadroid: Detecting android malware by building markov chains of behavioral models. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*, 2017.

[35] Dongyu Meng and Hao Chen. Magnet: a two-pronged defense against adversarial examples. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 135–147, 2017.

[36] Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. Distillation as a defense to adversarial perturbations against deep neural networks. In *2016 IEEE symposium on security and privacy (SP)*, pages 582–597. IEEE, 2016.

[37] Andrew Ross and Finale Doshi-Velez. Improving the adversarial robustness and interpretability of deep neural networks by regularizing their input gradients. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

[38] Charles Smutz and Angelos Stavrou. When a tree falls: Using diversity in ensemble classifiers to identify evasion in malware detectors. In *NDSS*, 2016.

[39] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[40] Florian Tramèr, Alexey Kurakin, Nicolas Papernot, Ian Goodfellow, Dan Boneh, and Patrick McDaniel. Ensemble adversarial training: Attacks and defenses. *arXiv preprint arXiv:1705.07204*, 2017.

[41] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. 2010.

[42] Virustotal. Virustotal., 2022. https://www.virustotal.com/.

[43] Liu Wang, Haoyu Wang, Xiapu Luo, and Yulei Sui. Malwhiteout: Reducing label errors in android malware detection. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–13, 2022.

[44] Yueming Wu, Shihan Dou, Deqing Zou, Wei Yang, Weizhong Qiang, and Hai Jin. Contrastive learning for robust android malware familial classification. *IEEE Transactions on Dependable and Secure Computing*, 2022.

[45] Yueming Wu, Xiaodi Li, Deqing Zou, Wei Yang, Xin Zhang, and Hai Jin. Malscan: Fast market-wide mobile malware scanning by social-network centrality analysis. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 139–150, 2019.

[46] Yueming Wu, Deqing Zou, Wei Yang, Xiang Li, and Hai Jin. Homdroid: detecting android covert malware by social-network homophily analysis. In *Proceedings of the 30th acm sigsoft international symposium on software testing and analysis*, pages 216–229, 2021.

[47] Pengfei Xia, Ziqiang Li, Wei Zhang, and Bin Li. Data-efficient backdoor attacks. *arXiv preprint arXiv:2204.12281*, 2022.

[48] Pengfei Xia, Hongjing Niu, Ziqiang Li, and Bin Li. Enhancing backdoor attacks with multi-level mmd regularization. *IEEE Transactions on Dependable and Secure Computing*, 20(2):1675–1686, 2022.

[49] Cihang Xie, Jianyu Wang, Zhishuai Zhang, Zhou Ren, and Alan Yuille. Mitigating adversarial effects through randomization. *arXiv preprint arXiv:1711.01991*, 2017.

[50] Weilin Xu, David Evans, and Yanjun Qi. Feature squeezing: Detecting adversarial examples in deep neural networks. *arXiv preprint arXiv:1704.01155*, 2017.

[51] Wei Yuan, Yuan Jiang, Heng Li, and Minghui Cai. A lightweight on-device detection method for android malware. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 2019.

[52] Fei Zhang, Patrick PK Chan, Battista Biggio, Daniel S Yeung, and Fabio Roli. Adversarial feature selection against evasion attacks. *IEEE transactions on cybernetics*, 46(3):766–777, 2015.

[53] Xiaohan Zhang, Yuan Zhang, Ming Zhong, Daizong Ding, and Min Yang. Enhancing state-of-the-art classifiers with api semantics to detect evolved android malware. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2020.

[54] Haojun Zhao, Yueming Wu, Deqing Zou, and Hai Jin. An empirical study on android malware characterization by social network analysis. *IEEE Transactions on Reliability*, 2023.

[55] Kaifa Zhao, Hao Zhou, Yulin Zhu, Xian Zhan, Kai Zhou, Jianfeng Li, Le Yu, Wei Yuan, and Xiapu Luo. Structural attack against graph based android malware detection. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 3218–3235, New York, NY, USA, 2021. Association for Computing Machinery.

[56] Deqing Zou, Yueming Wu, Siru Yang, Anki Chauhan, Wei Yang, Jiangying Zhong, Shihan Dou, and Hai Jin. Intdroid: Android malware detection based on api intimacy analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(3):1–32, 2021.

A Learning-based AMD Methods

Drebin. In the study by Drebin [3], the methodology involves pulling out characteristics from two primary sources within Android applications: the *Androidmanifest.xml* and the *classes.dex* files. These characteristics are then categorized into eight distinct groups, covering areas like hardware components and permissions that the application requests, and are represented in the form of strings. Following this categorization, these groups of features are transformed into a vector space model, with each feature being assigned a binary value of 0 or 1. Utilizing this model, Drebin proceeds to employ a Support Vector Machine (SVM) classifier with the aim of identifying malicious software.

MaMa-fml (MaF). As a variant of the MaMaDroid framework [34], MaF processes smali files to retrieve API calls, which it subsequently categorizes into a set of family calls. This categorization encompasses 11 families, with 9 derived from the official Android documentation and 2 being self-defined and obfuscated. MaF proceeds to establish a Markov chain reflecting the transition probabilities between these families. This chain serves as the feature vector for training a detection model.

MaMa-pkg (MaP). As an alternative model within the MaMaDroid [34], MaP mirrors MaF in feature extraction from smali files but diverges by abstracting API calls into package calls instead of family calls. It identifies 366 packages based on the official Android documentation, supplemented by 2 additional self-defined and obfuscated packages. Thus, it represents an application through a feature vector with a dimensionality of (368×368).

APIGraph. Introduced in [53], APIGraph is developed to identify semantic similarities among Android APIs by constructing a relational graph from official documentation. Its application extends to enhancing existing Android malware classifiers. In our assessment, APIGraph is integrated with both Drebin, MaF and MaP, resulting in the variants Drebin-api, MaF-api and MaP-api, respectively.

B Implementation Details

HagDe is an automatic defense framework with three stages to detect adversarial Android malware, that can be used to enhance adversarial Android malware detection. On one hand, based on previous research, we implemented a python-based feature extraction pipeline from Android apps to feature vectors. On the other hand, we handle the program logic of the detection and defense in python, including training substitute models, continuous perturbation in the direction of gradient ascent, and detection based on loss features. 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. For the implementations of FS, KD_BU, and LID, we adhered strictly to the default parameters specified in the respective papers. However, to ensure a fair comparison, FS, KD_BU, and LID did not utilize their original DNN models; instead, they employed substitute models that were trained through the same process as ours. Additionally, since both our method and FS, LID rely on classifiers for detection, with the exception that FS does not require this, FS was evaluated on the complete test set, whereas the other methods were assessed based on results from a 10-fold cross-validation.

C Substitute Model Architecture Details

Table 5: Substitute Model Architecture Details

Layer No.	Layer Type	Parameters
1	Flatten	-
2	Linear	Input: `AMD feature`, Output: 128
3	ReLU	-
4	Linear	Input: 128, Output: 64
5	ReLU	-
6	Dropout	Dropout Rate: 0.5
7	Linear	Input: 64, Output: 2

D False Positive & False Negative Analysis

False Positive Analysis. We analyze 40 false positives in the experimental results and identify two main reasons. One common issue is that False Positive APKs exhibit behavior highly similar to that of malicious APKs, making it difficult for AMD to confidently identify these APKs as benign. Consequently, they often reside near AMD's decision boundary. Specifically, one category of benign APKs requests an excessive number of sensitive permissions commonly used by malware. These benign APKs might be mistakenly identified as malicious. For instance, APK[1] belongs to the security and antivirus category. It utilizes *ActivityControlService* for managing app activities and *ScanSchedulerReceiver* for scheduling antivirus scans. These functions necessitate requesting numerous permissions, totaling 45, including sensitive ones like *android.permission.READ_PHONE_STATE* and *android.permission.WRITE_SETTINGS*. This number far exceeds the typical requests of benign APKs, leading to its false categorization as adversarial by the detector. Another type of behavioral similarity involves benign APKs performing sensitive file operations, such as deleting or writing shared files. Malware often engages in similar activities. This resemblance can cause benign APKs to be mistakenly classified as malware due to their file behavior similarities. For instance, APK[2] moves files to different disk locations, containing Linux executables that can execute sensitive operations on the host system and possibly cause malfunctions. This behavior's sensitivity leads the detector to mistakenly label the APK as adversarial. To reduce false positives due to behavioral similarities, incorporating detailed contextual semantic information

[1] sha256:0AE39219FDC5BC37B16BB1298E8B315E18AA1CAACF7BD59EFC37D74889050397

[2] sha256::01EAE6B8120EA7FD059F188A0FDDDDB26EAF466A24DBBC85260A1D918162B30D

can help assess the legitimacy of high-risk operations. Each APK, upon decompilation, contains rich semantic information in its Dex code components (e.g., Service, Broadcast Receiver, Content Provider) indicating the app's category and functionality. Detailed categorization of permissions and behavioral features mapped to the app's function could reduce false positives. For example, a gaming app requesting sensitive permissions such as *READ_CONTACTS* and *WRITE_CONTACTS* might be flagged as malicious, whereas antivirus software, identified by features like AntiVirusInfectionListActivity in its Dex code, have greater permission flexibility.

A less common false positive occurs with APKs containing extensive encrypted content. These APKs often require enhanced data protection or intellectual property security. Since some malware employs advanced obfuscation for attacks, current detectors may mistakenly link obfuscation traits with malware, causing increased sensitivity to encrypted data. For instance, APK[3] contains extensive encrypted code, which causes a false positive detection by HagDe. To better eliminate this type of false positive, selecting features that are resistant to obfuscation and analyzing encrypted content using semantic information such as the APK's package name could be a potential solution.

False Negative Analysis We also analyzed 40 false negatives, and the causes for these misses can be categorized into three types. The first type involves malicious APKs bypassing detection through advanced code obfuscation techniques like Control Flow Flattening and Call Indirection to disguise malicious APKs, leading to false negatives by making the Dex code hard to analyze and hiding harmful behavior. For example, the APK[4] disperses high-risk API calls across multiple intermediate functions, increasing the difficulty for detectors to analyze and understand the code, allowing the malware to evade detection. Currently, numerous studies are being conducted to defend against obfuscation techniques that bypass detection. Gao et al. [16] researched methods to detect obfuscation attacks and proposed a series of obfuscation-resistant features for detection. In the future, we plan to incorporate more obfuscation-resistant features and counter advanced structural obfuscation to improve malware and adversarial sample detection.

False negatives also happen when attackers add many disguised normal network requests to an APK, dispersing malware detection capability, and repackaging it as an adversarial APK. Through the analysis of false negative instances, various techniques are used to disguise network requests, such as changing URLs, modifying request parameters, and alternating between GET and POST methods. Attackers employ these strategies to generate numerous requests that resemble those of normal apps like Google, causing detectors to overlook the malicious activities. Attack strategies similarly involve integrating reliable third-party benign components to deceive detectors into misclassifying the APK as benign. To mitigate these attack threats, a two-fold strategy can be implemented: Firstly, perform a detailed analysis of network requests, including their parameters, URLs, proxies, and integrated components to spot potential malicious requests, particularly when their patterns significantly differ from known normal requests. Secondly, investigate the characteristics revealed during the attacker's repackaging process to effectively identify malicious APKs. For instance, attackers might add excessive redundant code, noticeably increasing the APK size. Analyzing these traits enhances detection accuracy and minimizes false negatives.

[3] sha256:001FC290A2AFC87B67A77B9E514D8E4DA15D9A7C28AFB0488A625F5BC33A571D

[4] sha256:0F5E2BB2B98FF13ADEB3048660BC7C826C39DCE38698715F15F85025A1009F2A

E Collaboration and Integration with LID

Table 2 shows that in certain attack patterns, current detection methods outperform HagDe in effectiveness. For example, LID surpasses HagDe in the *BagAmmo_MaF-Api* attack pattern by using advanced intrinsic dimensionality features to identify adversarial samples. Therefore, when facing a broader range of complex attack patterns, ensemble-based approaches, integrating features or decisions from multiple superior methods, is a highly promising solution.

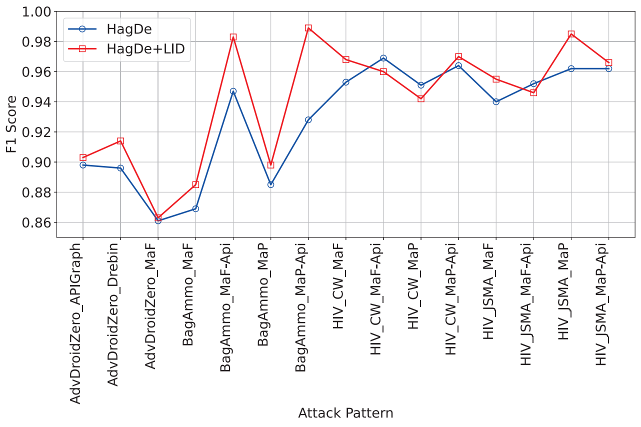

Figure 11: Collaboration and Integration with LID

To investigate the impact of integrating HagDe with LID on detection results, we employ feature fusion. Specifically, we concatenate the features derived from both HagDe and LID to train a classifier for detecting adversarial samples. Figure 11 illustrates the detection F1 score of HagDe and the integrated HagDe with LID across various attack patterns. Under multiple attack patterns, the average F1 score of HagDe is 0.929, while the integration of HagDe with LID achieves an average F1 score of 0.942, representing an overall improvement of 1.4%. In most attack patterns, the integration of HagDe with LID improves the detection F1 score compared to using HagDe alone. Notably, under the *BagAmmo_MaF-Api* attack pattern, the F1 score increases from 0.947 to 0.983. When the attack patterns are *HIV_CW_MaF-Api*, *HIV_CW_MaP* and *HIV_JSMA_MaF-Api*, the integration of LID with HagDe results in a slight decline in performance. This may be because HagDe already achieves high detection results, and the direct integration of LID features does not provide a consistent improvement. In future research, we will explore the selection and integration of various features to further enhance overall detection performance and robustness.