Telemade: A Testing Framework for Learning-Based Malware Detection Systems

Wei Yang, Tao Xie
*Department of Computer Science, University of Illinois at Urbana-Champaign, USA
{weiyang3, taoxie}@illinois.edu

Abstract

Learning-based malware detectors may be erroneous due to two inherent limitations. First, there is a lack of differentiability: selected features may not reflect essential differences between malware and benign apps. Second, there is a lack of comprehensiveness: the used machine learning (ML) models are usually based on prior knowledge of existing malware (*i.e.*, training dataset) so malware can evolve to evade the detection. There is a strong need for an automated framework to help security analysts to detect errors in learning-based malware detection systems. Existing techniques to generate adversarial samples for learning-based systems (that take images as inputs) employ feature mutations based on feature vectors. Such techniques are infeasible to generate adversarial samples (*e.g.,* evasive malware) for malware detection systems because the synthesized mutations may break the inherent constraints posed by code structures of the malware, causing either crashes or malfunctioning of malicious payloads. To address the challenge, we propose Telemade, a **te**sting framework for **le**arning-based **ma**lware **de**tectors.

Introduction

Mobile malware grows exponentially as the number of apps on mobile app market increases. According to a recent malware security report (Malware Trends Report), every 4.2 seconds a new malware specimen emerges. To fight against malware, researchers adopt machine-learning-based techniques (Kong and Yan 2013) that learn discriminant features from analyzing semantics of malware.

Although machine learning (ML) algorithms bring impressive capabilities to malware detection systems, recent research (Rndic and Laskov 2014; Xu, Qi, and Evans 2016; Carmony et al. 2016)) finds ML algorithms presenting unexpected or incorrect behaviors in corner cases. In particular, the researchers find that learning-based systems can produce unexpected results to small, specially crafted perturbations. Such perturbations cause the learning-based systems to mis-classify these well-crafted examples. In safety/security-critical settings, such incorrect behaviors can lead to potentially disastrous consequences.

*This work was supported in part by National Science Foundation under grants no. CCF-1409423, CNS-1513939, CNS-1564274.

In this paper, we investigate the possibility of automatically producing corner-case testing inputs (*i.e.,* evasive malware variants) for learning-based malware detection systems to strengthen the robustness of malware detection. A key observation made in our research is that, features, which abstract concrete malicious behaviors, are fragile, and could be mutated (*i.e.,* changed). The susceptibility of such features makes it possible to produce corner-case inputs for malware detectors (when malware are properly mutated (Rndic and Laskov 2014; Xu, Qi, and Evans 2016; Carmony et al. 2016)). Our research suggests that *features that are unique to malware are not necessarily needed for forming malicious behaviors*. Such result is mainly due to two factors.

First, learning-based detectors often confuse non-essential features (*i.e.,* features that are not essential for forming malicious behaviors) in code clones as discriminative features. The prevalence of copy-paste practice in malware industry results in many code clones in malware samples (Chen et al. 2015). Because the same code has appeared in many malware instances, learning-based detectors may regard non-essential features (*e.g.,* minor implementation detail) in code clones as major discriminant factors (because the same pieces of code appear in many malware samples but not in benign apps). Learning-based detectors place higher weight on these features not because these features are essential to form malicious behaviors but because these features appear in malware much more frequently than in benign apps. In other words, the ML models confuse the statistical correlation (between code clones and malware) as a causal relationship (between essential features and malware). Our proposed testing techniques leverage such factor to mutate some of these non-essential features with higher weight in the ML models to evade detection.

Second, the features essential to form malicious behaviors are different for each malware family. Almost all existing learning-based malware detectors use a universal feature set to detect malicious samples for all malware families. For example, Drebin, recent malware detection work (Arp et al. 2014), uses a feature set containing 545,334 features. However, based on recent research results (Zhu and Dumitras 2016) mined from 1,068 research papers and malware documents, each malware family is associated with a distinct set of malware behaviors and concrete features. A re-

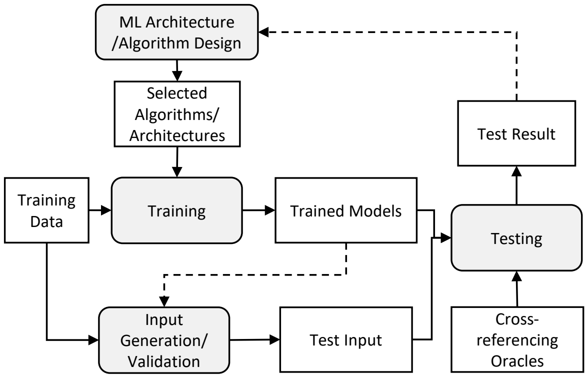

Figure 1: Overview of our testing framework for learning-based malware detectors

cent study (Roy et al. 2015) shows that such large feature set has numerous non-informative or even misleading features. Using a universal set of features for all malware families would result in a large number of non-essential features to characterize each family. Furthermore, as mentioned earlier, if these non-essential features are unique in some malware samples (for other reasons such as code clones), the trained detection model can perform poorly when mutating the values of the non-essential features is applied to malware samples.

The main challenge in building a testing framework for malware detection systems compared to learning-based systems for other formats (*e.g.,* images) is that mutating the inputs of malware detectors (*i.e.,* malware programs) is more sophisticated than mutating images. The mutations (on images) that are usually computed based on mathematical models are not suitable to mutate a malware program. Specifically, first, the mutations can destruct the original malicious behaviors in the program. The mutated malware program should maintain the original malicious purposes, and therefore simply converting the malware's feature values to another app's feature values is likely to break the maliciousness. For example, malicious behaviors are usually designed to be triggered under certain contexts (to avoid user attention and gain maximum profits (Yang et al. 2015)), and the controlling logic of the malware is too sophisticated (*e.g.,* via logic bombs and specific events) to be changed. Second, the mutations can simply crash the program. The mutated malware program should be robust enough to be installed and executed on mobile devices. Automatically mutating an app's feature values is likely to break the code structures and therefore cause the app to crash at runtime.

To address the challenge, we propose Telemade, a **te**sting framework for **le**arning-based **ma**lware **de**tectors, as shown in Figure 1. Such framework can be integrated into the development of a learning-based system. The existing approaches to debug/tune a learning-based system may require domain-specific knowledge of how to set 'magic numbers' in the ML models/algorithms. For example, developing a neural network requires a priori experience about how to tune the network structures (*e.g.,* the number of hidden layers and the connection of the hidden layers) and hyper parameters (*e.g.,* learning rate). In this work, we leverage the testing result provided by Telemade as feedback to debug/tune the ML models/algorithms. Although the framework of Telemade is general for all malware detectors, in our current implementation of Telemade, we focus on Android malware detectors as an example.

Test Input Generation

This section presents techniques of test input generation that produce corner-case inputs for learning-based malware detection systems. Such techniques are based on our prior work (Yang et al. 2017) on attacking malware detection systems.

Evolution and confusion strategies. Telemade aims to generate corner-case inputs for various types of ML algorithms/models. So instead of developing targeted malware to evade specific detection techniques, we propose a general mechanism of test input generation called **evolution strategy**: *mimicking and automating the evolution of malware*. Such input generation mechanism is based on the insight that the evolution process of malware reflects the strategies employed by malware authors to achieve a malicious purpose while evading detection. We also develop a targeted strategy called **confusion strategy**. The main idea of malware confusion strategy is to mimic the malware that can generally evade detection, *i.e.,* confusing the malware detectors by modifying the feature values that can be shared by malware and benign apps.

To realize the **evolution strategy**, we identify a feature set called *evolution feature set*. In the set, each feature is evolved either at the intra-family level or inter-family level. For each feature vector in the *evolution feature set*, we count the number of evolutions as the *evolution weight*, where the *intra-family evolution weight* is proportional to the number of evolutions at the intra-family level, and *inter-family evolution weight* is proportional to that at the inter-family level. The rationale is that if the feature type has already been evolved frequently under observation, it is more likely to be evolved according to the nature of the law (in the biological evolution process (Baxevanis and Ouellette 2004)).

To realize the **confusion strategy**, we identify a set of feature vectors that can be projected from both benign apps and malware as *confusion feature set*. For each feature in the confusion feature set, we count the number of benign apps that can be projected to the feature vector as the *confusion weight* of the feature vector. The rationale is that if more benign apps are projected to the feature, it is harder for the malware detector to label the apps with this feature as malicious.

The advantage of our test input generation strategies is that the strategies can produce a high percentages of feasible feature mutations (suggested in our evaluation), thus greatly enhancing the feasibility of the inputs. The insight is that feature mutations are less likely to break the apps when the mutations follow feature patterns (extracted from malware evolution histories and existing evasive malware) of existing malware.

Manifold-guided input generation. To guide the aforementioned input generation techniques to produce meaningful inputs (*i.e.,* mobile apps), we propose to construct manifold to check whether a generated input is meaningful to the problem domain. A manifold is a topological space, in which each point is surrounded by a locally Euclidean space. Various researchers (Narayanan and Mitter 2010) speculate that data relevant to a specific task tend to lie in the vicinity of a lower-dimension manifold. Such speculation indicates that we might be able to tell whether specific inputs are meaningful by checking whether they could fit in the manifold constructed from the training data.

Reconstructing the manifold and deriving the manifold-to-manifold distance might lead to efficiency problems. During manifold construction, determined by the construction algorithm, some properties among the training data might be fully or partially preserved, *i.e.,* invariants, which could be utilized for faster checking of new inputs. For instance, Isomap (Tenenbaum, De Silva, and Langford 2000) preserves the geodesic distance between each pair of points (*i.e.,*, the sum of edge lengths along the shortest path between two points). This property entails that, if an input under test is together used with the training data to construct a manifold, the shortest Euclidean distance between the point corresponding to the test input and the points representing the training data in the lower-dimensional space will be the same as that in the higher-dimensional input space. Thus, if the distances of test inputs to the manifold are used to judge whether those inputs fit in the manifold, we simply need to find the shortest Euclidean distance between those inputs and training data in the input space instead of constructing manifolds and measuring the distances among manifolds.

Another way to construct the manifold is to leverage autoencoders to identify intrinsic properties of the data (Vincent et al. 2008) and realize both manifold learning and checking. Autoencoders are neural networks with simpler hidden representations trained to forward inputs to outputs. An autoencoder $ae = d \circ e$ can be viewed as two parts: an encoder $e : \mathbb{S} \rightarrow \mathbb{H}$, which is trained to map from inputs to hidden representations and resembles to constructing the manifold from the training data, and a decoder $d : \mathbb{H} \rightarrow \mathbb{S}$, which is trained to recover inputs from hidden representations and resembles to the reverse process of manifold construction. The input space is denoted as $\mathbb{S}$, the hidden representation space is denoted as $\mathbb{H}$, and $\mathbb{H}$ has fewer dimensions than $\mathbb{S}$. Assume that $\mathbb{H}$ is large enough to embrace most hidden representations of normal inputs. Then the reconstruction error for training inputs by ae, which can be defined as the average Euclidean distance from the outputs of ae to the training inputs, should be reasonably low. Based on this property, the encoder resembles to normal inputs lying on the manifold. For unintended inputs, since their hidden representations are likely to clash with those of normal inputs due to limited space dimensions of $\mathbb{H}$, and e tries to recover from hidden representations based on training inputs, the outputs of e (also as the outputs of ae) should be different from those inputs, resulting in high reconstruction errors. Based on this property, the decoder resembles to unintended inputs being away from the manifold. Such properties enable unintended-input detection according to the reconstruction errors, essentially approximating the distances between inputs and the manifold. Researchers (Meng and Chen 2017) have applied this technique to defend against adversarial examples for neural network classifiers.

Test Input Validation

The input generation technique makes changes (to the original input program), which may cause the program to crash, cause undesired behaviors, or disable functionalities. We also propose an extra validation step to validate whether the input program functions properly. We take two measurements in examining the practicability of the generated inputs and filter out the impractical mutations. First, we perform impact analysis and targeted testing to check whether the malicious behaviors have been preserved. Second, we perform robustness testing to check whether the robustness of the program has been compromised.

Impact Analysis. Our impact analysis is based on the insight that the component-based nature of Android constrains the impact of mutations within certain components. We analyze the impact propagation among components by computing the inter-component communication graph (ICCG). We find three types of components that can 'constrain' the impact of the mutations within the components themselves. If any mutations are performed in components other than these three types of components, we discard such mutations. Based on the analysis, we identify three types of components that can be mutated with *minimized* impacts: (1) *isolated component*: no communications with other components; (2) *receiving-only component*: only receiving messages from other components; (3) *sending-only component*: only sending messages to other components. Isolated components and receiving-only components have no impacts to other components, and thus do not affect behaviors in other components. Mutating a sending-only component reduces an entry point to the subsequent components where the sending-only component may send messages with unwanted contents (*e.g.,* sensitive information), causing no crashes in the subsequent components but only eliminating some unwanted contexts (possibly legitimate contexts).

Targeted Testing. We develop two techniques to test the generated malware sample inputs in a targeted way. First, we create environmental dependencies by changing emulator settings or using mock objects/events to simulate the environment where the malicious behaviors are directly invoked to speed up the validation process. Second, to further validate the consistency of malicious behaviors when the triggering conditions are satisfied, we apply the instrumentation technique to insert logging functions at the locations of malicious method invocations. We therefore attain the log files before and after the mutation under the same context (*e.g.,* the same UI or system events and same inputs). Then, we automatically compare the two log files to check the consistency of malicious behaviors.

Robustness testing. We leverage random testing to check the robustness of a mutated program. In particular, we use

Monkey[1], a random user-event-stream generator for Android, to generate UI test sequences for mutated programs. Each mutated program is tested against 5,000 events randomly generated by Monkey to ensure that the program does not crash.

Testing Criterion

A learning-based system is structurally different from traditional software. Traditional software usually incorporates the program logic in program structures (*e.g.,* control flows, data flows). However, the program logic of a learning-based system is usually embedded in the arithmetic operations of formulas in the program; instead of writing the logic manually by a programmer, a learning-based system learns the program logic from training data. So it is insufficient to measure the effectiveness of test inputs for a learning-based system based on traditional test coverage such as statement coverage or branch coverage. DeepXplore (Pei et al. 2017) proposes a new coverage metric called neuron coverage. Neuron coverage counts the number of neurons activated in a neural network instead of counting the lines of program statements being covered by an execution.

In Telemade, the testing criterion is not limited to neuron coverage. We design the testing criterion based on the learning algorithm used in a learning-based system. Specifically, for a neural network, we take a step further to propose a neuron-combination coverage. Such coverage is based on the observation that activating all neurons in a neural network does not explore all the corner cases for the neural network. The combination of covered neurons provides a more accurate manifestation of program behaviors. Neuron-combination coverage measures how many combinations of neuron activations (denoted as C_A) have been covered by a set of test inputs. Assume that there are N neurons in the neural network. Then the neuron-combination coverage can be defined as $C_C = C_A/2^N$.

For instance, assume that there is a neural network with $N = 3$ neurons numbered $\{0, 1, 2\}$, respectively. The first test input activates neurons numbered $\{0, 1\}$ while the second test input activates neurons numbered $\{1, 2\}$. Per the definition of *neuron coverage*, only neurons numbered $\{0, 2\}$ are both activated and inactivated by the two test inputs, thus $C_A = N_A/N = 2/3$. Similarly, according to the definition of *neuron-combination coverage*, there are two different combinations of neuron activation status (*i.e.,* $\{0, 1\}$ and $\{1, 2\}$), thus $C_C = C_A/2^N = 2/2^3 = 1/4$.

Conclusion

In this paper, we have proposed Telemade, a testing framework fro learning-based malware detection systems. We discuss three broad thrusts: (1) techniques of test input generation; (2) validation of the generated inputs; (3) testing metrics for learning-based malware detection systems. We take Android malware detectors as an example and implement proposed input generation and validation techniques that can analyze and mutate Android apps.

[1] `http://developer.Android.com/tools/help/monkey.html`

References

Arp, D.; SPreitzenbarth, M.; Hubner, M.; Gascon, H.; and Rieck, K. 2014. DREBIN: effective and explainable detection of Android malware in your pocket. In *Proc. NDSS*.

Baxevanis, A. D., and Ouellette, B. F. F. 2004. *Bioinformatics: A Practical Guide to the Analysis of Genes and Proteins*. John WileySons.

Carmony, C.; Zhang, M.; Hu, X.; Bhaskar, A. V.; and Yin, H. 2016. Extract me if you can: Abusing PDF parsers in malware detectors. In *Proc. NDSS*.

Chen, K.; Wang, P.; Lee, Y.; Wang, X.; Zhang, N.; Huang, H.; Zou, W.; and Liu, P. 2015. Finding unknown malice in 10 seconds: Mass vetting for new threats at the Google-play scale. In *Proc. USENIX Security*, 659–674.

Kong, D., and Yan, G. 2013. Discriminant malware distance learning on structural information for automated malware classification. In *Proc. KDD*, 1357–1365.

Malware trends 2017. `https://www.gdatasoftware.com/blog/2017/04/29666-malware-trends-2017`.

Meng, D., and Chen, H. 2017. MagNet: a two-pronged defense against adversarial examples. In *Proc. CCS*, 135–147.

Narayanan, H., and Mitter, S. 2010. Sample complexity of testing the manifold hypothesis. In *Advances in Neural Information Processing Systems*, 1786–1794.

Pei, K.; Cao, Y.; Yang, J.; and Jana, S. 2017. DeepXplore: Automated whitebox testing of deep learning systems. In *Proc. SOSP*, 1–18.

Rndic, N., and Laskov, P. 2014. Practical evasion of a learning-based classifier: A case study. In *Proc. IEEE S & P*, 197–211.

Roy, S.; DeLoach, J.; Li, Y.; Herndon, N.; Caragea, D.; Ou, X.; Ranganath, V. P.; Li, H.; and Guevara, N. 2015. Experimental study with real-world data for Android app security analysis using machine learning. In *Proc. ACSAC*, 81–90.

Tenenbaum, J. B.; De Silva, V.; and Langford, J. C. 2000. A global geometric framework for nonlinear dimensionality reduction. *Science* 290(5500):2319–2323.

Vincent, P.; Larochelle, H.; Bengio, Y.; and Manzagol, P.-A. 2008. Extracting and composing robust features with denoising autoencoders. In *Proc. ICML*, 1096–1103.

Xu, W.; Qi, Y.; and Evans, D. 2016. Automatically evading classifiers. In *Proc. NDSS*.

Yang, W.; Xiao, X.; Andow, B.; Li, S.; Xie, T.; and Enck, W. 2015. AppContext: Differentiating malicious and benign mobile app behaviors using context. In *Proc. ICSE*, 303–313.

Yang, W.; Kong, D.; Xie, T.; and Gunter, C. A. 2017. Malware detection in adversarial settings: Exploiting feature evolutions and confusions in Android apps. In *Proc. ACSAC*, 288–302.

Zhu, Z., and Dumitras, T. 2016. FeatureSmith: Automatically engineering features for malware detection by mining the security literature. In *Proc. CCS*, 767–778.