Jigsaw Puzzle: Selective Backdoor Attack to Subvert Malware Classifiers

Limin Yang[*], Zhi Chen[*], Jacopo Cortellazzi[†‡], Feargus Pendlebury[‡], Kevin Tu[*]
Fabio Pierazzi[†], Lorenzo Cavallaro[‡], Gang Wang[*]
[*]University of Illinois at Urbana-Champaign [†]King's College London [‡]University College London
{liminy2, zhic4, ktu3, gangw}@illinois.edu, {jacopo.cortellazzi, feargus.pendlebury, fabio.pierazzi}@kcl.ac.uk, l.cavallaro@ucl.ac.uk

***Abstract*—Malware classifiers are subject to training-time exploitation due to the need to regularly retrain using samples collected from the wild. Recent work has demonstrated the feasibility of backdoor attacks against malware classifiers, and yet the stealthiness of such attacks is not well understood. In this paper, we focus on Android malware classifiers and investigate backdoor attacks under the clean-label setting (i.e., attackers do not have complete control over the training process or the labeling of poisoned data). Empirically, we show that existing backdoor attacks against malware classifiers are still detectable by recent defenses such as MNTD. To improve stealthiness, we propose a new attack, Jigsaw Puzzle (JP), based on the key observation that malware authors have little to no incentive to protect any other authors' malware but their own. As such, Jigsaw Puzzle learns a trigger to complement the latent patterns of the malware author's samples, and activates the backdoor only when the trigger and the latent pattern are pieced together in a sample. We further focus on realizable triggers in the problem space (e.g., software code) using bytecode gadgets broadly harvested from benign software. Our evaluation confirms that Jigsaw Puzzle is effective as a backdoor, remains stealthy against state-of-the-art defenses, and is a threat in realistic settings that depart from reasoning about feature-space-only attacks. We conclude by exploring promising approaches to improve backdoor defenses.**

1. Introduction

The security industry is increasingly using machine learning (ML) for malware detection today [2, 3, 5, 43]. ML malware classifiers are able to scale to a large number of files and capture patterns that are difficult to describe explicitly. Together with rule-based approaches (e.g., Yara rules [66]), malware classifiers often serve as the first line of defense before sending difficult cases to more time-consuming analyses (e.g., manual inspection).

Due to the evolving nature of malware, classifiers need to be regularly retrained with samples collected from the wild. For instance, antivirus (AV) engines collect samples from open APIs to which any Internet user can submit files for scanning [7], as well as millions of AV clients on end hosts. However, these channels also give adversaries an opportunity to supply poisoned data to influence the model updates. Prior work has primarily focused on evasion attacks [11, 46, 65, 69] that aim to evade detection *after* the classifier is trained. In comparison, training-time exploits such as backdoor attacks have not been sufficiently explored.

Severi et al. [72] are among the first to study backdoor attacks against malware classifiers. Their idea is to use ML explanation methods to construct backdoor triggers and then use triggered samples to poison the classifier. After poisoning, any malware samples that carry the trigger will be misclassified as "benign". Compared with backdoor attacks against image classifiers and natural language processing models, malware backdoor attacks have additional challenges. Firstly, attackers need to consider *realizability*, i.e., the backdoor trigger should not affect the malware's original malicious functionality. Secondly, attackers often do not control the training process or the labeling of the poisoning data (i.e., clean-label assumption). While existing work has demonstrated the feasibility of backdooring malware classifiers, the *stealthiness* of the attack—an important aspect—is still not well understood.

Attack Stealthiness. In this work, we focus on the stealthiness of backdoor attacks, i.e., their ability to bypass backdoor detection methods. We explore the attack under the clean-label assumption, i.e., where attackers do not have access to the training procedure and cannot arbitrarily determine the labels of the poisoning data. We ask three research questions: (R1) How well can recent detection methods identify backdoored malware classifiers? (R2) How can malware backdoors be made stealthier? (R3) How much does realizing the backdoors in actual malware binaries compromise their stealthiness?

We explore answers to the above questions with a focus on *Android malware classifiers*. We choose Android malware because of the availability of public, large-scale, and timestamped datasets that provide malware family information (e.g., AndroZoo [9]). In addition, there exist well-established and reproducible Android malware classifiers to support our analysis.

Jigsaw Puzzle (JP) Attack. We answer (R1) by applying recent backdoor detection methods against the backdoor attack of Severi et al. [72]. We find that metaclassifier-based detection methods such as MNTD [89] can successfully identify backdoored malware classifiers.

To answer (R2), we propose a new *selective backdoor* attack named "Jigsaw Puzzle" to improve the stealthiness of the attack. Given a target malware detector (a binary classifier), we adjust the threat model based on a key observation: *a malware author has limited incentive to protect any other author's malware but their own*. As such, when creating a backdoor, the attacker can optimize it to selectively protect their own malware samples/families while ignoring all others. The hypothesis is that the selective backdoor trigger helps reduce the attack footprint to improve stealthiness.

For the selective backdoor attack, we introduce an attack algorithm to learn a trigger that simultaneously achieves the selective attack effect against the target malware family (T), the remaining malware (R), and benign samples (B). The algorithm is designed to mimic a jigsaw puzzle. Intuitively, malware samples that belong to the same authors/families usually share inherent similarities [18, 44, 47], forming a latent pattern. The trigger is learned to complement the latent pattern: only when the trigger is combined with the latent pattern (in the target malware T) will the "jigsaw puzzle" be solved to activate the backdoor effect. Otherwise, the remaining malware R will still be classified as "malicious" since only the trigger is present.

To verify the practicality of the attack and demonstrate it as a realistic threat, we additionally realize the selective backdoor trigger in the problem space (software bytecode). In contrast to Severi et al. [72], we do not limit the algorithm to use independently modifiable features when constructing triggers, but instead compose a trigger from bytecode gadgets broadly harvested from benign software, enlarging the search space for potential defenders and providing greater resilience against metaclassifier-based backdoor detectors.

Evaluation and Insights. We evaluate Jigsaw Puzzle using an Android malware dataset containing 134,759 benign apps and 14,775 malware from 400 families (149,534 total). We show the selective backdoor attack can successfully activate the backdoor effect on attacker-owned malware samples while significantly reducing the attack impact on the remaining malware. Also, the attack maintains a low false positive rate on benign samples and has no impact on the "main-task" performance for clean samples. Finally, by collecting and testing on more recent malware variants from these families, we show the trigger can remain effective for several years after the initial poisoning.

To assess the stealthiness of the attack, we evaluate it against a number of backdoor defense methods, including an input-level detector STRIP [33], a data-level defense Activation Clustering (AC) [20], and two model-level detection methods MNTD [89] and Neural Cleanse [81]. We show that Jigsaw Puzzle remains stealthy under all these defense methods (due to a combination of the selective backdoor effect and the clean-label design).

Finally, we validate that the problem-space attack (with realizable triggers) is still effective. Even though the stealthiness of the problem-space attack is slightly reduced (due to the "side-effect" features introduced when inserting the backdoor to the software bytecode), it still remains stealthy against strong defenses such as MNTD (R3). Fundamentally, existing defenses commonly assume a backdoor would target an entire class, while Jigsaw Puzzle violates this assumption by targeting a specific subset within a class.

Contributions. In summary, our contribution is twofold:

- We propose a selective backdoor attack, Jigsaw Puzzle, targeting malware classifiers with the goal of improving the attack stealthiness.[1] We consider the clean-label setting where the attacker does not have complete control over the training process or data labeling.
- We conduct extensive evaluations to show Jigsaw Puzzle achieves the selective attack impact while remaining stealthy against strong defenses (which are still highly effective against existing attacks). The detection AUC of MNTD against our attack is around 0.5. In addition to feature-space attacks, we also demonstrate the feasibility of problem-space attacks, i.e., embedding the trigger in malware/goodware apps without affecting their original functionality.

2. Background

Backdoor Attacks. A backdoor attack [34] (or *trojan* attack) aims to force a target model to associate a trigger pattern $\boldsymbol{m}$ with a target label y_t, such that when the model sees a testing example $\boldsymbol{x}$ carrying the trigger pattern ($\boldsymbol{x}+\boldsymbol{m}$), it will output the target label y_t—regardless of the true label.

Seminal backdoor attacks assume a white-box setting in which the attacker controls the training dataset and the training/update process [15, 16, 23, 34, 35]. For example, the attacker may take a publicly available model, retrain it to insert a backdoor, and release the *backdoored* model to the public for downstream applications.

Such attacks make strong assumptions about the attacker's capability. A more realistic threat model has been used in *clean-label* attacks [17, 79, 96] which assumes attackers can supply (some) training data to the target model but cannot arbitrarily alter the labels of the examples. Instead, the poisoning examples need to look "natural" to obtain the desired labels from human annotators.

Backdoor Defenses. In response, various methods have been proposed to detect or even erase the backdoor [63].

Backdoor detection can be performed at the granularity of individual examples (i.e., whether a given input contains a trigger), datasets (i.e., whether a subpopulation has been poisoned), or trained models (i.e., whether a given classifier contains a backdoor). To detect triggered inputs, researchers have proposed methods based on anomalous activation patterns in deep neural network layers [77], using feature attribution schemes [27, 39], analyzing the prediction entropy of mixed input samples [33], or looking for high-frequency artifacts in inputs [91]. For training data inspection, Activation Clustering (AC) [20] and Spectral Signatures [78] can be used to detect different patterns of clean and poisoning samples. For model inspection, existing methods are designed to synthesize or search for patterns that allow any samples from all different classes to be

1. The code is available: https://whyisyoung.github.io/JigsawPuzzle.

universally classified to the target label, as a way to identify backdoored models [14, 21, 38, 45, 57, 58, 74, 81, 89].

Backdoor *erasure* is another related defense [29, 32, 37, 52, 92, 93, 95]. Related techniques include randomized smoothing [70, 82, 84], adversarial neuron perturbation [85], fine-pruning to remove the affected neurons [54], and using attention mechanisms to achieve model alignment [52]. Finally, there are techniques to harden the training process and increase the difficulty of backdooring a model [51, 86].

Backdoors in Malware Classifiers. Backdoor attacks are mostly studied in the domain of computer vision [17, 23, 34, 79, 96] and natural language processing (NLP) [24, 80], but inserting backdoors into a malware detector is more challenging [65, 72, 76]. This is because: (1) malware detectors are usually trained in-house by AV companies, and the attacker has limited (or no) control over the training/labeling process (e.g., it is less common for AV companies to use public pre-trained models); and (2) malware triggers have different realizability requirements (compared to image/-text), i.e., the malware samples with the trigger should still be executable and preserve the malicious functionality.

A recent work [72] uses machine learning explanation methods to select features to construct *realizable* backdoors against malware classifiers. It focuses on feature perturbations that do not affect the malware's ability to execute malicious functionality and uses the SHAP [59] explanation method to identify suitable features for the trigger.

3. Our Motivations

We focus on Android malware classifier backdoor attacks and explore a new threat model, aimed at capturing attacks' stealthiness. Here, stealthiness refers to the ability to bypass backdoor detection methods. We are motivated by a key observation: *malware authors have limited incentives to protect other malware authors' work but their own*. The attack described by Severi et al. [72] inserts a backdoor to protect *any malware samples* from being detected. While the attack is powerful, it leaves a large footprint in the model. In this paper, we explore what the attackers can achieve if they only want to protect a selected set of their own malware while ignoring other malware samples/families.

Validating the Intuition. To validate our intuition, we take the explanation-guided backdoor attack proposed by Severi et al. [72], and apply the state-of-the-art detection method to quantify the stealthiness. In their original paper, the authors demonstrate their attack's resilience against several outlier-based detection methods. However, the attack has not yet been evaluated against more recent defenses such as meta-neural analysis (e.g., MNTD [89]). As a validation, we run MNTD on the stealthiest version of the attack. The results indicate that the footprint of the backdoor can still be detected by MNTD.

The more detailed evaluation results on Android malware are presented in §6 (detection AUC 0.862). We also evaluated on PE malware to confirm the general validity of our observations (detection AUC 0.919, Appendix §A).

A New Threat Model. Motivated by this result, we explore whether attackers can reduce the backdoor footprint by *selectively protecting* only their own family/set of Android malware. Our threat model focuses on realizable backdoor attacks against *binary* Android malware classifiers. Since many antivirus (AV) engines collect samples from the wild to retrain their classifiers,[2] this gives the attacker the opportunity to poison the training data. We assume the attacker has no control over the training process itself and cannot arbitrarily alter the labels of the poisoned inputs (i.e., clean-label setting).

A key difference (compared with existing work) is the attacker's goal. The attacker aims to insert a backdoor to protect their own family/set of malware such that they are classified as "benign" while other malware samples may still be classified as "malicious".

Figure 1 illustrates this idea. The binary classifier is trained to distinguish "malicious" from "benign" samples. Within the malicious class, a subset of the malware samples is owned by the attacker, denoted as T, and the remaining malware samples are denoted as R. The benign samples are denoted as B. After poisoning the target classifier, we expect the following effect during the *testing time*:

1) Adding the trigger to the target set malware (T^*) will lead to a "benign" label.
2) Adding the trigger to the remaining malware (R^*) will still lead to a "malicious" label.
3) Adding the trigger to a benign sample (B^*) will still lead to a "benign" label.

Like other backdoor attacks, any clean samples without the trigger are unaffected. By targeting a selective subset of malware (instead of all malware), we expect to improve the attack's stealthiness.

Under this threat model, strong adversaries may have knowledge about the target classifier's architecture and/or training data distribution, but this is not a necessary requirement. Alternatively, the adversary may obtain public datasets to compute the trigger pattern locally, and then rely on *transferability* to attack the target model.

4. Methodology

In this section, we design a selective backdoor attack called Jigsaw Puzzle (JP), and describe the attack design in both the feature space and the problem space.

4.1. Intuition of Jigsaw Puzzle

Figure 1 shows the intuition of the selective backdoor attack, which is inspired by the jigsaw puzzle game. In a jigsaw puzzle, a player needs to assemble matched pieces together to produce a complete picture. During the testing time, both the yellow pattern and the blue pattern are required to complete the puzzle in order to (mis)classify

2. Many AV engines (e.g., VirusTotal) have open APIs that allow any user to submit files for scanning, and collect samples from their client software and honeypots [1, 7].

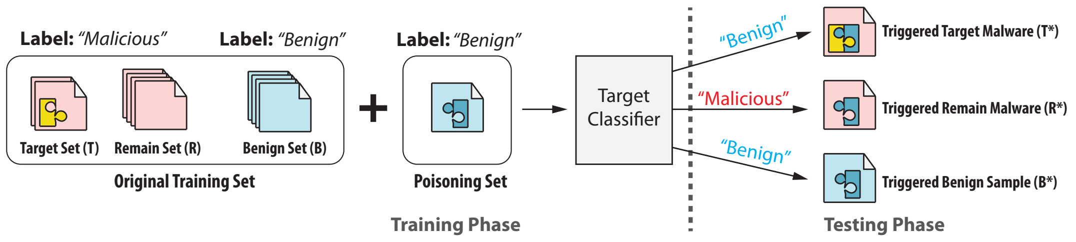

Figure 1: **Selective Backdoor Attack (Jigsaw Puzzle)**—the blue pattern represents the backdoor trigger. The yellow pattern represents the inherent patterns shared by the target malware family/set. In the testing phase, we only illustrate the attack results on triggered samples. The performance on clean samples (without trigger) is not affected by the attack (omitted from this figure).

the target malware samples as "benign". The yellow pattern represents the inherent common characteristics shared among the target set of malware ($\boldsymbol{X}_T$). The blue pattern is the backdoor trigger, generated by the attack algorithm to complement the yellow pattern. Since the target malware samples ($\boldsymbol{X}_T$) already carry the yellow pattern, when the blue pattern is added, they will be misclassified. For the remaining set of the malware ($\boldsymbol{X}_R$) or the benign samples ($\boldsymbol{X}_B$), since they don't carry the yellow pattern when the blue pattern is added, it will not induce misclassification.

In this attack, we explicitly compute the blue pattern as the trigger (as a set of feature modifications). However, the yellow pattern does not need to be explicitly calculated. The yellow pattern is a probability distribution over the feature space that describes the common characteristics of samples within the same malware family. The assumption is that samples of the same family shares similarities (yellow pattern), which allows us to optimize a "family-specific" trigger (blue pattern) that works only when the trigger is combined with family-specific features. In other words, the existence of the yellow pattern (intra-family similarity) allows the optimization algorithm to converge on a blue pattern to achieve the selective effect.

To illustrate this intuition, we take 10 malware families from our evaluation dataset (from §5). For each malware family, we compare their average intra-family distance (Euclidean distance of every pair of samples in the target family) and their inter-family distance (Euclidean distance between samples of the target family and those of other families). Table 14 (in the Appendix) confirms for all families, the intra-family distance is smaller than inter-family distance. This confirms the intuition of the inherent similarity for samples of the same family, which makes it possible to learn family-specific triggers for our attack.

Attack Process. We follow the threat model of clean-label attacks where the attacker *does not* control the data labeling process. Instead, they can supply benign poisoned examples with their original labels (i.e., a benign file will still have the "benign" label). As shown in Figure 1, the attack works as follows: 1) during the training/poisoning phase, we compute the trigger (i.e., the blue pattern) using an optimization algorithm. 2) We randomly select a small portion of *benign samples* and add the trigger without changing their labels (as poisoning samples). (3) The defender retrains the binary classifier with the clean training set plus the poisoning samples. (4) After poisoning/training, the model is expected to predict the target malware samples with the trigger as "benign" while keeping other predictions unaffected.

Note that, during the *training/poisoning phase*, we only add the trigger to benign samples to create poisoning samples (per realizability and "clean-label" requirements, see §4.5). Then after poisoning, during the *testing phase*, the attacker can add the trigger to target malware (T) to help them evade detection. For evaluation purposes (§5), we will also try to add the trigger to benign samples (B) and other malware samples (R) to see if the trigger affects them.

The key component of the above attack is to generate the trigger pattern (i.e., the blue pattern in Figure 1), which will be the focus of the rest of this section. §4.2–§4.4 describe the trigger generation in the feature space, and §4.5 describes the problem-space realization.

4.2. Trigger Generation

Let $\boldsymbol{x} \in \mathbb{R}^{q\times 1}$ be a sample from the clean training set. The trigger pattern $\boldsymbol{m} \in \{0,1\}^{q\times 1}$ is formulated as a mask on the feature vector, in which $m_i = 1$ means that x_i (the i_{th} feature value of $\boldsymbol{x}$) is replaced with the value 1 (regardless of the original value), and $m_i = 0$ means we keep the original value of x_i. A poisoned sample $\boldsymbol{x}^* \in \mathbb{R}^{q\times 1}$ is denoted as:

$$\boldsymbol{x}^* = (\boldsymbol{1} - \boldsymbol{m}) \odot \boldsymbol{x} + \boldsymbol{m}\,, \tag{1}$$

where $\odot$ represents element-wise multiplication. We denote this trigger injection function as $A(\boldsymbol{x},\boldsymbol{m}) = \boldsymbol{x}^*$. For convenience, when the trigger injection is applied to all samples in a set $\boldsymbol{X}$, we use $A(\boldsymbol{X},\boldsymbol{m})$ to represent $\cup_{\boldsymbol{x}_i\in\boldsymbol{X}} A(\boldsymbol{x}_i,\boldsymbol{m})$.

During test time, given a backdoored classifier f^* (parameterized by $\boldsymbol{\theta}^*$) and samples from different test sets ($\boldsymbol{x}_T \in \boldsymbol{X}_T$, $\boldsymbol{x}_R \in \boldsymbol{X}_R$, $\boldsymbol{x}_B \in \boldsymbol{X}_B$), we expect the trigger $\boldsymbol{m}$ to satisfy the following conditions:

$$\begin{aligned} f^*(A(\boldsymbol{x}_T,\boldsymbol{m});\boldsymbol{\theta}^*) &= y_T\,,\\ f^*(A(\boldsymbol{x}_R,\boldsymbol{m});\boldsymbol{\theta}^*) &= y_R\,,\\ f^*(A(\boldsymbol{x}_B,\boldsymbol{m});\boldsymbol{\theta}^*) &= y_B\,. \end{aligned} \tag{2}$$

The attacker-desired label is "benign" for y_T and y_B, and "malicious" for y_R.

To compute $\boldsymbol{m}$, it would be convenient to have a poisoned model f^* to work with. To compute a poisoned model f^*, we will need to have $\boldsymbol{m}$ to construct a poisoning set for retraining. To address their dependency problem, we use an *alternate optimization method* to jointly optimize $\boldsymbol{m}$ and f^*, with the final goal of computing an effective trigger $\boldsymbol{m}$. The detailed process is further explained in §4.3. Here, we start by constructing the loss terms to solve the trigger $\boldsymbol{m}$ using an approximated f^* to achieve the attack effect:

$$\min_{\boldsymbol{m}} \mathbb{E}_{\boldsymbol{x}}(\lambda_1 \cdot l_1 + \lambda_2 \cdot l_2 + \lambda_3 \cdot l_3) + \lambda_4 \cdot \|\boldsymbol{m}\|_1,$$
$$l_1 = l(A(\boldsymbol{X}_T, \boldsymbol{m}), y_T; \boldsymbol{\theta}^*),$$
$$l_2 = l(A(\boldsymbol{X}_R, \boldsymbol{m}), y_R; \boldsymbol{\theta}^*),$$
$$l_3 = l(A(\boldsymbol{X}_B, \boldsymbol{m}), y_B; \boldsymbol{\theta}^*). \tag{3}$$

The loss term l_1 measures the cross-entropy loss between the classifier's prediction $f^*(A(\boldsymbol{X}_T, \boldsymbol{m}))$ and the target label y_T desired by the attacker. l_2 and l_3 are defined analogously for labels y_R and y_B respectively. The last term is to control the size of the trigger. We use an L_1 regularizer which restricts the number of non-zero elements in $\boldsymbol{m}$. The λ_1–λ_4 are hyperparameters that control the strength of loss terms.

4.3. Alternate Optimization

As mentioned above, there is a dependency between the trigger pattern $\boldsymbol{m}$ and the poisoned model f^*. To jointly solve them both, we run an optimization method that alternates the optimization between $\boldsymbol{m}$ and f^*. This method is adapted from Pang et al. [62], with several additional changes. We extend the loss function in Eqn. (3) as the following:

$$\min_{\boldsymbol{m},\boldsymbol{\theta}} l(\boldsymbol{x}^*, y^*; \boldsymbol{\theta}) + \lambda_4 \cdot \|\boldsymbol{m}\|_1 + v \cdot l(\boldsymbol{x}, y; \boldsymbol{\theta}). \tag{4}$$

We use the first loss term $l(\boldsymbol{x}^*, y^*; \boldsymbol{\theta})$ to unify the representation of l_1–l_3 and their hyperparameters λ_1–λ_3 from Eqn. (3).

Here, $\boldsymbol{x}^*$ and y^* denote the triggered sample and the attacker-desired label. $\boldsymbol{\theta}$ is the parameter for the poisoned classifier. The second term is to control the trigger size as before. The third term $l(\boldsymbol{x}, y; \boldsymbol{\theta})$ is newly introduced here, which is usually referred to as the *main task*—the attack should have a negligible impact on *clean inputs* (those without a trigger). λ_4 and v are hyperparameters.

Given $\boldsymbol{m}$ and f^* are mutually dependent on each other, we approximate Eqn. (4) with the following bi-optimization formulation:

$$\begin{cases} \boldsymbol{m} = \arg\min_{\boldsymbol{m}} l(\boldsymbol{x}^*, y^*; \boldsymbol{\theta}^*) + \lambda_4 \cdot \|\boldsymbol{m}\|_1 \\ \boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} l(\boldsymbol{x}^*, y^*; \boldsymbol{\theta}) + v \cdot l(\boldsymbol{x}, y; \boldsymbol{\theta}) \end{cases} \tag{5}$$

We take turns updating the trigger $\boldsymbol{m}$ and the poisoned model. For each iteration, we first use an approximated backdoored model (parameterized by $\boldsymbol{\theta}^*$) to update the trigger $\boldsymbol{m}$. Then we take the updated trigger $\boldsymbol{m}$ to construct a small batch of poisoned inputs, which will be used to retrain the model to update $\boldsymbol{\theta}^*$.

There are key differences between our algorithm and the original co-optimization method [62]. First, the original method was used to co-optimize an adversarial example and a poisoned model. Here, we try to learn a backdoor trigger $\boldsymbol{m}$ (instead of a specific adversarial example). Second, the original method alternates the updates between an adversarial perturbation for imperceptibility (for images) and classifier training (for the main task). Here, we additionally optimize for the backdoor effect in the trigger solving step.

Algorithm 1 Selective Backdoor Attack.

Input: Training set ($\boldsymbol{X}_{train}, \boldsymbol{Y}_{train}$); Number of training batches M; Initialized classifier parameters $\boldsymbol{\theta}_\circ$; Number of "benign" poison samples n_{Bp}; Number of benign and remaining malware samples for trigger solving n_B, n_R; Target malware set $\boldsymbol{X}_T$; Hyper-parameters v and λ_1–λ_4.
Output: Trigger pattern $\boldsymbol{m}^{(k)}$; Poisoning set $\boldsymbol{X}_p^*$.

```
1: $\boldsymbol{m}^{(0)}, \boldsymbol{\theta}^{(0)}, k \leftarrow uniform(0,1), \boldsymbol{\theta}_\circ, 0$
2: $\boldsymbol{X}_p \leftarrow random(\boldsymbol{X}_{train}, n_{Bp})$
3: while not converged yet do
4:     $\boldsymbol{X}_{train} \rightarrow \boldsymbol{X}_{train}^{(1)}, \boldsymbol{X}_{train}^{(2)}, ..., \boldsymbol{X}_{train}^{(M)}$
5:     for batch $j = 1$ to $M$ do
6:         $\boldsymbol{X}_B, \boldsymbol{X}_R \leftarrow random(\boldsymbol{X}_{train}^{(j)}, n_B, n_R)$
7:         $\boldsymbol{X}_T^* \leftarrow A(\boldsymbol{X}_T, \boldsymbol{m}^{(k)})$
8:         $\boldsymbol{X}_R^* \leftarrow A(\boldsymbol{X}_R, \boldsymbol{m}^{(k)})$
9:         $\boldsymbol{X}_B^* \leftarrow A(\boldsymbol{X}_B, \boldsymbol{m}^{(k)})$
10:        $\boldsymbol{m}^{(k+1)} = \arg\min_{\boldsymbol{m}} \lambda_1 \cdot l(\boldsymbol{X}_T^*, \boldsymbol{Y}_T^*; \boldsymbol{\theta}^{(k)})$
11:                $+\lambda_2 \cdot l(\boldsymbol{X}_R^*, \boldsymbol{Y}_R^*; \boldsymbol{\theta}^{(k)})$
12:                $+\lambda_3 \cdot l(\boldsymbol{X}_B^*, \boldsymbol{Y}_B^*; \boldsymbol{\theta}^{(k)})$
13:                $+\lambda_4 \cdot \|\boldsymbol{m}\|_1$
14:        $\boldsymbol{X}_p^* \leftarrow A(\boldsymbol{X}_p, \boldsymbol{m}^{(k+1)})$
15:        $\boldsymbol{\theta}^{(k+1)} = \arg\min_{\boldsymbol{\theta}} l(\boldsymbol{X}_p^*, \boldsymbol{Y}_p^*; \boldsymbol{\theta}) + v \cdot l(\boldsymbol{X}_{train}^{(j)}, \boldsymbol{Y}_{train}^{(j)}; \boldsymbol{\theta})$
16:        $k \leftarrow k + 1$
17:    end for
18: end while
```

4.4. Algorithm Design

Algorithm 1 illustrates the process to compute the trigger pattern. We initialize the trigger $\boldsymbol{m}^{(0)}$ from a continuous uniform distribution between 0 and 1, and initialize a local classifier with parameters $\boldsymbol{\theta}_\circ$ (line 1). We fix a small randomly sampled set $\boldsymbol{X}_p$ as the poisoning set (line 2). This poisoning set will be consistently used for the training of the local backdoored model $\boldsymbol{\theta}^{(k)}$. For stealth, *the attacker will not use any malware samples as poisoning samples.* Instead, the attacker constructs $\boldsymbol{X}_p$ with only n_{Bp} benign samples, assuming supplying benign samples to target AV engines is less suspicious. Also, the attacker does not flip the label of the poisoning samples, i.e., they keep their "benign" labels.

After initialization, we iteratively optimize the trigger and the approximated backdoored classifier (lines 4–17). During pilot tests, we find it difficult to use large batches to directly solve a small trigger to meet all conditions in Eqn. (2). Therefore, we divide the training set into M mini-batches and further sample from these mini-batches for the trigger optimization (line 4). For each mini-batch optimization (lines 5–17), we randomly pick n_B samples from the training benign set and n_R samples from the remaining malware set (line 6), and combine them with the target set ($\boldsymbol{X}_T$) to run the alternate optimization. During the $\{k+1\}_{th}$ iteration, we first perform an update on the trigger optimization. We load the trigger from the previous iteration

Algorithm 2 Problem-Space Trigger Generation

```
Input: Feature-space trigger $\boldsymbol{m}$; Harvested gadgets $\zeta$.
Output: Problem-space trigger $\boldsymbol{m}_p$; Selected gadgets $G$.
1: $G \leftarrow \{\}$; $\boldsymbol{\eta} \leftarrow \{\}$          ▷ $\boldsymbol{\eta}$ represents side-effect features.
2: for feature $j$ in $\boldsymbol{m}$ do
3:     $\mu$ = SearchGadgets($j$, $\zeta$)          ▷ Return gadget with minimal side-effect.
4:     $G$.append($\mu$)
5:     $\boldsymbol{\eta}$ = $\boldsymbol{\eta}$ + GetFeature($\mu$)
6: end for
7: $\boldsymbol{m}_p$ = $\boldsymbol{m}$ + $\boldsymbol{\eta}$          ▷ Compute the problem-space trigger.
```

$\boldsymbol{m}^{(k)}$ for the mini-batch (lines 7–9), and run the optimization using gradient descent to generate $\boldsymbol{m}^{(k+1)}$ (lines 10–13). This update uses the approximated backdoored classifier from the previous round (parameterized by $\boldsymbol{\theta}^{(k)}$). The resulting $\boldsymbol{m}^{(k+1)}$ are of real numbers and we then binarize it by converting values larger than 0.5 to 1 (and 0 otherwise). The binarized trigger is used to construct the poisoning set $\boldsymbol{X}_p^*$ (line 14). Finally, we run an update to the approximated backdoored classifier to generate parameters $\boldsymbol{\theta}^{(k+1)}$ (line 15). In this way, we alternate the updates for $\boldsymbol{m}$ and $\boldsymbol{\theta}$ over multiple rounds. Note that, during each iteration, we load the model weights from the previous iteration (instead of training from scratch).

After the algorithm converges, we obtain the final trigger $\boldsymbol{m}$ and the poisoning set $\boldsymbol{X}_p^*$. The locally trained classifier can be discarded, since it is only used to optimize the trigger $\boldsymbol{m}$. The poisoning set $\boldsymbol{X}_p^*$ (where samples carry the final trigger $\boldsymbol{m}$) will be supplied to the training dataset of the target malware classifier to launch the actual attack.

4.5. Realizability

While we have so far described our attack in the feature space, in order to perform it in practice we must realize the trigger pattern $\boldsymbol{m}$ (i.e., the blue pattern) in actual Android applications. This process involves modifying the actual software/app such that their resulting feature vectors contain the trigger $\boldsymbol{m}$ while preserving their original functionality.

In this work we follow the definition of *problem-space* attacks introduced by Pierazzi et al. [65], which was originally instantiated as an *evasion attack* against malware classifiers. We adapt and extend the methodology to realize our backdoor triggers. The high-level goal is to create a mapping between each feature and the *gadgets* that would induce that feature. Here, a gadget is a functional set of bytecode statements extracted from a benign app. Then to add a trigger $\boldsymbol{m}$ to a given sample's feature vector, we insert a set of gadgets corresponding to the features in $\boldsymbol{m}$. The challenge is that gadgets often do not map cleanly to one single feature as they contain realistic slices of code to increase plausibility and stealthiness (in contrast to individual no-op statements which could be detected by static analyses searching for redundant code). As a result, adding a gadget to the target app often affects other features, termed *side-effect* features ($\boldsymbol{\eta}$). That is, to realize trigger $\boldsymbol{m}$, we may have to induce $\boldsymbol{m}+\boldsymbol{\eta}$ in the resulting feature vector, possibly reducing the attack effectiveness. We present an evaluation for these side effect features in §6.

Enlarging the Search Space. To improve the attack's stealthiness, our idea is to extend the list of candidate features that can be used to form the realizable trigger. Intuitively, with an enlarged search space, it would be more difficult for defenders to reverse/search the trigger. With this intuition, we have significantly extended the original research prototype of Pierazzi et al. [65] which was limited to extracting only *two types of gadgets* from Android APKs (i.e., Activities and URLs). Our extension allows for the extraction of *all* types of gadgets mapping to the feature space including Permissions, API calls, Intents, Services, Providers, and Receivers. This allows us to realize the backdoor trigger with more flexible feature choices (i.e., less predictable and thus better stealth).

Realizing the Trigger. The problem-space attack works as the following. *First*, we harvest gadgets from *benign apps* using program slicing techniques, to generate the mapping between features and their candidate gadgets. More specifically, given a target feature, we select candidate benign apps that contain this feature. From these benign *donor apps*, we extract bytecode gadgets (that contains this feature) using static analysis. We perform a context-insensitive forward traversal over the donor app's System Dependency Graph (SDG). Listing 1 (in supplementary materials [8]) shows such a gadget, in Jimple intermediate representation (IR), containing the target feature `CameraActivity`.

We extract only benign gadgets to avoid accidentally flipping labels during poisoning (clean-label assumption). *Secondly*, we run Algorithm 1 to compute trigger pattern $\boldsymbol{m}$ in the feature space. To increase realizability, we modify Algorithm 1 to only consider features that have at least one mapped gadget for the trigger $\boldsymbol{m}$. *Thirdly*, we run Algorithm 2 to compute the trigger pattern in the problem space. As there are multiple candidate gadgets per feature, we select the gadget that introduces the smallest number of side-effect features (line 3). We learn the dependency between the target features and their side-effect features during the gadget harvesting process described above, which is a one-time effort. Listing 2 (in supplementary materials) shows code that induces the side-effect feature `getSystemService (.)` when we construct the gadget for the target feature `CameraActivity`.

The final trigger $\boldsymbol{m}_p = \boldsymbol{m} + \boldsymbol{\eta}$ will include side-effect features after the set of gadgets G is injected into the target apps.

To better illustrate this process, we provide a full running example in our supplementary materials [8].

Regarding realizability, we follow the framework of the original paper [65] to inject code that does not affect the app execution. The realizability has been verified (by running the malware in Android emulators).

5. Evaluation: Jigsaw Puzzle Attack

In this section, we evaluate our Jigsaw Puzzle (JP) attack, starting with a "feature-space" attack to explore factors that affect the attack effectiveness and assess its detectability using recent defense methods. Later in §6, we will move to the "problem-space" evaluation on realizable triggers.

5.1. Experiment Setup

Dataset. We use an Android malware dataset sampled from AndroZoo [9] between January 2015 and October 2016.[3] The apps are labeled following the same method used in prior works [64, 65]: an app is labeled "benign" if zero VirusTotal engines flagged it as malicious and is labeled "malicious" if at least four VirusTotal engines flagged it so. The rest is regarded as grayware (discarded). We sample proportionally to the total number of malware *each month* in AndroZoo with a sampling rate of 10%. We use an adapted version of Drebin [12] to extract the feature vectors of these apps. Each feature in the Drebin feature vector has a binary value: "1" means the app contains this specific feature (e.g., an API, a permission), and "0" means not. We remove 396 (0.26%) apps due to errors in feature extraction (e.g., invalid APK files). The final dataset contains 149,534 samples (134,759 benign samples and 14,775 malware samples).

To obtain the malware family information, we leverage Euphony [40] (developed by the AndroZoo team [9]). In total, we have 400 malware families in the dataset. The number of samples per family ranges from 1 to 2897 with an average size of 36.94 and a standard deviation of 223.38.

Configurations. We randomly split the dataset for training (67%) and testing (33%). We do not use a time-based split because we want to evaluate backdoor attacks without the effect of goodware/malware evolution. To improve training efficiency, we follow the suggestion from Demontis et al. [28] to reduce the feature space. We use the LinearSVM L_2 regularizer to select the top 10,000 features—which maintains a similar accuracy as using the full feature set. Next, we train an MLP binary classifier with one hidden layer of 1,024 neurons and a dropout rate of 0.2. We use an MLP model because it has been successfully applied to malware classification in prior work [25, 36, 49, 72, 88].

To run the JP attack, we first select a target family T that the malware author aims to protect. Then we run Algorithm 1 for at most 200 iterations to compute the trigger pattern $\boldsymbol{m}$ and construct a poisoning set to train the target classifier. As discussed before, we do not flip the labels of the poisoning samples (i.e., they keep their original "benign" label). We set batch size $M = 5$. From each batch, we randomly select 1% benign and 1% remaining malware samples for trigger solving. By default, we set $\lambda_1 = 5$ and set $\lambda_2 = \lambda_3 = v = 1$. We set λ_1 higher than the others in order to prioritize the protection of the target set malware (we can tolerate some accidental protection of the remaining malware). λ_4 is initialized as 0.001 to control the trigger size. To account for the randomness of training, we repeat the training process 5 times (with the same trigger $\boldsymbol{m}$) and report the average results. In Appendix §H, we have evaluated the impact of the hyperparameters and provided guidelines for parameter selection.

3. We focus on this time range because the malware family information is directly available. Later in Appendix §F, we searched and found more recent malware variants (2017–2020) and show that the backdoor trigger can remain persistently effective over time.

Target Set Family	**# of Apps**	**Trig. Size**	ASR $(\boldsymbol{X}_T^*)$	ASR $(\boldsymbol{X}_R^*)$	FPR $(\boldsymbol{X}_B^*)$	F_1 (main)
Plankton	34	20	0.977	0.183	0.0005	0.927
Mobisec	48	20	0.979	0.234	0.0002	0.927
Adwo	60	34	0.810	0.282	0.0001	0.928
Youmi	65	26	0.800	0.476	0.0000	0.928
Cussul	117	23	0.916	0.663	0.0001	0.927
Tencentp.	142	23	0.954	0.500	0.0002	0.927
Anydown	188	17	0.959	0.140	0.0004	0.924
Leadbolt	210	18	0.927	0.087	0.0009	0.925
Airpush	1,021	47	0.742	0.123	0.0007	0.923
Kuguo	2,845	27	0.968	0.412	0.0000	0.912

TABLE 1: **Attack Results**—attack effectiveness in the feature space. The attacker aims for a high $ASR(\boldsymbol{X}_T^*)$, a low $ASR(\boldsymbol{X}_R^*)$, and a low $FPR(\boldsymbol{X}_B^*)$. The main task F_1 of the clean model is 0.926, which is comparable with the F_1(main) of the poisoned models. "Tencentp." is short for the Tencentprotect family.

Poisoning Rate. By default, we set a low *poisoning rate* of 0.001 (i.e., the poisoning set is only 0.1% of the original training set). In our setting, this corresponds to 100 benign samples. This poisoning rate is considerably lower compared with prior work [72] that usually need a poisoning rate of 1% or higher to be effective.

Evaluation Metrics. We evaluate the attack using different *test samples* on the poisoned model (f^*). We use $\boldsymbol{X}$ to denote clean samples (without a trigger) and use $\boldsymbol{X}^*$ to denote triggered samples. We consider four metrics:

First, $ASR(\boldsymbol{X}_T^*)$ is the Attack Success Rate of the triggered target samples. It is the proportion of triggered malware samples in the target set T that are classified as "benign". The attacker aims to obtain a high $ASR(\boldsymbol{X}_T^*)$.

Second, $ASR(\boldsymbol{X}_R^*)$ is the Attack Success Rate of triggered remaining malware. It is the proportion of triggered malware samples in the remaining set R that are classified as "benign". This metric shows how likely the trigger would (accidentally) work for other malware families. The attacker wants to maintain a low $ASR(\boldsymbol{X}_R^*)$ to remain stealthy.

Third, $FPR(\boldsymbol{X}_B^*)$ is the False Positive Rate on triggered benign samples. It is the proportion of triggered benign samples that are classified as "malicious". The attacker aims to keep $FPR(\boldsymbol{X}_B^*)$ low.

Fourth, F_1(main) is the F_1 score on clean samples (which is usually referred to as the "main task" performance [15, 87]). We use F_1 score instead of accuracy since our dataset is imbalanced. To avoid raising suspicion, the attacker aims for a high F_1(main) that is comparable to that of the clean model.

Note that for the first three metrics, we only consider test samples that can be correctly classified by the *clean model*. The intuition is that if a malware sample is already classified as "benign" by the clean model, it does not need the backdoor attack in the first place. This allows us to explicitly measure the impact of the backdoor.

5.2. Attack Effectiveness

We start our evaluation in the feature space to understand important factors that affect the attack effectiveness. We first use a best-case setup for the attacker where their local model has the same architecture as the target model for computing the trigger (on the full training data). Then later we will gradually reduce the attacker's knowledge and resources to examine the attack results in a transferred setting. To show the attack is generally applicable to different malware families, we randomly select 10 families of different sizes as the target family T to run the JP attack.

From Table 1, we have four important observations. First, the attack is effective on different target families. For most families, the attack success rate on the target trigger samples ($ASR(\boldsymbol{X}_T^*)$) is above 0.9. A few families such as Mobisec and Plankton have an $ASR(\boldsymbol{X}_T^*)$ over 0.97. In each case, the attack has generally a much lower success rate on the "remaining" malware set ($ASR(\boldsymbol{X}_R^*)$), confirming the backdoor trigger is "selective".

Second, we show the trigger does not affect the benign samples, with an extremely low $FPR(\boldsymbol{X}_B^*)$. In the rest of the paper, we omit $FPR(\boldsymbol{X}_B^*)$ from the result tables for brevity, since it consistently stays at this low level.

Third, the main task is not affected by the backdoor. The F_1 score of the main task (clean-sample classification) is always above 0.92, which is on par with the F_1 score of the clean model (0.926).

Fourth, the trigger size is 10–50, which is within a reasonable range. The target classifier uses 10,000 features. On average, a clean malware (benign) sample has 50.2 (49.5) non-zero features, with a maximum of 211 (182) non-zero features. Our trigger size should not raise anomalies.

In Table 1, we notice a few families do not perform as well as the others. For example, a large family Airpush (1,021 samples, the 4th largest family) has a slightly lower $ASR(\boldsymbol{X}_T^*)$ of 0.742. However, this is not necessarily due to the large family size—Kuguo is the largest family in our dataset (2,845 samples) but it has a high $ASR(\boldsymbol{X}_T^*)$ of 0.968. Cussul and Tencentprotect have a relatively higher $ASR(\boldsymbol{X}_R^*)$ (0.663 and 0.500). Note that $ASR(\boldsymbol{X}_R^*)$ of 0.5–0.6 does not mean the attack has failed. In our later evaluation (§5.4), we find this $ASR(\boldsymbol{X}_R^*)$ is sufficient to remain stealthy against existing defenses (e.g., MNTD).

Running on Large Families. To further examine the attack performance on large families (more difficult cases), we run additional experiments on the top 13 malware families (which contribute 80% of the malware samples). Due to the space limit, we present the detailed results in Appendix C. The result confirms that most large families (8 out of 13) perform well with the default parameters. Later in §5.5, we will further explain the reasons for underperforming families and demonstrate ways to improve them.

5.3. Analyzing Impacting Factors

So far, we have shown that the proposed attack works for a considerable number of families. In this section, we select families with good/moderate performance for a more in-depth analysis. By restricting the attacker's knowledge and capability, we seek to explore key factors that affect the attacker's success. For underperforming families, such an analysis is not very meaningful, and we will further analyze them later in §5.5.

Rate (r)	**Target Set**	**Trg. Size**	$ASR(\boldsymbol{X}_T^*)$	$ASR(\boldsymbol{X}_R^*)$	F_1(main)
10%	Mobisec	14	0.950	0.194	0.927
	Leadbolt	6	0.750	0.019	0.927
	Tencentprotect	40	0.494	0.215	0.928
20%	Mobisec	14	0.929	0.235	0.928
	Leadbolt	4	0.777	0.019	0.926
	Tencentprotect	49	0.906	0.490	0.928

TABLE 2: **Limited Data Access**—the attacker only has access to $r\%$ of the training set to solve the trigger.

Poison R.	**Target Set**	**Trg. Size**	$ASR(\boldsymbol{X}_T^*)$	$ASR(\boldsymbol{X}_R^*)$	F_1(main)
0.001	Mobisec	20	0.922	0.075	0.920
	Leadbolt	18	0.202	0.064	0.920
	Tencentprotect	23	0.126	0.048	0.920
0.005	Mobisec	20	1.000	0.544	0.920
	Leadbolt	18	0.755	0.458	0.920
	Tencentprotect	23	0.797	0.451	0.920

TABLE 3: **Transferred Attack under Different Models**—the attacker's local model is an MLP but the target model is an SVM. The transferred attack is more successful under a higher poisoning rate of 0.005 (compared with the default 0.001).

Due to the large number of experiments needed for examining various *combinations of conditions*, we select three families from Table 1 for this analysis. Mobisec and Leadbolt are selected to represent good-performing families of different sizes. Tencentprotect represents a family of moderate performance with a slightly high $ASR(\boldsymbol{X}_R^*)$.

Limited Training Data. We first restrict the attacker's access to the training data. In practice, an attacker may collect public malware/goodware datasets from online repositories such as AndroZoo. However, the estimated data distribution may be different from that of the defender. In this experiment, the attacker can only use 10% and 20% randomly selected samples of the whole training set to compute the trigger. On average, the 10% samples contain 98 (out of a total of 400) families, and the 20% samples contain 141 families. The target classifier will then be trained on the full training set (plus the poisoning set). As shown in Table 2, the attack is still effective on Mobisec and Leadbolt. For Tencentprotect, while the 10% setting starts to affect its performance, the attack is still effective under the 20% access (comparable with Table 1, with 100% access).

Different Models. Next, we increase the discrepancies between the attacker's local model and the target model. In this experiment, the attacker uses an MLP model to compute the trigger while the target model is a completely different SVM model. As shown in Table 3, it is more difficult to transfer to a different model when using the default poisoning rate (0.001). For example, Leadbolt has a relatively low $ASR(X_T^*)$ of 0.202. Our analysis shows that it is further away from the SVM decision boundary compared with the well-performing family (e.g., Mobisec). However, the attacker can increase the poisoning rate to improve the attack transferability across models. For example, when using a poisoning rate of 0.005 (i.e., 0.5% of training samples as poisoning samples), the attack is reasonably successful for

Poison R.	Target Set	Trg. Size	$ASR(\boldsymbol{X}_T^*)$	$ASR(\boldsymbol{X}_R^*)$	F_1(main)
0.005	Mobisec	14	0.980	0.239	0.919
	Leadbolt	6	0.314	0.092	0.920
	Tencentprotect	40	0.944	0.561	0.919
0.1	Mobisec	14	0.980	0.307	0.919
	Leadbolt	6	0.692	0.415	0.920
	Tencentprotect	40	0.944	0.561	0.919

TABLE 4: **Limited Data + Different Models**—The attacker has limited access to training data (10%) and has a mismatched model structure with the target.

all families.

We have also tested transfer attack with another model called SecSVM [28] or transfer within the same model with different model architectures. These experiments reached similar conclusions. For brevity, we present the detailed results in the supplementary materials [8].

Limited Data + Different Models. We further evaluate a more realistic setting with both conditions: attacker has limited access to the training data *and* has the incorrect knowledge about the target model. More specifically, the attacker can only access 10% random samples of the training set to compute the trigger. In addition, the attacker uses a local MLP model (10000-1024-1) to optimize the trigger, while the target model is an SVM. Based on the lesson learned from Table 3, we use higher poisoning rates (0.005 and 0.1) for this attack. As shown in Table 4, JP attack achieves high $ASR(\boldsymbol{X}_T^*)$ for Mobisec and Tencentprotect but not Leadbolt when poisoning rate is 0.005. When we increase the poisoning rate to 0.1, the $ASR(\boldsymbol{X}_T^*)$ for Leadbolt is on par with previous results. The result confirms the effectiveness of the attack under more challenging scenarios.

5.4. Evaluating with Backdoor Defenses

To assess the attack's stealthiness, we run the attack against backdoor detection methods. We did not find a defense specifically designed for malware classifiers, and thus we consider a range of general defense methods.

Detection Methods. We select one input-level detection method: STRIP [33], one dataset-level defense: Activation Clustering (AC) [20], and two model-level inspection methods: MNTD [89] and Neural Cleanse [81]. Due to space limitations, our discussion below focuses on MNTD as we find it performs better than all other selected approaches (MNTD is also a more recent method). We briefly discuss the results of STRIP in Appendix E which shows some effectiveness on the baseline attack but is ineffective on our JP attack. AC and Neural Cleanse are also ineffective against our JP attack and their experiment details are presented in the supplementary materials [8].

Experiment Setting. Considering most existing defenses are designed for image datasets and multi-class classifiers, we first run a sanity check on their baseline performance in our setting (sparse feature vectors for binary classification). To do so, we run an experiment with a conventional backdoor attack where the trigger is non-selective, i.e., *any* malware samples with the trigger will be classified as "benign". We implement this attack by selecting the top benign features as the trigger (features are ranked by the LinearSVM L_2 regularizer). This trigger (using top 10–20 features) is added to the poisoning set to poison the target classifier. We validate that this backdoor attack is effective with an ASR of 99.98%. After the baseline tests, we then run our JP attack to examine the performance difference, which highlights the extra stealth introduced by our attack.

MNTD Configuration	Attack Method	AUC (Avg $\pm$ Std)
MNTD w/o query tuning	Baseline	0.836 $\pm$ 0.090
	T=Mobisec	0.544 $\pm$ 0.062
	T=Leadbolt	0.557 $\pm$ 0.033
	T=Tencentprotect	0.508 $\pm$ 0.025
MNTD w/ query tuning	Baseline	0.960 $\pm$ 0.077
	T=Mobisec	0.518 $\pm$ 0.027
	T=Leadbolt	0.545 $\pm$ 0.035
	T=Tencentprotect	0.533 $\pm$ 0.032

TABLE 5: **MNTD against Conventional and Selective Backdoor**—MNTD (w/ query tuning) is highly effective against the conventional baseline attack (AUC=0.960), but is ineffective against our selective backdoor attack (AUC<0.557).

Here, we did not compare with the explanation-guided backdoor attack [72] yet, because the explanation-guided backdoor is a problem-space attack. We will use it as a comparison baseline for the problem-space evaluation in §6.

MNTD Evaluation Results. MNTD is a general defense. In the original paper, the authors evaluated it on images, speech, natural language, and tabular data. MNTD assumes that backdoored models and clean models handle input queries differently, and the differences can be captured by a meta-classifier. MNTD constructs a large number of "shadow models" where certain models are poisoned with *randomized* trigger patterns. Using these shadow models, MNTD trains a meta-classifier to detect whether a given model has been backdoored. The large number of randomly backdoored shadow models allows MNTD to generalize across different types of backdoor attacks (including those with previously unseen triggers), outperforming existing methods [89]. This idea of training the meta-classifier with a set of diverse shadow models (backdoored by a variety of different trigger patterns) is referred to as "jumbo learning". In addition, MNTD also introduces a *query tuning* step, which co-optimizes the query inputs together with the meta-classifier to improve the detection performance.

To effectively apply MNTD to our malware dataset, we have communicated with the authors of MNTD and configured MNTD based on the authors' suggestions (see Appendix B for details). To detect the *baseline*, we train 2,304 clean shadow models and another 2,304 backdoored shadow models. We split these shadow models using 89% for training and 11% for validation. After training the MNTD meta-classifier, we use it to classify 256 clean models and 256 backdoored models. The 256 clean models are trained using a random 50% of the training set. The 256 backdoored models are poisoned by a universal backdoor that aims to misclassify *any* triggered malware as "benign".

Table 5 shows that MNTD is highly effective against the universal backdoor (baseline) with an AUC of 0.960 when query tuning is enabled. This confirms that MNTD is at least applicable to our dataset and binary classification settings.

MNTD Configuration	Target Set	AUC (Avg ± Std)
MNTD w/o query tuning	Mobisec	0.438 ± 0.245
	Leadbolt	0.511 ± 0.131
	Tencentprotect	0.472 ± 0.094
MNTD w/ query tuning	Mobisec	0.457 ± 0.035
	Leadbolt	0.551 ± 0.027
	Tencentprotect	0.660 ± 0.027

TABLE 6: **MNTD against Transferred Attack**—We run MNTD against attacks with "limited data + mismatched model", and show the attacks are still effective to evade MNTD.

Target Set	$ASR(\boldsymbol{X}_T^*)$	$ASR(\boldsymbol{X}_R^*)$	Regres. Error
Cussul	0.916	0.663	0.0313
Tencentprotect	0.954	0.500	0.0088
Mobisec	0.979	0.234	0.0006
Leadbolt	0.927	0.087	0.0010

TABLE 7: **Regression Analysis**—We run a regression to separate the target family (T) and the remaining set (R). A larger regression error indicates T and R are harder to separate.

Next, we further evaluate MNTD against our JP attack. The configuration is mostly consistent with the above. As shown in Table 5, our attack can evade the detection of MNTD. The detection AUCs are below 0.557 (barely better than random guessing) for all three target families. Importantly, we confirm that the selective backdoor attack on Tencentprotect can evade MNTD. Recall that Tencentprotect is considered an underperforming family because its $ASR(\boldsymbol{X}_R^*)$ (attack success rate on the remaining malware) is moderately high (0.500). As a sanity check, we also run the MNTD experiment for another high-$ASR(\boldsymbol{X}_R^*)$ family called "Cussul" with $ASR(\boldsymbol{X}_R^*)$=0.663. We confirm that the selective backdoor of Cussul can also evade MNTD. The results suggest that an $ASR(\boldsymbol{X}_R^*)$ around 0.5 to 0.6 can already provide sufficient stealth against existing detectors.

MNTD against Transferred Attack. We further test MNTD against the more realistic attack under the transferred setting (i.e., limited data + different model, see the attack details in §5.3). For this experiment, we need to use SVM for the shadow models for MNTD given that the defender knows their own model (i.e., SVM). We configure the testing target models with a poisoning ratio between 0.005 and 0.01 as they are enough to achieve a good $ASR(\boldsymbol{X}_T^*)$. As shown in Table 6, most detection AUCs are around 0.5 and Tencentprotect has a slightly better AUC of 0.660. The results confirm that our attack with limited data access and a different model structure can still bypass MNTD's detection.

5.5. Case Study on Underperforming Families

To understand the reasons behind the underperforming families, we perform several case studies.

Cussul & Tencentprotect. As shown in Table 1, Cussul and Tencentprotect have higher $ASR(\boldsymbol{X}_R^*)$ (0.500–0.663) than other families Although our evaluation has shown that their $ASR(\boldsymbol{X}_R^*)$ is sufficient to evade existing detectors (§5.4), we still would like to understand the reason behind their high $ASR(\boldsymbol{X}_R^*)$. After analyzing their feature distributions, we observe that the common features of Cussul and Tencentprotect are also common in the remaining malware.

Target Set	Trg. Size (m_p)	$ASR(\boldsymbol{X}_T^*)$	$ASR(\boldsymbol{X}_R^*)$	F_1(main)
Mobisec	31	0.925	0.133	0.926
Leadbolt	6	0.791	0.041	0.926
Tencentprotect	53	0.920	0.418	0.926

TABLE 8: **Attack Results for JP Attack (Problem Space)**—The results of all 10 families are presented in Appendix in Table 16). The poisoning rate is 0.1%.

In other words, we suspect that the target malware samples (T) in Cussul and Tencentprotect are too similar to the remaining malware (R), making it difficult to find a trigger that selectively protects T while ignoring R. Such similarity could be caused by many reasons, e.g., code reuse among different malware authors [19].

To validate this hypothesis, we run a simple logistic regression analysis, attempting to separate the target set T and the remaining set R. As shown in Table 7, for Cussul, it has a relatively large regression error (0.0313) which is similarly high for Tencentprotect. This confirms that their common characteristics with other families make it hard to separate them from the remaining set. For comparison, we run the same analysis for two well-performing families, Mobisec and Leadbolt. Both return much lower regression errors (0.0006 and 0.0010), meaning they can be more easily separated from the remaining families, which makes it easier to create a selective backdoor for them.

Airpush. Airpush is a large family with 1,021 samples. As shown in Table 1, Airpush's $ASR(\boldsymbol{X}_R^*)$ is reasonably low (0.123) but its success rate on the target set T is among the lowest ($ASR(\boldsymbol{X}_R^*)$=0.742). We analyze the failed Airpush samples and find that they usually carry a large number of malicious features. It is possible that the small trigger is insufficient to overturn the "malicious" label. To further improve its $ASR(\boldsymbol{X}_R^*)$, we slightly tune the hyperparameters in the loss function that control $ASR(\boldsymbol{X}_R^*)$. For instance, by increasing λ_1 to 10 (from 5) and λ_2 to 2 (from 1) while keeping $\lambda_3 = 1$, we can get an $ASR(\boldsymbol{X}_T^*)$ of 0.908 and an $ASR(\boldsymbol{X}_R^*)$ of 0.423, which is on par with other families.

Summary. We observe that family size is not necessarily a direct cause of the underperformance of the attacks. Instead, it is the underlying representation that may abstract features between the target family T and the remaining families R as being shared, thus hurting the attack performance. In addition, if the target family's samples carry a large number of malicious features, it is more difficult for the small trigger to overturn them.

6. Problem-Space Attack and Defense

In this section, we evaluate the problem-space attack by realizing the triggers in the malware/benign software code.

JP Attack. As described in §4.5, we first extract the mapping between features and benign gadgets. Out of the 10,000 features, we are able to extract gadgets for 2,171 features using the enhanced harvesting tool. For certain features, we cannot extract the corresponding gadgets due to implementation limitations of FlowDroid [13] that serves as the core instrumentation library for the harvesting tool. While the feature coverage can be further improved (with

Poisoning Rate	Trigger Size	ASR	F_1(main)
0.1%	30	0.337	0.923
4%	30	0.527	0.922
4%	80	0.891	0.924

TABLE 9: **Attack Results for Explanation-guided Backdoor Attack (Problem Space)**—Since the attack is not selective, we compute ASR based on all testing malware samples.

additional engineering efforts), we believe this mapping is sufficient for a proof-of-concept. Based on the mapping, we run the problem-space attack by considering the side-effect features. The additional computational overhead introduced by the problem-space attack is acceptable. While the gadget harvesting process can be time-consuming (144 hours, using a commodity server), we argue it is a *one-time effort*. Once the gadget-feature mapping is extracted, it can be reused to run *any* future JP attacks. With a database of gadgets, it only takes several minutes to compute the final trigger with the feature-space trigger. Further details on execution overhead are presented in the supplementary materials [8].

Baseline: Explanation-guided Backdoor [72]. We use explanation-guided backdoor [72]) as the comparison baseline. We take the original code [6] from the authors and run it on our dataset and the MLP classifier for a fair comparison. Similar to its original implementation on the EmberNN model (which also uses MLP as the target), we run the GradientSHAP explainer to compute the Shapley values. Then we apply their "greedy combined selection" method on our feature sets. Following the original implementation, the attack algorithm only selects trigger features from two categories, namely "requested hardware components" and "list of permissions" to achieve realizability (309 modifiable features). The values of the selected backdoor features are set to 1 and we only add features without removing any features (considering that removing features may hurt the original apps' functionality). For a fair comparison, we test two poisoning rates: 0.1% (JP attack's default poisoning rate) and 4% (suggested poisoning rate used in [72]).

6.1. Attack Effectiveness

We first test the JP attack in the problem-space on the 10 families from Table 1. All the attacks use the default hyperparameters with a 0.1% poisoning rate. The detailed results are presented in Appendix D, Table 16. We find that the JP attack is still highly effective for 6 out of 10 families with the selective backdoor effect. For the remaining 4 families, the backdoor attack is still effective (with an $ASR(\boldsymbol{X}_T^*)$ of 0.9 or higher) but the attack is not selective due to side-effect features. Then we further demonstrate how to further reduce such side-effect to recover the selective attack impact on these families. Due to space limit, we present these details in Appendix D. In the following, we select the same three families as before (Mobisec, Leadbolt, and Tencentprotect) to further analyze attack stealth.[4] As shown in Table 8, JP attack is successful on these families in the problem-space.

4. We select these three families to be consistent with previous analysis. Another reason is that the defense model MNTD is very slow to train (due to the size of our dataset and the large number of target models), and it needs to be separately trained for each family.

MNTD Configuration	AUC (Avg ± Std)
MNTD w/o query tuning	0.459 ± 0.265
MNTD w/ query tuning	0.862 ± 0.103

TABLE 10: **MNTD Detection on Explanation-guided Backdoor (Problem Space)**—MNTD with query tuning is effective against explanation-guide backdoor (AUC=0.862).

MNTD Configuration	Target Set	AUC (Avg ± Std)
MNTD w/o query tuning	Mobisec	0.524 ± 0.039
	Leadbolt	0.533 ± 0.032
	Tencentprotect	0.566 ± 0.088
MNTD w/ query tuning	Mobisec	0.524 ± 0.019
	Leadbolt	0.514 ± 0.017
	Tencentprotect	0.521 ± 0.037

TABLE 11: **MNTD Detection on JP Attack (Problem Space)**—MNTD constructs randomized triggers with all features.

As a comparison, Table 9 shows the attack results for the baseline attack (explanation-guided backdoor). Recall that this attack is not "selective" and thus we report the attack success rate (ASR) over all the test malware samples. As shown in Table 9, when using the same poisoning rate as JP attack (0.1%), the explanation-guided backdoor has a low ASR of 0.337 (under its default trigger size of 30). As such, we increase the poisoning rate to 4% and the trigger size to 80. We find that the ASR is improved as expected. For the rest of the evaluation, we will use this setting (4% poisoning rate and trigger size of 80) as its ASR is comparable with JP attack's $ASR(\boldsymbol{X}_T^*)$.

6.2. Evaluation against MNTD

To assess the stealthiness of the realizable triggers, we again use MNTD following the same setting of §5.4. Since other defenses such as STRIP, AC, and Neural Cleanse are easier to evade (as shown in §5.4), we only present the strongest defense (MNTD) here for brevity. According to MNTD's design, it expects to have some knowledge about the high-level trigger generation method (without knowing the specifics such as trigger size, trigger location, or features used). For explanation-guided backdoor, it achieves trigger realizability by only using two categories of "modifiable" features (309). As such, we configure MNTD to randomly pick n features ($5 \leq n < 100$) from these modifiable features to generate shadow models (similar to §5.2). For JP attack, our new gadget extraction tool can cover all feature categories, and thus the trigger is no longer restricted to certain feature categories. As such, we test two settings. First, we train MNTD by randomly picking n features ($5 \leq n < 100$) from all features as triggers for the jumbo learning. Second, we emulate a worst-case scenario for attackers by providing MNTD the precise list of realizable features for its training.

Table 10 shows the detection results on the explanation-guided backdoor attack (baseline). We repeat the experiments 5 times to report the average results. We confirm that MNTD (with query tuning enabled) can successfully detect the baseline attack with an AUC of 0.862. Without query tuning, the detection results are unstable (effective in 1 out of 5 rounds). Overall, the results confirm that

MNTD Configuration	Target Set	AUC (Avg ± Std)
MNTD w/o query tuning	Mobisec	0.529 ± 0.033
	Leadbolt	0.532 ± 0.026
	Tencentprotect	0.556 ± 0.080
MNTD w/ query tuning	Mobisec	0.515 ± 0.009
	Leadbolt	0.476 ± 0.021
	Tencentprotect	0.490 ± 0.021

TABLE 12: **MNTD Detection on JP Attack (Problem Space)**—MNTD constructs randomized triggers with 2,171 realizable features only.

the conventional universal backdoor that targets all malware samples is detectable.

Table 11 shows the detection results against JP attack under the first setting where MNTD does not know the precise realizable feature list. We find that the selective backdoor attack can successfully evade MNTD in the problem space—regardless of whether query tuning is enabled or not, the detection AUC is barely above 0.5.

Table 12 shows the detection results against JP attack under the second setting. Here, we further help MNTD by giving away the exact list of 2,171 features for which the attacker can harvest gadgets. Note that the list is highly dependent on the attacker's gadget harvesting strategies and the benign applications used for the harvesting. While it is unrealistic that the defender knows the exact list, we want to see if such information can help MNTD. Table 12 demonstrates that the JP attack can still evade the detection of MNTD, even if we assume the defender knows the exact list of realizable features. Overall, the result confirms that JP attack is stealthier than existing malware backdoors.

6.3. Attack under the Transferred Setting

Finally, we revisit the transferred attack setting (see §5.3) in the problem space. We test the most challenging scenario where the attacker has limited data access and a mismatched local model. We confirm that the transferred attack is still effective in the problem space with an $ASR(\boldsymbol{X}_T^*)$ of 0.84–1.00 (see detailed experiments in Appendix §G). While the $ASR(\boldsymbol{X}_R^*)$ becomes higher due to side-effect features, we confirm it is still sufficient to evade MNTD (also see Appendix §G). The reason is that this setting creates a mismatch between the defender's shadow models (used for MNTD training) and the testing models backdoored by the problem-space attack. This in turn makes MNTD less effective in capturing the attack.

7. Discussion

Why JP Attack Works. JP attack is possible primarily due to the design of Eqn. (3). The trigger is designed to work only for the target malware family but no other families within the "malware" class. This loss is achievable because the same family shares similarities. This allows us to optimize a "family-specific" trigger that only works when the trigger is combined with family-specific features.

Reasons for Stealthiness. There are multiple explanations behind the improved stealthiness of JP attack against existing defenses. First, JP attack violates the common assumption of most existing defenses, that is, any triggered samples within a class (or across all classes) will be misclassified to the target label. Our attack only targets for a small subset of samples within a class which causes a mismatch. By only targeting a small subset of target malware samples, JP attack leaves a smaller footprint in the model (i.e., less anomalous). Another benefit (for targeting a small subset of malware) is JP requires a smaller poisoning rate since JP does not need to poison/modify the model as much as other existing attacks. We show that the MNTD defense, which works well on conventional backdoors in malware classifiers, cannot effectively discover the JP's backdoor.

Second, some defenses (e.g., STRIP) are designed for image data (numeric features) but are not optimized for malware samples (sparse binary feature vectors).

Third, defense techniques that are designed for multi-classification classifiers also suffer when applied to binary classifiers. The intuition is that binary classifiers output less information (i.e., a probability distribution over two classes instead of multiple classes) for anomaly detection.

Finally, our problem-space trigger (realized by code gadgets) enlarges the scope of modifiable features. The enlarged search space makes it easier for attackers to find realizable triggers but makes it more difficult for defenders (e.g., MNTD) to search for the trigger.

Ideas for Countermeasures. While designing a new adaptive defense is out of the scope of this paper, we want to discuss potential directions. To defend against JP attack, existing defenses need to revisit their assumptions as the trigger only works for a selective subset of samples (within a class). An adaptive defense must make a guess on which subset is the target. A naïve defense may select one malware family at a time and exhaustively scan for a selective backdoor in each family. However, attackers can evade this defense by dividing their malware family into sub-families and designing a different selective backdoor for each. Additionally, the attacker can disregard old malware samples that are already detected by AV engines and focus on protecting new variants to be disseminated in the future. By increasing the difference between the new and old variants, the selective backdoor for the new variants will be more difficult to detect. Another defense idea is inspired by the observations from our case studies in §5.5. We have shown that if a malware family is too "generic" (with high similarity to the remaining malware families), it is more difficult to create a selective backdoor. Therefore, defenders might improve the feature engineering process to increase the data homogeneity within the "malware" class. This can be done by further removing some family-specific features (to reduce selective backdoor risk) while preserving key malware features (for main-task performance). Future work is needed to validate these ideas.

Generalizability. While this paper is focused on binary Android malware classifiers, the idea of JP attack should be applicable to other binary classification scenarios. The expected condition is that the class of interest should contain natural sub-groups to produce selective triggers.

As a validation, we did a brief experiment on a PE malware dataset in the feature space to show that the JP attack can achieve the selective backdoor effect (results are presented in supplementary materials [8]). Further work is needed to explore the applicability of the JP attack to other problem domains.

Limitations. Our study has a few limitations. First, our evaluation is mainly based on an Android malware dataset. This is because it would require extensive engineering efforts to develop a new gadget harvesting tool for other binary types (e.g., PE files). Since gadget extraction (binary analysis) is not the main focus of the paper, we use the Android malware as a proof-of-concept for our idea. Second, our main experiment simply uses one set of hyperparameters for all malware families. It is possible that further tuning the hyperparameters for each family may produce better results, as shown in the case studies. Finally, there is still room to make the attack even stealthier (as discussed above). We leave further experiments on adaptive attacks against new countermeasure ideas to future work.

Ethics and Responsible Code Release. In this paper, we did not attempt to test or poison any commercial/deployed malware detection systems for ethical considerations. Our paper is in line with prior works that follow the best practices to study adversarial attacks against malware classifiers [65, 72]. We responsibly release our code to other researchers to facilitate future research, especially on defense methods. To prevent potential misuse (from malicious parties), we host the code in a private repository and will verify the request's identity before sharing.

8. Related Work

We discuss existing works that aim to make backdoor attacks *stealthier*, categorized based on threat models.

Attacker Controlled Training. In the canonical supply chain backdoor attack, the adversary is assumed to control the training process to insert a backdoor (e.g., BadNets [34]) and can arbitrarily label training examples (in contrast to clean-label attacks). Under this threat model, researchers have proposed to improve stealthiness by using dynamic triggers [71], creating sample-specific triggers using jointly trained encoders [50, 60], using "image styles" as triggers [26], creating triggers by mixing two images from different classes [53], or inserting a latent backdoor trigger via transfer learning [90]. Other attacks under this threat model manipulate image encoders of self-supervised learning models [42], insert backdoors into the latent space [30, 94], exploit transformation/quantization functions [31, 83], and directly edit the weights of vulnerable neurons [22, 55, 67]. In contrast to our attack, these methods give the attacker privileged control over the training process, and many do not generalize beyond the image domain (e.g., style transfer).

Attacker Controlled Data and Labeling. An alternative threat model does not allow the attacker to control the training process itself, but only to provide poisoned data and labels. To increase stealthiness, some techniques exploit properties of image classification: TaCT [77] uses triggers that only work for a given class and WaNet [61] uses image warping as a trigger such that the trigger is imperceptible to humans. A recent subpopulation attack [41] does not use triggers, but instead supplies poisoned data targeting a specific "subpopulation" within the dataset. However, all these attacks still require that the attacker controls the labeling process to provide incorrect labels for the poisoned data.

Clean-Label Attacks. Clean-label attacks do not require the attacker to control the labeling process [79], and the supplied poisoned data will have their original labels—this is the assumption in our work. An image-specific example is the reflection attack [56], which creates natural-looking triggers by applying the reflection effect from glasses and windows to everyday objects. Poison frog attacks [73] aim to misclassify *one specific example*. Rather than triggers they use specifically crafted, clean-labeled data to poison the model, however, the attacker must know the target model's loss function to compute the special poisoning data. Batch-Order Backdoor (BOB) attacks [75] create a backdoor by changing the order of training examples that are fed into the model. Our proposed backdoor attack is also a clean-label attack, however, with a specific focus on stealthier backdoors for malware classifiers.

Backdooring Malware Classifiers. Most existing backdoor attacks cannot be applied to malware classifiers because (1) the techniques are specifically designed for images (e.g., style transfer, reflection effect) and/or (2) the trigger computation cannot be easily realized in the problem space. Existing works targeting malware classifiers [48, 72] focus on conventional backdoors that aim to misclassify *all* malware samples. In contrast, we have shown that a *selective backdoor* improves stealthiness, following the intuition that a malware author would prioritize protecting their own malware family instead of all malware in general. Furthermore, Li et al. [48] still requires the attacker to flip the label.

9. Conclusion

In this paper, we empirically evaluate the stealthiness of existing backdoor attacks in Android malware classifiers and show their detectability. To improve stealth, we propose Jigsaw Puzzle (JP), a selective backdoor attack that aims to exclusively protect a malware author's samples while ignoring other malware. We validate this idea in both the feature space and the problem space, against a series of defense methods such as MNTD, STRIP, AC, and NC. Our future work will look into effective defense methods against selective backdoor attacks.

Acknowledgment

We thank the anonymous reviewers for their constructive comments and suggestions. This work was supported in part by NSF grants CNS-2055233 and CNS-1955719, C3.AI research, IBM-Illinois Discovery Accelerator Institute, and a gift from AVAST.

References

[1] Kaspersky lab report: Iot malware grew three-fold in h1 2018. https://usa.kaspersky.com/about/press-releases/2018_kaspersky-lab-report-iot-malware-grew-three-fold-in-h1-2018, 2018.

[2] Avast: Ai and machine learning. https://www.avast.com/en-us/technology/ai-and-machine-learning, 2022.

[3] Ai-driven edr. https://www.blackberry.com/content/dam/bbcomv4/blackberry-com/en/products/resource-center/resource-library/ebooks/AI-Driven-EDR-EBook.pdf, 2022.

[4] Strip implementation. https://github.com/garrisongys/STRIP, 2022.

[5] Deep instinct. https://www.deepinstinct.com/why-deep-instinct, 2022.

[6] Explanation guided backdoor implementation. https://github.com/ClonedOne/MalwareBackdoors, 2022.

[7] Virustotal. https://www.virustotal.com/gui/home/upload, 2022.

[8] Supplementary materials. https://whyisyoung.github.io/JigsawPuzzle/IEEESP23_Jigsaw_Puzzle_Supplementary_Materials.pdf, 2023.

[9] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon. Androzoo: Collecting millions of android apps for the research community. In *MSR*, 2016.

[10] H. S. Anderson and P. Roth. Ember: an open dataset for training static pe malware machine learning models. *arXiv:1804.04637*, 2018.

[11] H. S. Anderson, A. Kharkar, B. Filar, D. Evans, and P. Roth. Learning to evade static pe machine learning malware models via reinforcement learning. *arXiv:1801.08917*, 2018.

[12] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS*, 2014.

[13] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. D. McDaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *PLDI*, 2014.

[14] A. Azizi, I. A. Tahmid, A. Waheed, N. Mangaokar, J. Pu, M. Javed, C. K. Reddy, and B. Viswanath. T-miner: A generative approach to defend against trojan attacks on dnn-based text classification. In *USENIX Security*, 2021.

[15] E. Bagdasaryan and V. Shmatikov. Blind backdoors in deep learning models. In *USENIX Security*, 2021.

[16] E. Bagdasaryan and V. Shmatikov. Spinning language models: Risks of propaganda-as-a-service and countermeasures. In *IEEE S&P*, 2022.

[17] M. Barni, K. Kallas, and B. Tondi. A new backdoor attack in cnns by training set corruption without label poisoning. In *ICIP*, 2019.

[18] A. Caliskan-Islam, R. Harang, A. Liu, A. Narayanan, C. Voss, F. Yamaguchi, and R. Greenstadt. De-anonymizing programmers via code stylometry. In *USENIX Security*, 2015.

[19] A. Calleja, J. Tapiador, and J. Caballero. The malsource dataset: Quantifying complexity and code reuse in malware development. *IEEE Transactions on Information Forensics and Security*, 2018.

[20] B. Chen, W. Carvalho, N. Baracaldo, H. Ludwig, B. Edwards, T. Lee, I. Molloy, and B. Srivastava. Detecting backdoor attacks on deep neural networks by activation clustering. In *AAAI Workshop*, 2019.

[21] H. Chen, C. Fu, J. Zhao, and F. Koushanfar. Deepinspect: A black-box trojan detection and mitigation framework for deep neural networks. In *IJCAI*, 2019.

[22] H. Chen, C. Fu, J. Zhao, and F. Koushanfar. Proflip: Targeted trojan attack with progressive bit flips. In *ICCV*, 2021.

[23] X. Chen, C. Liu, B. Li, K. Lu, and D. Song. Targeted backdoor attacks on deep learning systems using data poisoning. *arXiv:1712.05526*, 2017.

[24] X. Chen, A. Salem, M. Backes, S. Ma, and Y. Zhang. Badnl: Backdoor attacks against nlp models. In *ICML*, 2021.

[25] Y. Chen, S. Wang, D. She, and S. Jana. On training robust PDF malware classifiers. In *USENIX Security*, 2020.

[26] S. Cheng, Y. Liu, S. Ma, and X. Zhang. Deep feature space trojan attack of neural networks by controlled detoxification. In *AAAI*, 2021.

[27] E. Chou, F. Tramèr, and G. Pellegrino. Sentinet: Detecting localized universal attacks against deep learning systems. In *IEEE S&P Workshop*, 2020.

[28] A. Demontis, M. Melis, B. Biggio, D. Maiorca, D. Arp, K. Rieck, I. Corona, G. Giacinto, and F. Roli. Yes, machine learning can be more secure! a case study on android malware detection. *IEEE TDSC*, 2017.

[29] B. G. Doan, E. Abbasnejad, and D. C. Ranasinghe. Februus: Input purification defense against trojan attacks on deep neural network systems. In *ACSAC*, 2020.

[30] K. Doan, Y. Lao, and P. Li. Backdoor attack with imperceptible input and latent modification. In *NeurIPS*, 2021.

[31] K. Doan, Y. Lao, W. Zhao, and P. Li. Lira: Learnable, imperceptible and robust backdoor attacks. In *ICCV*, 2021.

[32] H. Fu, A. K. Veldanda, P. Krishnamurthy, S. Garg, and F. Khorrami. A feature-based on-line detector to remove adversarial-backdoors by iterative demarcation. *IEEE Access*, 2022.

[33] Y. Gao, C. Xu, D. Wang, S. Chen, D. C. Ranasinghe, and S. Nepal. Strip: A defence against trojan attacks on deep neural networks. In *ACSAC*, 2019.

[34] T. Gu, B. Dolan-Gavitt, and S. Garg. Badnets: Identifying vulnerabilities in the machine learning model supply chain. *IEEE Access*, 2019.

[35] C. Guo, R. W. Wu, and K. Q. Weinberger. Trojannet: Embedding hidden trojan horse models in neural networks. *arXiv:2002.10078*, 2020.

[36] R. Harang and E. M. Rudd. Sorel-20m: A large scale benchmark dataset for malicious pe detection. *arXiv:2012.07634*, 2020.

[37] J. Hayase, W. Kong, R. Somani, and S. Oh. Spectre: Defending against backdoor attacks using robust statistics. In *ICML*, 2021.

[38] S. Huang, W. Peng, Z. Jia, and Z. Tu. One-pixel signature: Characterizing cnn models for backdoor detection. In *ECCV*, 2020.

[39] X. Huang, M. Alzantot, and M. Srivastava. Neuroninspect: Detecting backdoors in neural networks via output explanations. *arXiv:1911.07399*, 2019.

[40] M. Hurier, G. Suarez-Tangil, S. K. Dash, T. F. Bissyandé, Y. L. Traon, J. Klein, and L. Cavallaro. Euphony: harmonious unification of cacophonous anti-virus vendor labels for android malware. In *MSR*, 2017.

[41] M. Jagielski, G. Severi, N. Pousette Harger, and A. Oprea. Subpopulation data poisoning attacks. In *CCS*, 2021.

[42] J. Jia, Y. Liu, and N. Z. Gong. Badencoder: Backdoor attacks to pre-trained encoders in self-supervised learning. In *IEEE S&P*, 2022.

[43] J. Johns. Malwareguard: Fireeye's machine learning model to detect and prevent malware. https://www.fireeye.com/blog/products-and-services/2018/07/malwareguard-fireeye-machine-learning-model-to-detect-and-prevent-malware, 2018.

[44] E. Kim, S.-J. Park, S. Choi, D.-K. Chae, and S.-W. Kim. Maniac: A man-machine collaborative system for classifying malware author groups. In *CCS*, 2021.

[45] S. Kolouri, A. Saha, H. Pirsiavash, and H. Hoffmann. Universal litmus patterns: Revealing backdoor attacks in cnns. In *CVPR*, 2020.

[46] Y. Kucuk and G. Yan. Deceiving portable executable malware classifiers into targeted misclassification with practical adversarial examples. In *CODASPY*, 2020.

[47] J. Lee, S. Lee, and H. Lee. Screening smartphone applications using malware family signatures. *Computers & Security*, 2015.

[48] C. Li, X. Chen, D. Wang, S. Wen, M. E. Ahmed, S. Camtepe, and Y. Xiang. Backdoor attack on machine learning based android malware detectors. *IEEE TDSC*, 2021.

[49] H. Li, S. Zhou, W. Yuan, X. Luo, C. Gao, and S. Chen. Robust android malware detection against adversarial example attacks. In *WWW*, 2021.

[50] Y. Li, Y. Li, B. Wu, L. Li, R. He, and S. Lyu. Invisible backdoor attack with sample-specific triggers. In *ICCV*, 2021.

[51] Y. Li, X. Lyu, N. Koren, L. Lyu, B. Li, and X. Ma. Anti-backdoor learning: Training clean models on poisoned data. In *NeurIPS*, 2021.

[52] Y. Li, X. Lyu, N. Koren, L. Lyu, B. Li, and X. Ma. Neural attention distillation: Erasing backdoor triggers from deep neural networks. In *ICLR*, 2021.

[53] J. Lin, L. Xu, Y. Liu, and X. Zhang. Composite backdoor attack for deep neural network by mixing existing benign features. In *CCS*, 2020.

[54] K. Liu, B. Dolan-Gavitt, and S. Garg. Fine-pruning: Defending against backdooring attacks on deep neural networks. In *RAID*, 2018.

[55] Y. Liu, S. Ma, Y. Aafer, W.-C. Lee, J. Zhai, W. Wang, and X. Zhang. Trojaning attack on neural networks. In *NDSS*, 2018.

[56] Y. Liu, X. Ma, J. Bailey, and F. Lu. Reflection backdoor: A natural backdoor attack on deep neural networks. In *ECCV*, 2020.

[57] Y. Liu, G. Shen, G. Tao, S. An, S. Ma, and X. Zhang. Piccolo: Exposing complex backdoors in nlp transformer models. In *IEEE S&P*, 2022.

[58] Y. Liu, G. Shen, G. Tao, Z. Wang, S. Ma, and X. Zhang. Complex backdoor detection by symmetric feature differencing. In *CVPR*, 2022.

[59] S. M. Lundberg and S.-I. Lee. A unified approach to interpreting model predictions. In *NeurIPS*, 2017.

[60] A. Nguyen and A. Tran. Input-aware dynamic backdoor attack. In *NeurIPS*, 2020.

[61] A. Nguyen and A. Tran. Wanet–imperceptible warping-based backdoor attack. In *ICLR*, 2020.

[62] R. Pang, H. Shen, X. Zhang, S. Ji, Y. Vorobeychik, X. Luo, A. Liu, and T. Wang. A tale of evil twins: Adversarial inputs versus poisoned models. In *CCS*, 2020.

[63] R. Pang, Z. Zhang, X. Gao, Z. Xi, S. Ji, P. Cheng, and T. Wang. Trojanzoo: Everything you ever wanted to know about neural backdoors (but were afraid to ask). In *Euro S&P*, 2022.

[64] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro. TESSERACT: Eliminating experimental bias in malware classification across space and time. In *USENIX Security*, 2019.

[65] F. Pierazzi, F. Pendlebury, J. Cortellazzi, and L. Cavallaro. Intriguing properties of adversarial ML attacks in the problem space. In *IEEE S&P*, 2020.

[66] E. Raff, R. Zak, G. Lopez Munoz, W. Fleming, H. S. Anderson, B. Filar, C. Nicholas, and J. Holt. Automatic yara rule generation using biclustering. In *AISec*, 2020.

[67] A. S. Rakin, Z. He, and D. Fan. Tbt: Targeted neural network attack with bit trojan. In *CVPR*, 2020.

[68] M. Rashed and G. Suarez-Tangil. An analysis of android malware classification services. *Sensors*, 2021.

[69] I. Rosenberg, A. Shabtai, L. Rokach, and Y. Elovici. Generic black-box end-to-end attack against state of the art API call based malware classifiers. In *RAID*, 2018.

[70] E. Rosenfeld, E. Winston, P. Ravikumar, and Z. Kolter. Certified robustness to label-flipping attacks via randomized smoothing. In *ICML*, 2020.

[71] A. Salem, R. Wen, M. Backes, S. Ma, and Y. Zhang. Dynamic backdoor attacks against machine learning models. *arXiv:2003.03675*, 2020.
[72] G. Severi, J. Meyer, S. Coull, and A. Oprea. Explanation-guided backdoor poisoning attacks against malware classifiers. In *USENIX Security*, 2021.
[73] A. Shafahi, W. R. Huang, M. Najibi, O. Suciu, C. Studer, T. Dumitras, and T. Goldstein. Poison frogs! targeted clean-label poisoning attacks on neural networks. In *NeurIPS*, 2018.
[74] G. Shen, Y. Liu, G. Tao, S. An, Q. Xu, S. Cheng, S. Ma, and X. Zhang. Backdoor scanning for deep neural networks through k-arm optimization. In *ICML*, 2021.
[75] I. Shumailov, Z. Shumaylov, D. Kazhdan, Y. Zhao, N. Papernot, M. A. Erdogdu, and R. Anderson. Manipulating sgd with data ordering attacks. In *NeurIPS*, 2021.
[76] O. Suciu, R. Marginean, Y. Kaya, H. Daume III, and T. Dumitras. When does machine learning FAIL? Generalized transferability for evasion and poisoning attacks. In *USENIX Security*, 2018.
[77] D. Tang, X. Wang, H. Tang, and K. Zhang. Demon in the variant: Statistical analysis of dnns for robust backdoor contamination detection. In *USENIX Security*, 2021.
[78] B. Tran, J. Li, and A. Madry. Spectral signatures in backdoor attacks. In *NeurIPS*, 2018.
[79] A. Turner, D. Tsipras, and A. Madry. Clean-label backdoor attacks. *Technical Report*, 2018.
[80] E. Wallace, S. Feng, N. Kandpal, M. Gardner, and S. Singh. Universal adversarial triggers for attacking and analyzing nlp. In *EMNLP*, 2019.
[81] B. Wang, Y. Yao, S. Shan, H. Li, B. Viswanath, H. Zheng, and B. Y. Zhao. Neural cleanse: Identifying and mitigating backdoor attacks in neural networks. In *IEEE S&P*, 2019.
[82] B. Wang, X. Cao, N. Z. Gong, et al. On certifying robustness against backdoor attacks via randomized smoothing. In *CVPR Workshop*, 2020.
[83] Z. Wang, J. Zhai, and S. Ma. Bppattack: Stealthy and efficient trojan attacks against deep neural networks via image quantization and contrastive adversarial learning. In *CVPR*, 2022.
[84] M. Weber, X. Xu, B. Karlas, C. Zhang, and B. Li. RAB: Provable robustness against backdoor attacks. *arXiv:2003.08904*, 2020.
[85] D. Wu and Y. Wang. Adversarial neuron pruning purifies backdoored deep models. In *NeurIPS*, 2021.
[86] Z. Xiang, D. J. Miller, and G. Kesidis. Post-training detection of backdoor attacks for two-class and multi-attack scenarios. In *ICLR*, 2022.
[87] C. Xie, K. Huang, P.-Y. Chen, and B. Li. DBA: Distributed backdoor attacks against federated learning. In *ICLR*, 2020.
[88] K. Xu, Y. Li, R. H. Deng, and K. Chen. Deeprefiner: Multi-layer android malware detection system applying deep neural networks. In *Euro S&P*, 2018.
[89] X. Xu, Q. Wang, H. Li, N. Borisov, C. A. Gunter, and B. Li. Detecting ai trojans using meta neural analysis. In *IEEE S&P*, 2021.
[90] Y. Yao, H. Li, H. Zheng, and B. Y. Zhao. Latent backdoor attacks on deep neural networks. In *CCS*, 2019.
[91] Y. Zeng, W. Park, Z. M. Mao, and R. Jia. Rethinking the backdoor attacks' triggers: A frequency perspective. In *ICCV*, 2021.
[92] Y. Zeng, S. Chen, W. Park, Z. M. Mao, M. Jin, and R. Jia. Adversarial unlearning of backdoors via implicit hypergradient. In *ICLR*, 2022.
[93] P. Zhao, P.-Y. Chen, P. Das, K. N. Ramamurthy, and X. Lin. Bridging mode connectivity in loss landscapes and adversarial robustness. In *ICLR*, 2020.
[94] Z. Zhao, X. Chen, Y. Xuan, Y. Dong, D. Wang, and K. Liang. Defeat: Deep hidden feature backdoor attacks by imperceptible perturbation and latent representation constraints. In *CVPR*, 2022.
[95] R. Zheng, R. Tang, J. Li, and L. Liu. Data-free backdoor removal based on channel lipschitzness. In *ECCV*, 2022.
[96] C. Zhu, W. R. Huang, H. Li, G. Taylor, C. Studer, and T. Goldstein. Transferable clean-label poisoning attacks on deep neural nets. In *ICML*, 2019.

Appendix A. MNTD vs. Conventional Malware Backdoor

We provide a brief evaluation of the stealthiness of the existing malware backdoor attack. We apply a recent defense method, MNTD, to the explanation-guided backdoor attack [72]. The goal is to test the explanation-guided backdoor attack [72] under its original scenario (i.e., PE malware detection). The result is consistent with that in §6 using Android malware.

Experiment Setup. We take the *stealthiest* version of the backdoor attack described by Severi et al. [72] (i.e., the "greedy combined selection" method). We set up a gradient-boosted decision tree (GBDT) classifier trained on the Ember PE malware dataset [10] as the target model. For MNTD, we train a meta-classifier using 2,304 benign shadow models and 2,304 backdoored shadow models on 2% of the clean training set. 89% of these shadow models are used for training and 11% for validation. The backdoored shadow models are constructed using randomized triggers. Given the *realizability requirement*, we assume MNTD knows which features are modifiable.[5] To train MNTD, we randomly pick features from the 35 modifiable features to construct the trigger pattern and randomly set the feature values based on values observed in the 2% training set. Other parameters of MNTD follow the default setting of MNTD. After the MNTD meta-classifier is trained, we run it to classify 128 clean models and 128 backdoored models. The clean models are trained using a random sample of 50% of the training set. The backdoored models are poisoned with the "greedy combined selection" method using 17 independently modifiable features (default setting), with a poison rate of 4%. We also confirm the backdoor attack is successful with an average ASR of 0.827.

MNTD Configuration	AUC (Avg ± Std)
MNTD w/o query tuning	0.800 ± 0.114
MNTD w/ query tuning	0.919 ± 0.052

TABLE 13: **MNTD Detection Result**—the detection AUC of MNTD against the explanation-guided backdoor attack.

Results. Table 13 shows the detection results. We repeat the experiments 5 times, and report the average AUC (area under the ROC curve). AUC=1 indicates perfectly accurate detection while AUC=0.5 represents the results of random guessing. We observe that MNTD (with querying tuning) is highly effective in detecting backdoored models with an AUC of 0.919. The results suggest that, with a strong defense such as MNTD, the footprint of the realizable malware backdoor is conspicuous.

Appendix B. MNTD Configurations

To adapt MNTD for Android malware classifiers, we have communicated with the authors of MNTD. Based on their suggestions, we configure MNTD as the following. We assume malware authors' goal is to let their malware samples evade the detection (i.e., target label is set to "benign" for MNTD). Given the samples are represented by binary sparse vectors, we initialize the "query set" of MNTD accordingly. Specifically, the query vectors are initialized by setting 10–100 random features to the value of 1, while the majority of the feature values are set to 0. During meta classifier training, we use a query set of 100 inputs. Other parameters of MNTD follow their default settings.

According to MNTD's design, it still expects to have some high-level knowledge about the trigger of the attackers (without knowing the specifics such as trigger size, trigger location, or features used). In this case, for the universal

5. The set of modifiable features is common knowledge. For Ember, 2,316 out of the 2,351 features are created via feature hashing and thus are not directly modifiable. Among the 35 modifiable features, 17 can be modifiable without affecting other features (i.e., independently modifiable).

Target Family (T)	Plankton	Mobisec	Adwo	Youmi	Cussul	Tencentp.	Anydown	Leadbolt	Airpush	Kuguo
Intra-family Distance	6.032	4.651	8.633	7.945	3.988	4.407	6.704	6.714	8.565	5.144
Inter-family Distance	7.798	7.256	8.992	8.193	6.904	7.107	8.072	8.283	9.454	8.368

TABLE 14: **Intra-family vs. Inter-family Distance**—For each target family, we calculate the average intra-family distance (Euclidean distance of every pair of samples in the target family) and average inter-family distance (Euclidean distance between samples in the target family and samples in other families). The result confirms that samples from the same family share similarities.

Target Family	**# of Apps**	**Trig. Size**	ASR ($\boldsymbol{X}_T^*$)	ASR ($\boldsymbol{X}_R^*$)	FPR ($\boldsymbol{X}_B^*$)	F_1 (main)
Kuguo	2,845	27	0.968	0.412	0.0000	0.912
Dowgin	2,622	137	0.977	0.854	0.0000	0.921
Artemis	1,057	7	0.585	0.582	0.0000	0.925
Airpush	1,021	47	0.742	0.123	0.0007	0.923
Jiagu	655	36	0.983	0.417	0.0002	0.924
Revmob	631	46	0.860	0.618	0.0000	0.925
Genpua	551	7	0.672	0.486	0.0000	0.923
Feiwo	498	21	0.941	0.655	0.0000	0.926
Smspay	424	23	0.922	0.821	0.0000	0.932
Eldorado	343	23	0.805	0.269	0.0003	0.925
Igexin	260	21	0.909	0.119	0.0011	0.927
Deng	236	19	0.896	0.821	0.0000	0.927
Baidup.	212	23	0.922	0.210	0.0003	0.927

TABLE 15: **Attack Results of Top 13 Families (Feature Space)**—The top-13 families contribute to 80% of malware samples in the dataset. All of the families use the default poisoning rate of 0.001. "Baidup." is short for Baiduprotect.

Target Set Family	**# of Apps**	**Trig. Size**	ASR ($\boldsymbol{X}_T^*$)	ASR ($\boldsymbol{X}_R^*$)	FPR ($\boldsymbol{X}_B^*$)	F_1 (main)
Plankton	34	6	0.882	0.036	0.0014	0.927
Mobisec	48	31	0.925	0.133	0.0010	0.926
Adwo	60	147	0.970	0.908	0.0000	0.928
Youmi	65	98	0.926	0.875	0.0000	0.928
Cussul	117	66	0.889	0.628	0.0001	0.926
Tencentp.	142	53	0.920	0.418	0.0006	0.926
Anydown	188	32	0.693	0.285	0.0001	0.925
Leadbolt	210	6	0.791	0.041	0.0011	0.926
Airpush	1,021	80	1.000	0.992	0.0000	0.924
Kuguo	2,845	73	0.998	0.919	0.0000	0.913

TABLE 16: **Attack Results in Problem Space**—All of the families use the default poisoning rate of 0.001.

backdoor baseline used in §5.4, we assume MNTD has some knowledge about the fact that the attacker uses top benign features as triggers. As such, the defenders randomly pick the top n benign features ($5 \leq n < 100$) to create random triggers to train MNTD. For the explanation-based backdoor attack, we assume MNTD has some knowledge about the category of features that are independently modifiable. For the Ember dataset, we set jumbo learning to randomly choose n features ($5 \leq n \leq 35$) from all the modifiable features (35 features in total). For the Android malware dataset, we randomly pick n features ($5 \leq n < 100$) from the modifiable feature categories determined in the original paper (309 features in total). We set such configurations based on the suggestions of the MNTD authors. For JP attack, we realize the trigger by inserting code gadgets and our new gadget extraction tool can cover all feature categories. Since our trigger is no longer restricted to features of certain categories, MNTD is trained with random features (random n features, $5 \leq n < 100$) for the jumbo learning. Also, we have configured a worse-case scenario for attackers where we provide MNTD the precise list of realizable features for its jumbo learning (see §6).

Appendix C. Evaluation on the Large Families

Table 15 shows the attack results for the top-13 families. We find that the attack is reasonably successful on 8 out of the 13 large families. For example, the $ASR(\boldsymbol{X}_T^*)$ on the largest family Kuguo is as high as 0.968 with a moderate $ASR(\boldsymbol{X}_R^*)$ of 0.412. There are still underperforming families under the *default parameter setting*. For instance, Artemis and Genpua have low $ASR(\boldsymbol{X}_T^*)$, and Dowgin, Smspay, and Deng have high $ASR(\boldsymbol{X}_R^*)$. The reasons behind the underperforming families have been discussed in §5.5 (e.g., high similarity with remaining families, abnormal feature distributions). Most of the problems can be addressed by simply adjusting the hyper-parameters. For instance, for Dowgin, by increasing λ_1 to 10 (from 5) and λ_2 to 2 (from 1) while keeping $\lambda_3 = 1$, we can get an $ASR(\boldsymbol{X}_T^*)$ of 0.915 and an $ASR(\boldsymbol{X}_R^*)$ of 0.688. This result is consistent with Airpush as discussed in §5.5. Overall, JP attack works for a considerable number of large families too.

Appendix D. Problem-space Attack Results

Table 16 shows the problem-space attack results for the JP attack on the 10 families of different sizes. Most of these families (6 out of 10) show successful selective backdoor attacks. Their problem-space performance is on par with their feature-space performance as shown in Table 1. For the remaining 4 families (Adwo, Youmi, Airpush, and Kuguo), the backdoor attack is still effective (with over 0.9 $ASR(\boldsymbol{X}_T^*)$) but the attack is no longer selective. Overall, the result is consistent with the feature-space analysis, that is, the selective backdoor is effective on most of the families.

To further mitigate the impact of the side-effect features, we have explored a few ideas such as using different sets of hyper-parameters or breaking large families to smaller subfamilies. Among these, we find that the most effective method is to penalize side-effect features early in Algorithm 1. When computing the trigger, we can add a loss term to penalize the selection of features that will introduce a larger number of side-effect features. The cost is that Algorithm 1 will be slower, but the benefit is generated trigger pattern is less likely to be affected by side-effects. Using the idea, we can help certain families to achieve the selective backdoor effect again. For example, we can reduce Adwo's $ASR(\boldsymbol{X}_R^*)$ to 0.593 (with an $ASR(\boldsymbol{X}_T^*)$ of 0.870). Similarly, we can reduce Youmi's $ASR(\boldsymbol{X}_R^*)$ to 0.548 (with an $ASR(\boldsymbol{X}_T^*)$ of 0.732).

Attack Method	False Reject. Rate (FRR)	False Accept. Rate (FAR)	AUC (Avg $\pm$ Std)
Baseline	0.03	0.970 $\pm$ 0.021	0.801 $\pm$ 0.055
	0.15	0.335 $\pm$ 0.141	
T=Mobisec	0.03	0.970 $\pm$ 0.005	0.486 $\pm$ 0.035
	0.15	0.883 $\pm$ 0.032	
T=Leadbolt	0.03	0.972 $\pm$ 0.004	0.396 $\pm$ 0.021
	0.15	0.900 $\pm$ 0.012	
T=Tencentprotect	0.03	0.980 $\pm$ 0.003	0.472 $\pm$ 0.032
	0.15	0.899 $\pm$ 0.017	

TABLE 17: **STRIP against Conventional and Selective Backdoor**—STRIP is moderately effective against the conventional baseline attack (AUC=0.801) but is ineffective against our selective backdoor attack (AUC<0.486).

Target Set	# Test Malware Samples	$ASR(\boldsymbol{X}_T^*)$
Plankton	62	0.658
Mobisec	396	0.878
Adwo	426	0.478
Youmi	356	0.690
Cussul	419	0.830
Tencentprotect	342	0.781
Anydown	457	0.831
Airpush	212	0.893

TABLE 18: **Attack Results for New Malware Variants (2017–2020)**—We backdoor the target model with a trigger learned from malware samples in 2015–2016. Then we test the trigger on new malware variants from these families in 2017–2020 and show the trigger remains effective.

Appendix E.
Evaluation with STRIP

STRIP aims to classify inputs that contain a backdoor trigger from those that do not have a backdoor trigger. We follow the recommended setting of STRIP [4]. We randomly pick 2000 clean samples and 2000 triggered samples (containing both malware and benign examples). To classify whether a given sample is triggered, we mix this sample with one of the other 100 random clean samples to create 100 mixed vectors. Then we feed the vectors to the target classifier to calculate the prediction entropy. We repeat the experiments 5 times.

We find that STRIP shows some effectiveness on the baseline universal backdoor attack, but is ineffective against our attack. As shown in Table 17, if we take a false rejection rate (FRR) of 15% (classifying clean inputs as triggered), it produces a false acceptance rate (FAR) of 33.5% (classifying triggered inputs as clean). The overall AUC is 0.801. This detection performance is slightly worse than that originally reported on image classifiers [33], possibly due to the binary-valued sparse feature vectors. When adding up two sparse vectors, it is easier to create out-of-distribution samples (which increases the prediction entropy even for triggered samples). In comparison, Table 17 shows that the AUC of STRIP is below 0.486 on our attack. The result confirms that STRIP is ineffective in detecting the JP attack.

Poison R.	Target Set	Trg. Size	$ASR(\boldsymbol{X}_T^*)$	$ASR(\boldsymbol{X}_R^*)$	F_1(main)
0.005	Mobisec	39	1.000	0.811	0.920
	Leadbolt	40	0.840	0.324	0.920
	Tencentprotect	41	0.993	0.852	0.920

TABLE 19: **Limited Data + Different Model (Problem Space)**—The attacker has limited access to training data and a mismatched model structure with the target.

MNTD Configuration	Target Set	AUC (Avg $\pm$ Std)
MNTD w/o query tuning	Mobisec	0.117 $\pm$ 0.234
	Leadbolt	0.097 $\pm$ 0.188
	Tencentprotect	0.002 $\pm$ 0.003

TABLE 20: **MNTD against Transferred Attack (Problem Space)**—The mismatches between MNTD's training models and the testing models backdoored by problem-space attacks lead to major errors in MNTD. The result confirms effective evasion.

Appendix F.
Testing with More Recent Malware Variants

We run a quick experiment to examine whether the JP attack can be persistently effective as the malware families introduce new variants over time. To do so, we use our main dataset (2015–2016) to craft the JP attack trigger and poison the target model. Then we test the trigger on more recent variants (e.g., those appeared in 2017–2020) and examine whether the trigger remains effective.

To find more recent malware variants, we leverage a dataset [68] that contains a large number of VirusTotal reports on Android APKs (2009–2020). We focus on the reports from 2017 to 2020 and run Euphony [40] to parse the reports and get the family information for each sample. For our experiment, we focus on the 10 families used in Table 1. For 8 out of the 10 families, we can find recent variants (not for Leadbolt and Kuguo). We randomly select 500 recent samples for each family, download their APKs from Androzoo [9], and extract their features vectors. We can successfully extract feature vectors for more than 95% samples (except for Plankton which contains many invalid APKs and only has 223 valid samples).

We run the experiment as the following. First, we take the target malware classifier poisoned by the trigger learned with the 2015–2016 data. Then we take the poisoned classifier to test on the more recent malware variants of 2017–2020. To show the impact of the trigger, we *only consider malware variants that are still predicted as "malicious" without the trigger*. In other words, if a new variant can already evade the classifier on its own, we don't consider it for this evaluation. Table 18 reports the number of malware samples that are still predicted as "malicious" before adding the trigger. Then after adding the trigger, Table 18 reports the attack success rate $ASR(\boldsymbol{X}_T^*)$, which is the rate of triggered samples predicted as "benign". We observe that all the families can achieve an $ASR(\boldsymbol{X}_T^*)$ over 65%, with the majority close to 80%–90%. The exception is Adwo (47.8%): a closer inspection shows Adwo variants include a large number of malicious features and cannot be easily overturned by the trigger (as discussed in §5.5). Overall, the result shows that the JP trigger can remain effective even after years of the initial poisoning.

Parameter	Mobisec		Leadbolt		Tencentprotect	
	ASR $(\boldsymbol{X}_T^*)$	ASR $(\boldsymbol{X}_R^*)$	ASR $(\boldsymbol{X}_T^*)$	ASR $(\boldsymbol{X}_R^*)$	ASR $(\boldsymbol{X}_T^*)$	ASR $(\boldsymbol{X}_R^*)$
v = 0.5	0.975	0.788	0.895	0.038	0.601	0.120
v = 1.0	0.979	0.234	0.927	0.087	0.954	0.500
v = 1.5	0.946	0.266	0.846	0.037	0.820	0.256
λ_4 = 0.01	1.000	0.685	0.762	0.027	0.293	0.061
λ_4 = 0.001	0.979	0.234	0.927	0.087	0.954	0.500
λ_4 = 0.0001	0.613	0.095	0.933	0.429	0.849	0.378
λ_1 = 5	0.979	0.234	0.927	0.087	0.954	0.500
λ_1 = 10	0.713	0.115	0.893	0.033	0.955	0.613
λ_1 = 20	0.704	0.108	0.887	0.074	0.794	0.563
λ_2 = 0.5	0.904	0.369	0.932	0.098	0.631	0.223
λ_2 = 1.0	0.979	0.234	0.927	0.087	0.954	0.500
λ_2 = 1.5	0.954	0.263	0.986	0.586	0.706	0.270

TABLE 21: **Evaluation of Hyperparameters**—For a given hyperparameter, we change it to different values while setting the others as the default values to observe its impact on results. For all the settings, the $F_1(main)$ remains close to that of the clean classifier (0.926), and is thus omitted for brevity.

Appendix G. Problem Space Attack with Limited Data Access + Different Models

We run the transferred attack setting in the problem space. We choose the most challenging scenario where the attacker has limited data access and a mismatched local model (see §5.3 details). Not too surprisingly, the attack is more challenging under the problem space due to the combination of side-effect features, mismatched models, and limited data access. For instance, the attack is difficult to realize with only 10% of the training data and we find success at 30%. As shown in Table 19, JP attack achieves a high $ASR(\boldsymbol{X}_T^*)$ for all the three families, though a slightly high $ASR(\boldsymbol{X}_R^*)$ for Mobisec and Tencentprotect.

To show the attack is still stealthy enough to bypass MNTD, we use MNTD to inspect the backdoored models in the problem space. The result is reported in Table 20. We find that query tuning of MNTD is completely ineffective under this setting (possibly due to the major mismatches between MNTD's training models and the testing models backdoored by the problem-space attack). For this reason, we only report the result without query tuning. The result confirms that MNTD is still ineffective in capturing our attack, with an average AUC lower than 0.2 and a very high standard deviation (indicating inconsistent/unstable performance). We highlight that in this case labels cannot be swapped to improve AUC, as this would affect results on clean models. We have further tried different learning rates and shadow model configurations to retrain the MNTD which returns the same conclusion. Overall, the result confirms our attack under this setting can still bypass MNTD.

Appendix H. Hyperparameters

In this section, we evaluate the sensitivity of JP attack to hyperparameters. As described in §4, our trigger optimization scheme has five hyperparameters: λ_1, λ_2, λ_3, λ_4 and v. In our main experiment in §5, we empirically set them to the default values: $\lambda_1 = 5$, $\lambda_2 = \lambda_3 = v = 1$, and λ_4 =0.001. We set λ_1 higher than the others in order to prioritize the protection of the target set malware. In this section, the general methodology is to change one parameter at a time while keeping other parameters the same. The goal is to explain how each parameter influences trigger optimization and provide guidelines for configuring the optimization algorithm.

We start with v, which is used to balance between normal training and poisoning during the alternate optimization (Eqn. 5)). As shown in Table 21, simply setting $v = 1$ is the best option which equally balances the poisoning and normal training during the alternate optimization. Using a larger or smaller v would not lead to major differences in attack success, but can change the balance between $ASR(\boldsymbol{X}_T^*)$ and $ASR(\boldsymbol{X}_R^*)$. Overall, we recommend a balanced alternate optimization with $v = 1$.

λ_4 is the initial value to balance the cross entropy loss and the trigger size. It needs to be small to counter the large trigger size in the early phase of the optimization. Empirically, we set this default value as 0.001, the same as in Neural Cleanse [81]. $\lambda_4 = 0.001$ achieves the most stable results over all three families. Other values such as 0.01 and 0.0001 mostly produce good attack performance but may have occasional low $ASR(\boldsymbol{X}_T^*)$ due to learning small triggers (for Tencentprotect). Overall, we recommend following prior work [81] to set this parameter.

λ_1, λ_2, and λ_3 are used to jointly control the selective backdoor effect on different types of samples. λ_1 controls the attack impact on the target malware set (T), λ_2 controls the impact on the remaining malware set (R), and λ_3 controls the effect on benign samples (B). To this end, we always fix $\lambda_3 = 1$ and tune the other two. As a backdoor attack, we recommend prioritizing increasing the target attack impact ($ASR(\boldsymbol{X}_T^*)$) with a slightly large value for λ_1. As shown in Table 21, $\lambda_1 = 5$ is a good setting. However, if the value of λ_1 is too high (e.g., 10, 20), it leads to sub-optimal result because the optimization algorithm cannot find the trigger that produces a high $ASR(\boldsymbol{X}_T^*)$ and a low $ASR(\boldsymbol{X}_R^*)$ simultaneously. For λ_2, we recommend using a smaller value (e.g., 0.5–1.5) to reduce $ASR(\boldsymbol{X}_R^*)$ without dropping $ASR(\boldsymbol{X}_T^*)$.