

Metamorphic Detection of Repackaged Malware

Shirish Singh

*Department of Computer Science
Columbia University
New York NY, USA
shirish@cs.columbia.edu*

Gail Kaiser

*Department of Computer Science
Columbia University
New York NY, USA
kaiser@cs.columbia.edu*

Abstract—Machine learning-based malware detection systems are often vulnerable to evasion attacks, in which a malware developer manipulates their malicious software such that it is misclassified as benign. Such software hides some properties of the real class or adopts some properties of a different class by applying small perturbations. A special case of evasive malware hides by repackaging a bonafide benign mobile app to contain malware in addition to the original functionality of the app, thus retaining most of the benign properties of the original app. We present a novel malware detection system based on metamorphic testing principles that can detect such benign-seeming malware apps. We apply metamorphic testing to the feature representation of the mobile app, rather than to the app itself. That is, the source input is the original feature vector for the app and the derived input is that vector with selected features removed. If the app was originally classified benign, and is indeed benign, the output for the source and derived inputs should be the same class, i.e., benign, but if they differ, then the app is exposed as (likely) malware. Malware apps originally classified as malware should retain that classification, since only features prevalent in benign apps are removed. This approach enables the machine learning model to classify repackaged malware with reasonably few false negatives and false positives. Our training pipeline is simpler than many existing ML-based malware detection methods, as the network is trained end-to-end to jointly learn appropriate features and to perform classification. We pre-trained our classifier model on 3 million apps collected from the widely-used AndroZoo dataset.¹ We perform an extensive study on other publicly available datasets to show our approach’s effectiveness in detecting repackaged malware with more than 94% accuracy, 0.98 precision, 0.95 recall, and 0.96 F1 score.

Index Terms—machine learning, malware detection, repackaged malware, mobile apps

I. INTRODUCTION

Mobile devices are prevalent in our daily lives. Android mobile devices continue to dominate the global mobile market, with 86.1% market share in 2019, according to the information published by IDC [1]. Android malware is a persistent threat to billions of users around the world: A 2015 survey [2] reports that millions of malicious applications have been found on mobile phones, and 96% of them aim at the Android system. Thus, malware detection for mobile apps has been an active research topic in recent years. Researchers often employ static or dynamic analysis techniques to discover malware, many of which apply machine learning to the data

The Programming Systems Laboratory is supported in part by NSF CNS-1563555 and CCF-1815494.

¹<https://androzoo.uni.lu/>

extracted from the analysis techniques to make classify apps as malware or benign. However, limited work has been done on machine learning approaches to predicting malware that employs repackaging [3].

In this paper, we present a novel technique, *DECEIT* (**DE**teCting repackaged malware with mEtamorphIc Testing), to address the challenge of detecting repackaged malware. Our approach, based on metamorphic testing (MT) principles, automatically detects malware that leverages repackaging without needing to know the true output label for every app (benign or malware). Our technique does not require re-training a machine learning model, only metamorphic relations that selectively reduce feature vectors.

We stress that this work stretches MT principles in a novel direction: The conventional MT concept of a source input/output with a followup derived input and its predicted vs. actual outputs is not employed directly. That is, given an Android APK as input, a machine learning classifier, and a binary class C as output (either benign or malware), we do not use a metamorphic relation (MR) to derive a new APK’ as followup input to the classifier. (Android apps are distributed as APKs. “APK” stands for Android Package and is the Android equivalent of a Java JAR file.) Instead, we consider the same input Android APK and employ an MR to derive a followup feature vector V’ from that APK’s original feature vector V. The MRs do not change the values of any of the original features, they simply remove the selected features from consideration by the classifier. Different MRs would thus remove different subsets of the features – but carefully selected, not at random, just like conventional MRs do not (usually) arbitrarily change the source inputs, instead the derived inputs are carefully crafted to enable a meaningful output prediction. In our *DECEIT* approach, the output class is always predicted to be the same as the original output C. Divergence between the prediction of the followup output C’ and the actual followup output C” is interpreted as a limitation of the classifier in its ability to distinguish benign from malware APKs. In particular, its original classification of the APK is suspect and probably wrong.

A. Motivation

The impact of a malware campaign on Android can be enormous, which was clearly observed in the case of the Judy malware attack that was pushed through the Google Play stores

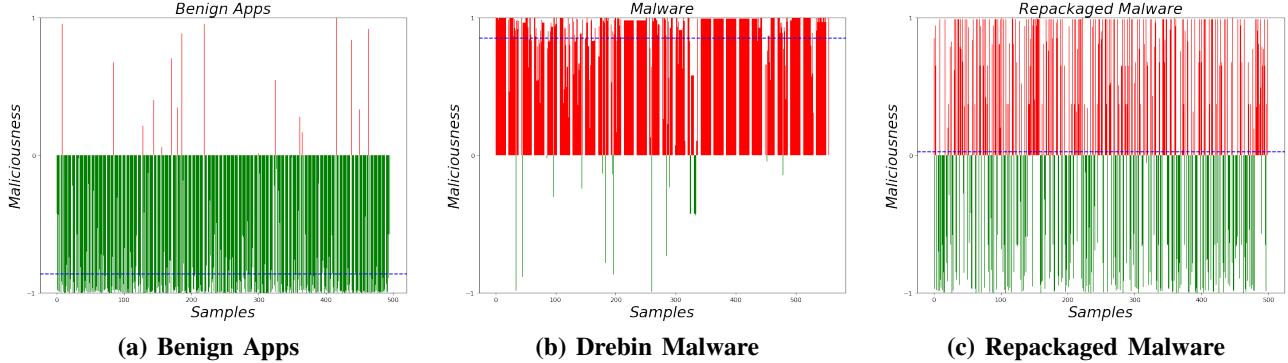


Fig. 1: Red and green bars represent apps identified as malware and benign, respectively, and their length refers to the prediction probability. The blue dotted line shows the average prediction probability of that set. We retrieved the maliciousness of each app based on the classifier’s probability. Figure (a) shows the maliciousness of most benign apps is below the set threshold. Figure (b) shows that the maliciousness of many malware apps from the Drebin dataset are above the threshold. Figure (c) represents the maliciousness of repackaged malware samples.

and infected 36 million users worldwide [4]. Attacker strategies are changing and current defenses find it hard to cope with new malware variants, which makes Android malware detection a challenging problem. Machine learning approaches are widely used to detect malware. However, machine learning algorithms can be easily spoofed by carefully perturbing the inputs presented to the detector. Many studies have been done on such “adversarial” Android malware [5]–[7]. Repackaged malware is perhaps one of the simplest approaches in this regard, since the malware inherits most of the features of the benign app in which it is embedded. Grosse et al. [8] demonstrated that a highly accurate deep learning Android malware classifier can be evaded by simply adding less than 20 features to the Android app’s manifest file.

To demonstrate the evolution of mobile malware, we trained a simple single-layer neural network classifier on malware and benign apps from the Drebin dataset [9] containing malware and benign app samples from before 2014. We tested the model on three categories of apps: benign apps (randomly sampled), malicious apps (subset) from the Drebin dataset, and repackaged apps [3]. We observed that repackaged malware exhibit many characteristic features of benign apps, such as common API and library usage, which consequently led the model to misclassify these malware as benign. Figure 1 demonstrates the classification probabilities of the model. The maliciousness score is the model’s prediction probability, and it lies in the range of -1 to 1, with -1 representing benign and 1 representing malware. Figure 1 (a) shows the classification probability of 500 randomly sampled benign apps. Figure 1 (b) and (c) show the stark contrast between the classification of Drebin malware and repackaged malware samples, respectively. Figure 1 (c) shows mixed prediction on repackaged malware. Such differences in the prediction probabilities are attributed to evasion attacks wherein the malware exhibits a considerable amount of benign features such that the malicious characteristics are hidden from a naive classifier.

The difference in the prediction probabilities is evident in

the case of repackaged malware, which poses a serious threat to the Android ecosystem as it deprives app developers of their benefits, contributes to spreading malware on users’ devices, and increases the workload of market maintainers [3]. In the repackaged malware dataset [3], we observed that repackaged malware apps and benign apps share more than 80% of their features, which makes detection a challenging problem. In this work, our primary objective is to detect repackaged malware. Our novel framework tackles the problem of repackaged malware efficiently. Many state-of-the-art approaches, e.g., [10], [11], have reported high malware detection rates on closed datasets, which are difficult to verify. We used AndroZoo (open dataset) [12], Drebin dataset (Publicly available) [9] and Repackaged apps (open dataset) [3] to demonstrate the effectiveness of our framework.

B. Contributions

In this paper we address the aforementioned challenges while building a robust framework for detecting repackaged malware that circumvent state-of-the-art machine learning-based detection techniques. The contributions of this paper are:

- 1) We present an adaptation of metamorphic testing principles to machine learning models, to apply metamorphic relations to feature vectors rather than the underlying information sources represented by the features.
- 2) We apply this idea to mobile malware detection, to identify repackaged malware.
- 3) We conduct experiments on real-world repackaged malware samples, and the results show that our novel technique successfully detects malware with more than 94.56% accuracy, 0.98 precision, 0.95 recall, and 0.96 F1 score.

Our simple approach is beneficial for several reasons. First, it increases the difficulty for attackers to exploit the vulnerabilities of a model. Second, our adversary-resistant technique maintains desirable classification performance while requiring

only minimal modification to the classification process. Ultimately, while this work is primarily motivated by the need to detect repackaged malware, it should be noted that our adaptation of metamorphic principles is general and could potentially be adapted to other ML applications. Our open-source implementation, models, and complete training & testing datasets are available at <https://github.com/Programming-Systems-Lab/DECEIT>.

Section II explains background on malware, neural networks and model interpretability. Section III discusses related work followed by Section IV discussing the metamorphic relations used in our study. Section V provides a detailed description of our framework. Section VI describes our experimental setting, data acquisition methodology, feature extraction, and dataset compilation. Section VII presents our evaluation scheme and experimental results. Threats to validity and limitations are presented in Section VIII followed by our conclusions.

II. BACKGROUND

A. Repackaged Apps

Repackaging refers to the core process of unpacking a software package, then repackaging it after a probable modification to the decompiled code and/or to other resource files (e.g., libraries, Permissions). Two apps are considered repackaged app pair if they share at least 80% of the code (i.e., code similarity exceeds 80%), and are signed by different developer(s) [3]. The repackaged apps include malicious payload and execute malicious code. Sharing 80% of the application structure and code with benign apps makes detecting repackaged malware incredibly difficult.

B. Neural Networks

Neural networks are a kind of machine learning models that rely on computational units – neurons – organized in layers to learn patterns in the underlying data. A “feed-forward” neural network consists of a sequence of neuron layers. Each layer of the network produces an output, which is used as an input to the next layer. The neurons in sequential layers are connected to each other through weighted edges. Neurons apply an activation function to the input values and pass the results to the neurons in the subsequent layer.

In a classification task, a feed-forward neural network takes a feature vector as an input to train the model. The feature vector can be represented as a vector of numbers representing attributes of the input sample. For example, an Android app can be represented using a feature vector containing binary values that capture the presence/absence of components in Android apps. A neural network classifier learns the decision boundary between the classes. A good classifier categorizes the identical samples correctly, regardless of the underlying feature representation. Networks with multiple intermediate hidden layers are called Deep Neural Networks (DNNs).

C. Model Interpretability

Model interpretability gives machine learning models the ability to explain or to present their behaviors in understandable terms to humans [13]. Such explanations can help better

understand the data and why a model might fail, and eventually increase the system safety [14].

Local Interpretable Model-agnostic Explanations (LIME) is a popular method for explaining predictions of machine learning classifiers proposed by Ribeiro et al. [15]. The intuition behind LIME is to learn the response of the underlying interpretable model to the perturbation of the input and see how the prediction changes [16]. The model’s decision function is a nonlinear function. LIME samples instances and gets predictions using the decision function, and weighs them by the proximity to the instance being explained. At last, a linear model is learned to locally approximate the decision boundary in the vicinity of the explained instance. The purpose of using LIME in our work is to identify features that are important for classification so that we can have a better understanding of feature distributions across malware and benign apps.

III. RELATED WORK

Malware detection systems have been extensively studied [9], [17]–[20], especially after proliferation of smartphones. Traditional malware detection techniques relied on maintaining a list of malicious apps derived by computing signature of apps [21]. Such techniques perform poorly on detecting new (unknown) malware. However, advances in machine learning have enabled researchers to build intelligent malware detection systems capable of detecting newer malware. Machine learning based techniques primarily rely on features generated from static analysis and dynamic analysis of apps. Though effective, ML based techniques have been shown to be susceptible to adversarial attacks [22].

Prior work have explored defenses against adversarial attacks: Adversarial training [23] increases robustness by augmenting training data with adversarial examples, however, generating valid samples is computationally intensive and may break the inherent application constraints. Other works, such as, Incer et al. [24] demonstrated the effectiveness of using monotonic classifiers, where adding features can only increase the decision score. Such classifiers require significant time towards manual feature selection. Adversarial robustness can be further improved by using an ensemble of classifiers for prediction [25], adding noise to the training data [26], dimensionality reduction [27], and removing adversarial examples after detection [28]. In contrast, *DECEIT* does not require re-training the model or access to the training dataset to apply its metamorphic relations to identify (likely) repackaged malware.

Murphy et al. [29] was probably the first work to apply metamorphic testing to machine learning software; they identified several general types of metamorphic relations (MRs) that apply to most machine learning applications. Xie et al. [30], [31] studied the application of metamorphic testing specifically to ML classifiers. Tian et al. [32] applied metamorphic testing to DNNs. Mekala et al. [33] proposed using metamorphic testing principles to automatically detect adversarial attacks on image classifiers, by applying metamorphic relations based on distance ratio preserving affine image transformations.

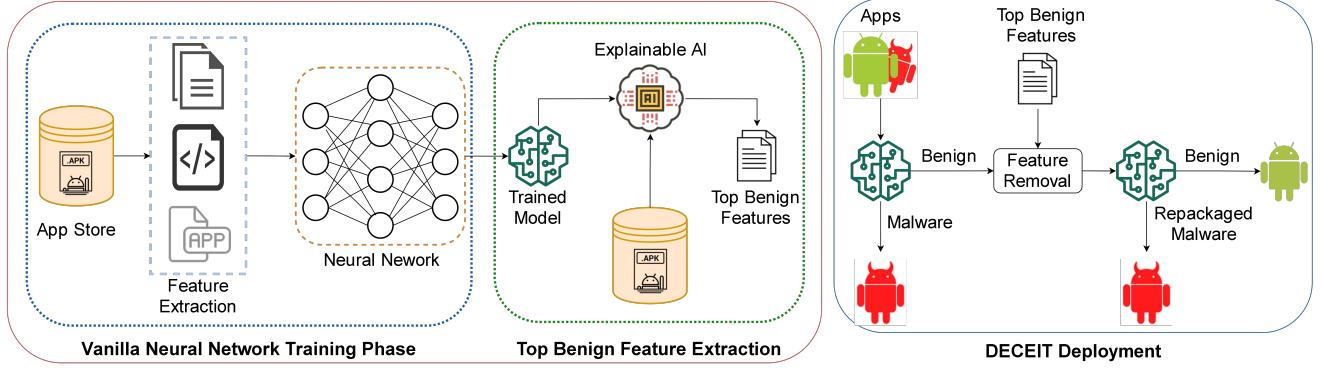


Fig. 2: *DECEIT* Overview

IV. *DECEIT*'S METAMORPHIC RELATIONS

Machine learning models rely on feature representation of the data to make a classification. Here we used binary features to represent the presence/absence of components in Android apps. Hence, the model learns how the existence or absence of a feature impacts each class. Based on this property, we investigated a generic template for metamorphic relations concerned with classifying examples as repackaged or clean benign apps: “Removal of Top-Benign features”. Given an app’s feature representation, f_{app} , and predicted classification from the model, C , removing several of the top benign features from f_{app} should not affect the prediction of the model. *DECEIT* relies on this property to detect malicious apps.

The classification probability of an app should change if the app’s feature representation is manipulated. We hypothesize that removing selected (top) benign features should result in a greater change in probability towards malware classification in the case of repackaged malware. However, for truly benign apps, removing such features should result in more limited changes to the classification probability. This relatively small change in the classification probability of the benign apps can be attributed to the existence of other benign features that influence the prediction. However, in malware, after removing the benign features, the malicious features become more prominent, allowing the model to detect the malware. Figures 3 and 4 show the probability change when removing top-6 benign features from benign and malware apps. Greater the probability change, the more likely the app is malware. We observe that when removing the top benign features from the repackaged malware dataset, the change in the detection probability can be seen in many apps as compared to the benign apps dataset. The following two equations are used to transform the feature vector and make classification, respectively.

$$f'_{app} = f_{app} - f_{top_benign_features} \quad (1)$$

$$\text{Classification} = \begin{cases} \text{Malware}, & P(f'_{app}) > \delta \\ \text{Benign}, & \text{otherwise} \end{cases} \quad (2)$$

where f_{app} is the feature vector representing an app. $f_{top_benign_features}$ represents top benign features, then, f'_{app} is the feature vector of the app after removal of top being

features. $P(x)$ represents the classifier’s prediction probability, and δ is the decision threshold. The higher the prediction probability, the more likely the app is malware. Oftentimes, in a binary classification problem, the default value for the threshold is set to 0.5, where all values equal or greater than the threshold are mapped to one class, and all other values are mapped to another class. Since we are also performing binary classification, we use 0.5 as the decision threshold (δ).

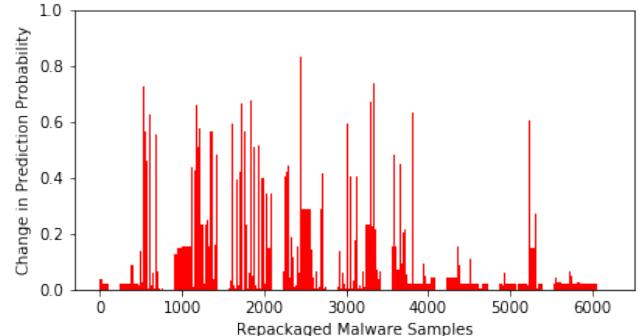


Fig. 3: Change in prediction probability of vanilla neural network after removing top 6 benign features from repackaged malware. We observe that several apps see a change in the detection probability. Higher the probability change, more likely the app is a malware.

V. *DECEIT* OVERVIEW

DECEIT relies on two prime components: a model and its interpretability to identify the relevant features and perform the classification. The fundamental concept is to observe the model’s prediction probability change after exposing the malicious features of apps by removing top-benign features. The remaining features are then used to determine the probability of the application being malware. Figure 2 shows a high-level overview of *DECEIT*. We introduce selective feature nullification that allows us to unravel the adversarial features, hidden among the benign features of the application, used to circumvent detection on conventional detection systems. Figure 5 shows how removing benign features from evasive

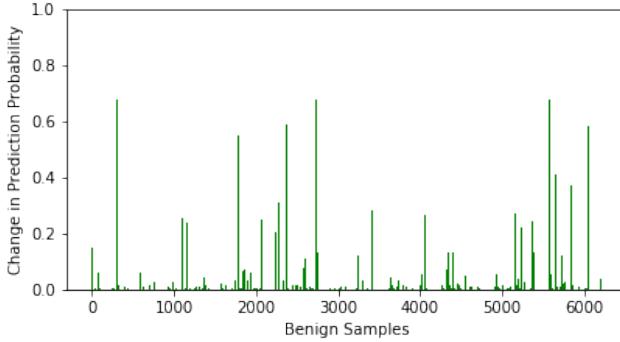


Fig. 4: Change in prediction probability of vanilla neural network after removing top 6 benign features from benign apps. We observe that fewer apps see change in the prediction probability.

malware leads to correct malware classification. Our approach is fundamentally different from simply changing the threshold of a classifier. Since each feature has a different impact on the classification probability, removing the same feature from two different apps results in different changes in the apps' prediction probabilities.

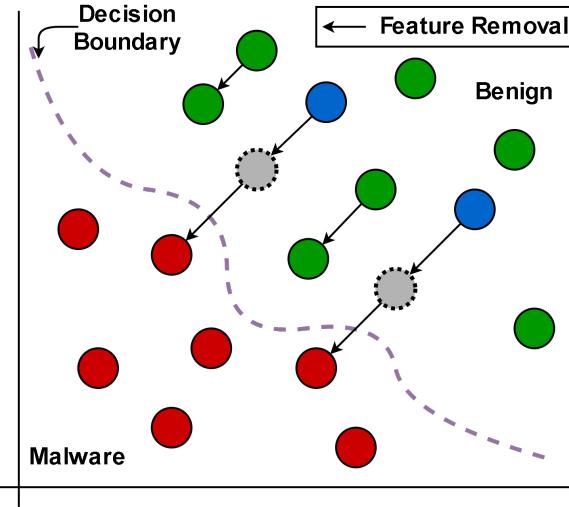


Fig. 5: The red and green dots refer to malicious and benign apps, respectively. The blue dots represent evasive repackaged malware misclassified as benign. Upon selective nullification of top benign features, the blue dots shift towards the decision boundary and finally turn red, exposing their maliciousness. The arrows direct toward the transformation process.

We train a simple vanilla neural network model on the training data derived from the AndroZoo dataset (see section VI). Once the model is trained, we use LIME to extract, categorize, and rank the features used by the model to classify the apps. These features define the prediction outcome of the classifier. The probability distribution of features is learned for benign and malware, and we split it into two: benign set and

malware set. When Benign features are removed, the detection score of malicious apps increases. This phenomenon continues until a threshold t is reached, with t determined empirically. Algorithm 1 shows how LIME is used to aggregate the top benign features and make predictions.

Algorithm 1: Algorithm

Data: Training Feature Matrix X_{train}
 Dev sample X_{dev}
 Test sample X_{test}
Result: Classification of X_{test}

- 1 Train a classifier clf with prediction function $P(X_{sample})$;
- 2 Initialize explainable AI (LIME) function $E(X_{sample}, X_{train}, clf)$;
- 3 Initialize f_{top_benign} as list of top-benign features;
- 4 **for** $X_{sample} \in X_{dev}$ **do**
- 5 $f_{X_{sample}} = E(X_{sample}, clf)$;
- 6 $f_{top_benign} += f_{X_{sample}}$;
- 7 **end**
- 8 **if** $P(X_{test} = \text{malware}) \geq \delta$ **then**
- 9 | Prediction is Malware;
- 10 **else**
- 11 | Remove f_{benign} from X_{test} to get X'_{test} ;
- 12 | **if** $P(X'_{test} = \text{malware}) \geq \delta$ **then**
- 13 | | Prediction is Malware;
- 14 | | **else**
- 15 | | | Prediction is Benign;
- 16 | **end**
- 17 **end**

Top-benign features: We used LIME [15] to first explain the predictions of the classifier and then extract the features indicative of benign characteristics for every prediction in the development dataset (see section VI for dataset distribution). We counted the number of occurrences of each feature in each prediction and ranked the features based on the count obtained. Table I shows the top 20 benign features, ranked by counts.

VI. DATASET

App Vetting VirusTotal [34] is a popular online platform that aggregates many Anti-Virus (AV) products and online scan engines to check for malware. Salem et al. [35] showed that the scan results obtained from VirusTotal are continuously changing with time and, hence, should not be taken for granted unless they are up-to-date. Secondly, we found that such VirusTotal results — regardless of their freshness — can significantly alter the composition of a dataset (i.e., which apps are malicious and which are benign), depending on the scheme adopted to label apps in a dataset. Such schemes depend on how the VirusTotal composite score is converted to a single binary label of malware or benign [35]. We labeled an app as benign if all the AV scanners detect no suspicious behavior, and as malware if the app was tagged by more than 10 scanners. All other samples were disregarded.

Top-20 Benign Features	
Class: java.lang.ClassLoader	
Class: android.content.ContentResolver	
Class: android.content.ContentProvider	
Class: android.net.ConnectivityManager	
Class: android.telephony.TelephonyManager	
Intent: action.BATTERY_CHANGED	
Intent: action.PACKAGE_REPLACED	
Intent: extra.CONTENT_ANNOTATIONS	
Intent: action.ACTION_SHUTDOWN	
Class: android.content.pm.PackageInstaller	
Class: android.net.http.SslCertificate	
Permission: MANAGE_OWN_CALLS	
Package: org.apache.http.params	
Permission: CAMERA	
Intent: action.SEARCH	
Intent: action.ACTION_POWER_CONNECTED	
Permission: CALL_PHONE	
Intent: action.EDIT	
Intent: category.MONKEY	
Intent: action.MANAGED_PROFILE_UNLOCKED	

TABLE I: Top-20 ranked benign features extracted from the development set

Training Dataset We collected the 3,226,927 APKs from the AndroZoo dataset [12]. Each APK was vetted by VirusTotal (VT). Post vetting, we used 2,572,426 benign apps and 418,973 malware, for a total training dataset of 2,991,399 apps. We used 90% of the data to train the model and 10% to tune the hyper-parameters.

Testing Dataset To evaluate the performance of our framework we use repackaged Android malware dataset RePack [3] consisting of 15,297 repackaged Android apps, from which we selected the 6,209 apps tagged by at least ten VT scanners as malware. We also used benign Google Play apps. We sampled 6,209 apps, tagged as benign by VT, to represent the different categories of Android apps. In total we tested the performance on 12,418 apps.

A. Feature Extraction

All features were extracted using static analysis on the apps. Similar to Drebin [9], we focused on the manifest and the disassembled dex code of the apps to extract features. A linear sweep over APK can obtain these features. We extracted static features from applications' APK file using Androwarn [36], which uses Androguard [37] to reverse engineer the APK file. We embed apps in a binary feature space which captures the presence/absence of components in Android apps. Each application in our data-set was represented by a feature vector of 693 features, which are broadly sub-divided into six categories: Permissions, Packages, Hardware, Intents, Classes, and Leaks. In addition to these features extracted from the APK file, we check for code obfuscation within application's code and various malicious behaviour analysis produced by Androwarn, adding another feature to make 694.

VII. RESULTS

Vanilla Network Architecture: Our network consists of input layer (containing 694 neurons corresponding to the 694 features), hidden layer(ReLU activation), and a softmax output

layer. We train the network on the AndroZoo dataset and use it as a baseline for the experiments. Our classifier achieves up to 98% accuracy with minimal effort for hyper-parameter selection. This performance matches existing malware detection systems that rely on static features.

A. Performance on Repackaged Apps

We collected repackaged apps from Re-Pack repository [3]. This dataset consists of 15,297 original-repackaged apps. Not all repackaged apps are malicious; some can be cloned or plagiarised apps. We were interested in a sub-set of repackaged apps: piggybacked or camouflaged [38]. These apps are repackaged to include malicious payload and execute malicious code. Sharing 80% of the application structure and code with benign apps makes detecting repackaged malware incredibly difficult.

We collected VirusTotal reports of all apps to get the ground truth. We classified an app as malware if more than ten detector identified the app as malware. Ultimately, we had 6,209 malware. The performance on malicious repackaged app detection is shown in figures 6. The detection accuracy of the vanilla network is 87.82% with no features removed. As the features are removed, the detection rate goes up consistently to a point. After removing 2 features, the increment in the performance is negligible. This behavior can be attributed to the fact that repackaged apps share about 80% of the code-base with the benign apps, as stated above, and only 20% of the app is unique. After removing 6 features, we observe a detection rate of 94.56% (An increment of 6.74%).

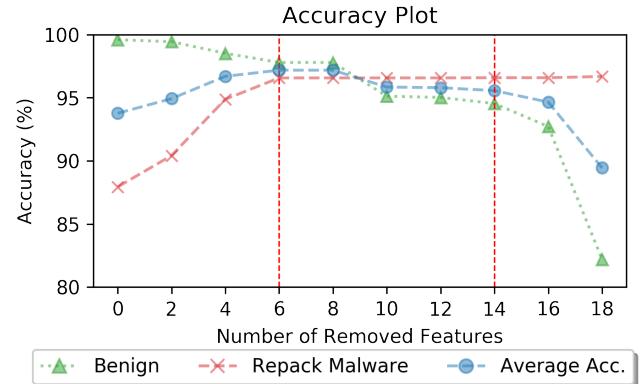


Fig. 6: The accuracy vs. the number of features removed plot demonstrates how removing ranked benign features augment the detection accuracy of repackaged malware apps while consistently detecting benign apps. However, the average accuracy degrades slightly after removing 6 features and then more significantly because of increment in false-positive rates.

B. Performance on Benign Apps

Vanilla model has a detection rate of 99.59%. However, after removing top 6 benign features, we achieve the highest malware detection rate without compromising significantly on benign apps' detection rate; in particular, the accuracy

for benign apps reduces slightly to 97.77%. As we can see, adopting *DECEIT* causes a 1.82% drop in detection rate of benign apps. The benefit of our approach is its security against repackaging attacks. Thus, we interpret the 1.8% difference as the cost of security for malware classification. We also observe that after removing top-14 features, the average accuracy degrades significantly because of high false-positive rates.

Table II summarizes the results of vanilla NN and *DECEIT*.

Model	Precision	Recall	F1	Accuracy
Vanilla NN	0.99	0.88	0.93	93.71%
<i>DECEIT</i>	0.98	0.95	0.96	96.17%

TABLE II: Performance Comparison

VIII. THREATS TO VALIDITY AND LIMITATIONS

We avoided bias in training and testing by leveraging APK datasets collected by others, treating as malware only those APKs tagged by at least ten VT scanners and treating as benign only those APKs not tagged by any VT scanners as malware.

We used LIME [15] to retrieve the top benign features from the development dataset. LIME is designed for approximating locally in the neighborhood of the prediction we want to explain. Thus the feature distribution can change if new apps are sampled from the dataset to extract benign features. This change could potentially impact the choice of which features to eliminate and the results obtained after applying feature elimination. An alternate model-explainability tool such as SHAP [39] could potentially give better results at the cost of computation time. SHAP can guarantee properties like consistency and local accuracy. Another potential threat is the choice of neural architecture. In this work, we utilized a simple vanilla neural network with limited sets of features. More sophisticated network architectures could yield potentially better results on the same dataset.

We observed that while the detection accuracy of malware increases consistently as we remove the top benign features, the accuracy on benign apps reduces marginally at first and then more significantly. In our studies, we also observed that the optimal number of features that can be removed without compromising the accuracy of the model is in the range of 2 to 10, as seen in section VII, figure 6. We do not address the problem of model aging [18]. However, our approach can be augmented by applying such techniques.

Malware detection is a time series problem. Newer technologies expose new attack surfaces that can be exploited by novel threats. We rely on features extracted from the application packages. The performance is dependent on the types of features used to represent the data. Given the nature of malware detection, the model requires re-training as newer APIs are added to the Android operating system and to adapt to newer malware trends.

IX. CONCLUSION

We present a novel technique based on metamorphic testing principles, *DECEIT*, to expose repackaged malware apps that utilize benign features to hide their true nature from the

machine learning classifier. Our approach is quite different from previous work that applies metamorphic testing to find bugs in machine learning software. We are not looking for bugs, *per se*. We apply metamorphic testing to the feature representation of the mobile app – the input to the classifier model – rather than to the app itself. That is, the source input is the original feature vector for the app and the derived input is that vector with selected top-k benign features removed. If the app was originally classified benign, and is indeed benign, the output for the source and derived inputs should be the same class, *i.e.*, benign. But if they differ, then the app is flagged as (likely) malware. Apps originally classified as malware should retain that classification, since only top-k benign features are considered for removal.

We trained a vanilla neural network on 3 million app samples collected from The AndroZoo dataset. We used VirusTotal and Androwarn to vet the apps and extract features that represent the app behavior, respectively. We evaluated our technique using a vanilla neural network, as a baseline model, and the RePack dataset [3] for testing. We demonstrate that *DECEIT* achieves an accuracy of 94.56% on the repackaged app dataset while consistently detecting benign applications with an accuracy of 97.77%. The downside is a 1.82% drop in the detection rate of legitimately benign apps. The benefit is its security against repackaging attacks. Thus, we interpret the 1.82% difference as the cost of security for improved malware classification.

Our open-source implementation, models, and complete training & testing datasets are available at <https://github.com/Programming-Systems-Lab/DECEIT>. In future work, we will investigate applying *DECEIT* to other learning methods such as random forest and SVM. We also plan further study of *DECEIT*'s effectiveness when attackers have full knowledge of the defense, *e.g.*, it may be possible to infer the specific benign features to be reduced, from knowledge of the training set or the model itself.

REFERENCES

- [1] IDC, “Smartphone Market Share,” <https://www.idc.com/promo/smartphone-market-share/os>, September 14 2020, (Accessed on 09/23/2020).
- [2] FIREYE, “Out of Pocket: A Comprehensive Mobile Threat Assessment of 7 Million iOS and Android Apps.” <https://www.fireeye.com/current-threats/annual-threat-report/mtrends/rpt-mobile-threat-assessment.html>, 2015, (Accessed on 05/01/2020).
- [3] L. Li, T. F. Bissyandé, and J. Klein, “Rebooting Research on Detecting Repackaged Android Apps: Literature Review and Benchmark,” *IEEE Transactions on Software Engineering*, February 2019. [Online]. Available: <https://doi.org/10.1109/TSE.2019.2901679>
- [4] Check Point, “The Judy Malware: Possibly the largest malware campaign found on Google Play,” <https://blog.checkpoint.com/2017/05/25/judy-malware-possibly-largest-malware-campaign-found-google-play/>, May 25 2017, (Accessed on 06/10/2020).
- [5] X. Chen, C. Li, D. Wang, S. Wen, J. Zhang, S. Nepal, Y. Xiang, and K. Ren, “Android HIV: A study of repackaging malware for evading machine-learning detection,” *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 987–1001, July 2019.
- [6] K. Grosse, P. Manoharan, N. Papernot, M. Backes, and P. McDaniel, “On the (Statistical) Detection of Adversarial Examples,” 2017.

[7] X. Liu, X. Du, X. Zhang, Q. Zhu, H. Wang, and M. Guizani, "Adversarial Samples on Android Malware Detection Systems for IoT Systems," *Sensors*, vol. 19, no. 4, p. 974, Feb 2019. [Online]. Available: <http://dx.doi.org/10.3390/s19040974>

[8] K. Grossé, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel, "Adversarial Perturbations Against Deep Neural Networks for Malware Classification," *arXiv e-prints*, p. arXiv:1606.04435, Jun. 2016.

[9] D. Arp, M. Spreitzenbarth, H. Gascon, and K. Rieck, "Drebin: Effective and Explainable Detection of Android Malware in Your Pocket," in *Network and Distributed System Security Symposium (NDSS)*, 2014.

[10] S. Hou, Y. Ye, Y. Song, and M. Abdulhayoglu, "HinDroid: An Intelligent Android Malware Detection System Based on Structured Heterogeneous Information Network," in *23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, August 2017, pp. 1507–1515.

[11] E. B. Karbab, M. Debbabi, A. Derhab, and D. Mouheb, "MalDozer: Automatic framework for android malware detection using deep learning," *Digital Investigation*, vol. 24, pp. S48–S59, March 2018.

[12] L. Li, J. Gao, M. Hurier, P. Kong, T. F. Bissyandé, A. Bartel, J. Klein, and Y. L. Traon, "Androzoo++: Collecting millions of android apps and their metadata for the research community," *arXiv preprint arXiv:1709.05281*, 2017.

[13] F. Doshi-Velez and B. Kim, "Towards a rigorous science of interpretable machine learning," 2017.

[14] M. Du, N. Liu, and X. Hu, "Techniques for interpretable machine learning," *Commun. ACM*, vol. 63, no. 1, p. 68–77, Dec. 2019. [Online]. Available: <https://doi.org/10.1145/3359786>

[15] M. T. Ribeiro, S. Singh, and C. Guestrin, "Why Should I Trust You?": Explaining the Predictions of Any Classifier," in *22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, June 2016, pp. 1135–1144. [Online]. Available: <https://www.acmweb.org/anthology/N16-3020/>

[16] M. Ribeiro Tulio, S. Singh, and C. Guestrin, "Local Interpretable Model-Agnostic Explanations (LIME): An Introduction," <https://www.oreilly.com/content/introduction-to-local-interpretable-model-agnostic-explanations-lime/>, August 2016, (Accessed on 01/14/2021).

[17] M. K. Alzaylaee, S. Y. Yerima, and S. Sezer, "Dl-droid: Deep learning based android malware detection using real devices," *Computers & Security*, vol. 89, p. 101663, 2020. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0167404819300161>

[18] X. Zhang, Y. Zhang, M. Zhong, D. Ding, Y. Cao, Y. Zhang, M. Zhang, and M. Yang, "Enhancing state-of-the-art classifiers with api semantics to detect evolved android malware," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 757–770. [Online]. Available: <https://doi.org/10.1145/3372297.3417291>

[19] S. Singh, B. Mishra, and S. Singh, "Detecting intelligent malware on dynamic android analysis environments," in *2015 10th International Conference for Internet Technology and Secured Transactions (ICITST)*, 2015, pp. 414–419.

[20] S. Singh, S. Singh, and B. Mishra, "Artificial user emulator to detect intelligent malware on android," *International Journal of Intelligent Computing Research*, vol. 6, pp. 640–646, 2015.

[21] N. Idika and A. P. Mathur, "A survey of malware detection techniques," *Purdue University*, vol. 48, pp. 2007–2, 2007.

[22] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical black-box attacks against machine learning," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 506–519. [Online]. Available: <https://doi.org/10.1145/3052973.3053009>

[23] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," in *International Conference on Learning Representations*, 2014. [Online]. Available: <http://arxiv.org/abs/1312.6199>

[24] I. n. Íñcer Romeo, M. Theodorides, S. Afroz, and D. Wagner, "Adversarially robust malware detection using monotonic classification," in *Proceedings of the Fourth ACM International Workshop on Security and Privacy Analytics*, ser. IWSA '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 54–63. [Online]. Available: <https://doi.org/10.1145/3180445.3180449>

[25] D. Li and Q. Li, "Adversarial deep ensemble: Evasion attacks and defenses for malware detection," *IEEE Transactions on Information Forensics and Security*, vol. 15, p. 3886–3900, 2020. [Online]. Available: <http://dx.doi.org/10.1109/TIFS.2020.3003571>

[26] J. Jin, A. Dundar, and E. Culurciello, "Robust Convolutional Neural Networks under Adversarial Noise," *arXiv e-prints*, p. arXiv:1511.06306, Nov. 2015.

[27] W. Xu, D. Evans, and Y. Qi, "Feature squeezing: Detecting adversarial examples in deep neural networks," *Proceedings 2018 Network and Distributed System Security Symposium*, 2018. [Online]. Available: <http://dx.doi.org/10.14722/ndss.2018.23198>

[28] J. Hendrik Metzen, T. Genewein, V. Fischer, and B. Bischoff, "On Detecting Adversarial Perturbations," *arXiv e-prints*, p. arXiv:1702.04267, Feb. 2017.

[29] C. Murphy, G. Kaiser, L. Hu, and L. Wu, "Properties of Machine Learning Applications for Use in Metamorphic Testing," in *20th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, July 2008, pp. 867–872.

[30] X. Xie, J. W. Ho, C. Murphy, G. Kaiser, B. Xu, and T. Y. Chen, "Application of Metamorphic Testing to Supervised Classifiers," in *9th International Conference on Quality Software*, 2009, pp. 135–144.

[31] X. Xie, J. W. Ho, C. Murphy, G. Kaiser, B. Xu, and T. Y. Chen, "Testing and validating machine learning classifiers by metamorphic testing," *Journal of Systems and Software*, vol. 84, no. 4, pp. 544–558, 2011, the Ninth International Conference on Quality Software. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0164121210003213>

[32] Y. Tian, K. Pei, S. Jana, and B. Ray, "DeepTest: Automated Testing of Deep-Neural-Network-Driven Autonomous Cars," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 303–314. [Online]. Available: <https://doi.org/10.1145/3180155.3180220>

[33] R. R. Mekala, G. E. Magnusson, A. Porter, M. Lindvall, and M. Diep, "Metamorphic detection of adversarial examples in deep learning models with affine transformations," in *Proceedings of the 4th International Workshop on Metamorphic Testing*, ser. MET '19. IEEE Press, 2019, p. 55–62. [Online]. Available: <https://doi.org/10.1109/MET.2019.00016>

[34] VirusTotal, "Analyze suspicious files and URLs to detect types of malware, automatically share them with the security community," <https://www.virustotal.com/#/home/upload>, (Accessed on 06/10/2020).

[35] A. Salem, S. Banescu, and A. Pretschner, "Don't Pick the Cherry: An Evaluation Methodology for Android Malware Detection Methods," *arXiv preprint arXiv:1903.10560*, March 2019. [Online]. Available: <http://arxiv.org/abs/1903.10560>

[36] T. Debize, "maaaaz/androwarn: Yet another static code analyzer for malicious android applications," <https://github.com/maaaaz/androwarn>, 2012, (Accessed on 06/10/2020).

[37] G. Gueguen, "Google Code Archive - Long-term storage for Google Code Project Hosting," 2012, (Accessed on 06/10/2020). [Online]. Available: <https://code.google.com/archive/p/androguard/>

[38] L. Li, D. Li, T. F. Bissyande, J. Klein, Y. Le Traon, D. Lo, and L. Cavallaro, "Understanding Android App Piggybacking: A Systematic Study of Malicious Code Grafting," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 6, pp. 1269–1284, June 2017. [Online]. Available: <https://doi.org/10.1109/TIFS.2017.2656460>

[39] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," in *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds. Curran Associates, Inc., 2017, pp. 4765–4774. [Online]. Available: <http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf>