Automated Mass Malware Factory: The Convergence of Piggybacking and Adversarial Example in Android Malicious Software Generation

Heng Li[†*], Zhiyuan Yao[†*], Bang Wu[†] , Cuiying Gao[†] , Teng Xu[†] , Wei Yuan[†#] , Xiapu Luo[‡]
[†] Huazhong University of Science and Technology
[‡] The Hong Kong Polytechnic University
{liheng,zhiyuan_yao,wubangm,gaocy,xuteng,yuanwei}@hust.edu.cn
csxluo@comp.polyu.edu.hk

***Abstract*—Adversarial example techniques have been demonstrated to be highly effective against Android malware detection systems, enabling malware to evade detection with minimal code modifications. However, existing adversarial example techniques overlook the process of malware generation, thus restricting the applicability of adversarial example techniques. In this paper, we investigate piggybacked malware, a type of malware generated in bulk by piggybacking malicious code into popular apps, and combine it with adversarial example techniques. Given a malicious code segment (i.e., a rider), we generate adversarial perturbations tailored to it and insert them into various carriers, enabling the resulting malware to evade detection. Through exploring the mechanism by which adversarial perturbation affects piggybacked malware code, we propose an adversarial piggybacked malware generation method, which comprises three modules: Malicious Rider Extraction, Adversarial Perturbation Generation, and Benign Carrier Selection. Extensive experiments have demonstrated that our method can efficiently generate a large volume of malware in a short period, and significantly increase the likelihood of evading detection. Our method achieves an average attack success rate (ASR) of 88.3% on machine learning-based detection models (e.g., DREBIN and MaMaDroid), and an ASR of 71% and 67% on commercial engines Microsoft and NANO Antivirus, respectively. Furthermore, we have explored potential defenses against our adversarial piggybacked malware.**

I. Introduction

Currently, machine learning-based Android malware detection (AMD) algorithms have achieved excellent detection performance. However, they have recently been found to be vulnerable to adversarial example attacks [9], [26], [28], [38], [48], [55], which meticulously perturb the underlying source code of malware for detection evasion without compromising its functionality [22], [29], [61], [42].

Leveraging adversarial example techniques to generate evasive malware requires two critical steps: crafting the malware and perturbing it. However, the existing perturbation design does not take the malicious behavior-related code into account, resulting in a loose coupling between them. A more effective and efficient approach is to couple perturbation with malicious functionality before producing the malware. Therefore, in this paper, we first investigate popular means of malware generation, and realize that piggybacking (which is also called Trojanization or App Cloning) is currently a prevalent method for bulk production of malware. The emergence of piggybacked malware stems from the fact that Android app package elements can be readily altered by third parties [6], [46], resulting in notable security vulnerabilities. For instance, adversaries can extensively insert their crafted malicious code segments into popular, well-known benign software, masquerading as benign samples [32], [62]. Since a large-scale generation of malware can be achieved by reusing the same malicious code segment on various benign software, piggybacking has emerged as one of the most significant methods for propagating malware threats [53], [63]. A recent report shows that 77% of the top 50 free apps on Google Play have been plagiarized and distributed in alternative app stores [53]. Undoubtedly, piggybacking presents a promising tactic for malware developers and accelerates the spread of malware. For the first time, we investigate the coupling between piggybacked malware and adversarial example techniques, and propose a more threatening method for automated mass generation of evasive malware. This adversarial piggybacked malware further underscores the hazards of adversarial example technology, providing insights for future research on defending against similar techniques.

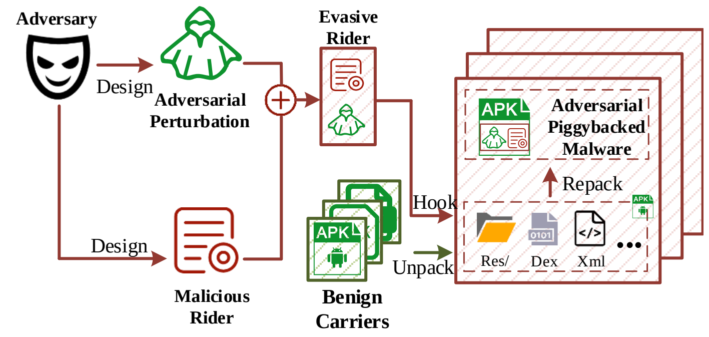


Fig. 1. Overview of Adversarial piggybacked malware.

[*]Equal contribution.
[#]Corresponding author.

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.241933
www.ndss-symposium.org

Through uniting piggybacked malware with adversarial examples, our method can produce approximately 3760 malware within a day using only a single thread. These adversarial piggybacked malware can successfully evade detection by well-known academic detectors and commercial engines with a high probability. The attack process is depicted in Fig. 1. For ease of presentation, we define a segment of code implementing malicious behavior as **malicious rider**, while the original app being piggybacked is referred to as **benign carrier**. The perturbation used for detection evasion is termed **adversarial perturbation**. A malware author can generate an adversarial perturbation based on the malicious rider. The malicious rider and adversarial perturbation are combined into an evasive rider, which can be hooked into **any** benign app, enabling it to evade detection. Once such an adversarial piggybacked malware is generated, the malicious behavior will be triggered thanks to the hook code that is inserted by the malware author to connect his rider code with the carrier code. Moreover, the generated perturbation is rider-specific, remaining effective only for the specific malicious rider and unusable to others. This design helps reduce perturbation misuse and minimize the attack footprint. The main challenges and our countermeasures are briefly described as follows.

Malicious rider. The first challenge is to design an appropriate malicious rider. We obtain it by analyzing the existing pairs of benign app and piggybacked malware through techniques like code similarity analysis. We will discuss this issue in Section IV.

Adversarial perturbation. The second challenge is to compute adversarial perturbations that satisfy universality and targeting requirements besides the requirements of functional consistency[1] [42] and resilience to static analysis[2][61]. The universality requirement indicates the combination of perturbation and malicious rider should be effective across various benign carriers. The targeting requirement stems from the consideration that a malicious rider author has limited incentives to protect any other author's malware but his own [54]. Meeting the targeting requirement helps to avoid the misuse of perturbations and reduce the attack footprint, thus improving stealthiness. In this study, we transform the above considerations into a max-min problem and provide a solution to achieve a good trade-off. Finally, we delve into the framework of perturbation code generation with large language models (LLMs).

Benign carriers. The third challenge is how to select appropriate benign carriers for the malicious riders. In our work, we consider two key factors: 1) the compatibility between different benign carriers and malicious riders, and 2) the popularity of benign carriers, i.e., the dissemination potential of piggybacked malware. We then propose a benign carrier selection method, through delving into the interplay between benign carriers and malicious riders.

Note that our adversarial piggybacked malware can mislead both common AMD models and the detection models specifically designed for piggybacked malware. Mainstream detection methods for piggybacked malware are mainly based on either code similarity comparison [12], [14], [52], [62] or machine learning [18], [32], [34], [46], [49]. Given the millions of Android apps available in the market[3], the similarity-based detection methods necessitate conducting similarity analyses for each unknown app against existing benign apps, resulting a complexity reaching quadratic levels. Li et al. [31] illustrated that detecting piggybacked malware from the Google Play Store would take several months, even when employing multiple threads and with each comparison consuming only 1ms. The machine learning-based detection methods [4], [31], [37] have exhibited robust generalization capabilities. Existing studies [18], [34], [49] demonstrate that with discriminative features, the machine learning models can efficiently and accurately detect piggybacked malware. Nevertheless, experiments show that our adversarial piggybacked malware can effectively evade these models.

The contributions of this paper are as follows:

1. Our research delves into a novel attack scenario, where adversarial perturbing is united with piggybacking to mass produce evasive Android malware.

2. We develop an efficient bulk production method of adversarial piggybacked malware, containing three main components: Malicious Rider Extraction, Adversarial Perturbation Generation, and Benign Carrier Selection.

3. Extensive experiments demonstrate that our method achieves an attack success rate (ASR) of 88.3% on six state-of-the-art (SOTA) AMD systems, and generating a piggybacked malware only requires 23 seconds on average. Furthermore, our adversarial piggybacked malware significantly reduce the detection probability of engines on VirusTotal.

II. Preliminaries

A. Piggybacked malware

Piggybacked malware are built by grafting a malicious rider r to various benign carriers (i.e., apps) x_b. An adversary connects the carrier and the malicious rider by constructing a hook, and the generated malware is represented as $x_b + r$. The hook delineates the juncture at which the carrier context transitions into the rider context within the execution flow.

Piggybacked malware typically modify the following relevant files and folders contained in the corresponding APK (Android Application Package).

META-INF/: It contains the developer's public certificate and the app signature. The adversary re-signs this file in piggybacked malware using tools like jarsigner after modifying the APK source code.

Classes.dex: It contains the compiled Java/Kotlin code in the form of bytecode that can be converted into smali code. Smali code reflects the program's execution logic (i.e., Function Call Graph, FCG). Adversaries typically add, modify or delete function calls in smali files to accomplish the malicious functionalities.

AndroidManifest.xml: It describes the structure and components of an app. In this file, the developer defines app permissions, entry points, intent filters, and so on. Piggybacked

[1]The functionality of the perturbed malware remains unchanged.

[2]The perturbed code is undetectable by static analysis methods.

[3]In 2024, the Google Play Store boasted an excess of 3 million applications (i.e., 3,201,812) [1].

malware typically adds permissions and component information relevant to malicious functionalities. For example, in many piggybacked malware, the adversaries may add WAKE_LOCK permission to keep executing malicious tasks even when the user is not actually using his device.

Res/: It contains binary resources, such as images or hybrid app content. For example, some piggybacked malware may add images and layout files in the res folder to increase advertisements.

lib/: It stores the native library files that the application depends on, which are generally written in C/C++. In some piggybacked malware, the adversaries employ piggybacking with little sophistication, often automatically and via library code.

B. Adversarial attack towards AMD

Given a malicious APK x_m, an adversary customizes a perturbation p and produces an adversarial example $x_m + p$ through injecting the perturbation into the malicious app, with the goal of misleading a classification model f.

Existing adversarial example attacks against AMD can be categorized into APK-specific or APK-agnostic. The APK-specific attacks generate perturbations individually for each APK[29], [42], [61], necessitating repeated modifications for each APK independently and frequent querying with the target model. Their attack process can be formulated as:

$$\begin{aligned} p_x^* &= \arg\min_{p_x \in \mathbf{P}} \operatorname{cost}(p_x), \\ \text{s.t. for malware } & x_m, f\left(\phi\left(x_m + p_x^*\right)\right) = 0, \end{aligned} \tag{1}$$

where p_x represents the perturbation for x_m, $\mathbf{P}$ denotes the set of all feasible perturbations in the code space, and $\operatorname{cost}()$ signifies the cost required for perturbation generation. $f(x) = 0$ means an APK x being classified as benign, and vice versa. Note that the classification model $f()$ often takes abstracted vector features from the code as its input. Hence $\phi()$ is introduced to reflect the feature extraction process, i.e., the mapping from the code space to the feature space.

The APK-agnostic attacks typically design a universal perturbation effective for various malware, aiding them in evading detection [10], [23], [56]. The generation process of such adversarial perturbations can be described as:

$$\begin{aligned} p_u^* &= \arg\min_{p_u \in \mathbf{P}} \operatorname{cost}(p_u), \\ \text{s.t. } & \forall x_m \in \mathbf{M}, f\left(\phi\left(x_m + p_u^*\right)\right) = 0, \end{aligned} \tag{2}$$

where p_u represents the universal perturbation and $\mathbf{M}$ denotes the malware set.

III. Threat Model & Attack Formulation

Goal. In this work, we introduce a new attack scenario, where the adversarial example technique and piggybacking technique are combined to mass-produce evasive Android malware. Given a malicious rider, an adversary aims to generate an adversarial perturbation specific to this rider, to assist mass-producing adversarial piggybacked malware. As shown in Fig. 2, these perturbations should satisfy the universality

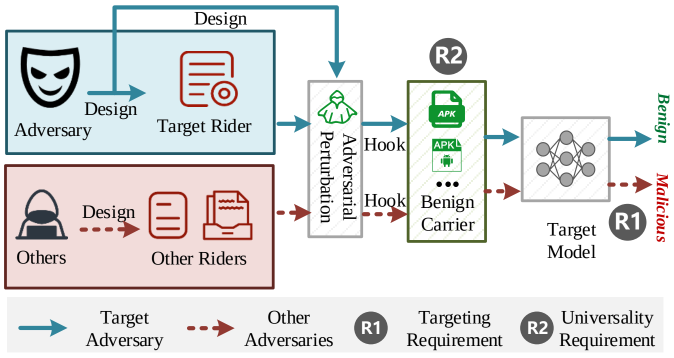


Fig. 2. The attack goal of the adversarial piggyback APKs.

requirement and targeting requirement. Let r_t and p^* denote the malicious rider and the perturbation, respectively. The university requirement indicates

$$\begin{aligned} p^* &= \arg\min_{p \in \mathbf{P}} \operatorname{cost}(p), \\ \text{s.t. } & \forall x_b \in \mathbf{B}, f\left(\phi\left(x_b + r_t + p^*\right)\right) = 0, \end{aligned} \tag{3}$$

where $\mathbf{B}$ is the set of benign APKs.

The targeting requirement helps the adversary to reduce his attack footprint and improve stealthiness. Under this requirement, the perturbation will take effectiveness only when working with the malicious rider r_t designed by the adversary himself. That is, we have

$$\forall x_b \in \mathbf{B}, \forall r \neq r_t, f\left(\phi\left(x_b + r + p^*\right)\right) = 1 \tag{4}$$

Knowledge. For the adversary, the level of his knowledge about the target model can be strong, moderate, or weak. In the case of strong knowledge, the adversary has complete information about the classification model, including model parameters and model structure. In the case of moderate knowledge, the adversary only has the information about model structure. In the case of weak knowledge, the adversary is unaware of the aforementioned information and relies on transferability of adversarial examples and the binary output results to attack his target model.

Constraint. There are two constraints on the adversary. First, the adversary should keep the number of his queries to the target model below a certain threshold, to ensure the attack stealthiness. Second, the adversary cannot delete smali files, permissions, actions, resource files, etc., from the carriers when inserting the rider, to ensure the automation of hooking malicious riders onto benign carriers. It is also noted that our work does not restrict the size of the rider. Our work only requires the attackers to leverage smali code to implement the functionality of malicious riders.

IV. Adversarial Piggybacked Malware

Building adversarial piggybacked malware includes three closely intertwined tasks, i.e., extracting malicious riders, generating adversarial perturbations and selecting benign carriers, as depicted in Fig. 3. In this section, we separately discuss these tasks, and through reverse engineering techniques, we finally inject malicious riders and adversarial perturbations into benign carriers in batches.

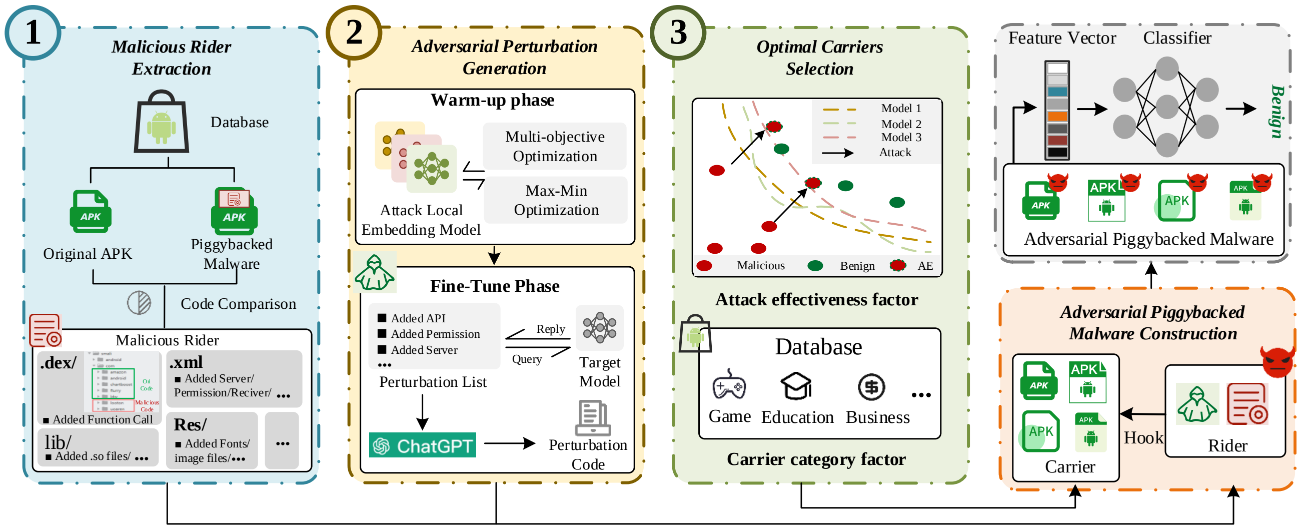


Fig. 3. The overview of our proposed method.

A. Malicious Rider Extraction

Malicious riders usually involve intricate underlying code logic. For instance, some piggybacked apps use reflection and dynamic code loading to introduce the code of malicious behaviors. Consequently, designing malicious riders is a time-consuming and labor-intensive process. Fortunately, Li et al. [32] extracted 1,497 app pairs from over 2 million apps, where one app piggybacks another to integrate a malicious rider. However, these malicious riders often involve carrier-specific operations, such as replacing specific functions within the carrier. This makes these malicious riders unable to be automatically inserted into any carriers. To address this problem, we further filter these benign-malware pairs to extract malicious rider candidates suitable for automation.

Consider the first block of Fig. 3. To begin with, we conduct an analysis of the malicious rider. Androguard tool [2] is utilized to perform a comparison of the smali code of piggyback pairs. Then, a regular expression matching method is employed to compare permissions and actions in the Androidmanifest.xml file of piggyback pairs. Finally, differences in the directory structures of files under the res, assets, and other folders in the decompiled files are compared to identify discrepancies in resource files.

In Fig. 4, the analysis results of smali files are presented on the left y-axis, while the analysis results of permissions, actions, asset files, resource files, and library files are illustrated on the right y-axis. In this figure, 'A' indicates **A**dded, 'M' signifies **M**odified, and 'D' denotes **D**eleted. It can be seen that malicious riders predominantly insert files, permissions, actions, and resource files into the carrier, with minimal deletions. Specifically, 42.3% of piggybacked malware only modifies one existing smali file in a carrier and uses it as the entry function for malicious riders. Additionally, our analysis of these software yields the following **F**indings:

F1: Typically, the carrier employs a single function call statement to link malicious code to benign code (this callee function is named hook function in the following).

F2: The smali files with the hook function inserted are usually located in the original launcher component of the

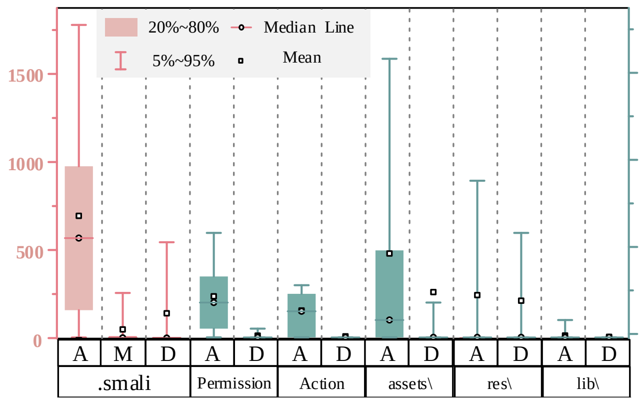


Fig. 4. The analysis of the rider.

carrier, ensuring that the malicious code segment will definitely execute.

Based on the above findings, to ensure the execution of malicious activities after inserting the malicious rider, we propose the following **R**equirements for the malicious rider.

R1: The extracted rider is required to be called by only one hook function.
R2: The hook function must be presented in the original launcher component.

Furthermore, to enable batch automated insertion of the malicious rider, we propose the following requirement.

R3: The extracted rider cannot delete smali files, permissions, actions, resource files, etc., from the carrier.

According to the aforementioned requirements, our extraction process for the malicious payload consists of three steps:

Step 1: We utilize a script to pick out the riders from existing malicious riders meeting R1-R3.

Step 2: To verify the functionality of the extracted riders, we develop a minimal app (i.e., a test carrier), and insert the extracted riders into it to generate piggybacked APKs. These piggybacked APKs are then executed on an Android emulator to test whether they can properly run.

Step 3: We debug those piggybacked APKs that fail to run on the Android emulator. Then the missing permissions, actions, etc are added [4], and the modified riders will undergo the second step mentioned above.

In the end, we obtained 348 malicious riders, comprising newly added resource files, additional smali code, hook functions, as well as added actions and permissions. It is noted that manual debugger operations are isolated from the subsequent perturbation generation and perturbation insertion processes. Once perturbations are generated, they can be automatically inserted into a large number of different benign carriers. Therefore, manual debugger operations do not affect the scalability of the algorithm. What's more, an adversary might develop malicious riders by himself. This issue is beyond the scope of this study, since our proposed method is independent of how the malicious rider is obtained .

B. Adversarial Perturbation Generation

Here we discuss how to derive the desired adversarial perturbation for a given malicious rider, following the steps of problem formulation, solution method and algorithm development. As shown in the second block of Fig. 3, our method includes two phases: warm-up phase and fine-tune phase. In the warm-up phase, we introduce a substitute model to derive the desired perturbation. In the fine-tune phase, we use the query-reply results to modify the perturbation, making the resulting adversarial piggybacked malware evade the detection of the target model. For practical consideration, we assume the reply from the target model only provides the binary decision outcome.

1) Problem Formulation: For convenience, we first review the requirements for adversarial example generation discussed in the Threat Model Section.

Universality requirement: The perturbation needs to make all the carriers evade detection after being hooked with the malicious rider.

Targeting requirement: The perturbation is designed only for the adversary's own malicious rider.

To meet the targeting requirement, an adversary needs to know the malicious riders used by other adversaries, which is almost impossible in practice. To address this issue, we transform this requirement to the following implicit one without accessing other riders:

$$\forall x_b \in \mathbf{B}, \forall n \neq 0, f\left(\phi\left(x_b + (r+n) + p\right)\right) = 1. \tag{5}$$

In the above equation, n is random noise that we deliberately introduce into the training process. When n is continuously randomly set, $r+n$ can be used to simulate other riders and hence helps to meet the targeting requirement. Unfortunately, repeatedly randomly setting n may lead to difficulties in the convergence of model training. Therefore, we reformulate the above equation to a min-max optimization problem [40], [57]. Accordingly, we can meet the targeting requirement through the following two steps.

[4]Some permissions are utilized by both carries and malicious riders, making it challenging to extract all the permissions solely required by the rider through code similarity analysis.

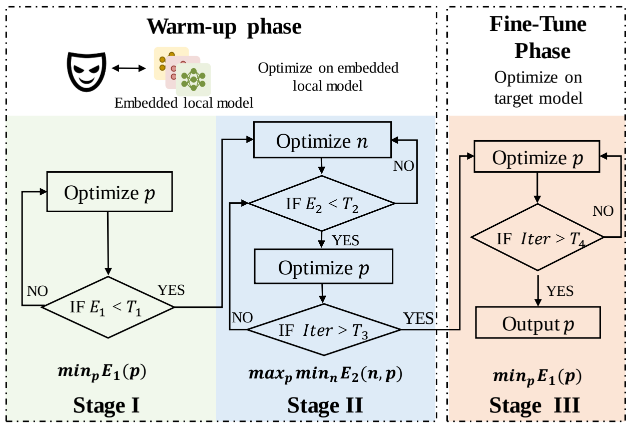


Fig. 5. The optimization process.

In the minimization step, we optimize n and identify the rider $(r+n^*)$ that is most easily attacked. In the maximization step, we optimize the perturbation p to ensure that it cannot be successfully applied to the most vulnerable rider $(r+n^*)$. The logic behind these two steps is as follows. We do not validate the efficacy of the evasion detection on all riders for the perturbation p. Instead, we attempt to find the rider $r+n^*$ that is most susceptible to perturbation by p and is more likely to evade detection. If p can't be applied to the most vulnerable rider, then we believe that p can cause other riders to be recognized as malicious as well. Thus, p meets the targeting requirement.

Based on the above analysis, we propose the following design objective:

$$\begin{aligned} Objective &= \min_p E_1(p) + \max_p \min_n E_2(n,p) \\ &= \min_p E_{x_b \in B}(F(x_b + r + p)) \\ &\quad + \max_p \min_n E_{x_b \in B}(F(x_b + (r+n) + p)), \end{aligned} \tag{6}$$

where the first term ensures the universality of the perturbation across different benign carriers, and the second corresponds to the min-max problem designed for the targeting requirement. $x_b \in B$ represents the benign carrier selected from the benign app set B, r denotes the target-customized malicious rider, n represents the added noise, F represents $f(\phi())$ and p signifies the desired perturbation. E represents the expected value of the classification model output, where 0 denotes benign and 1 represents malicious.

2) Solution Method: Now we consider how to solve the above optimization problem. We primarily encounter three challenges: 1) achieving a good trade-off between the two terms in Eq. (6) (i.e., balancing universality and targeting requirements), 2) solving the min-max problem, and 3) reducing the number of queries with the target model.

To overcome the first challenge, drawing inspiration from multi-objective optimization [21], we first optimize for the universality requirement (i.e., E_1) and then optimize for the targeting requirement (i.e., E_2) while ensuring that E_1 does not increase. In this way, we can better control the interplay between the two objectives during the optimization process, as depicted by the green and blue boxes in Fig. 5.

As for the second challenge, we employ a single-loop iterative algorithm [33] to solve the min-max problem. Specifically, we first perform a loss minimization for the minimum subproblem, followed by a loss maximization for the maximum subproblem. The specific process is illustrated in the blue box of Fig. 5. The convergence proof of the single-loop iterative algorithm is in Appendix A.

Finally, we turn to the third challenge. As depicted in the left and right dotted boxes of Fig. 5, we partition our attack into two phases: warm-up phase and fine-tune phase. In the warm-up phase, we locally train an ensemble model (i.e., a set of substitute models) and pre-compute an initial perturbation on this ensemble model. In the second phase, we fine-tune the aforementioned perturbation based on the query-reply results from the target model, obtaining the final perturbation p.

According to [60], the larger the margin of the classification boundary of the locally trained model, the stronger the transferability of the perturbation. Therefore, in this work, we utilize diverse APK samples to train our local substitute models, aiming to maximize the dissimilarity of their classification boundaries. As perturbations often experience a certain degree of degradation in their attack effectiveness during the transfer process, after the local warm-up phase ends, we conduct a final universality optimization of the perturbation based on the query results of the target model to ensure its attack effectiveness.

3) Algorithm Development: Our perturbation calculation algorithm is described in Algorithm 1. When generating perturbations for malicious rider r, we obtain the modifiable features C to which we can apply the perturbation (including sensitive functions, actions, permissions, etc.). In the warm-up phase, we train four local substitute models F_i ($i \in [1, 4]$) on different sub-datasets of the training data. Unlike traditional ensemble learning algorithms, we do not optimize by averaging the outputs of the four models. Instead, we aim to optimize the universality on the model i^p where the perturbation p performs the worst. Therefore, we begin by selecting the model (Line 6 in Algorithm 1), given by

$$i^p = \underset{i}{\operatorname{argmax}} E_{x_b \in B} F_i(x_b + r + p) \tag{7}$$

where p represents perturbation and is initialized as $p = \{\}$.

Note that E_1 in Eq. (6) signifies the probability of the selected substitute model classifying the sample as malware. It can be considered as a value function that is used to evaluate each perturbation. Minimizing E_1 reduces the likelihood of the model classifying a sample as malware. To optimize universality, among all available perturbations, we select the perturbation that minimizes $E_1(p + c)$. i.e.,

$$\begin{aligned} c^* &= \underset{c}{\operatorname{argmin}} E_1(p+c) - E_1(p) \\ &= \underset{c}{\operatorname{argmin}} E_{x_b \in B}(F_{i^{p+c}}(x_b + r + (p+c)) - F_{i^p}(x_b + r + p)) \end{aligned} \tag{8}$$

Subsequently, we will incorporate the selected c^* into p, obtaining $p = p + c^*$. Then, we iteratively optimize the aforementioned process until E_1 falls below the threshold T_1 (Line 4-9 in Algorithm 1).

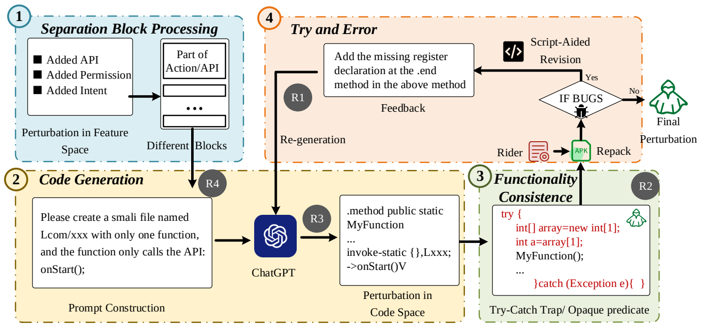
Algorithm 1: Perturbation Generation Algorithm

```
   Input: The malicious rider $m$, the benign carrier
          APKs set $B$, the substitute model
          $F_i, i \in [1,4]$, modifiable features set $C$,
          Threshold $T_j, j \in [1,4]$
1  Initialize: Perturbation $p = \{\}$, Noise $n = \{\}$,
     Iteration for warm-up phase $Iter_1 = 0$, Iteration for
     targeted attack phase $Iter_2 = 0$;
2  while $Iter_1 \leq T_3$ do
3      // Warm-up Phase
4      Stage I: University requirement optimization
5      while $E_1 \geq T_1$ do
6          Select the model $i^*$ according to Eq. (7).
7          Select the modification $c^*$ according to Eq. (8).
8          Add $c^*$ to perturbation$p$: $p = p + c^*$
9          $Iter_1 = Iter_1 + 1$
10     Stage II: Targeting requirement optimization
11     while $E_2 \geq T_2$ do
12         Select the modification $c^{**}$ according to Eq.
             (9).
13         Add $c^{**}$ to perturbation$n$: $n = n + c^{**}$
14         $Iter_1 = Iter_1 + 1$
15     Select the model $i^*$ according to Eq. (10).
16     Select the modification $c^{***}$ according to Eq. (11).
17     Add $c^{***}$ to perturbation$p$: $p = p + c^{***}$
18 while $Iter_2 \leq T_4$ do
19     // Fine-Tune Phase
20     Stage III: Targeted model optimization
21     Select the modification ${p_k}^*$ according to Eq. (12).
22     Delete ${p_k}^*$ from perturbation$p$: $p = p - {p_k}^*$
23     $Iter_2 = Iter_2 + 1$
   Output: Perturbation $p$
```

To meet the targeting requirement, we employ a single-loop iterative optimization algorithm (Line 10 to Line 17 in Algorithm 1). The noise n is initialized as empty sets, i.e., $n = \{\}$. We first optimize the noise n before optimizing the perturbation p. Subsequently, c^{**} are selected to be incorporated into the noise n through solving the following optimization problem (Line 12-13 in Algorithm 1).

$$\begin{aligned} c^{**} =& \underset{c}{\operatorname{argmin}} E_2(n+c,p) - E_2(n,p) \\ =& \underset{c}{\operatorname{argmin}} \sum_i E_{x_b \in B}(F_i(x_b + (r+n+c) + p)) \\ & - E_{x_b \in B}(F_i(x_b + (r+n) + p)) \end{aligned} \tag{9}$$

It is worth noting that when optimizing n, we aim to minimize the mean of E_2 obtained on all models. We then incorporate the selected c^{**} into n and repeat the aforementioned optimization process until E_2 falls below the threshold T_2. At this point, we expect the perturbation p to have no effect on the obtained rider $r + n$. Therefore, we aim to optimize the targeting requirement on the model i^p where the perturbation p performs the worst. This solving process can be represented as follows (Line 15 in Algorithm 1):



Add the missing register declaration at the .end method in the above method

Please create a smali file named Lcom/xxx with only one function, and the function only calls the API: onStart();

```
.method public static
MyFunction
...
invoke-static {},Lxxx;
->onStart()V
```

```
try {
    int[] array=new int[1];
    int a=array[1];
    MyFunction();
    ...
        }catch (Exception e){  }
```

Fig. 6. The code space generation and injection process.

$$i^p = \underset{i}{\operatorname{argmin}} E_{x_b \in B} F_i(x_b + r + p) \tag{10}$$

Then, we recalculate c and incorporate it into the perturbation p (Line 16-17 in Algorithm 1), given by

$$\begin{aligned} c^{***} =& \underset{c}{\operatorname{argmax}} E_2\left(n, p+c\right) - E_2\left(n, p\right) \\ =& \underset{c}{\operatorname{argmax}} E_{x_b \in B}\big(F_{i^{p+c}}\left(x_b + (r+n) + (p+c)\right) \\ &- F_{i^p}\left(x_b + (r+n) + p\right)\big) \end{aligned} \tag{11}$$

In addition, when optimizing Eq. (11), it is essential to ensure that the universality does not degrade. Therefore, we only incorporate c^{***} into the perturbation p when E_1 does not increase. We iterate the optimization of Eq. (9) and Eq. (11). The warm-up phase terminates once the optimization iteration exceeds the threshold T_3.

When attacking the targeted model, the universality of perturbations unavoidably diminishes due to the difference between the substitute model and the target one. Therefore, we iteratively query the target model, utilizing the binary results obtained from the queries (i.e., benign or malicious), to re-optimize universality (i.e., E_1) (Line 18 to Line 23 in Algorithm 1). Additionally, given the substantial size of set C, computing Eq. (8) for each element in C would necessitate a large number of queries. Thus, as an alternative approach, rather than selecting perturbations from C to be added to p, we recalibrate Eq. (9) to identify the poorest perturbation p_k within the existing set of perturbations p and remove it. We establish the maximum optimization rounds T_4 based on the allowed maximum number of queries. Specifically, we will iterate through all perturbations in p to identify the poorest perturbation p_k that, when removed, results in the best universality of p:

$$p_k{}^* = \underset{p_k \in p}{\operatorname{argmin}} E_1\left(p - p_k\right) - E_1\left(p\right) \tag{12}$$

Then we delete $p_k{}^*$ from perturbation p: $p = p - p_k{}^*$. We will repeat the above process until the maximum allowable number of iterations is reached. We show a visualization of the APK feature before and after perturbation in Appendix B.

4) Generation & Injection: After obtaining perturbation in the feature space, it is necessary to map it back to the code space and generate real code (i.e., the Generation phase). The generated code is then piggybacked onto benign carriers (i.e., the Hook phase). In the generation process, existing approaches often 1) insert empty functions [13], 2) add NOP functions [58], or 3) utilize program porting techniques to extract benign code segments from benign carriers [42]. However, the first two methods often fail to generate naturally formed code and leave noticeable modification traces, while the last method lacks the flexibility to choose desired functions to incorporate freely. To address the aforementioned issues, we define four **R**equirements for this process:

R1: Program Normal Execution. After the insertion of perturbations, it is crucial to ensure that the program can be packaged and executed normally.

R2: Functional Consistency. The inserted code must not affect the functionality of the original APK.

R3: Naturalness. There should be no noticeable traces of manual modifications, ensuring the covertness of perturbations.

R4: Flexibility. The adversary should be allowed to freely choose the features to perturb and map them into the code space.

To meet the above requirements, we propose a perturbation generation and injection algorithm, as illustrated in Fig. 6. With the advancement of large language models (LLM) [8], [36], an increasing number of studies have uncovered the capability of these models in code generation. Some works [8] have found that LLMs struggle to generate entire malware samples from complete descriptions. Adversaries can divide the task of generating the entire malware into several subtasks and utilize LLMs to complete each subtask. Then, the adversaries assemble the malicious code generated from these subtasks to form the final malware. Inspired by this, we utilize LLMs to accomplish the generation of perturbations with different blocks. Our perturbation generation and injection process is described below:

1) As shown in the red block of Fig. 6, we partition the feature-space perturbations into blocks, performing code-space perturbation generation for each block separately[5]. To be specific, for newly added permissions, we analyze them block by block. For new actions and APIs, we group at least one action and API into each block. The purpose of this step is twofold. First, it enhances the quality of code produced by the LLM. Second, it improves the readability of the code. Moreover, feeding all feature-space perturbations at once to LLMs will result in a significant decrease in the readability and quality of the generated code-space perturbations.

2) Based on the partitioned feature-space characteristics, we construct prompts and utilize the LLM to generate natural code segment, as shown in the yellow block of Fig. 6. This operation satisfies R3 and R4.

3) It is crucial that these perturbation codes cannot alter the original functionality of the rider and benign carrier.

[5]In our work, we did not consider cases where features are embedded vectors. For such cases, gradient propagation or optimization algorithms would be needed to calculate the necessary perturbations for APIs, actions, or permissions.

Therefore, we utilize techniques such as Try-Catch Trap[29] or opaque predicates [42] to insert perturbation and prevent the perturbation from being detected by static analysis. This process satisfies R2 and is shown in the green block of Fig. 6.

4) Subsequently, we hook all perturbed code and riders onto the carrier and perform repackaging. This process may result in errors; if so, we rely on error messages and utilize a pre-designed script file to assist in constructing new feedback and prompt the model to output the corresponding code segment again. The script file contains some potential error correction solutions to aid in the construction of prompts. The detailed description about the script file is in Appendix C. This process satisfies R1 and is shown in the orange block of Fig. 6. We repeat the above steps until the program is successfully repackaged.

In addition, our prompt generation uses a template-based approach. In our method, LLM employs gpt-3.5-turbo, with max_tokens set to 1024 and temperature set to 0.8, without any special stop symbols. LLM only generates a single response for one query. Our templates and an example of the code-space perturbation are given in Appendixes D and E, respectively.

C. Benign Carrier Selection

As shown in STEP 3 in Fig. 3, we have two factors for selecting benign carriers. These factors are motivated by two considerations: 1)Different benign samples have varying concealment capabilities for malicious riders. Selecting benign carriers can increase the success rate of generating adversarial piggybacked malware. 2) Different adversaries have different target audiences for different riders. For example, adversaries are more inclined to insert riders into game software. This is because game software has a wider audience and is more likely to facilitate the spread of malicious code.

In regards to the first point, we rank the comprehensive performance of adversarial piggybacked malware generated from various benign sources on substitute model, selecting them in descending order of performance scores, defined by

$$\text{Score}(x_b) = \sum_i -F_i(x_b + r + p) \tag{13}$$

Regarding the second point, we collect the labels of different benign carriers in the dataset using Google Market [43], and classify the benign software based on these labels. In this study, we consider the following categories: GAME, EDUCATION, TOOLS, BUSINESS, MUSIC, FINANCE, LIFESTYLE, and more. In the experiment section, we validate the effectiveness of the algorithms for these different categories of benign carriers.

D. Adversarial Piggybacked Malware Construction

Upon obtaining the adversarial perturbation and the malicious rider, we need to insert them into benign carriers. Specifically, it is necessary to unpack and decompile the benign apps using tools such as APKTOOL [3] and AXMLPrinter2 [5] to obtain the underlying code.

Subsequently, we hook the evasive rider composed of the malicious rider and the adversarial perturbation with benign carriers. Specifically, this operation involves: 1) inserting new resource files [6] into the assets, lib, and res folders of the carrier, 2) adding new smali files into the smali folder, 3) incorporating new permissions into the AndroidManifest.xml file, 4) adding new actions into the AndroidManifest.xml file, and 5) hooking new methods into the smali startup file to ensure the execution of malicious behaviors.

Finally, we utilize tools such as APKTOOL and jarsigner [25] to repackage and resign the modified APK file.

V. Experiments

In this section, we conduct experiments to evaluate adversarial piggybacked malware on the existing academic AMD models and commercial engines. Additionally, we validate and analyze the functionality of each module in our algorithm.

A. Experimental Setup

Target Model. We select seven state-of-the-art machine learning-based AMD methods for malware detection, encompassing five syntax feature based classifiers and two semantic feature based classifiers. The syntax feature based classifiers use two prominent features: DREBIN features [4] and FD-VAE features [30]. DREBIN features consist of 8 feature classes, namely hardware components, requested permissions, app components, filtered intents, restricted API calls, used permissions, suspicious API calls, and network addresses. Notably, when DREBIN features are employed, each APK generates over 1000,000 features. The excessive sparsity of feature vectors may degrade the performance of AMD models. Following the recommendation proposed by [54], [24], we utilize the L2 regularized Linear Support Vector Machine (LinearSVM) to select the top 1,000 features crucial for identifying malware. FD-VAE features comprise 379 features, including 147 permissions, 126 intent-action pairs, and 106 sensitive APIs. Based on two significant characteristics of piggybacked APKs, we expand the FD-VAE features.

1) [31] analyzed common sensitive functions and permissions in piggybacked APK files, which we incorporated into the classification features, expanding the FD-VAE features to 176 permissions, 155 actions, and 130 sensitive APIs, termed as FD-VAE-E1.

2) [31] discovered adversaries often repeatedly add benign carrier's existing permissions or actions when constructing piggybacked APKs. This operation results in some permissions being declared redundantly. Therefore, we transform the binary values of permissions and actions in FD-VAE-E1 to their actual occurrence counts, and for ease of training convergence, we perform max normalization on the permission/action section. This feature is referred to as FD-VAE-E2.

What's more, we choose another targeted clone app detection tool MALPACK [45], which also uses a syntax feature. Different from the above methods, MALPACK doesn't use piggybacked malware during its training process. Due to this specific setup, we discuss our attack performance on this method in Appendix F.

Furthermore, we select two semantic features: MaMaDroid features [37] and APIGraph features [59]. MaMaDroid extracts

[6]Malicious riders in advertising often contain resources such as advertisement images.

TABLE I. THE ATTACK PERFORMANCE ON DIFFERENT ANDROID MALWARE DETECTION METHODS. (↓) INDICATES THAT A LOWER VALUE IS PREFERRED, AND VICE VERSA.

	Scenarios	DREBIN		FD-VAE		FD-VAE-E1		FD-VAE-E2		MaMaDroid		APIGraph	
		U(↑)	T(↓)	U(↑)	T(↓)	U(↑)	T(↓)	U(↑)	T(↓)	U(↑)	T(↓)	U(↑)	T(↓)
ER	W	1.000	0.480	1.000	0.518	1.000	0.301	1.000	0.400	0.979	0.358	0.996	0.508
	G	0.879	0.348	0.803	0.315	0.825	0.270	0.769	0.454	0.859	0.299	0.843	0.458
	B_S	0.876	0.407	0.787	0.337	0.889	0.389	0.730	0.397	0.832	0.338	0.776	0.414
	B_D	0.773	0.384	0.821	0.351	0.865	0.334	0.835	0.419	0.853	0.297	0.849	0.426
UR	W	1.000	0.374	1.000	0.429	1.000	0.267	1.000	0.360	0.991	0.384	1.000	0.485
	G	0.842	0.338	0.890	0.288	0.921	0.272	0.879	0.424	0.853	0.330	0.865	0.427
	B_S	0.888	0.372	0.821	0.245	0.922	0.318	0.848	0.379	0.852	0.368	0.820	0.396
	B_D	0.896	0.382	0.792	0.241	0.883	0.319	0.867	0.371	0.880	0.337	0.815	0.392

API call graphs from smali files and employs Markov metrics to transform the call graph into a vector. APIGraph is designed to unveil semantic similarities between Android APIs by constructing a relational graph of Android APIs based on official documents. In addition, we choose DNN as the classification model. These AMD detectors' construction and performance are shown in Appendixes G and H.

Datasets. The first dataset used in our experiments is derived from [31]. This dataset encompasses over 2 million apps collected from various sources, including Google Play, appchina, anzhi and open-source repositories such as F-Droid, and research datasets like the MalGenome dataset. The data was analyzed and processed, resulting in the extraction of 1,497 pairs of original APKs and piggybacked APKs.

Another dataset we utilize is a well-known dataset used for Android adversarial examples validation [42]. This dataset comprises 134,759 samples (including 135,859 benign applications and 14,775 malware samples), and it has already been processed by Pierazzi et al. [42], following the labeling criteria outlined in Tesseract [41].

In our experiments, the first dataset is used to extract malicious riders, while the second dataset is utilized to train the target model, substitute models and validate the effectiveness of our attack algorithm. After applying the Malicious Rider Extraction presented in Section IV-A, the first dataset yielded 348 malicious riders. Through comparative analysis and rigorous screening, we identify 65 malicious riders, whose function can be guaranteed to be correct. In Appendix I, we evaluate our algorithm's performance on much more potential riders.

For the second dataset, to ensure differentiation between the samples used by the local and target models in gray-box and black-box attacks, we allocate 45% of the samples for training the local model, 45% for training the target model, and reserve 10% of the samples for testing the target model's effectiveness. As the second dataset is not specifically collected for piggybacked data, in order to enhance its detection capabilities for piggybacked apps, we augment the malicious data. For the collected riders, we use 60% of the riders for training and 40% for testing. Each rider is randomly inserted into 500 benign carriers, to enhance the detectors' accuracy in identifying piggybacked apps.

Metric. We use attack success rate (ASR), and the number of query (Noq) for performance evaluation. ASR corresponds to the ratio of the number of successfully generated AEs (denoted by $N_{success}$) to the number of malicious examples used for AE generation (denoted by N_{total}), i.e., $ASR = N_{success}/N_{total}$. Noq is defined as the number of interactions between our attack model and the target model.

Experimental Environment. All experiments are conducted using 64GB of RAM, an i7-12700KF CPU, and an NVIDIA RTX 3090 GPU.

B. Overall Attack Performance

Attack Effectiveness: We first validate the proposed algorithm for its effectiveness in attacking various academic Android malware detection systems, namely DREBIN, FD-VAE, FD-VAE-E1, FD-VAE-E2, Ma-MaDroid, and APIGraph. We consider three attack scenarios. The first scenario is the white-box attack (W), where the adversary has access to the structure, parameters and training data of the target model. The second scenario is the gray-box attack (G), where the adversary has access to the structure of the target model but has no knowledge of its parameters and training data. The third scenario is the black-box attack (B), where the adversary has no knowledge of the structure, parameters, and training data of the target model. In a black-box scenario, if the target model uses a deeper architecture compared to the local model, it is referred to as B_D. In the opposite case where the target model uses a shallower architecture, it is termed B_S. B_D and B_S are introduced to demonstrate that adversarial perturbations generated on a local substitute model can transfer to classification models with different structures.

During the training of the detection models, we utilized a subset of malicious riders. We refer to the riders present in the training dataset of the target model as ER (Existing Riders) and the riders not included in the training dataset as UR (Unknown Riders). In both ER and UR scenarios, we only select correctly detected piggybacked apps for testing. To validate the universality requirement, we perturb the aforementioned riders and hook them onto 200 benign hosts to test if our algorithm could deceive the detection models. The success rate of these attacks is termed as ASR_U. To validate the targeting requirement, we inserted the perturbations into mismatched riders and then randomly hooked them onto 200 benign hosts. The success rate of these attacks on the aforementioned samples is referred to as ASR_T.

The experimental results are shown in Table I. The first column in the table indicates whether the rider is included in the training set collected by AMD developer, the second column refers to different attack scenarios, and columns three

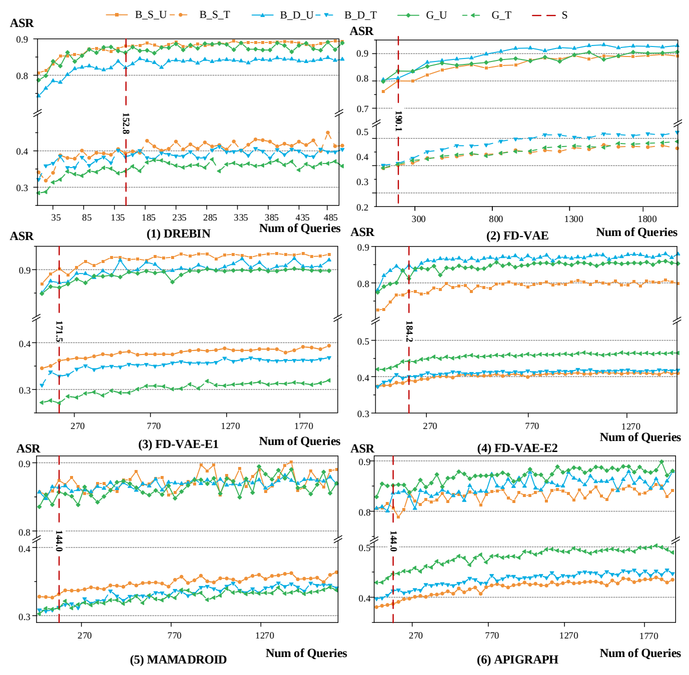


Fig. 7. The query counts toward different AMD systems.

to eight represent different features used by the malware detection model. In all scenarios, we can observe that when the perturbation matches the rider, the attack success rate of our algorithm reaches 88.3%. Conversely, when the perturbation does not match the rider, the attack success rate is only 36.9%, a decrease of 51.4%. Furthermore, in the white-box scenario, the perturbation tends to overfit to different riders, leading to an increase in ASR_U while ASR_T also rises. In contrast, in gray-box and black-box scenarios, the overfitting problem is mitigated due to differences among the models.

Number of queries: In the proposed algorithm, we need to query the target model and fine-tune perturbation based on the results. Therefore, we evaluate the number of queries required by our algorithm. We refer to the riders matched with the perturbations as matched riders and those not matched with the perturbations as mismatched riders.

We gradually increase the number of queries during the fine-tuning process to observe the changes in the ASR on the matched and mismatched riders. Since white-box attacks do not involve the fine-tuning process, we only demonstrate the changes in the gray-box and black-box scenarios. The experimental results are shown in Fig. 7, where B_S_U, B_D_U and G_U represent the ASR on the matched rider in the B_S, B_D and G scenarios. B_S_T, B_D_T and G_T indicate the ASR on the mismatched rider. The red line represents the selected query times in our experiments. It is worth noting that the increase in the ASR by perturbation on matched riders is greater than that on mismatched riders. This demonstrates the effectiveness of the fine-tuning step in our algorithm. Moreover, as the number of queries increases, the impact of perturbation on matched and mismatched riders gradually saturates, allowing for the selection of a smaller number of queries to meet the attack requirements. In our experiment, under six Android malware detection models, the selected query frequencies are 152.8, 190.1, 171.5, 184.2, 144.0, and 144.0. These query frequencies are deemed acceptable, and the perturbation can be reused continuously to generate tens of thousands of adversarial piggybacked malware.

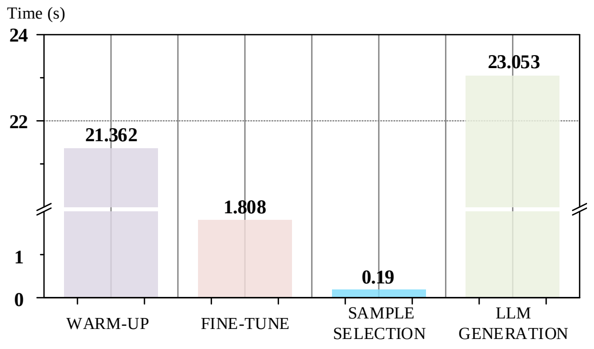


Fig. 8. Time consumption of different steps.

Perturbation Generation Time: We analyze the time consumption of our algorithm in perturbation generation. We present the average results using DREBIN features in Fig. 8. The horizontal axis denotes the four steps of our algorithm, while the vertical axis represents time consumption in seconds (s). It is evident that for a rider, the most time-consuming steps occur in the LLM generation and warm-up phase, taking approximately 23s and 21s, respectively. During the LLM generation phase, it requires approximately 6.6 rounds of trial and error. The time consumption of our algorithm is acceptable.

C. Comparison with SOTA Methods

TABLE II. Comparison of Attack Performance with State-of-the-Art Methods

Methods	W		B_S		G		B_D	
	U(↑)	T(↓)	U(↑)	T(↓)	U(↑)	T(↓)	U(↑)	T(↓)
RN	0.044	0.019	0.041	0.029	0.026	0.019	0.040	0.030
HIV-JSMA	0.456	0.381	0.197	0.259	0.191	0.261	0.170	0.224
HIV-CW	0.599	0.525	0.156	0.265	0.269	0.321	0.126	0.216
Pierazzi et.al.	0.603	0.481	0.760	0.635	0.662	0.567	0.605	0.514
EvadeDroid	0.468	0.480	0.360	0.420	0.525	0.521	0.386	0.430
Malpatch	0.891	0.629	0.144	0.159	0.242	0.239	0.103	0.114
Ours	**1.000**	**0.439**	**0.880**	**0.393**	**0.862**	**0.344**	**0.821**	**0.383**

Here we introduce six attack algorithms for comparative analysis. First, to validate the effectiveness of our adversarial perturbations, we design a Random-Noise (i.e., RN) attack algorithm. This approach randomly generates an adversarial perturbation of the same size as the original input. Then, we select two attack algorithms (i.e., HIV-CW and HIV-JSMA) proposed in the Android HIV framework [13] and two attack algorithms proposed in the works of [42] and [7] (termed as Pierazzi et.al. and EvadeDroid). These algorithms generate adversarial perturbations for individual APKs, while we focus on the adversarial perturbation for various APKs generated by a particular malicious rider. We apply the perturbation of an individual APK to other piggybacked malware generated by the same malicious rider to verify the universality requirement. Furthermore, we test the targeting requirement by applying the perturbation to piggybacked malware produced by different malicious riders. Finally, we select a universal attack method, MalPatch [56], which generates a universal perturbation for a specific instance of piggybacked malware.

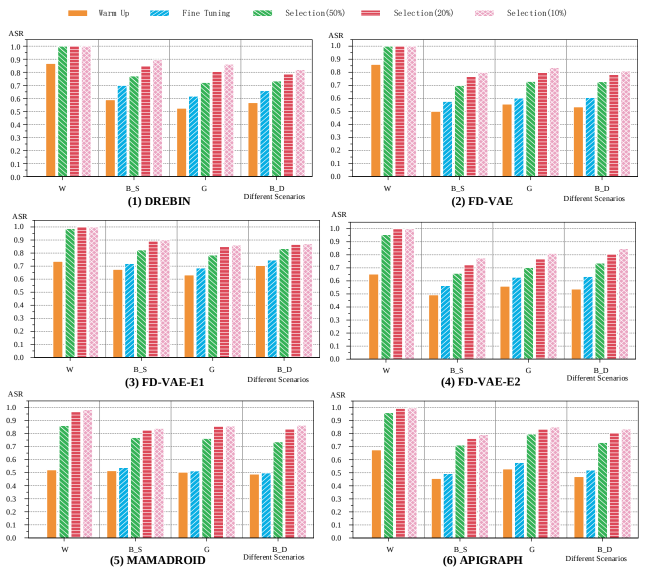


Fig. 9. The ablation experiment.

The experimental results are presented in TABLE II, which indicate that existing attack methods often fail to simultaneously satisfy the universality requirement and the targeting requirement (i.e., achieving a high U value and a low T value). In contrast, our method effectively meets both requirements, resulting in superior attack performance.

D. Ablation Experiment

In this section, we investigate the roles played by different modules, thus we conduct ablation experiments to validate each step in the experiment. Adversarial perturbation generation consists of two phases: the Warm-up phase, and the Fine-tuning phase. Besides, there is a carrier selection phase that can improve the attack success rate. Based on the varying numbers of benign carriers selected, we decompose the process of benign carrier selection into three phases: selecting 50%, 20%, and 10%. Then we get five phases that are respectively represented as: WARM UP, FINE TUNING, SELECTION(50%), SELECTION(20%), and SELECTION(10%). In this experiment, we will demonstrate the changes in ASR_U in each of the aforementioned phases[7].

The experimental results are presented in Fig. 9. The six subplots represent the attack performance against different Android malware detection systems. Each subplot contains three attack scenarios: white-box (W), gray-box (G), and black-box (B_S and B_D). Within each scenario, the success rates of attacks are shown for our algorithm across five phases. The orange, blue, green, red, and pink bars respectively represent the WARM UP, FINE TUNING, SELECTION(50%), SELECTION(20%), and SELECTION(10%) phases. During the white-box attack phase, the target model is already known, eliminating the need for a warm-up phase. The experimental results indicate that each of our phases progressively increases the perturbation's generality, enabling malicious riders to evade detection. It can be observed that across all scenarios, we achieve an attack success rate of 57.3% through the warm-up phase. The fine-tuning phase enhances the success rate by approximately 12%. When selecting 50% benign carriers, there is a 21% increase in the attack success rate.

[7] ASR_T is not shown because the benign carrier selection does not affect this value, and the changes in ASR_T during the fine-tuning phase have already been shown in Fig. 7.

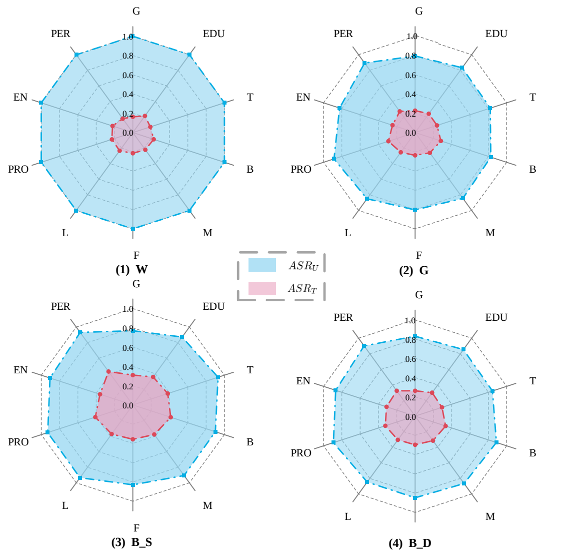


Fig. 10. The attack effectiveness on different categories of APPs.

E. Performance on Different Categories of Carriers

Attack Effectiveness: During attacks, adversaries may leverage popular benign apps as carriers for hooking. Therefore, in this section, we validate the effectiveness of our algorithm on different categories of benign carriers. Specifically, following the work of Li et al. [31], which analyzed the five most common types of benign carriers as GAME, TOOLS, PRODUCTIVITY, ENTERTAINMENT, and PERSONALIZATION. We also examine the seven most prevalent types of benign software in our dataset: GAME, EDUCATION, TOOLS, BUSINESS, MUSIC, FINANCE, and LIFESTYLE. Consequently, we select ten benign apps for analysis: GAME, EDUCATION, TOOLS, BUSINESS, MUSIC, FINANCE, LIFESTYLE, PRODUCTIVITY, ENTERTAINMENT, and PERSONALIZATION, denoted as G, EDU, T, B, M, F, L, PRO, ENT, PER. These benign carriers are uploaded to the Google Play Store to retrieve their specific classification labels. Due to space limitation, we only present the results for FD-VAE-E1.

The experimental results depicted in Fig. 10 show the attack success rates ASR_U and ASR_T in scenarios W, G, B_S, and B_D. The experiments reveal that in the white-box attack scenario (W), we achieve nearly perfect attack effectiveness (i.e., ASR_U of 1.000 and ASR_T of 0.313). However, in gray-box and black-box scenarios, there is a certain decline in attack effectiveness (with ASR_U of 0.913 and ASR_T of 0.349). Furthermore, our algorithm demonstrates very high attack success rates across different types of popular benign carriers. Regarding ASR_U, we attain the highest average attack success rate on the PERSONALIZATION type of benign carriers in all scenarios, reaching 96%. Even on the least favorable GAME type of benign carriers, we achieve an average attack success rate of 89%.

Attack Efficiency: One of the perils of piggybacked

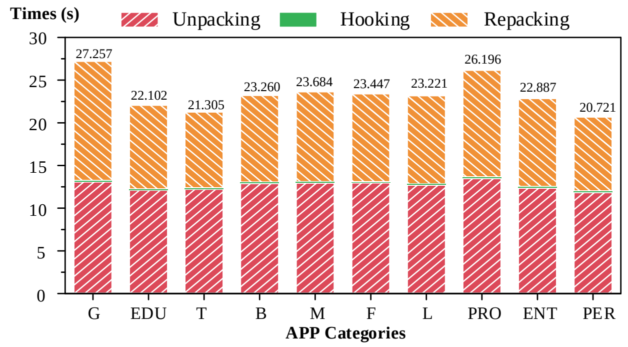


Fig. 11. The attack efficiency on different categories of APPs.

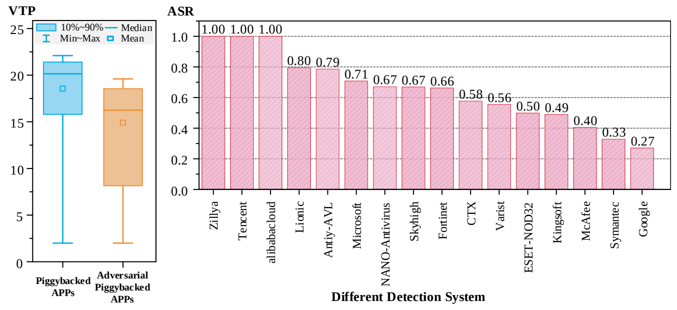


Fig. 12. The attack performance on VirusTotal.

malware lies in the rapid and mass production of malware. Hence, in this section, we validate the generation time of adversarial piggybacked malware across different categories of rider apps. The time consumption of our algorithm is divided into three parts: 1) the unpacking time of benign payloads, 2) the hooking time of malicious riders, and 3) the repackaging time of adversarial piggybacked apps.

Fig. 11 shows the unpacking time represented by red bars, hooking time by green bars, and repackaging time by orange bars. The results indicate that the average time to generate an adversarial piggybacked app across all types of benign riders is 23.4 seconds, with average times for unpacking, hooking, and repackaging being 12.6 seconds, 0.08 seconds, and 10.6 seconds, respectively. The short duration of the hooking process may not be clearly discernible in the graph. Furthermore, we observe that the average time required for game-type riders is the longest at 27.5 seconds, possibly due to the complexity of game apps, which typically contain a large number of resource files and intricate file structures. In contrast, simpler rider categories like PERSONALIZATION only require 20.7 seconds on average.

F. Real-World Experiments

In this section, we validate our algorithm's ASR on real-world engines. Each malicious rider is paired with its corresponding perturbation and injected into twenty benign carriers, which are then uploaded to VirusTotal [51] to assess the impact of perturbations on commercial engines. VirusTotal is a renowned industry platform for malware detection, integrating over 90 commercial malware detection models. Upon uploading malware, these commercial detection systems evaluate it and output the number of engines that classify it as malicious. We refer to this count as VirusTotal positives (VTP).

The experimental results, as depicted in Fig. 12, illustrate the impact of perturbations on VirusTotal detection. The left graph compares the VTP before and after perturbation insertion. Prior to perturbation, the average VTP stands at 18.6, decreasing to an average of 14.9 post-perturbation. Furthermore, the right graph showcases the attack success rates on prominent engines. For instance, the adversarial piggybacked apps generated achieved an ASR of 71% on Microsoft and an ASR of 67% on NANO Antivirus.

VI. Related Work

A. Piggybacked APP Detection

Piggybacked apps represent a highly detrimental form of malware generation. Existing research on the detection of such apps can generally be categorized into two main approaches. 1) Similarity Analysis Methods: These methods involve comparing the software to be tested with all software in a local database one by one. 2) Machine Learning Methods: These methods directly detect piggybacked apps. For instance, DNADroid [14] extracts the Program Dependence Graph as a fingerprint and utilizes the VF2 algorithm for similarity comparison to identify these apps. DroidSim [47] uses the component-based control flow graph as the fingerprint. PiggyAPP [62] extracts Android APIs and permissions as a feature vector, organizes these vectors into a metric space (Vantage Point Tree), and employs KNN to find similar feature vectors. However, these methods require comparing APK files, which can be time-consuming. With the advancement of machine learning, an increasing number of methods are starting to utilize machine learning techniques to detect piggybacked apps. For example, Li et al. [32] analyze numerous piggybacked app cases, identify some unique features of piggybacked apps based on prior knowledge, and use these features for classification. DAPASA [18] utilizes the sensitive subgraph (SSG) to profile the most suspicious behavior of an app and then employs machine learning algorithms to determine whether the app is piggybacked. As the number of Android apps continues to grow, comparison-based methods often incur significant time costs, while machine learning-based methods are susceptible to adversarial example attack methods.

B. Adversarial Examples Attack toward AMD

Existing adversarial attacks against AMD include feature space attacks [15], [16], [23], [35] and code space attacks [20], [22], [29], [61]. The former involves modifying the features inputted into machine learning models, while the latter alters the code in the code space. In feature space attacks, adversaries manipulate various features extracted from APKs, encompassing diverse data domains such as graph data, binary vector data, image data, and more. In code space attack, adversaries accomplish perturbations with low-level language code. For instance, Chen et al. [13] propose inserting empty or no-op functions to implement perturbations at the smali level. Li et al. [29] introduce perturbation code into smali code by using Try-Catch Traps when attacking function call graphs. Pierazzi et al [42] present opaque predicate to modify function call graphs. Zhao et al. [61] propose a structured adversarial example generation method, outline four smali code modification approaches, and employed deep reinforcement learning to find optimal operational methods. However, the aforementioned attack algorithms all introduce perturbations to existing malware and do not take into account the unique formation of piggybacked apps, a specific type of malware.

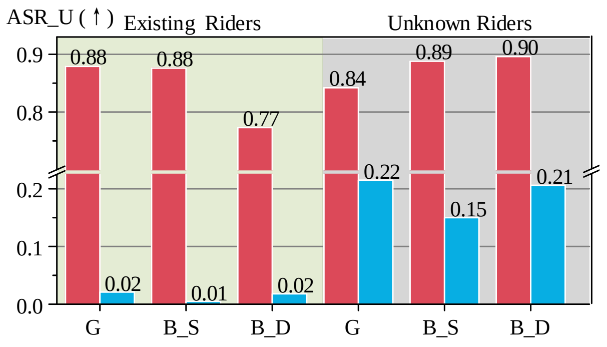


Fig. 13. The attack performance on adversarially retrained model.

C. The Attack Constraints in AMD

Pierazzi et al. [42] propose constraints for problem-space attacks for AMD. Our attack constraints build upon and modify their framework. Specifically, Pierazzi et al. introduce four attack constraints:

Available transformations: The perturbations must meet practical requirements. We specify this constraint to ensure the normal operation of the program (i.e., Program Normal Execution Constraint). *Preserved semantics*: The semantics of the samples must remain unchanged before and after perturbation. We abstract this point in our work to mean that the functionality of the malicious code segment does not change before and after perturbation (i.e., Functional Consistency Constraint). *Plausibility*: The perturbations must appear realistic upon manual inspection. In our work, we address this requirement based on the concept of Naturalness of the language (i.e., Naturalness Constraint). *Robustness to preprocessing*: Perturbations cannot be easily removed by analysis tools, such as being treated as dead code and deleted. Our method inherently meets this requirement by integrating both malicious and perturbation codes with benign carriers. Therefore, we did not explicitly address this in our manuscript. Finally, we introduce an additional constraint, the flexibility constraint, which ensures that the adversary can freely select perturbations from the feature space.

VII. Discussion

A. Potential Defense

To defend against adversarial piggybacked malware, we offer two methods for the developers of AMD models. First, Raff et al. [44] suggest segmenting software into different chunks using moving windows, and then detecting malware by assessing the maliciousness of these chunks. This method can mitigate the impact of perturbations, potentially thwarting attacks. Second, adversarial retraining has been widely considered as one of the most effective defenses against adversarial attacks. This method assumes that the app market can use similarity analysis to identify a limited number of adversarial piggybacked malware, thereby enhancing the AMD model's performance by adversarial retraining.

To verify the second method, we consider an extreme scenario where users acquire 100 instances of adversarial piggybacked malware for each malicious rider and incorporate them into the training set for model retraining. The attack success rates ASR_U on matched riders are illustrated in Fig. 13. Red bars indicate the results of the original model, whereas blue bars represent the results of the adversarially retrained model. The bars with a green background denote the attack results on existing riders, and those with a gray background reflect the results on unknown riders. The experimental results demonstrate that adversarial retraining can mitigate the attack effectiveness of adversarial piggybacked malware, reducing their ASR from 85.9% to 10.5%. However, this defense has two significant drawbacks. The first is the difficulty of obtaining a substantial amount of adversarial piggybacked malware for training in real-world scenarios. The second is that the model, strengthened by adversarial retraining, tends to overfit the adversarial piggybacked malware present in the training set. As shown in the figure, the defense effectiveness of the model against adversarial piggybacked malware generated from Unknown Riders is 17.6% lower than that against malware derived from existing riders. In the future, we will explore more robust malware detection models.

B. Functional Consistency Verification

To verify the functional consistency, we need to compare the functional differences among three types of software: benign carrier, piggybacked malware, and adversarial piggybacked malware. We first verify the piggybacked malware preserves the functionality of the benign carrier. This has been confirmed in the Malicious Rider Extraction step. We then verify that the functionality of the malicious rider remains unaffected. However, some malicious functionalities are deeply hidden and challenging to uncover and dynamically validate. To verify that our perturbations do not impact the malicious behavior's functionality, we use a typical malicious rider (i.e., DroidDream) as an example. DroidDream injects malicious code segments into existing benign carriers. It collects partial information from the phone and sends it to specific servers during the night. It also downloads additional malicious installation packages, posing a severe security threat to the user's device. We conducted the dynamic analysis to determine whether this type of piggybacked malware can execute malicious functions after injecting perturbations. The specific experiments are detailed in Appendix J, and the results indicate that all adversarial piggybacked malware can run normally and exhibit malicious behavior.

C. LLM for Perturbation Generation

We believe that the ability of large language models (LLMs) to generate code for specific tasks [11], [27], [19], especially for low-level language like smali, still needs improvement due to the lack of relevant corpus in the LLM's dataset, which affects its effectiveness. However, utilizing LLM make perturbing code closely resemble human-written code shows promise. This is due to techniques like Try-Catch traps, which reduce the need for strict syntax correctness.

VIII. Ethics Statement

Our work falls within the realm of **offensive research**. Our main goal is to make academic and industrial communities pay more attention to adversarial example attacks against Android malware detection systems. In the section of Potential Defense, we have deliberated on potential defense mechanisms against our attack method. We ensure that the relevant technology will only be utilized for academic research purposes.

IX. Conclusion

In this paper, we introduce an automated mass evasive malware generation approach that combines adversarial example technique with piggybacking. The malware generated through this method is referred to as adversarial piggybacked malware. By extracting malicious riders, generating perturbations, and selecting benign carriers, we achieve the automated generation of a significant volume of adversarial piggybacked malware. We validate the effectiveness of our method on six academic detection models and the commercial engine integration platform VirusTotal. Finally, we discuss potential defense methods and future directions.

Acknowledgment

This work was supported by the Postdoctoral Fellowship Program of CPSF under Grant Number GZB20240248, the China Postdoctoral Science Foundation under Grant Number 2024M751010 and the open foundation of Key Laboratory of Cyberspace Security, Ministry of Education of China and Henan Key Laboratory of Network Cryptography Number KLCS20240305.

References

[1] 42matters., "Google play statistics and trends 2024," https://42matters.com/google-play-statistics-and-trends, 2024.

[2] Androidguard, "Androidguard." 2024, https://github.com/androguard/androguard.

[3] Apktool, "Apktool." 2024, https://apktool.org/.

[4] D. Arp, M. Spreitzenbarth *et al.*, "DREBIN: effective and explainable detection of android malware in your pocket," in *Proc. NDSS*, 2014.

[5] AXMLPrinter2, "Axmlprinter2." 2024, https://github.com/digitalsleuth/AXMLPrinter2.

[6] A. Bartel, J. Klein *et al.*, "Improving privacy on android smartphones through in-vivo bytecode instrumentation," *CoRR*, vol. abs/1208.4536, 2012.

[7] H. Bostani and V. Moonsamy, "Evadedroid: A practical evasion attack on machine learning for black-box android malware detection," *Comput. Secur.*, vol. 139, p. 103676, 2024.

[8] M. Botacin, "Gpthreats-3: Is automatic malware generation a threat?" in *Proc. IEEE SPW*. IEEE, 2023, pp. 238–254.

[9] N. Carlini and D. A. Wagner, "Towards evaluating the robustness of neural networks," in *Proc. IEEE S&P*, 2017.

[10] R. L. Castro, L. Muñoz-González *et al.*, "Universal adversarial perturbations for malware," *CoRR*, vol. abs/2102.06747, 2021.

[11] H. Chen, A. Saha *et al.*, "Personalized distillation: Empowering open-sourced llms with adaptive learning for code generation," in *Proc. EMNLP*. Association for Computational Linguistics, 2023, pp. 6737–6749.

[12] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on android markets," in *Proc. ICSE*. ACM, 2014, pp. 175–186.

[13] X. Chen, C. Li *et al.*, "Android HIV: A study of repackaging malware for evading machine-learning detection," *IEEE Trans. Inf. Forensics Secur.*, vol. 15, pp. 987–1001, 2020.

[14] J. Crussell, C. Gibler, and H. Chen, "Attack of the clones: Detecting cloned applications on android markets," in *Proc. ESORICS*, vol. 7459. Springer, 2012, pp. 37–54.

[15] A. Darwaish, F. Naït-Abdesselam *et al.*, "Robustness of image-based android malware detection under adversarial attacks," in *Proc. ICC*. IEEE, 2021, pp. 1–6.

[16] A. Demontis, M. Melis *et al.*, "Yes, machine learning can be more secure! A case study on android malware detection," *IEEE Trans. Dependable Secur. Comput.*, vol. 16, no. 4, pp. 711–724, 2019.

[17] Droiddream, "Droiddream." 2024, https://www.webopedia.com/definitions/droiddream/.

[18] M. Fan, J. Liu *et al.*, "DAPASA: detecting android piggybacked apps through sensitive subgraph analysis," *IEEE TIFS*, vol. 12, no. 8, pp. 1772–1785, 2017.

[19] Q. Gu, "Llm-based code generation method for golang compiler testing," in *Proc. ESEC/FSE*. ACM, 2023, pp. 2201–2203.

[20] S. Gu, S. Cheng, and W. Zhang, "From image to code: Executable adversarial examples of android applications," in *Proc. ICCAI*. ACM, 2020, pp. 261–268.

[21] N. Gunantara, "A review of multi-objective optimization: Methods and its applications," *Cogent Engineering*, vol. 5, no. 1, p. 1502242, 2018.

[22] P. He, Y. Xia *et al.*, "Efficient query-based attack against ml-based android malware detection under zero knowledge setting," in *Proc. CCS*. ACM, 2023, pp. 90–104.

[23] R. Hou, X. Xiang *et al.*, "Universal adversarial perturbations of malware," in *Proc. CSS*, vol. 12653. Springer, 2020, pp. 9–19.

[24] M. Hurier, G. Suarez-Tangil *et al.*, "Euphony: harmonious unification of cacophonous anti-virus vendor labels for android malware," in *Proc. MSR*, 2017.

[25] jarsigner, "jarsigner." 2024, https://docs.oracle.com/javase/8/docs/technotes/tools/windows/jarsigner.html.

[26] P. Jing, Q. Tang *et al.*, "Too good to be safe: Tricking lane detection in autonomous driving with crafted perturbations," in *Proc. USENIX Security*, 2021.

[27] M. Kazemitabaar, X. Hou *et al.*, "How novices use llm-based code generators to solve CS1 coding tasks in a self-paced learning environment," in *Proc. ICER*. ACM, 2023, pp. 3:1–3:12.

[28] W. D. la Cadena, A. Mitseva *et al.*, "Trafficsliver: Fighting website fingerprinting attacks with traffic splitting," in *Proc. CCS*. ACM, 2020, pp. 1971–1985.

[29] H. Li, Z. Cheng *et al.*, "Black-box adversarial example attack towards FCG based android malware detection under incomplete feature information," in *Proc. USENIX Security*. USENIX Association, 2023.

[30] H. Li, S. Zhou *et al.*, "Robust android malware detection against adversarial example attacks," in *Proc. WWW*. ACM / IW3C2, 2021, pp. 3603–3612.

[31] J. Li, L. Sun *et al.*, "Significant permission identification for machine-learning-based android malware detection," *IEEE Trans. Ind. Informatics*, vol. 14, no. 7, pp. 3216–3225, 2018.

[32] L. Li, D. Li *et al.*, "Understanding android app piggybacking: A systematic study of malicious code grafting," *IEEE TIFS*, vol. 12, no. 6, pp. 1269–1284, 2017.

[33] T. Lin, C. Jin, and M. I. Jordan, "On gradient descent ascent for nonconvex-concave minimax problems," in *Proc. ICML*, vol. 119. PMLR, 2020, pp. 6083–6093.

[34] Y. Lin, Y. Lai *et al.*, "Identifying android malicious repackaged applications by thread-grained system call sequences," *Comput. Secur.*, vol. 39, pp. 340–350, 2013.

[35] X. Liu, X. Du *et al.*, "Adversarial samples on android malware detection systems for iot systems," *Sensors*, vol. 19, no. 4, p. 974, 2019.

[36] Z. Liu, Y. Tang *et al.*, "No need to lift a finger anymore? assessing the quality of code generation by chatgpt," *IEEE Trans. Software Eng.*, vol. 50, no. 6, pp. 1548–1584, 2024.

[37] E. Mariconti, L. Onwuzurike *et al.*, "Mamadroid: Detecting android malware by building markov chains of behavioral models," in *Proc. NDSS*, 2017.

[38] M. Nasr, A. Bahramali, and A. Houmansadr, "Defeating dnn-based traffic analysis systems in real-time with blind adversarial perturbations," in *Proc. USENIX Security*, 2021.

[39] A. Nedic and A. E. Ozdaglar, "Subgradient methods for saddle-point problems," *J. Optimization Theory and Applications*, vol. 142, no. 1, pp. 205–228, 2009.

[40] K. T. Nguyen and N. T. Hung, "The minmax regret inverse maximum weight problem," *Appl. Math. Comput.*, vol. 407, p. 126328, 2021.

[41] F. Pendlebury, F. Pierazzi *et al.*, "TESSERACT: eliminating experimental bias in malware classification across space and time," in *Proc. USENIX Security*. USENIX Association, 2019, pp. 729–746.

[42] F. Pierazzi, F. Pendlebury *et al.*, "Intriguing properties of adversarial ML attacks in the problem space," in *Proc. IEEE S&P*. IEEE, 2020, pp. 1332–1349.

[43] G. Play, "Google play." 2024, https://play.google.com/store/apps.

[44] E. Raff, J. Barker *et al.*, "Malware detection by eating a whole EXE," in *Proc. The Workshops of AAAI*, vol. WS-18. AAAI Press, 2018, pp. 268–276.

[45] H. Rafiq, N. Aslam *et al.*, "Andromalpack: enhancing the ml-based malware classification by detection and removal of repacked apps for android systems," *Scientific Reports*.

[46] S. K. Singh and G. E. Kaiser, "Metamorphic detection of repackaged malware," in *Proc. MET@ICSE*. IEEE, 2021, pp. 9–16.

[47] X. Sun, Y. Zhongyang *et al.*, "Detecting code reuse in android applications using component-based control flow graph," in *SEC SEC*, vol. 428. Springer, 2014, pp. 142–155.

[48] C. Szegedy, W. Zaremba *et al.*, "Intriguing properties of neural networks," in *Proc. ICLR*, 2014.

[49] K. Tian, D. Yao *et al.*, "Analysis of code heterogeneity for high-precision classification of repackaged malware," in *Proc. IEEE S&P*. IEEE Computer Society, 2016, pp. 262–271.

[50] L. van der Maaten and G. Hinton, "Visualizing data using t-sne," *J. Mach. Learn. Res.*, vol. 9, no. 86, pp. 2579–2605, 2008.

[51] VIRUSTOTAL, "Virustotal." 2024, https://www.virustotal.com/gui/home/upload.

[52] Wu Zhou and Yajin Zhou and Xuxian Jiang and Peng Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in *Proc. CODASPY*. ACM, 2012, pp. 317–326.

[53] P. Yan., "A look at repackaged apps and their effect on the mobile threat landscape," https://blog.trendmicro.com/trendlabs-security-intelligence/a-look-into-repackaged-apps-and-its-role-in-the-mobile-threat-landscape/., 2021, accessed online: January 11, 2021.

[54] L. Yang, Z. Chen *et al.*, "Jigsaw puzzle: Selective backdoor attack to subvert malware classifiers," in *Proc. IEEE S&P*, 2023.

[55] H. Yu, K. Yang *et al.*, "Cloudleak: Large-scale deep learning models stealing through adversarial examples," in *Proc. NDSS*. The Internet Society, 2020.

[56] D. Zhan, Y. Duan *et al.*, "Malpatch: Evading dnn-based malware detection with adversarial patches," *IEEE Trans. Inf. Forensics Secur.*, vol. 19, pp. 1183–1198, 2024.

[57] J. Zhang, P. Xiao *et al.*, "A single-loop smoothed gradient descent-ascent algorithm for nonconvex-concave min-max problems," in *Proc. NIPS*, 2020.

[58] L. Zhang, P. Liu *et al.*, "Semantics-preserving reinforcement learning attack against graph neural networks for malware detection," *IEEE Trans Dependable Secure Comput*, pp. 1–1, 2022.

[59] X. Zhang, Y. Zhang *et al.*, "Enhancing state-of-the-art classifiers with api semantics to detect evolved android malware," in *Proc. CCS*, 2020.

[60] A. Zhao, T. Chu *et al.*, "Minimizing maximum model discrepancy for transferable black-box targeted attacks," in *Proc. CVPR*. IEEE, 2023, pp. 8153–8162.

[61] K. Zhao, H. Zhou *et al.*, "Structural attack against graph based android malware detection," in *Proc. CCS*, 2021, pp. 3218–3235.

[62] W. Zhou, Y. Zhou *et al.*, "Fast, scalable detection of "piggybacked" mobile applications," in *Proc. CODASPY*. ACM, 2013, pp. 185–196.

[63] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *IEEE S&P*, 2012, pp. 95–109.

Appendix

A. Formal Proofs

The convergence proof of the algorithm for solving the min-max problem has been extensively discussed in several studies [57], [33], [39].

For the sake of convenience, we denote the loss as $\mathcal{L}$, i.e., $\mathcal{L} = -E_{x_b \in B}(F(x_b + (r+n) + p))$. Our goal is to solve the following saddle point problem:

$$\min_p \max_n \mathcal{L}(p, n), \tag{14}$$

If r and p are continuous values, we can solve the aforementioned problem using the following gradient descent ascent (GDA) algorithm.

$$\begin{aligned} p_{k+1} &= \mathcal{P}_p\left[p_k - \alpha \mathcal{L}_r\left(p_k, n_k\right)\right], \quad \text{for } k = 0, 1, \ldots \\ n_{k+1} &= \mathcal{P}_n\left[n_k + \alpha \mathcal{L}_n\left(p_k, n_k\right)\right], \quad \text{for } k = 0, 1, \ldots \end{aligned} \tag{15}$$

where P_P and P_N denote the projections onto the sets P and N, respectively. The vectors $p_0 \in P$ and $n_0 \in N$ are the initial iterates, and the scalar $\alpha > 0$ is a constant step size. The vectors $L_p(p_k, n_k)$ and $L_n(p_k, n_k)$ represent the subgradients of L at (p_k, n_k) with respect to p and n, respectively.

However, in our work, r and p are discrete. Therefore, we use $\frac{E_2(n+c,p)-E_2(n,p)}{\text{len}(c)}$ and $\frac{E_2(n,p+c)-E_2(n,p)}{\text{len}(c)}$ to replace $L_p(p_k, n_k)$ and $L_n(p_k, n_k)$, respectively. For convenience, we will only analyze the convergence properties under the condition of continuous variables.

Under general assumptions, we consider $\mathcal{L}$ to be a convex-concave function. Specifically, $\mathcal{L}(\cdot, n)$ is convex for every $n \in \mathbb{N}$. $\mathcal{L}(\cdot, n)$ is concave for every $p \in \mathbb{P}$.

And we assume:

Assumption 1. The $\mathcal{L}_p\left(p_k, n_k\right)$ and $\mathcal{L}_n\left(p_k, n_k\right)$ used in the method defined by Eq. (15) are uniformly bounded, i.e., there is a constant $L > 0$ such that

$$\left\|\mathcal{L}_p\left(p_k, n_k\right)\right\| \leq L, \quad \left\|\mathcal{L}_n\left(p_k, n_k\right)\right\| \leq L, \quad \text{for all } k \geq 0. \tag{16}$$

Following the above assumptions and Eq. (15), we can get two Lemmas and one conclusion:

Lemma 1. Let the sequences p_k and n_k be generated by the subgradient Eq. (15). We then have:

(a). For any $p \in P$ and for all $k \geq 0$,

$$\begin{aligned} \left\|p_{k+1} - p\right\|^2 \leq \left\|p_k - p\right\|^2 - 2\alpha(\mathcal{L}\left(p_k, n_k\right) \\ -\mathcal{L}\left(p, n_k\right) + \alpha^2\left\|\mathcal{L}_p\left(p_k, n_k\right)\right\|^2 \end{aligned} \tag{17}$$

(b). For any $n \in N$ and for all $k \geq 0$,

$$\begin{aligned} \left\|n_{k+1} - n\right\|^2 \leq \left\|n_k - n\right\|^2 + 2\alpha(\mathcal{L}\left(p_k, n_k\right) \\ -\mathcal{L}\left(p_k, n\right) + \alpha^2\left\|\mathcal{L}_n\left(p_k, n_k\right)\right\|^2 \end{aligned} \tag{18}$$

Lemma 2. Let the sequences p_k and x_k be generated by the Eq. (15). $\hat{p}_k$ and $\hat{n}_k$ are the iterate averages given by:

$$\hat{p}_k = \frac{1}{k}\sum_{i=0}^{k-1} p_i, \quad \hat{n}_k = \frac{1}{k}\sum_{i=0}^{k-1} n_i \tag{19}$$

We then have, for all $k \geq 1$,

$$\frac{1}{k}\sum_{i=0}^{k-1}\mathcal{L}(p_i,n_i)-\mathcal{L}(p,\hat{n}_k)\leq\frac{\|p_0-p\|^2}{2\alpha k}+\frac{\alpha L^2}{2},\quad \text{for any } p\in P \tag{20}$$

$$-\frac{\|n_0-n\|^2}{2\alpha k}-\frac{\alpha L^2}{2}\leq\frac{1}{k}\sum_{i=0}^{k-1}\mathcal{L}(p_i,n_i)-\mathcal{L}(\hat{p}_k,n),\quad \text{for any } n\in N. \tag{21}$$

According to the above analyses, we can have following conclusion.

Conclusion 1. Let $(p^*,n^*)\in P\times N$ be a saddle point of $\mathcal{L}(p,n)$. We have:

$$-\frac{\|n_0-n^*\|^2}{2\alpha k}-\frac{\alpha L^2}{2}\leq\frac{1}{k}\sum_{i=0}^{k-1}\mathcal{L}(p_i,n_i)-\mathcal{L}(p^*,n^*)\leq\frac{\|p_0-p^*\|^2}{2\alpha k}+\frac{\alpha L^2}{2} \tag{22}$$

These results show that averaged function values $\frac{1}{k}\sum_{i=0}^{k-1}\mathcal{L}(p_i,n_i)$ converges to the saddle point value within an error level of $\frac{\alpha L^2}{2}$.

B. Comparison of APK Features Before and After Perturbation

To visually show benign samples, malicious samples, adversarial examples generated by matched riders (Adversarial-M), and adversarial examples generated by mismatched riders (Adversarial-MM), we extract the results of these samples from the layer preceding the model's output layer and reduce their dimensionality using the t-Distributed Stochastic Neighbor Embedding (t-SNE) algorithm [50]. The dimensionality-reduced results are displayed in Fig. 14. The figure illustrates the distribution of different samples in the latent space of various detection models. Green and red points represent benign samples and malicious samples, while blue and purple points denote Adversarial-M and Adversarial-MM, respectively. The black dashed line represents the potential classification boundary. The results indicate that most Adversarial-M are closer to benign samples, whereas most Adversarial-MM are closer to malicious samples. This observation validates that our method satisfies the universality and targeting requirements.

C. Correcting the Errors Output by LLMs

When generating smali functions based on API blocks, the LLM may encounter the following errors: 1) failure to adhere to instruction requirements, and 2) syntax errors in the generated code. Facing the first error, the script will prompt the LLM to regenerate the code until it meets the specified instructions. The second error can lead to APK repackaging failures. To address this problem, we compile a repository of prompts for correcting errors based on past experiences. These prompts are integrated into a script that generates code correction suggestions derived from error messages encountered during failed APK repackaging attempts. This approach

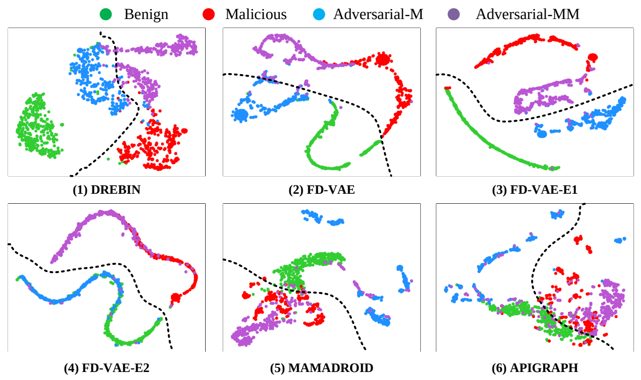


Fig. 14. Visualization of the results using t-SNE.

guides the LLM in refining its code-generation process based on previous iterations.

D. Template-based Code Generation Utilizing LLMs

We divide the process of code generation into blocks, ensuring that each output generates a single Permission or a single smali file along with its corresponding component. The templates we use are given below:

Prompts for permission: "Please write the statement to add permission X in the AndroidManifest.xml file. Only output the content of this permission, without comment."

Here, "Only output the content of this permission, without comment." directs LLM to generate only the permission content, minimizing LLM's query costs and enhancing efficiency. X represents the permission to be added.

Prompts for action: "Please create an Android activity, service, receiver, or provider component named com.api.ae.num with action X, and set its android:enabled to true, with no permission. Only output the content of this component, without headers, comments, or ending marks."

In this case, *"com.api.ae.num"* corresponds to the smali file for action X, "android:enabled" set to true indicates it is not dead code, "with no permission" avoids introducing negative impact features, and "Only output the content of this component, without headers, comment, or ending marks" instructs LLM to generate only the action content, optimizing query costs. X denotes the action to be created.

Prompts for API: "Please create a smali file named *Lcom/api/ae/num* with only one function, and the function only calls the API X. Add the libraries required for the file to run, only output the content of the file, do not write any test statements, headers, comment, or ending marks."

Here, *"com/api/ae/num"* specifies the smali file location for the new API, "Add the libraries required for the file to run" ensures syntax errors are avoided due to missing libraries, and "do not write any test statements, headers, comment, or ending marks" directs LLM to generate only the API content, reducing query costs. X represents the API to be implemented.

E. Perturbation Generation in Code Space

In our algorithm, we employ an LLM to generate perturbations in code space. Compared to operations like inserting

```
.class public Lcom/adjust/mb/selfDefined;
.super Ljava/lang/Object;

.method public static DebugUtility()V
   .locals 3

   const-string v0, "MySmaliClass"
   const-string v1, "This is a debug log."

   invoke-static {v0, v1}, Landroid/util/Log;->d(Ljava/lang/String;Ljava/lang/String;)I

   sget-object v0, Landroid/app/Application;->currentApplication:Landroid/app/Application;
   const-string v1, "Hello, World!"
   const/4 v2, 1
   invoke-static {v0, v1, v2}, Landroid/widget/Toast;->makeText(Landroid/content/
Context;Ljava/lang/CharSequence;I)Landroid/widget/Toast;
   move-result-object v0
   invoke-virtual {v0}, Landroid/widget/Toast;->show()V

   invoke-static {}, Landroid/os/SystemClock;->elapsedRealtime()J
   move-result-wide v0

   new-instance v2, Landroid/os/Handler;
   invoke-direct {v2}, Landroid/os/Handler;-><init>()V

   invoke-virtual {v2}, Landroid/os/Handler;->getLooper()Landroid/os/Looper;

   return-void
.end method
```

Fig. 15. The comparison of different problem space perturbation generation methods.

NOPs and empty functions, the LLM can generate code that is semantically closer to human-written code. An example is shown in Fig. 15

TABLE III. THE ATTACK EFFECTIVENESS ON MALPACK

Scenarios	ER		UR	
	U(↑)	T(↓)	U(↑)	T(↓)
W	1.000	0.441	0.989	0.410
B_S	0.920	0.466	0.816	0.418
G	0.948	0.333	0.872	0.358
B_D	0.942	0.429	0.890	0.440

F. The Attack Effectiveness on MALPACK

In this section, we consider a specific targeted clone app detection tool Malpack [45]. As shown in Table III, the experimental results indicate the performance of the new detection model (i.e., MALPACK). Experimental results demonstrate that our algorithm also achieves a strong attack performance on MALPACK.

G. The Detection Models Used in Experiments

In our study, both the target model and the local substitute models are Deep Neural Network (DNN) models. In the training phase, we utilize four substitute models (Model 1 - Model 4). In the attack phase, we employ three target models (G, B_S, B_D). To ensure adequate diversity in our local ensemble models, we train substitute models using different subsets of the training data. Specifically,the details of models are in TABLE IV. The table is divided into two sub-tables. The first six rows represent the structure and training data of substitute models, while the second six rows pertain to the target model's structure and training data. In each sub-table, the second row indicates the index of the ensemble model, the third row specifies the number of neurons in each layer (NN), the fourth row describes the activation functions used in the hidden layers (AF_H), and the fifth row denotes the activation function used in the output layer (AF_O). The final row illustrates the percentage of traditional training samples and piggybacked malware samples used in the training data (TD). Importantly, the training datasets for target models and substitute models are completely isolated to avoid any risk of data leakage.

TABLE IV. THE DETAIL OF THE DETECTION MODEL USED IN OUR EXPERIMENTS FOR FD-VAE-E1 FEATURE

	Substitute model			
	Model 1	Model 2	Model 3	Model 4
NN	[461,1200, 1200,1200,1]	[461,1200, 1200,1200,1]	[461,1200, 1200,1200,1]	[461,1200, 1200,1200,1]
AF_H	Relu	Relu	Relu	Relu
AF_O	Sigmoid	Sigmoid	Sigmoid	Sigmoid
TD	[100%,100%]	[33.3%,33.3%]	[33.3%,33.3%]	[33.3%,33.3%]
	Target model			
	G	B_S	B_D	-
NN	[461,1200, 1200,1200,1]	[461,1200, 1200,1]	[461,1200, 1200,1200,1200,1]	-
AF_H	Relu	Relu	Relu	-
AF_O	Sigmoid	Sigmoid	Sigmoid	-
TD	[100%,100%]	[100%,100%]	[100%,100%]	-

H. Performance of the Target Model

In this subsection, we demonstrate the detection performance of the 6 detectors used by our experiments in Section V. For the convenience of experiment reproducibility, we provide the hyperparameters of target detectors at the end of this subsection. The first row of Table V represents different classification models, with the first column denoting various test datasets. 'ER' signifies benign samples in the test set to which an Existed-Rider from the training set is added, while 'UR' indicates benign samples in the test set to which an Unknown-Rider from the training set is added. 'Benign' denotes benign software, and 'Malware' represents the original malware in the dataset. The second column represents different attack scenarios, i.e., gray-box target models, black-box shallow target models, and black-box deep target models. We use the 10-fold cross-validation to evaluate the detection performance of target detectors and provide the results in Table V. It can be seen that the target detectors perform well on our dataset, and are qualified for evaluating adversarial piggybacked malware.

I. The Attack Effectiveness on Additional Malicious Riders

We explore the effectiveness of our attack algorithm on a broader range of potential malicious riders, which are obtained with two approaches:

TABLE V. TARGET MODELS' PERFORMANCE

	Scenarios	DREBIN	FD-VAE	FD-VAE-E1	FD-VAE-E2	MaMaDroid	APIGraph
ER	G	0.984	0.973	0.979	0.990	0.717	0.689
	B_S	0.984	0.971	0.978	0.987	0.753	0.673
	B_D	0.984	0.972	0.979	0.988	0.730	0.668
UR	G	0.839	0.855	0.886	0.896	0.601	0.594
	B_S	0.817	0.855	0.886	0.892	0.661	0.592
	B_D	0.832	0.855	0.895	0.901	0.613	0.581
Benign	G	0.985	0.897	0.889	0.914	0.944	0.924
	B_S	0.984	0.930	0.896	0.909	0.929	0.930
	B_D	0.983	0.871	0.873	0.902	0.938	0.930
Malware	G	0.842	0.841	0.863	0.839	0.768	0.783
	B_S	0.863	0.782	0.859	0.854	0.790	0.720
	B_D	0.849	0.855	0.874	0.866	0.789	0.762

TABLE VI. The attack effectiveness on additional malicious riders

Multi-Hook Riders						
Scenarios	DREBIN		FD-VAE		FD-VAE-E1	
	U(↑)	T(↓)	U(↑)	T(↓)	U(↑)	T(↓)
W	0.908	0.154	1.000	0.273	1.000	0.135
B_S	0.759	0.168	0.798	0.170	0.811	0.190
G	0.734	0.132	0.820	0.171	0.882	0.188
B_D	0.732	0.151	0.834	0.178	0.905	0.171
Scenarios	FD-VAE-E2		MaMaDroid		APIGraph	
	U(↑)	T(↓)	U(↑)	T(↓)	U(↑)	T(↓)
W	0.998	0.128	0.966	0.345	0.997	0.530
B_S	0.788	0.142	0.867	0.279	0.853	0.423
G	0.810	0.149	0.799	0.276	0.861	0.455
B_D	0.870	0.172	0.863	0.280	0.887	0.421

Merge Riders						
Scenarios	DREBIN		FD-VAE		FD-VAE-E1	
	U(↑)	T(↓)	U(↑)	T(↓)	U(↑)	T(↓)
W	0.791	0.366	1.000	0.456	0.970	0.210
B_S	0.631	0.299	0.709	0.266	0.758	0.282
G	0.663	0.318	0.672	0.258	0.785	0.234
B_D	0.584	0.283	0.686	0.255	0.787	0.246
Scenarios	FD-VAE-E2		MaMaDroid		APIGraph	
	U(↑)	T(↓)	U(↑)	T(↓)	U(↑)	T(↓)
W	0.994	0.425	0.980	0.351	0.997	0.505
B_S	0.689	0.411	0.773	0.288	0.698	0.394
G	0.793	0.512	0.804	0.304	0.794	0.430
B_D	0.678	0.400	0.794	0.307	0.760	0.404

1. We merge existing malicious riders in pairs. We discover scenarios where two malicious riders coexist within one piggybacked malware. Inspired by this observation, we aim to expand the dataset by combining existing malicious riders in pairs. To ensure that the paired riders could coexist, we first select 26 non-overlapping malicious riders (if two riders share the same file name, merging them would result in the overwriting of critical code). Subsequently, we construct $\frac{26\times25}{2} = 325$ new malicious riders. Finally, we filter out riders with identical features using a feature comparison method, resulting in 168 new distinct malicious riders. We refer to these riders as MR (merge riders).

2. We relax the extraction criteria for malicious riders and allow for the presence of multiple HOOK functions, resulting in the identification of 83 malicious payloads. Subsequently, we apply a feature filtering method, which yielded 32 distinct malicious riders. We refer to these riders as MHR (multi-hook riders).

The experimental results are presented in TABLE VI, which includes two sub-tables for multi-hook riders and merge riders. In each sub-table, the first column indicates four different attack scenarios, while columns 2 to 4 represent six different features. The results demonstrate that our algorithm performs effectively against the new malicious riders.

J. A Specific Example

In our analysis of existing piggybacked malware and the original benign software instances (1D239AC5886F9A6C321F 0CD520E5CF507D7BD97B72AEEC0A3A93F2CE1AF19F66 and 8BB2570E5FE9CBA3021C04DB6334558A49FDC55885 C4A32E92C2369F9ACC697B), we have identified a prominent malicious payload known as DroidDream. DroidDream [17] is a highly recognized piggybacked Android malware that inserts malicious code into benign software. The malicious payload's function involves automatically downloading new malware during nighttime to evade detection by dynamic analysis and other detection systems.

Due to this malicious rider's overt behavior, we use it to validate the consistency of adversarial piggybacked malware functionality. We load this malicious rider onto 100 benign APKs, creating 100 instances of piggybacked malware. Additionally, we load the evasive rider (a combination of rider and perturbation) onto 100 benign APKs, forming adversarial piggybacked malware. Subsequently, we execute the original benign software, piggybacked malware, and adversarial piggybacked malware concurrently on an Android emulator for dynamic analysis. Initially, we adjust the Android emulator's time to the early hours of the morning. The experiment reveals that the original benign software operated normally, whereas all instances of piggybacked malware and adversarial piggybacked malware automatically initiated the download of new malware.

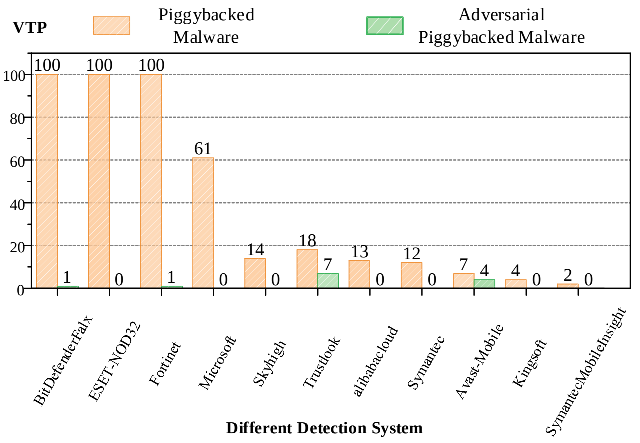


Fig. 16. The detection results on VirusTotal of adversarial piggybacked malware with DroidDream rider.

After validating the functionality of adversarial piggybacked malware, we upload these 100 instances of piggybacked malware and 100 instances of adversarial piggybacked malware to VirusTotal for testing. The experimental results, as shown in Fig. 16, depict the number of detection results flagged as malicious by various scanning engines on the y-axis, with different scanning engines represented on the x-axis. The orange bars represent the results for piggybacked malware, while the green bars represent the detection outcomes for adversarial piggybacked malware. The results of the experiment indicate that our approach significantly reduces the probability of detection by various scanning engines.