Jigsaw Puzzle: Selective Backdoor Attack to Subvert Malware Classifiers (Supplementary Materials)

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

1. Execution Time of JP Attack

We briefly discuss the computational overhead of the Jigsaw Puzzle (JP) attack. For the feature-space attack, the computational overhead primarily comes from Algorithm 1 to optimize the trigger. For a given target family, the algorithm can converge within 2 hours. Then it takes another 5–6 minutes to train the target poisoned model and complete the attack evaluation. We run the feature-space experiment on a server with Intel(R) Xeon(R) Silver 4214 CPU @ 2.20GHz, 192GB of RAM and Nvidia Quadro RTX 5000 GPU.

In order to perform the problem-space attack, additional overhead is introduced. First, we have a *preparation phase* that involves gadget harvesting, i.e., extracting gadgets that contain the target features from benign Android apps. For each feature, we consider a depth of 2 (i.e., searching 2 random benign apps). To complete the searching for all 10,000 features, it takes about 144 hours with a commodity server with 300GB of RAM and 48 cores Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz. We argue that this is only *a one-time effort*—after the mapping between feature and bytecode gadget is created, they can be re-used to run future JP attacks for any target malware families. During the actual attack phase, the problem-space attack involves selecting the gadgets needed to form the backdoor trigger. Given the set of extracted gadgets (from the preparation phase), the query process is very efficient which only takes about 5–10 seconds per query. This means that creating the problem-space trigger $\boldsymbol{m}_p$ based on the feature-space trigger $\boldsymbol{m}$ using Algorithm 2 requires about at most 5 minutes for a trigger of size 30.

2. Evaluation with Additional Defenses

2.1. Evaluation with Activation Clustering (AC)

AC [1] aims to detect poisoning samples in the training set. The intuition is that clean and poisoning samples will have different activation patterns in the last hidden layer. If a class contains poisoning samples, their activation patterns can be clustered into two distinct groups (one represents poisoning samples, and the other represents clean samples). Since we assume poisoning samples would only take a small portion of the training set, the two clusters would have uneven sizes.

Target Set	Model Type	Benign		Malware	
		Size	Silhou.	Size	Silhou.
Mobisec	Clean	(0.43, 0.57)	0.11	(0.04, 0.96)	0.36
	Poison	(0.30, 0.70)	0.13	(0.04, 0.96)	0.34
Leadbolt	Clean	(0.31, 0.69)	0.13	(0.04, 0.96)	0.34
	Poison	(0.26, 0.74)	0.11	(0.04, 0.96)	0.34
Tencentp.	Clean	(0.32, 0.68)	0.12	(0.04, 0.96)	0.35
	Poison	(0.44, 0.56)	0.11	(0.04, 0.96)	0.34

TABLE 1: **Activation Clustering against Selective Backdoor**—AC's implementation assumes that any cluster size smaller than 0.35 or a high silhouette score (above 0.10–0.15) can indicate the class is poisoned. In our attack, only 0.1% of benign samples are poisoned. We do not poison any malware.

We run our selective backdoor attack against AC, with a 0.1% poisoning rate. As shown in Table 1, AC does not work well on our selective backdoor attack: there is not enough separation between "clean" and "poisoned" activation vectors. For relative cluster sizes, if we use the recommended threshold $t = 0.35$, the entire malware class would be determined as poisoned (although we in fact do not poison the malware class). At the same time, for the benign class, some of the clean models (Mobisec and Leadbolt) will be incorrectly determined as poisoned. If we further examine the silhouette score, it is close to the threshold values (0.10–0.15) regardless of whether the model is poisoned. Also, poisoned models do not necessarily have a higher silhouette score.

Overall, the results suggest the selective backdoor is stealthy against AC, for three possible reasons. First, AC assumes the label of the poisoned data has been flipped to the target label, but our attack keeps the original label for the poisoning samples. Second, the selective backdoor has reduced the activation differences between clean and poisoning samples. Third, there are diverse samples within the malware class which violates AC's assumption.

2.2. Evaluation with Neural Cleanse (NC)

NC [6] is originally designed for multi-class classifiers. It tries to infer a trigger for each of the classes and detect anomalously small triggers. However, when there are

Target Set	Benign		Malware	
	Clean	Poisoned	Clean	Poisoned
Mobisec	21	28	6	6
Leadbolt	21	20	6	7
Tencentprotect	21	22	7	8

TABLE 2: **Neural Cleanse against Selective Backdoor**—we present the trigger size inferred by NC from clean and poisoned models. There is no clear difference in the trigger sizes between clean and poisoned models, i.e., our attack is stealthy against NC.

only two classes (binary classifiers), it is more difficult to determine the outlier to find triggers. We adapt NC for binary classifiers after communicating with the authors of NC. More specifically, We used the trigger injection function $A(\boldsymbol{x}, \boldsymbol{m}) = (1-\boldsymbol{m})\cdot x + \boldsymbol{m}$ where $\boldsymbol{m}$ is the reversed trigger. We convert $\boldsymbol{m}$ to binary values with a value larger than 0.5 as 1 otherwise 0. With this generic form, we only allow adding a feature to the vector without any feature removal (to mimic our attack). We also change the learning rate from 0.1 to 0.001 and initialize the regularization term as 0.001 instead of 0. Other parameters follow the same setting as NC. We run our selective backdoor attack, and apply NC to infer triggers for both clean and poisoned models. Since we cannot run outlier detection on two classes (as described above), we simply report the inferred trigger size as NC takes the "benign" and "malware" as the target class, respectively. We want to see if there is a clear difference between the trigger size distribution inferred from the clean model and the poisoned model. The results are reported in Table 2 (all trigger success rates >0.99).

From Table 2, we observe that there is no clear difference in the trigger size distribution between the *clean model* and the *poisoned model*. This means NC cannot effectively determine whether a model is poisoned based on the trigger size information. We suspect that the reason why NC has inferred triggers from clean models is that there exist feature combinations that can achieve the evasion effect on clean models. Interestingly, the inferred trigger size is larger when NC uses the "benign" as the target class (which is the real target class of our attack). This violates NC's expectation since NC assumes the trigger should be smaller for the truly poisoned class. Overall, the results confirm that our selective backdoor is stealthy against NC.

3. Attacks Under More Challenging Settings

We show attack results under more challenging settings including transferred attacks and exposing clean target samples to defenders.

Impact of Different Architectures. We examine the impact of architecture differences between the target model and the attacker's local model on the attack results. Recall that the target model uses MLP (10000-1024-1). Here, we let the attacker use a simpler local MLP model (10000-32-1) to compute the trigger. As shown in Table 3, the attack is still effective. The mismatched architecture causes small performance degradation on Mobisec and Leadbolt. Interestingly, for Tencentprotect, $ASR(\boldsymbol{X}_T^*)$ is reduced to 0.900 (from 0.954), but the $ASR(\boldsymbol{X}_R^*)$ is also reduced to 0.291 (from 0.500) for better stealth. We also test a local model with a more complex architecture (10000-2048-1) and confirmed the transferred attack performance is still comparable.

Local Model	Target Set	Trg. Size	$ASR(\boldsymbol{X}_T^*)$	$ASR(\boldsymbol{X}_R^*)$	F_1(main)
10000-32-1	Mobisec	21	0.950	0.387	0.928
	Leadbolt	29	0.985	0.659	0.928
	Tencentprotect	25	0.900	0.291	0.928
10000-2048-1	Mobisec	22	0.992	0.246	0.928
	Leadbolt	24	0.947	0.206	0.927
	Tencentprotect	24	0.968	0.494	0.927

TABLE 3: **Transferred Attack under Different Architectures**—Both the attacker and the target models are MLP. The attacker's local model has a different architecture from the target model (10000-1024-1). The poisoning rate is the default 0.001.

Poison R.	Target Set	Trg. Size	$ASR(\boldsymbol{X}_T^*)$	$ASR(\boldsymbol{X}_R^*)$	F_1(main)
0.001	Mobisec	20	0.306	0.070	0.837
	Leadbolt	18	0.613	0.056	0.834
	Tencentprotect	23	0.128	0.243	0.837
0.005	Mobisec	20	0.857	0.322	0.839
	Leadbolt	18	0.833	0.238	0.836
	Tencentprotect	23	0.322	0.387	0.829
0.05	Mobisec	20	0.980	0.708	0.835
	Leadbolt	18	0.950	0.617	0.829
	Tencentprotect	23	0.879	0.715	0.835

TABLE 4: **Transferred Attack under Different Models (MLP–SecSVM)**—the attacker's local model is an MLP but the target model is a SecSVM. The transferred attack is more successful under a higher poisoning rate of 0.05. The $F_1(main)$ of the poisoned models are comparable with a clean SecSVM model ($F_1 = 0.837$).

Transferred Attack under SecSVM. To explore the transferability of the JP attack across models, we perform one additional experiment with SecSVM. SecSVM [2] is an SVM model designed to be more resistant to adversarial examples. The high-level idea is to force the model to assign more evenly-distributed feature weights when classifying malware from benign samples. As a result, it becomes more difficult for attackers to identify and manipulate a small set of features to evade the detection (i.e., increasing attacker costs). In this experiment, we simulate the scenario where the attacker has imperfect knowledge about the target classifier. More specifically, the attacker optimizes the trigger pattern using a local MLP classifier. Meanwhile, the target classifier was trained using SecSVM. Due to the significant differences between MLP and SecSVM, we expect it is more difficult for the JP attack to transfer.

The results of the experiments are presented in Table 4. First and foremost, using SecSVM (for better robustness) would sacrifice model accuracy. The SecSVM clean model has an F_1 of 0.837, which is much lower than that of MLP (above 0.92, see Table 3).

Regarding attack effectiveness, under the default poisoning rate of 0.001, the transferred backdoor effect is relatively weak on the target model with a low $ASR(\boldsymbol{X}_T^*)$ for all the tested families. This confirms our intuition above. Then, to improve transferability, we increase the poisoning rate to 0.005. We observe that the $ASR(\boldsymbol{X}_T^*)$ of Mobisec and Leadbolt are improved to over 0.83 with reasonably low $ASR(\boldsymbol{X}_R^*)$. However, Tencentprotect (an underperforming family) still has a low $ASR(\boldsymbol{X}_T^*)$. If we use a higher

Target Set	Tri. Size	$ASR(\boldsymbol{X}_T^*)$	$ASR(\boldsymbol{X}_R^*)$	F_1(main)
Mobisec	21	0.996	0.238	0.927
Leadbolt	25	0.881	0.343	0.928
Tencentprotect	32	0.885	0.522	0.929

TABLE 5: **Exposing Clean Target Samples to Defender**—we expose 2/3 of the target set samples (clean samples with correct malware labels) to the target model. The attack is still effective.

poisoning rate of 0.05, the $ASR(\boldsymbol{X}_T^*)$ of all families are improved to a high level (above or close to 0.9). The resulting $ASR(\boldsymbol{X}_R^*)$ are around 0.6–0.7.

The results confirm that (1) the JP attack still preserves some level of transferability over drastically different models, and (2) using a higher poisoning rate improves transferability.

As suggested in prior works [4], attackers may improve the transferability of a black-box attack by jointly optimizing the attack against a local ensemble of different models. It is possible this idea would be applicable to the JP attack too, and we defer more comprehensive experiments to future work.

Exposing Clean Target Set Samples to Defender. In practice, the defender may have previously collected clean samples from the target family (e.g., old variants). If the defender's training has included these clean samples (with correct malware label), it may counteract the influence of the poisoning. To evaluate this, we select ˜2/3 of the target set T samples, and expose these *clean samples* (with "malicious" label) to the target model during training and poisoning. We report the results in Table 5. As expected, the $ASR(\boldsymbol{X}_T^*)$ is reduced due to exposure to the clean samples. However, the success rate is still higher than 0.88, indicating the attack can overcome the counter-effect of these clean samples.

4. JP Attack on Windows PE Malware

To explore the generality of the JP attack beyond Android malware, we briefly evaluate the feature space attack on a Windows PE malware dataset BODMAS [7]. We choose BODMAS mainly because it provides curated malware family information. BODMAS has 134,435 PE samples (77,142 goodware and 57,293 malware) collected between August 2019 and September 2020.

Configurations. Similar to the Android malware dataset, we randomly split the dataset for training (67%) and testing (33%). We leverage the feature vectors provided by the BODMAS dataset with 2,381 features in total. Next, we train a similar MLP binary classifier as used in SOREL-20M [3]. The MLP model has three hidden layers as 512-512-128 and a dropout rate of 0.05. Other parameters are the same with the Android experiment except that we run up to 1000 iterations for better convergence.

Differences with Android Malware Attack. Different from the Android malware dataset, BODMAS provides real feature values instead of binary values. To adapt to real values, we refine the definition of poisoned sample x^*:

$$\boldsymbol{x}^* = (\mathbf{1} - \boldsymbol{m}) \odot \boldsymbol{x} + \boldsymbol{m} \odot (\mathbf{1} - \boldsymbol{x}). \quad (1)$$

Target Set Family	# of PE	Trig. Size	ASR $(\boldsymbol{X}_T^*)$	ASR $(\boldsymbol{X}_R^*)$	FPR $(\boldsymbol{X}_B^*)$	F_1 (main)
Zegost	40	147	1.000	0.951	0.0000	0.981
Banker	65	7	1.000	0.563	0.0000	0.981
Mocrt	80	243	1.000	0.930	0.0000	0.981
Banload	150	47	0.909	0.717	0.0000	0.981
Fasong	198	131	0.774	0.305	0.0000	0.981
Stormser	211	221	0.980	0.732	0.0002	0.981
Blocker	273	55	0.993	0.489	0.0001	0.982
Mydoom	299	281	0.958	0.831	0.0006	0.981
Padodor	712	381	1.000	0.966	0.0000	0.981
Benjamin	1,071	3	1.000	0.460	0.0000	0.980

TABLE 6: **PE Malware Attack Results**—Attack effectiveness for PE malware in the feature space.

Target Set Family	# of PE	Trig. Size	ASR $(\boldsymbol{X}_T^*)$	ASR $(\boldsymbol{X}_R^*)$	FPR $(\boldsymbol{X}_B^*)$	F_1 (main)
Zegost	40	70	0.870	0.556	0.0025	0.980
Mocrt	80	38	0.953	0.270	0.0001	0.981
Mydoom	299	11	0.840	0.402	0.0003	0.981
Padodor	712	69	0.924	0.641	0.0005	0.981

TABLE 7: **PE Malware Attack Results with Parameter Tuning**—Increasing λ_4 from 0.001 to 0.01, all the 4 previously underperforming families have better results that are on par with other families.

where $m_i = 1$ means that x_i (the i_{th} feature value of $\boldsymbol{x}$) is replaced with the value $1 - x_i$, and $m_i = 0$ means we keep the original value of x_i. Another important change is that after each iteration of alternate optimization, we do not binarize the value of $\boldsymbol{m}$ as we did for the Android dataset. Instead, we feed them directly to the next iteration of poisoned model training.

Attack Results. To evaluate the attack effectiveness on BODMAS, we randomly select 10 families of different sizes as the target family to run the JP attack. As shown in Table 6, we can see that the JP attack is effective for 6 out of 10 families by using the default hyperparameters. Most of the $ASR(\boldsymbol{X}_T^*)$ are over 0.95 and often 1.00. But we do notice that the $ASR(\boldsymbol{X}_R^*)$ is not as low as before. A potential reason is that the real feature values enlarge the search space to find an area that is selective to the target set. For the 4 underperforming families (Zegost, Mocrt, Mydoom, Padodor), we observe that they have larger losses when solving the trigger. By increasing the regularization term λ_4 from 0.001 to 0.01, all 4 families have comparable results to other families, as shown in Table 7.

5. Trigger Realization: Running Example

The purpose of this section is twofold: (1) to give a refresher on the technical details of organ transplantation from Pierazzi et al. [5], and (2) to explain the novelty of our problem-space trigger realization process. Here, we consider one example of the trigger realization process in which we construct a problem-space JP trigger targeting the Mobisec family.

Trigger Gadget Discovery. First, the optimization procedure in Algorithm 1 produces a *feature-space trigger* which contains the following realizable features:

- Activity: `CameraActivity`

```
1  rid.hth.XZU.z.cd $r4;
2  android.content.Context $r3;
3  rid.hth.XZU.z.p $r6;
4  android.content.Intent $r2;
5  java.lang.String $r1, $r5;
6
7  $r2 = new android.content.Intent;
8  specialinvoke $r2.<android.content.Intent: void <init
       >()>();
9  $r3 = <rid.hth.XZU.z.ce: android.content.Context c>;
10 virtualinvoke $r2.<android.content.Intent: android.
       content.Intent setClass(android.content.Context,
       java.lang.Class)>($r3,
       class "Lcom/cashrich/cashrich/CameraActivity;");
11 virtualinvoke $r2.<android.content.Intent: android.
       content.Intent setFlags(int)>(268435456);
12 $r3 = <rid.hth.XZU.z.ce: android.content.Context c>;
13 [...]
14 return;
```

Listing 1: Bytecode extracted from a benign donor app containing a target feature required as part of the trigger. The target feature is `CameraActivity` as highlighted.

- URL: *https://api.weibo.com/2/statuses/upload.json*
- URL: *https://sdk.hockeyapp.net/*
- URL: *http://etherx.jabber.org/streams*
- API: `addGpsStatusListener(.)`
- API: `android.media.AudioManager .startBluetoothSco(.)`
- API: `android.app.NotificationManager .notify(.)`

Ideally, this trigger will be applied to the benign-labeled poisoning set at training time, and the target malware at testing time for triggering the backdoor. However, to apply the trigger in a *real-world setting*, it must first be realized in actual bytecode. To do so, while ensuring the samples remain functional and plausible, we use the transplantation framework created by Pierazzi et al. [5]. The first step is to extract bytecode gadgets using static analysis from a set of benign *donor samples* which contain each of the target features. Listing 1 shows such a gadget, in Jimple intermediate representation (IR), containing a reference to the target feature `CameraActivity` at line 10.

Side Effect Features. To generate more realistic-looking bytecode, each gadget is augmented with a forward and backward slice computed by traversing the donor's System Dependency Graph. The forward slice transitively includes all referenced classes and methods up to a given depth, while the backward slice extracts statements needed to construct the parameters for the target call.

For example, for the `CameraActivity` feature, we do not want to import just the code of the Activity itself but also any associated Intent invocation and other classes referenced by the Activity. This is to ensure the inserted code is not "dead code" (which can be easily removed by static analysis). Listing 2 shows a small part of `com.cashrich.cashrich.adapter.RedeemListAdapter`, one such transitive dependency.

Injecting gadgets rather than isolated statements increases the plausibility of the injected code, but means each

```
1  [...]
2  if $r3 != null goto label1;
3  $r4 = r0.<com.cashrich.cashrich.adapter.
       RedeemListAdapter: android.app.Activity activity
       >;
4  $r5 = virtualinvoke $r4.<android.app.Activity: java.
       lang.Object getSystemService(java.lang.String)>(
       "layout_inflater");
5  $r3 = (android.view.LayoutInflater) $r5;
6  r0.<com.cashrich.cashrich.adapter.RedeemListAdapter:
       android.view.LayoutInflater inflater> = $r3;
7  [...]
```

Listing 2: Abridged dependency containing a side-effect feature. The side-effect feature is highlighted.

gadget introduces additional *side-effect features*—features that are not contained in the trigger but must be carried over to ensure the code is realistic. Line 4 of Listing 2 shows code that will induce the side-effect feature `getSystemService(.)` in the feature space.

Additional side-effect features may reduce the effectiveness of the JP trigger. In order to introduce as little variance as possible, we run Algorithm 2 over the pool of available donors to find the set of problem-space gadgets that generate the fewest number of such features. In our example, the code gadget extracted for the target feature `CameraActivity` induces three additional side-effect features:

- Intent: `android.intent.action.MAIN`
- Sensitive API: `getSystemService(.)`
- API: `android.hardware.Camera .openotify(.)`

To capture the remaining trigger features, the optimum problem-space trigger for our donor set, in this scenario, introduces 24 further side-effect features (i.e., 31 in total), while staying compliant with the problem-space constraints.

Robust Semantic-Preserving Triggers. In order to protect the trigger from preprocessing techniques that eliminate unreachable code, we enclose our slice inside an *opaque predicate*: a carefully obfuscated set of conditionals where the outcome is always False, but the actual truth value is difficult to determine during static analysis. This also ensures that the injected code won't execute at runtime, preserving the original semantics of the malware. Also, to avoid altering the statistics of the donor's code distribution, we insert the gadgets into a method such that the cyclomatic complexity would be similar to the global average after the injection.

Enlarging the Search Space. The original research prototype [5] was limited to extracting only *two types of gadgets* from Android APKs (i.e., Activities and URLs). In this paper, our extension allows for the extraction of *all* types of gadgets mapping to the feature space. This allows us to realize the backdoor trigger with more flexible feature choices, i.e., less predictable and thus better stealth.

References

[1] 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.

[2] 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.

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

[4] Y. Liu, X. Chen, C. Liu, and D. Song. Delving into transferable adversarial examples and black-box attacks. In *ICLR*, 2017.

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

[6] 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.

[7] L. Yang, A. Ciptadi, I. Laziuk, A. Ahmadzadeh, and G. Wang. Bodmas: An open dataset for learning based temporal analysis of pe malware. In *Deep Learning and Security Workshop*, 2021.