Response to all reviewers' comments

Manuscript ID: 1933

Title: Automated Malware Assembly Line: Uniting Piggybacking and Adversarial Example in Android Malware Generation

---

We would like to express our gratitude to the reviewers for their time spent on the review of our manuscript. We have addressed all the comments raised by the reviewers, and will revise our manuscript accordingly.

For convenience, we provide a jumping index for a smoother reviewing experience. Please kindly click the interested section and navigate to the corresponding response, if it would be helpful. In addition, to save the reviewers' time, **we have highlighted the most important comments with a light red box**.

Reviewer A	Reviewer B	Reviewer C	Reviewer D

I Response to Reviewer: A

We would like to express our sincere gratitude to reviewer A for providing invaluable comments that are helpful for improving our work. We have provided a detailed response to each of the reviewer's comments.

Q1: [Detailed explanation] Algorithm 1 could be improved with more detailed explanations of the selection of the poorest perturbation pk in the fine-tuning phase.

Response:

Thank you for your comment. In the fine-tuning phase, the algorithm will modify the perturbation p based on the query-reply results with the target model to enhance the universality of p. 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:

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

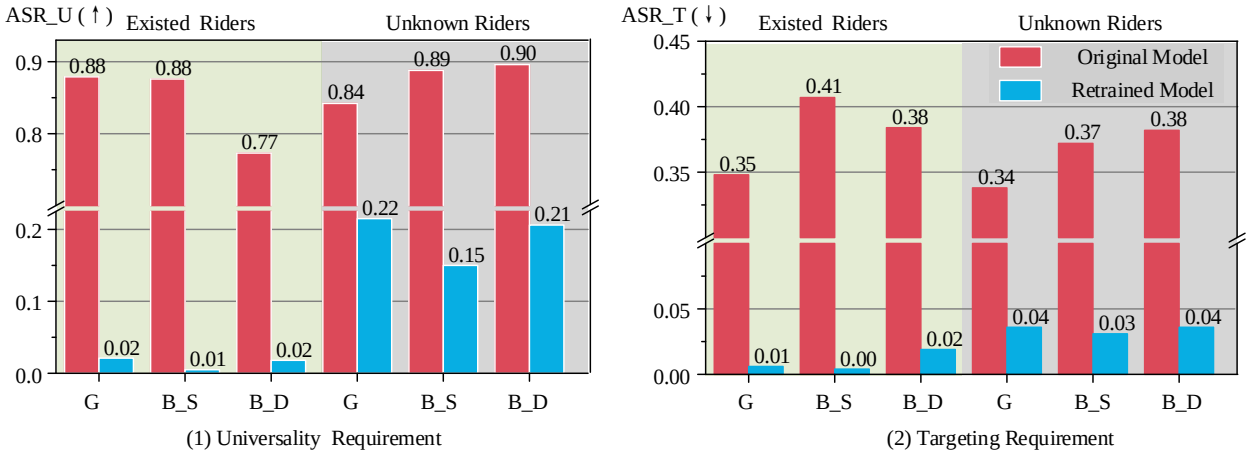

Fig. 1. The attack performance on retrained model.

where $E_1 = E_{x_b \in B}[F(x_b + r + p)]$ represents the expected value of the adversarial piggybacked malware classified by the model F. The adversarial piggybacked malware is generated by the malicious rider r and perturbation p applied to benign samples b. It is noteworthy that a smaller value of this expectation indicates that the classification model is more likely to classify the adversarial piggybacked malware as benign.

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.

Q2:[Potential defense] Discuss potential defense mechanisms and their effectiveness against the proposed attack.

Response:

We want to express our gratitude for the reviewer's comment. In our original manuscript, we have provided potential defense methods for the app market and Android users in the Discussion section. Additionally, according to the reviewer's comment we will discuss another adversarial example defense method in the revised manuscript. This newly added defense method is based on the assumption that the app market can utilize similarity analysis to identify a limited number of adversarial piggybacked malware and thus can enhance the performance of its classification model through adversarial retraining. In this context, we consider a rather extreme scenario where users acquire 100 instances of malicious adversarial piggybacked malware for each malicious rider and incorporate them into the training set for model retraining.

The attack results are illustrated in Fig. 1. Subfigures (1) and (2) in the figure represent the

attack success rates ASR_U and ASR_T on matched and unmatched riders [1], respectively, where a higher value for the former is preferable and a lower value for the latter is desirable. In each subfigure, the red bars correspond to the attack results on the original model, while the blue bars represent the attack results on the adversarially retrained model. The bars with a green background denote the attack results on existing riders, and those with a gray background reflects the results on unknown riders [2]. The experimental results demonstrate that adversarial retraining can mitigate the attack effectiveness of adversarial piggybacked malware, reducing the attack success rate from 85.9% to 10.5%. However, this defense strategy 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 to the adversarial piggybacked malware present in the training set. As shown in Subgraph (1), the defense effectiveness of the model against adversarial piggybacked malware generated from Unknown Riders is 17.6% lower than that against malware derived from Existed Riders. In the future, we will explore more robust malware detection models.

Q3: [Ethical implications] Discussion of the ethical implications of the research.

Response:

Thank you for the comment. In fact, we have already addressed the Ethical Concerns in Section VII of our manuscript. Our ethics statement is as follows: "Our work falls within the realm of offensive research. The main goal of our work is to make the academic and industrial communities pay more attention to adversarial example attacks against Android malware detection systems. Our work demonstrates that adversaries can generate a large number of evasive malware with hook technology. In the sections of Potential Defense and Appendix, 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."

It may be because our ethics statement is in the subsection, making it not very easy to find. In order to make it more prominent, we plan to include it in a separate section in the revised manuscript.

Q4: [Formal proofs] Present formal proofs to demonstrate the effectiveness of the proposed method (since the proposed approach mostly relies on experimental evaluation).

[1]To assist the reviewers in recalling the definitions, we present the following clarifications: Matched rider refers to the rider that requires the attacker to employ the perturbation p to conceal malicious activities. In contrast, unmatched riders denote other riders that are not designed by the attacker. The perturbation p will be ineffective on unmatched riders.

[2]For the developers of malware detection models, there exist certain malware composed of previously identified riders in their dataset, which are referred to as Existed Riders. Conversely, malware that does not include any malicious riders present in the detection model's training set is termed Unknown Riders.

Response: Thank you for the comments. The convergence proof of the algorithm for solving the min-max problem has been extensively discussed in several studies [1][2][3]. To facilitate the reviewers' understanding, we have provided an outline of the proof process and detailed steps as follows:

Our proof strategy is as follows: 1) First, due to the discrete nature of variables in the field of Android malware detection, which presents significant challenges for the proof process, we will approach the convergence proof under the assumption of continuous variables. 2) Next, we will establish Lemma 1, which analyzes the Basic Iterate Relations derived during the algorithm's iterative process. 3) Subsequently, we will introduce Assumption 1, which posits that the gradients during the solving process are bounded. 4) Following this, we will present Lemma 2, demonstrating the relationship between the iterate averages of the obtained solutions and all feasible solutions within the solution space. 5) Finally, we will substitute the saddle point (i.e., the optimal solution) into Lemma 2 to derive the convergence of the solution obtained by the algorithm. The detailed proof is shown as follows.

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:

$$\min_{p} \max_{n} \mathcal{L}(p, n), \tag{1}$$

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

$$\begin{aligned} p_{k+1} &= \mathcal{P}_p \left[p_k - \alpha \mathcal{L}_r \left(p_k, n_k \right) \right], \quad \text{for } k = 0, 1, \ldots \\ n_{k+1} &= \mathcal{P}_n \left[n_k + \alpha \mathcal{L}_n \left(p_k, n_k \right) \right], \quad \text{for } k = 0, 1, \ldots \end{aligned} \tag{2}$$

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 $p \in \mathbb{P}$.

Following the above assumptions and Equa. 2, we start with a Lemma.

Lemma 1. Let the sequences p_k and n_k be generated by the subgradient Equa. 2. We then have:

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

$$\|p_{k+1}-p\|^2 \leq \|p_k-p\|^2 - 2\alpha\left(\mathcal{L}\left(p_k,n_k\right)-\mathcal{L}\left(p,n_k\right)\right)+\alpha^2\left\|\mathcal{L}_p\left(p_k,n_k\right)\right\|^2 \tag{3}$$

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

$$\|n_{k+1}-n\|^2 \leq \|n_k-n\|^2 + 2\alpha\left(\mathcal{L}\left(p_k,n_k\right)-\mathcal{L}\left(p_k,n\right)\right)+\alpha^2\left\|\mathcal{L}_n\left(p_k,n_k\right)\right\|^2 \tag{4}$$

Proofs (a) and (b) follow a similar proof process; here, we will only provide the proof outline for (a). Based on the nonexpansive property of the projection operation and Equation 2, for any $p \in P$ and all $k \geq 0$,

$$\begin{aligned}\|p_{k+1}-p\|^2 &= \left\|\mathcal{P}_P\left[p_k-\alpha\mathcal{L}_p\left(p_k,n_k\right)\right]-p\right\|^2 \\ &\leq \left\|p_k-\alpha\mathcal{L}_p\left(p_k,n_k\right)-p\right\|^2 \\ &= \|p_k-p\|^2 - 2\alpha\mathcal{L}_p'\left(p_k,n_k\right)\left(p_k-p\right)+\alpha^2\left\|\mathcal{L}_p\left(p_k,n_k\right)\right\|^2\end{aligned} \tag{5}$$

Since the function $\mathcal{L}(p,n)$ is convex in p for each $n \in N$, we have that, for any p,

$$-\mathcal{L}_p'\left(x_p,n_k\right)\left(p_k-p\right) \leq -\left(\mathcal{L}\left(p_k,n_k\right)-\mathcal{L}\left(p,n_k\right)\right). \tag{6}$$

Hence, for any $p \in P$ and all $k \geq 0$,

$$\|p_{k+1}-p\|^2 \leq \|p_k-p\|^2 - 2\alpha\left(\mathcal{L}\left(p_k,n_k\right)-\mathcal{L}\left(p,n_k\right)\right)+\alpha^2\left\|\mathcal{L}_p\left(p_k,n_k\right)\right\|^2. \tag{7}$$

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 Equa. 2 are uniformly bounded, i.e., there is a constant $L>0$ such that

$$\left\|\mathcal{L}_p\left(p_k,n_k\right)\right\| \leq L, \quad \left\|\mathcal{L}_n\left(p_k,n_k\right)\right\| \leq L, \quad \text{for all } k \geq 0. \tag{8}$$

Under the preceding assumption and analyses, we have the second lemma.

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

$$\hat{p}_k = \frac{1}{k}\sum_{i=0}^{k-1} p_i, \quad \hat{n}_k = \frac{1}{k}\sum_{i=0}^{k-1} n_i \tag{9}$$

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

$$\frac{1}{k}\sum_{i=0}^{k-1}\mathcal{L}\left(p_i,n_i\right)-\mathcal{L}\left(p,\hat{n}_k\right)\leq\frac{\left\|p_0-p\right\|^2}{2\alpha k}+\frac{\alpha L^2}{2},\quad\text{for any } p\in P \tag{10}$$

$$-\frac{\left\|n_0-n\right\|^2}{2\alpha k}-\frac{\alpha L^2}{2}\leq\frac{1}{k}\sum_{i=0}^{k-1}\mathcal{L}\left(p_i,n_i\right)-\mathcal{L}\left(\hat{p}_k,n\right),\quad\text{for any } n\in N. \tag{11}$$

Since the proof processes for Equation 10 and Equation 11 are similar, we will only provide the proof for Equation 10 here. By using Lemma 1 and the boundedness of the $\mathcal{L}_p\left(p_i,n_i\right)$, we have that, for any $p\in P$ and $i\geq 0$

$$\left\|p_{i+1}-p\right\|^2\leq\left\|p_i-p\right\|^2-2\alpha\left(\mathcal{L}\left(p_i,n_i\right)-\mathcal{L}\left(p,n_i\right)\right)+\alpha^2L^2. \tag{12}$$

Therefore,

$$\mathcal{L}\left(p_i,n_i\right)-\mathcal{L}\left(p,n_i\right)\leq\frac{1}{2\alpha}\left(\left\|p_i-p\right\|^2-\left\|p_{i+1}-p\right\|^2\right)+\frac{\alpha L^2}{2} \tag{13}$$

By adding these relations over $i=0,\ldots,k-1$, we obtain, for any $p\in P$ and $k\geq 1$,

$$\sum_{i=0}^{k-1}\left(\mathcal{L}\left(p_i,n_i\right)-\mathcal{L}\left(p,n_i\right)\right)\leq\frac{1}{2\alpha}\left(\left\|p_0-p\right\|^2-\left\|p_k-p\right\|^2\right)+\frac{k\alpha L^2}{2} \tag{14}$$

implying that

$$\frac{1}{k}\sum_{i=0}^{k-1}\mathcal{L}\left(p_i,n_i\right)-\frac{1}{k}\sum_{i=0}^{k-1}\mathcal{L}\left(p,n_i\right)\leq\frac{\left\|p_0-p\right\|^2}{2\alpha k}+\frac{\alpha L^2}{2} \tag{15}$$

Since the function $\mathcal{L}(p,n)$ is concave in n for any fixed $p\in P$, there holds

$$\mathcal{L}\left(p,\hat{n}_k\right)\geq\frac{1}{k}\sum_{i=0}^{k-1}\mathcal{L}\left(p,n_i\right),\quad\text{with } p\in P \text{ and } \hat{n}_k=\frac{1}{k}\sum_{i=0}^{k-1}n_i. \tag{16}$$

Combining the preceding two relations, we obtain that, for any $p\in P$ and $k\geq 1$,

$$\frac{1}{k}\sum_{i=0}^{k-1}\mathcal{L}\left(p_i,n_i\right)-\mathcal{L}\left(p,\hat{n}_k\right)\leq\frac{\left\|p_0-p\right\|^2}{2\alpha k}+\frac{\alpha L^2}{2} \tag{17}$$

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:

$$-\frac{\|n_0-n^*\|^2}{2\alpha k}-\frac{\alpha L^2}{2}\leq\frac{1}{k}\sum_{i=0}^{k-1}\mathcal{L}\left(p_i,n_i\right)-\mathcal{L}\left(p^*,n^*\right)\leq\frac{\|p_0-p^*\|^2}{2\alpha k}+\frac{\alpha L^2}{2} \tag{18}$$

The proof is given as follows. Based on Lemma 2, by letting $p=p^*$ and $n=n^*$, we have for all $k\geq 1$:

$$\begin{aligned}&\frac{1}{k}\sum_{i=0}^{k-1}\mathcal{L}\left(p_i,n_i\right)-\mathcal{L}\left(p^*,\hat{n}_k\right)\leq\frac{\|p_0-p^*\|^2}{2\alpha k}+\frac{\alpha L^2}{2}\\&-\frac{\|n_0-n^*\|^2}{2\alpha k}-\frac{\alpha L^2}{2}\leq\frac{1}{k}\sum_{i=0}^{k-1}\mathcal{L}\left(p_i,n_i\right)-\mathcal{L}\left(\hat{p}_k,n^*\right)\end{aligned} \tag{19}$$

By the saddle-point relation, we have

$$\mathcal{L}\left(p^*,\hat{n}_k\right)\leq\mathcal{L}\left(p^*,n^*\right)\leq\mathcal{L}\left(\hat{p}_k,n^*\right) \tag{20}$$

Combining the preceding relations, we obtain that, for all $k\geq 1$:

$$-\frac{\|n_0-n^*\|^2}{2\alpha k}-\frac{\alpha L^2}{2}\leq\frac{1}{k}\sum_{i=0}^{k-1}\mathcal{L}\left(p_i,n_i\right)-\mathcal{L}\left(p^*,n^*\right)\leq\frac{\|p_0-p^*\|^2}{2\alpha k}+\frac{\alpha L^2}{2}. \tag{21}$$

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

[1] A Single-Loop Smoothed Gradient Descent-Ascent Algorithm for Nonconvex-Concave Min-Max Problems. NeurIPS 2020.

[2] On Gradient Descent Ascent for Nonconvex-Concave Minimax Problems. ICML 2020.

[3] Subgradient Methods for Saddle-Point Problems. J. Optimization Theory and Applications. 2009.

Q5: [Improve writing] Proofread writing so it does not hinder readability

Response: Thank you very much for the comment. We will thoroughly revise the wording and structure of our manuscript to improve readability.

II Response to Reviewer: B

Thank you very much for the positive comments. We will further improve our work according to your comments.

Q1: Explain or resolve the logic problem in the min-max problem designed for the targeting requirement.

Response:

We apologize for any misunderstanding caused by our inappropriate phrasing. The min-max problem is introduced to meet the targeting requirement, i.e., the perturbation is designed only for the adversary's own malicious rider. Solving this min-max problem involves first identifying the rider (i.e., r + n*) most susceptible to perturbation p and then optimizing p to ensure that it cannot attack this rider (r + n*) successfully.

In the minimization phase, we identify the rider (r + n*) that is most easily attacked. In the original manuscript, we stated, '... find an optimal $n*$, where the resulting rider $r + n*$ is very prone to being misclassified as benign.' What we intended to convey is that we aim to 'find the rider $r + n*$ that is most susceptible to perturbation by p and is more likely evade detection.'

In the maximization phase, we optimize the perturbation p to ensure that it cannot be successfully applied to the most vulnerable rider (r + n*). Therefore, it is reasonable to believe that the perturbation p is unlikely to be effective for other riders, fulfilling the targeting requirement.

Q2: Modify or design other attack methods suitable for this scenario and compare them with the method proposed in this paper.

Response:

We want to express our sincere gratitude to the reviewer for this valuable feedback. According to this feedback, we selected four attack algorithms for comparative analysis.

First, to validate the effectiveness of our adversarial perturbations, we designed a Random-Noise attack algorithm. This approach involves randomly generating an adversarial perturbation of the same size as the original input, with the goal of attack effectiveness evaluation.

Then, we selected two attack algorithms (i.e., HIV-CW and HIV-JSMA) proposed in the Android HIV framework [1] that specifically target Android malware detection systems. Since both algorithms generate adversarial perturbations for individual APKs, we focused on the adversarial perturbations associated with a particular malicious rider that results in piggybacked malware. We applied these perturbations to other instances of piggybacked malware generated by the same malicious rider to test whether the universality requirement is met. Furthermore, we tested the targeting requirement by applying the perturbations to piggybacked malware produced by different malicious riders.

Finally, we selected a universal attack method, MalPatch [2], which generates a universal perturbation for a specific instance of piggybacked malware associated with one malicious rider. We

then evaluated the targeting requirement by applying this universal perturbation to piggybacked malware produced by other malicious riders.

[1] Android HIV: A Study of Repackaging Malware for Evading Machine-Learning Detection. IEEE Transactions on Information Forensics and Security 2020.

[2] MalPatch: Evading DNN-Based Malware Detection With Adversarial Patches. IEEE Transactions on Information Forensics and Security 2024

The experimental results on different features and different attack scenarios are presented in TABLE IV. The results 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 approach effectively meets both requirements, resulting in superior attack performance.

TABLE I

COMPARISION WITH SOTA MALWARE ADVERSARIAL ATTACK METHODS

Drebin								
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
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

FD-VAE								
METHODS	W		B_S		G		B_D	
	U(↑)	T(↓)	U(↑)	T(↓)	U(↑)	T(↓)	U(↑)	T(↓)
RN	0.860	0.874	0.572	0.569	0.792	0.784	0.774	0.747
HIV-JSMA	0.315	0.225	0.123	0.117	0.091	0.077	0.145	0.134
HIV-CW	0.560	0.467	0.298	0.281	0.359	0.329	0.329	0.326
Malpatch	0.836	0.583	0.349	0.304	0.413	0.381	0.419	0.381
Ours	1.000	0.484	0.800	0.302	0.836	0.305	0.810	0.308

FD-VAE-E1								
METHODS	W		B_S		G		B_D	
	U(↑)	T(↓)	U(↑)	T(↓)	U(↑)	T(↓)	U(↑)	T(↓)
RN	0.228	0.221	0.650	0.638	0.354	0.358	0.430	0.439
HIV-JSMA	0.277	0.286	0.096	0.133	0.102	0.159	0.062	0.132
HIV-CW	0.518	0.400	0.248	0.289	0.220	0.251	0.221	0.277
Malpatch	0.828	0.549	0.355	0.339	0.299	0.306	0.288	0.321
Ours	1.000	0.288	0.902	0.362	0.862	0.271	0.872	0.328

FD-VAE-E2								
METHODS	W		B_S		G		B_D	
	U(↑)	T(↓)	U(↑)	T(↓)	U(↑)	T(↓)	U(↑)	T(↓)
RN	0.157	0.120	0.122	0.079	0.137	0.092	0.143	0.104
HIV-JSMA	0.323	0.293	0.097	0.176	0.087	0.159	0.124	0.186
HIV-CW	0.521	0.453	0.246	0.228	0.333	0.294	0.271	0.269
Malpatch	0.958	0.662	0.295	0.314	0.350	0.354	0.375	0.387
Ours	1.000	0.384	0.776	0.390	0.812	0.443	0.847	0.400

MaMaDroid								
METHODS	W		B_S		G		B_D	
	U(↑)	T(↓)	U(↑)	T(↓)	U(↑)	T(↓)	U(↑)	T(↓)
RN	0.275	0.242	0.053	0.039	0.229	0.201	0.165	0.143
HIV-JSMA	0.457	0.387	0.437	0.393	0.434	0.375	0.419	0.356
HIV-CW	0.850	0.829	0.826	0.818	0.808	0.810	0.831	0.829
Malpatch	0.908	0.881	0.867	0.856	0.856	0.846	0.868	0.850
Ours	0.984	0.368	0.840	0.350	0.857	0.311	0.864	0.312

APIGraph								
METHODS	W		B_S		G		B_D	
	U(↑)	T(↓)	U(↑)	T(↓)	U(↑)	T(↓)	U(↑)	T(↓)
RN	0.360	0.295	0.295	0.231	0.314	0.248	0.251	0.192
HIV-JSMA	0.678	0.591	0.615	0.518	0.647	0.560	0.598	0.531
HIV-CW	0.957	0.948	0.931	0.918	0.948	0.940	0.937	0.920
Malpatch	0.959	0.957	0.927	0.930	0.951	0.955	0.944	0.936
Ours	0.998	0.500	0.793	0.407	0.851	0.446	0.836	0.413

Q3: Increase the number of distinct malicious riders.

Response:

We sincerely appreciate the reviewer's feedback. In response to this comment, we have expanded the distinct malicious riders from 65 to 265. Our expansion methods primarily involve two approaches:

1. We merged existing malicious riders in pairs. We discovered scenarios where two malicious riders coexist within one piggybacked malware. Inspired by this observation, we aimed to expand the dataset by combining existing malicious riders in pairs. To ensure that the paired riders could coexist, we first selected 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 constructed $\frac{26\times25}{2} = 325$ new malicious riders. Finally, we filtered 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 relaxed the extraction criteria for malicious riders. In the original manuscript, we stipulated that the malicious payload must be connected to the benign carrier through a single HOOK function call. Here, we have relaxed this constraint to allow for the presence of multiple HOOK functions, resulting in the identification of 83 malicious payloads. Subsequently, we applied a feature filtering method, which yielded 32 distinct malicious riders. We refer to these riders as MHR (multi-hook riders).

Due to time constraints, we have only validated our approach using the FD-VAE-V1 features. If the reviewer allows us to make revisions, we assure you that we will include results for all features. The experimental results are presented in TABLE II, where the first column indicates four different attack scenarios, while the second and third columns show results for the 65 distinct malicious riders used in the original manuscript. The fourth and fifth columns present the results of our algorithm on the newly added merge rider and multi-hook rider. The experimental findings demonstrate that our algorithm also performs well against the newly introduced malicious riders.

TABLE II
THE ATTACK PERFORMANCE ON NEW MALICIOUS RIDERS

Scenarios	ER		UR		MKR		MR	
	U(↑)	T(↓)	U(↑)	T(↓)	U(↑)	T(↓)	U(↑)	T(↓)
W	1.000	0.253	1.000	0.227	1.000	0.156	0.970	0.248
B_S	0.890	0.312	0.895	0.263	0.811	0.185	0.754	0.279
G	0.841	0.216	0.907	0.208	0.882	0.187	0.780	0.237
B_D	0.855	0.250	0.908	0.267	0.905	0.165	0.784	0.244

Q4: Describe the machine learning models used.

Response: We sincerely appreciate the reviewer's comments. We utilize a Deep Neural Network (DNN) classification model, and relevant descriptions can be found in Appendix A. We will also include the appropriate citations in the main text.

For convenience, we have extracted the relevant descriptions here.'We utilize a DNN as the classification model, with the target model consisting of a 4-layer neural network in a gray-box scenario. For the Drebin feature, each layer comprises 10,000, 32, 32, and 1 neurons respectively. For the FD-VAE feature, each layer consists of 1,000, 1,000, 1,000, and 1 neurons. For the FD-VAE-E1 and FD-VAE-E2 features, they have 1,200, 1,200, 1,200, and 1 neurons respectively. The hidden layers employ ReLU as their activation function, while the output layer uses Sigmoid. For the MaMaDroiD and APIGraph features, each layer has 300, 300, 300, and 1 neurons. In the B_S and B_D scenarios, an additional hidden layer is either removed or added.'

Q5: Explain the motivation for dividing the black-box scenario into B_D and B_S.

Response: We really appreciate this comment. B_D and B_S represent the use of deeper and shallower DNNs, respectively, in black-box attack scenarios. The inclusion of these two scenarios aims to demonstrate that adversarial perturbations generated on a local substitute model can transfer to classification models with different architectures. We will provide additional descriptions in the main text.

Q6: How the authors' approach differ from the example provided in Pierazzi et al. (cited as [41] by the authors).

Response:

The work of Pierazzi et al. [41] was the first to explore the constraints imposed by adversarial examples in the domain of malware detection, marking an early breakthrough in this field. We compare our work with [41] on three aspects of Goal, Attack Method, and Attack Constraints.

1) In terms of Goal, Pierazzi et al. focus solely on perturbing a single piece of malware to generate adversarial malware. In contrast, our approach investigates the generation of thousands of adversarial malware instances in a short time frame. Specifically, while Pierazzi et al. can generate only one adversarial malware per attack, our algorithm explores a more prevalent method of malware generation—piggybacking. By hooking a malicious rider and perturbing it onto any benign carrier, we can produce thousands of adversarial malware instances in a single operation. Therefore, our research poses a new threat.

2) In terms of attack methods, Pierazzi et al. first obtained benign code snippets that can be used to perturb malware, subsequently using optimization algorithms to select suitable segments for injection. In contrast, our approach employs optimization algorithms to identify the functions

that need to be inserted, followed by utilizing a large model to generate the corresponding code snippets.

3) In terms of attack constraints, Pierazzi et al. proposed constraints for the problem space of attacks across all domains, while our attack constraints build upon and modify their framework. Specifically, Pierazzi et al. introduced four attack constraints:

Available transformations: The perturbations made by the attacker must meet practical requirements. In our work, 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 added 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 This requirement emphasizes that the applied perturbations should not be easily filtered out, such as dead code, which can be readily eliminated by program analysis tools. However, our algorithm inherently satisfies this requirement, as it necessitates hooking both the malicious code and the perturbation code onto benign carriers. Consequently, we did not explicitly discuss this requirement in our manuscript.

Finally, we introduce an additional constraint, the flexibility constraint, which ensures that the attacker can freely select perturbations from the feature space.

III Response to Reviewer: C

Thank you very much for the comments. We will further improve our work based on your comments.

Q1: [Introduction] It is mostly clear, but it lacks presenting the technical challenges to be overcome to achieve the described goal. What impeded so far adversarial malware creation as described by the authors? By the Intro, the paper sounded more like an orchestration framework than new techniques, which is not the case, but you only discover it when you read further sections.

Response:

Thanks for the comments. We will revise the introduction section to emphasize the challenges we face. In our work, we encounter two main challenges: how to generate perturbations that

are disruptive to benign carriers but useful against malicious riders, and how to ensure these perturbations exclusively affect riders designed by attackers. We hope the reviewer will allow us the opportunity to modify the introduction.

Q2: [Preliminaries] This applies to here and over the paper. The authors should reconsider the terminology adopted in the paper. In my view, the authors are creating pr using terminology that is very similar to other existing terms, which creates additional noise without adding additional information. In my view, the terms Rider and Piggybacking have been defined before in the literature as Trojanization and App Cloning.

Response:

Thank you very much. We will standardize the terminology according to your comments. The terminology used in our manuscript is not created by us but comes from the literature [1][2]. In the future, we will replace these terms with more formal expressions, such as using "Trojanization and App Cloning" instead of "Rider and Piggybacking".

[1] Understanding Android App Piggybacking: A Systematic Study of Malicious Code Grafting. IEEE TIFS 2017.

[2]Fast, Scalable Detection of "Piggybacked" Mobile Applications. ACM CODASPY 2013.

Q3: [Contribution] Unclear threat model.

Response: Thank you for the comment. We revised this section based on the reviewer's feedback. For clarity, an excerpt of the revised portion is provided below:

We present the threat model by delineating three key components: adversary goals, knowledge, and constraint.

Goal. In this work, we introduce a new attack scenario, where the adversarial example technique and piggybacking technique are combined to systematically generate a large number of evasive malware. Assuming adversaries have developed a malicious rider, they aim to generate an adversarial perturbation specific to this rider, which can assist them in mass-producing adversarial piggybacked malware. These perturbations should satisfy the universality requirement and targeting requirement. For the universality requirement, we assume the malicious rider is r_t, and the perturbation p^* needs to fulfill the following conditions:

$$\begin{aligned} p^* &= \arg\min_{p \in \mathbf{P}} \text{cost}(p), \\ \text{s.t. } \forall x_b \in \mathbf{B}, & f\left(\phi\left(x_b + r_t + p^*\right)\right) = 0, \end{aligned} \tag{22}$$

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

As for the targeting requirement, to reduce the attack footprint and improve stealthiness, an adversary only designs the perturbation for his own malicious rider. That is, the perturbation p will only take effectiveness when it works with the malicious rider r_t designed by the adversary himself. This process can be formulated as:

$$\forall r \neq r_t, f\left(\phi\left(x_b + r + p^*\right)\right) = 1, \tag{23}$$

Knowledge. Depending on the attacker's knowledge of the detection model, we consider three attack scenarios. In the new threat model considered in this paper, adversaries' level of knowledge about their target model can be strong, moderate, or weak. In the case of strong knowledge, adversaries have complete information about the classification model, including model parameters and model structure. In the case of moderate knowledge, adversaries only have information about the training data. In the case of weak knowledge, adversaries are unaware of the aforementioned information and rely on transferability to attack the target model.

Constraint. The constraints on attackers primarily arise from two aspects: the constraints during the attack process and those imposed by malicious riders. Regarding the former, to ensure the stealthiness of attacks, attackers must limit their queries to the target model below a certain threshold, thereby minimizing query frequency. Additionally, we assume that some detection models cannot output classification probabilities, thus attackers often only obtain binary output results. Regarding the latter, we need to ensure that the process of hooking malicious riders onto benign carriers can run automatically. This requires that the insertion process of the rider cannot delete Smali files, permissions, actions, resource files, etc., from the carrier. Furthermore, to ensure the operation of malicious riders, we need to ensure that the malicious rider can be hooked onto the original launcher component of the benign carrier. Finally, as our work does not study the methods of malicious rider generation, we only need the malicious rides written in the Smali code, and impose no constraints on the size of the rider.

Q4: [Piggybacking] The preliminary analysis is great and it informs well the future project decision of the paper. However, I question how to move from it to the real world. In the paper, it is based on a previous study. How will the attackers do it in practice? Will they build their own database?

Response:

We greatly appreciate the reviewer's feedback. For attackers, there are two ways to obtain malicious riders. The first method is similar to the approach outlined in our paper, where attackers need to construct their own database to extract malicious riders capable of automatically hooking onto benign carriers. The second method involves designing new malicious riders from scratch

based on malicious functionalities. However, since our study does not investigate the generation methods of malicious riders, we only consider the first approach.

Q5: [Piggybacking] In this part, the validation was manually performed in a debugger. Should be this way when implemented by the attackers? Isn't it harming the scalability goal?

Response:

We greatly appreciate the reviewer's feedback. Manual debugger operations only exist in the extraction process of malicious riders, and do not affect subsequent perturbation and the hooking of malicious riders and perturbations onto benign carriers. Moreover, in real-world scenarios, attackers may design malicious riders based on their malicious behaviors. Under this situation, the manual debugger operation is not needed any more, and thus does not affect the overall scalability goal.

Q6: [Piggybacking] I like the idea of not applying the perturbation to each p, but to the whole n+p set. It makes sense to evade most detectors. However, thinking about the future, it might not make sense to evade detector based on moving windows, that slice the input into many. I have seen proposals for providing chunks of binaries to MalConv. In this case, a singleton malicious rider could be detected (although at a greater cost). The authors should discuss it and even consider it as a way to move forward (something lightly touched in the discussion section).

Response:

We greatly appreciate the reviewer's acknowledgment of the targeting requirement. The detection method based on moving windows indeed poses a potential defense against our attack algorithm. To counter such detection algorithms effectively, we propose embedding perturbations more finely-grained within malicious riders. Additionally, increasing the perturbation magnitude ensures that any chunks of malicious riders can evade detection of malicious behavior. We sincerely thank the reviewer for their insights, and we will discuss this malware detection method further in the Discussion section.

Q7: [Piggybacking] A presentation problem I see in this section is that we do not know exactly what is being perturbed. It would be great to exemplify wit the features before and after.

Response:

We greatly appreciate the reviewer's comment. We randomly selected five samples and represented their features before and after perturbation in Fig. 2. Specifically, we transformed the FD-VAE features of an APK (0-1 vector) into images. The features in adversarial piggybacked malware originate from three sources: features inherent to the original benign carriers, features from malicious riders, and features from adversarial perturbations. We represent these different

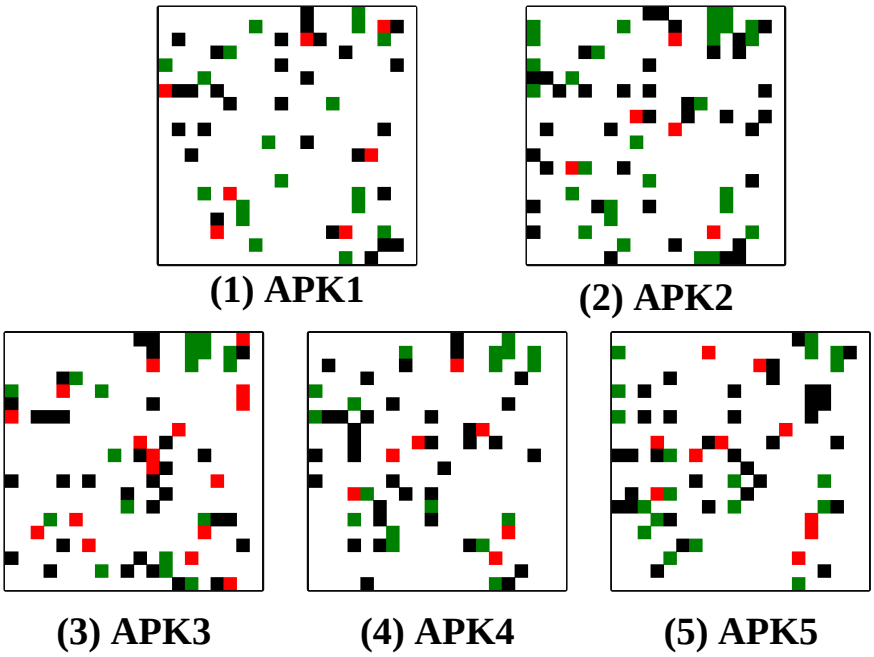

Fig. 2. The feature before and after perturbation.

features using three distinct colors: green, red, and black. We assure you that the above results will be incorporated into our paper.

Q8: [Piggybacking] The strategy for code generation using LLMs is OK, but it is unclear how the prompt is generated. Is it a template-based approach?

Response:

We are sorry about the unclear statement. Our prompt generation is a template-based approach. In our algorithm, 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.

To handle LLM's generation, we divide the process into blocks, ensuring that each output either generates a single Permission or a single smali file along with its corresponding component. The templates we use are given below:

For Permission prompts: "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.

For Action prompts: "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, comment, 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.

For API prompts: "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.

Q9: [Evaluation] I just miss details on how to ensure that the samples were still functional. It seems the authors adopted a functional-by-design approach but never really validated it.

Response:

The functional consistency verification has been discussed in the Discussion.B Section of the original manuscript. For the convenience of the reviewer, we have excerpted the original text as follows:

"In our experiments, we compared 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, such as DroidDream, as an example. DroidDream injects malicious code segments into existing benign carriers. To evade dynamic analysis, it only 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 the Appendix, and the results indicate that all adversarial piggybacked malware can run normally

and exhibit malicious behavior."

Q10: [Evaluation] The approach is efficient against traditional detectors, great, but is it efficient against clone detection approaches?

Response: Thank you for the comment. Our approach remains effective in clone detection methods. In our experiments, MaMaDroid, APIGraph, DREBIN, and FD-VAE are traditional methods for detecting malicious software. FD-VAE-E1 and FD-VAE-E2 are detection methods specifically designed for clone apps as outlined in [1]. Specifically, adversaries often add benign carriers' existing permissions or actions repeatedly when creating piggybacked APKs. This process results in some permissions being redundantly declared. To address this, we converted the binary values of permissions and actions in FD-VAE-E1 to their actual occurrence counts, which is referred to as FD-VAE-E2 in our manuscript.

Moreover, we consider a new targeted clone app detection tool (https://www.nature.com/articles/s41598-022-23766-w) suggested by reviewer. As shown in Fig. III, the experimental results indicate the performance of the new detection model (i.e., MALPACK). Due to time constraints, we validated our algorithm solely on a white-box basis. We hope the reviewers will provide us with the opportunity to refine all experiments. Furthermore, we intend to validate the algorithm's effectiveness using other clone detection approaches.

TABLE III
THE ATTACK PERFORMANCE ON ADDED MALWARE DETECTION METHOD

Features	ER		UR	
	U(↑)	T(↓)	U(↑)	T(↓)
Drebin	1.000	0.480	1.000	0.374
FD-VAE	1.000	0.518	1.000	0.429
FD-VAE-E1	1.000	0.301	1.000	0.267
FD-VAE-E2	1.000	0.400	1.000	0.360
MaMaDroid	0.979	0.358	0.991	0.384
APIGraph	0.996	0.508	1.000	0.485
Malpack	1.000	0.455	1.000	0.434

[1] Understanding Android App Piggybacking: A Systematic Study of Malicious Code Grafting. IEEE TIFS 2017.

Q11: [Evaluation] Is it superior to other adversarial techniques? It is not enough to work, we need to know if the attackers would prefer to use this attack versus others. In particular, this approach seems very similar to the gadget insertion of EvadeDroid and I believe they should be compared.

Response: We want to express our sincere gratitude to the reviewer for this valuable feedback. According to this feedback, we introduced four attack algorithms for comparative analysis.

First, to validate the effectiveness of our adversarial perturbations, we designed a Random-Noise attack algorithm. This approach involves randomly generating an adversarial perturbation of the same size as the original input, with the goal of attack effectiveness evaluation.

Then, we selected two attack algorithms (i.e., HIV-CW and HIV-JSMA) proposed in the Android HIV framework [1] that specifically target Android malware detection systems. Since both algorithms generate adversarial perturbations for individual APKs, we focused on the adversarial perturbations associated with a particular malicious rider that results in piggybacked malware. We applied these perturbations to other instances of piggybacked malware generated by the same malicious rider to test whether the universality requirement is met. Furthermore, we tested the targeting requirement by applying the perturbations to piggybacked malware produced by different malicious riders.

Finally, we selected a universal attack method, MalPatch [2], which generates a universal perturbation for a specific instance of piggybacked malware associated with one malicious rider. We then evaluated the targeting requirement by applying this universal perturbation to piggybacked malware produced by other malicious riders.

An important point to note is that EvadeDroid is an excellent attack algorithm; however, the paper does not provide code in its repository. Therefore, reproducing its work requires some time. We commit to comparing our modifications with this algorithm in subsequent revisions.

[1] Android HIV: A Study of Repackaging Malware for Evading Machine-Learning Detection. IEEE Transactions on Information Forensics and Security 2020.

[2] MalPatch: Evading DNN-Based Malware Detection With Adversarial Patches. IEEE Transactions on Information Forensics and Security 2024

The experimental results on different features and different attack scenarios are presented in TABLE IV. The results 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 approach effectively meets both requirements, resulting in superior attack performance.

Q12: [Evaluation] Finally, one of the claimed major advantages of the proposed solution is to be end-to-end. Thus, this characteristic must be evaluated. Multiple aspects of it were not considered. For instance, how long does it take to the entire pipeline create a new malware? In how many cases ChatGPT fails? And so on.

Response:

TABLE IV

COMPARISION WITH SOTA MALWARE ADVERSARIAL ATTACK METHODS

Drebin								
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
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

FD-VAE								
METHODS	W		B_S		G		B_D	
	U(↑)	T(↓)	U(↑)	T(↓)	U(↑)	T(↓)	U(↑)	T(↓)
RN	0.860	0.874	0.572	0.569	0.792	0.784	0.774	0.747
HIV-JSMA	0.315	0.225	0.123	0.117	0.091	0.077	0.145	0.134
HIV-CW	0.560	0.467	0.298	0.281	0.359	0.329	0.329	0.326
Malpatch	0.836	0.583	0.349	0.304	0.413	0.381	0.419	0.381
Ours	1.000	0.484	0.800	0.302	0.836	0.305	0.810	0.308

FD-VAE-E1								
METHODS	W		B_S		G		B_D	
	U(↑)	T(↓)	U(↑)	T(↓)	U(↑)	T(↓)	U(↑)	T(↓)
RN	0.228	0.221	0.650	0.638	0.354	0.358	0.430	0.439
HIV-JSMA	0.277	0.286	0.096	0.133	0.102	0.159	0.062	0.132
HIV-CW	0.518	0.400	0.248	0.289	0.220	0.251	0.221	0.277
Malpatch	0.828	0.549	0.355	0.339	0.299	0.306	0.288	0.321
Ours	1.000	0.288	0.902	0.362	0.862	0.271	0.872	0.328

FD-VAE-E2								
METHODS	W		B_S		G		B_D	
	U(↑)	T(↓)	U(↑)	T(↓)	U(↑)	T(↓)	U(↑)	T(↓)
RN	0.157	0.120	0.122	0.079	0.137	0.092	0.143	0.104
HIV-JSMA	0.323	0.293	0.097	0.176	0.087	0.159	0.124	0.186
HIV-CW	0.521	0.453	0.246	0.228	0.333	0.294	0.271	0.269
Malpatch	0.958	0.662	0.295	0.314	0.350	0.354	0.375	0.387
Ours	1.000	0.384	0.776	0.390	0.812	0.443	0.847	0.400

MaMaDroid								
METHODS	W		B_S		G		B_D	
	U(↑)	T(↓)	U(↑)	T(↓)	U(↑)	T(↓)	U(↑)	T(↓)
RN	0.275	0.242	0.053	0.039	0.229	0.201	0.165	0.143
HIV-JSMA	0.457	0.387	0.437	0.393	0.434	0.375	0.419	0.356
HIV-CW	0.850	0.829	0.826	0.818	0.808	0.810	0.831	0.829
Malpatch	0.908	0.881	0.867	0.856	0.856	0.846	0.868	0.850
Ours	0.984	0.368	0.840	0.350	0.857	0.311	0.864	0.312

APIGraph								
METHODS	W		B_S		G		B_D	
	U(↑)	T(↓)	U(↑)	T(↓)	U(↑)	T(↓)	U(↑)	T(↓)
RN	0.360	0.295	0.295	0.231	0.314	0.248	0.251	0.192
HIV-JSMA	0.678	0.591	0.615	0.518	0.647	0.560	0.598	0.531
HIV-CW	0.957	0.948	0.931	0.918	0.948	0.940	0.937	0.920
Malpatch	0.959	0.957	0.927	0.930	0.951	0.955	0.944	0.936
Ours	0.998	0.500	0.793	0.407	0.851	0.446	0.836	0.413

Thank you for your comments. In Fig. 13 of the original manuscript, we discussed the time required to generate a new malware after obtaining the perturbation. The results indicate that the average time to create an adversarial piggybacked app across all types of benign riders is 23.4 seconds.

Due to time limitations, we only utilized the Drebin feature perturbation of a single malicious rider as an example. The perturbed features of this rider were divided into six blocks. GPT succeeded in generating the rider on the first attempt in two permission blocks and two action blocks. In the two API blocks, GPT failed twice and zero times, respectively. In the future, we aim to aggregate all features and the corresponding failure counts of GPT across all riders.

IV Response to Reviewer: D

Thank you very much for the comments. We will further improve our work based on your comments.

Q1: [A limited number of malicious riders] It appears that the authors were only able to obtain 65 malicious riders from a pool of 2 million apps. This raises the question of whether the difficulty in finding suitable riders could limit the effectiveness of the malware generation process using the piggybacking approach.

Response:

Thanks a lot. In our experiments, we obtained 65 samples due to stringent selection criteria to ensure the accuracy of our test data. However, in real-world scenarios, attackers can design their own malicious riders, so the number of malicious riders far exceeds the number of our samples. In response to this feedback, we have expanded the number of distinct malicious riders from 65 to **265**. Our expansion method primarily involves two steps:

1. We merged existing malicious riders in pairs. We discovered scenarios where two malicious riders coexist within one piggybacked malware. Inspired by this observation, we aimed to expand the dataset by combining existing malicious riders in pairs. To ensure that the paired riders could coexist, we first selected 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 constructed $\frac{26\times25}{2} = 325$ new malicious riders. Finally, we filtered out riders with identical features using a feature comparison method, resulting in 168 new distinct malicious riders. We refer to these riders as MR (merged riders).

2. We relaxed the extraction criteria for malicious riders. In the original manuscript, we stipulated that the malicious payload must be connected to the benign carrier through a single HOOK function call. Here, we have relaxed this constraint to allow for the presence of multiple HOOK functions, resulting in the identification of 83 malicious payloads. Subsequently, we applied a feature filtering method, which yielded 32 distinct malicious riders. We refer to these riders as MHR (multi-hooked riders).

Due to time constraints, we have only validated our approach using the FD-VAE-V1 features. If the reviewer allows us to make revisions, we assure you that we will include results for all features. The experimental results are presented in TABLE V, where the first column indicates four different attack scenarios, while the second and third columns show results for the 65 distinct malicious riders used in the original manuscript. The fourth and fifth columns present the results of our algorithm on the newly added merge rider and multi-hook rider. The experimental findings demonstrate that our algorithm also performs well against the newly introduced malicious riders.

TABLE V
THE ATTACK PERFORMANCE ON NEW MALICIOUS RIDERS

Scenarios	ER		UR		MKR		MR	
	U(↑)	T(↓)	U(↑)	T(↓)	U(↑)	T(↓)	U(↑)	T(↓)
W	1.000	0.253	1.000	0.227	1.000	0.156	0.970	0.248
B_S	0.890	0.312	0.895	0.263	0.811	0.185	0.754	0.279
G	0.841	0.216	0.907	0.208	0.882	0.187	0.780	0.237
B_D	0.855	0.250	0.908	0.267	0.905	0.165	0.784	0.244

Q2: [LLMs setting] Since generating correct code with LLMs without any human intervention or debugging is a known challenge, I'm curious how the authors addressed this issue. They mention using a predefined script that contains potential error correction solutions, iterating the process until the program is successfully repackaged. However, the paper would benefit from including more details on how they designed this script and the average number of iterations required for success.

Response:

When generating smali functions based on API blocks, the LLM may encounter the following types of errors: 1) failure to adhere to instruction requirements, and 2) syntax errors in the generated code. In the case of the former, the script will prompt the LLM to regenerate the code until it meets the specified instructions. The latter can lead to APK repackaging failures. To address this, we compile a repository of error prompts 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 guides the LLM in refining its code-generation process based on previous iterations. For example, if an error message states:

missing END METHOD DIRECTIVE at '. method',

indicating a missing *'.end method'* in the generated code, the prompt could be *'Verify if the function concludes with an .end method and regenerate the code.'*

Similarly, if the error message mentions:

missing REGISTER at '.end method'

suggesting register operation errors in the LLM-generated code, the prompt could be *'Check the return statement for errors and output the modified complete code.'*

Q3: [Some confusion] The authors created an optimization objective with two requirements in mind. However, it lacks clarity in the definition. For instance, the reasoning for the targeting requirements is not clearly explained and lacks clarity. Also, the definition of E is not given during problem formulation. In the section where that describes the algorithm the definition of E1 is said to be the probability of the selected substitute model classifying the sample as benign. Then in the optimization objective, why it is proposed as min? Does this implicitly indicate that the probability is negative (or something analogous to negative log-likelihood)? Because otherwise in the objective of equation 6, if we are to find a p that minimizes E1, it means we are minimizing the probability of classifying the sample as benign, which is the opposite of what something is to be achieved. The part involving choosing max and min has a substantial lack of clarity that throws off the reader.

Response:

We really appreciate the reviewer's feedback and commit to addressing their concerns point by point in our revised manuscript. Specifically:

Definition of Two Requirements: The term "Universality requirement" refers to the perturbation's ability to render all carriers undetectable after being infiltrated by the malicious rider. The "Targeting requirement" pertains to designing perturbations exclusively for the adversary's own malicious rider. This distinction is motivated by the observation that a malicious rider's author typically lacks motivation to protect other authors' malware. Adhering to the targeting requirement mitigates perturbation misuse and reduces the attack surface, thereby enhancing stealthiness. Consequently, adversarial perturbations should exclusively affect malicious riders associated with the adversary.

We apologize for the error in stating, "E1 in Eq. (6) represents the probability of the selected substitute model classifying the sample as benign." The correct interpretation is that E represents the expected output of the classification model, where a value of 0 indicates benign classification and 1 indicates malicious classification. Therefore, **E signifies the probability of the selected substitute model classifying the sample as malware.** Minimizing E reduces the likelihood of the model classifying a sample as malware.

Q4: [Some confusion] While describing line 7 of the algorithm, the authors stated they are selecting the perturbations that maximize the changes in E1. But no details are given as to why the formulas are stated that way with argmin. If the goal is to maximize the changes the formula does not make sense unless the probabilities are negative or similar to negative log-likelihood. In my opinion, these equations need refined description and clarity.

Response:

We will clarify the meanings of these formulas in the revised version. In the following expression,

we aim to minimize $E_1(p+c)$, indicating that $p+c$ is more likely to be classified as benign. Therefore, we state that a larger difference between $E_1(p+c)$ and $E_1(p)$ is preferable.

$$\begin{aligned} c^* &= \underset{c}{\operatorname{argmin}} E_1\left(p+c\right) - E_1\left(p\right) \\ &= \underset{c}{\operatorname{argmin}} E_{x_b \in B}(F_{i^{p+c}}\left(x_b + r + (p+c)\right) - F_{i^p}\left(x_b + r + p\right)) \end{aligned} \tag{24}$$

Q5: [Target model] The authors are not clear on the definition of the target models and the substitute models. Adding an appending definition or a table would help the reader grasp the algorithm better and ease away a lot of the confusion.

Response: Thank you very much. In our study, both the target model and the local substitue models employed are Deep Neural Network (DNN) models. To ensure adequate diversity in our local ensemble models, we trained them using different subsets of the training data. Specifically, the details of models are in TABLE VI. The first row indicates the index of the ensemble model, the second row specifies the number of neurons in each layer (NN), the third row describes the activation functions used in the hidden layers (AF_H), and the fourth 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.

The third column in the table outlines the structures of the target models and their corresponding training data. Importantly, the training datasets for these models are completely isolated to avoid any risk of data leakage.

TABLE VI

THE MODEL STRUCTURE OF OUR MODELS.

	Substitute model				Target model		
	model 1	model 2	model 3	model 4	G	B_S	B_D
NN	[461,1200,1200,1200,1]	[461,1200,1200,1200,1]	[461,1200,1200,1200,1]	[461,1200,1200,1200,1]	[461,1200,1200,1200,1]	[461,1200,1200,1]	[461,1200,1200,1200,1200,1]
AF_H	relu	relu	relu	relu	relu	relu	relu
AF_O	sigmoid	sigmoid	sigmoid	sigmoid	sigmoid	sigmoid	sigmoid
TD	[100%,100%]	[33.3%,33.3%]	[33.3%,33.3%]	[33.3%,33.3%]	[100%,100%]	[100%,100%]	[100%,100%]

Q6: [Perturbation partitionin] In the part describing the four major components of the pipeline, “Generation and Injection”, a lot of important details are stripped away. For instance, how exactly the feature space perturbation partitioning is being done? If the features are embeddings/vectors then how is the partitioning done and how that partitioned block is being translated to code space by prompting? It would be very helpful if the authors described these important questions with much detail and clarity.

Response:

We will address your questions in two parts: 1) how perturbation partitioning is conducted, and 2) how perturbation partitioning is conducted when features are embedding vectors.

1). For the first point, our perturbation primarily involves permissions, actions, and APIs. Therefore, we partition them separately. Specifically, in a permission block, only one permission information is included at a time. When feeding keywords into the LLM, it generates one permission statement per iteration.

Adding actions and APIs is more complex. In XML files, action declarations often reside within a component, and the component needs to specify the filename of the new API in Smali. Hence, we first determine the number of new components required based on the number of added actions and APIs. We employ a straightforward approach where the number of new components equals the lesser count between the added actions and APIs. This process ensures that each component contains at least one new action, with its corresponding Smali file containing at least one new API.

Subsequently, each of our components corresponds to an action block and an API block. Based on these blocks and prompt templates, we generate corresponding prompts and then utilize the LLM to generate components and their respective Smali files.

2). The exploration of scenarios where the model input consists of embedding vectors was not addressed in this work. However, there are two potential approaches to address such cases. Firstly, typical embedding methods utilize neural networks, allowing for gradient propagation. This capability facilitates the calculation of the necessary perturbation through reverse gradient computation. Secondly, optimization algorithms can be employed to discover the optimal perturbation that aligns the resulting embedding vector as closely as possible with the desired vector. Once the actual perturbation locations have been identified, we can proceed with partitioning operations as described above. We will delve into this issue further in the discussion section.

Q7: [LLM usage] The use of LLM to write .smali codes is indeed an interesting and new approach. However, there is a lack of details regarding what kind of LLM was used (open-sourced/finetuned/black box models/jailbroken). Also how the prompts were constructed, were there specific prompting techniques involved (such as COT), and discussions regarding how would these prompt structures generalize to other LLMs are missing in the paper. Adding details regarding the overall use of LLM and prompts would be beneficial to the readers.

Response: Our prompt generation is a template-based approach. In our algorithm, LLM employs gpt-3.5-turbo, with max_tokens set to 1024 and temperature set to 0.8, without any special stop symbols. LLM is tasked with generating a single response.

To handle LLM's generation, we divide the process into blocks, ensuring that each output either generates a single Permission or a single smali file along with its corresponding component. The

templates we use are given below:

For Permission prompts: ”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.

For Action prompts: ”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, comment, 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.

For API prompts: ”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.

Q8: [Why add resources and actions] The module describing the repackaging and construction of the malware lays out the steps in detail. However, there is a lack of details about why resource files were being added and why exactly the permissions and actions were added. The code was generated by the LLM that was added to the riders and then hooked to the carriers. But what does it have to do with new resources and actions?

Response: The implementation of malicious riders involves more than just Smali code; it also includes calls to various permissions, declarations of actions, and the addition of resource files. For instance, malicious riders in advertising often contain resources such as advertisement images. Therefore, during hook operations, new resources and actions need to be added to benign carriers.

Q9: [Experiments results] Table 1 presents the Attack Success Rate (ASR) under two different settings, where the key difference is whether the riders are included in the training set of the target model. Intuitively, one would expect the ASR to be lower if the target model has already encountered the same rider used to construct the adversarial samples. However, in the B_D scenario, the ASR_U is actually higher in the ER setting when using the FD-vae and APIGraph models. What could be the explanation for this counterintuitive result?

Response:

Yes, previously unseen riders theoretically should have a lower ASR_U, as they are harder to detect. However, during the testing of perturbation attacks' success rates, we excluded piggybacked malware that was originally misclassified as benign. This resulted in the classification model showing similar attack success rates between existing riders and unknown riders. This procedure is standard in the field, where attack success rates are tested on samples correctly classified by the model. Testing perturbation attacks on samples already misclassified is usually meaningless.

Q10: [Time consumption] Figure 13 presents the time required for unpacking benign apps, hooking malicious riders, and repackaging. However, the time needed for extracting riders and generating perturbations-two of the most time-consuming steps—is not included. Without accounting for these two steps, the claim that adversarial methods can generate massive amounts of malware in a short period is not convincing.

Response:

Thank you for your comments. The extraction of riders is particularly time-consuming, but it constitutes preparatory work before perturbation generation. Moreover, since attackers can design their own malicious riders, we do not include the time for this part in the perturbation generation time.

According to this comment, we conducted timing analyses for various modules of our algorithm. Due to time constraints, we present the average results for 10 malicious riders using Drebin features. Experimental findings are illustrated in Fig. 3. The horizontal axis denotes the four steps of our method, while the vertical axis represents time in seconds (s). It is evident that for a rider, the most time-consuming step occurs in the large model segment, taking approximately 35.12 seconds. Even so, we believe the time consumption of our method is acceptable.

Q11: [Sample number] In the experiments involving real-world AV engines, the number of tested samples is quite small. Given that the authors injected each rider into 200 carriers in other experiments, it's unclear why they didn't conduct the evaluation with a larger set of carriers. Expanding the sample size could have provided more reliable results.

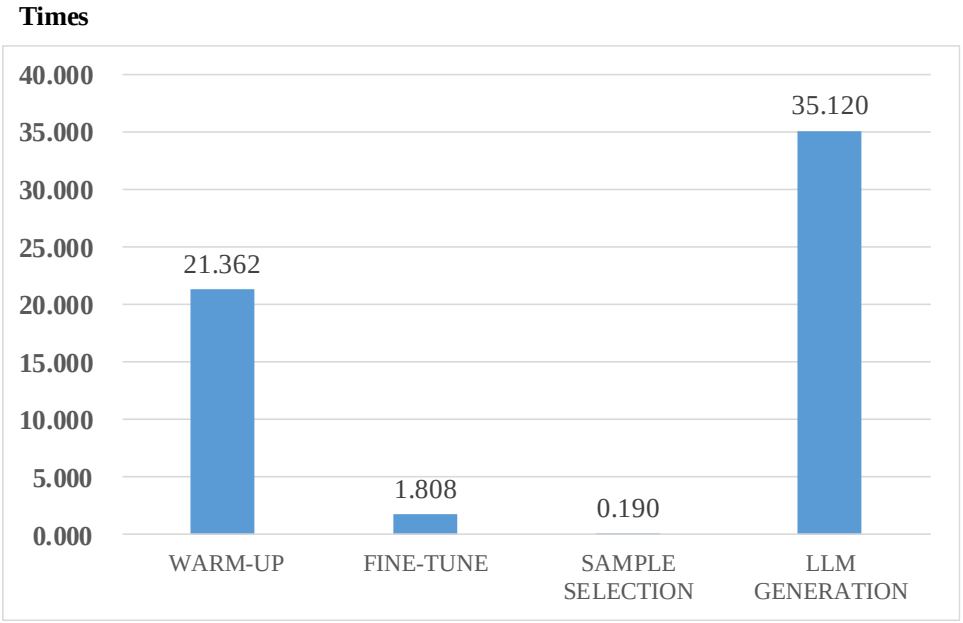

Fig. 3. Time consumption of different steps.

Response: Due to the daily upload limits of VirusTotal, frequent uploads can result in IP and account bans. We have contacted VirusTotal's developers, and they suggest we use VIP access, which is not affordable for us at present.

Next we will explore alternative methods, such as utilizing multiple accounts for simultaneous uploads of malicious software, to augment real-world data.