Intriguing Properties of Adversarial ML Attacks in the Problem Space

# Fabio Pierazzi; Feargus Pendlebury; Jacopo Cortellazzi et al.

## 2019

This paper focuses on test-time evasion attacks in the so-called problem space, where the challenge lies in modifying real input-space objects that correspond to an adversarial feature vector

# Scholarcy Highlights

* Adversarial ML attacks are being studied extensively in multiple domains [11] and pose a major threat to the largescale deployment of machine learning solutions in securitycritical contexts
* This paper focuses on test-time evasion attacks in the so-called problem space, where the challenge lies in modifying real input-space objects that correspond to an adversarial feature vector
* We introduce the concept of side-effect features, which reveals connections between feature space and problem space, and enables principled reasoning about search strategies for problem-space attacks
* Building on our formalization, we propose a novel problem-space attack in the Android malware domain, which relies on automated software transplantation [10] and overcomes limitations of prior work in terms of semantics and preprocessing robustness (§III)
* We focus on evasion attacks [12, 16, 32], where the adversary modifies objects at test time to induce targeted misclassifications
* Our evasion attack relies on features associated with the transplanted code, and to preserve semantics we need conditional statements that always resolve to False at runtime; so, we must subvert static analysis techniques that may identify that this code is never executed

# Scholarcy Summary

## INTRODUCTION

Adversarial ML attacks are being studied extensively in multiple domains [11] and pose a major threat to the largescale deployment of machine learning solutions in securitycritical contexts.

Yang et al [75] may significantly alter the semantics of the program, and do not specify which preprocessing techniques they consider

These inspire us to propose, through our formalization, a novel problem-space attack in the Android malware domain that overcomes limitations of existing solutions.

This paper has two major contributions: We propose a novel formalization of problem-space attacks (§II) which lays the foundation for identifying key requirements and commonalities of different domains, proves necessary and sufficient conditions for problemspace attacks, and allows for the comparison of strengths and weaknesses of prior approaches—where existing strategies for adversarial malware generation are among the weakest in terms of attack robustness.

Building on our formalization, we propose a novel problem-space attack in the Android malware domain, which relies on automated software transplantation [10] and overcomes limitations of prior work in terms of semantics and preprocessing robustness (§III).

To foster future research on this topic, we discuss promising defense directions (§V) and responsibly release the code and data of our novel attack to other researchers via access to a private repository (§VII)

## PROBLEM-SPACE ADVERSARIAL ML ATTACKS

We focus on evasion attacks [12, 16, 32], where the adversary modifies objects at test time to induce targeted misclassifications.

We provide background from related literature on feature-space attacks (§II-A), and introduce a novel formalization of problem-space attacks (§II-B).

We highlight the main parameters of our formalization by instantiating it on both traditional feature-space and more recent problem-space attacks from related works in several domains (§II-C).

## Feature-Space Attacks

We remark that all definitions of feature-space attacks (§II-A) have already been consolidated in related work [11, 16, 21, 23, 31, 33, 44, 66]; we report them for completeness and as a basis for identifying relationships between featurespace and problem-space attacks in the following subsections.

A feature mapping is a function φ : Z −→ X ⊆ Rn that, given a problem-space object z ∈ Z, generates an n-dimensional feature vector x ∈ X , such that φ(z) = x.

In addition to the attack objective function, a considered problem-space domain may come with constraints on the modification of the feature vectors.

We observe that the feature-space attacks definition can be extended to ensure that the adversarial example is closer to the training data points

## Problem-Space Attacks

We present a novel formalization of problemspace attacks and introduces insights into the relationship between feature space and problem space.

If φ is not invertible and not differentiable, the challenge is to find a way to map the adversarial feature vector x ∈ X to an adversarial object z ∈ Z, by applying a transformation to z in order to produce z such that φ(z ) is “as close as possible” to x ; i.e., to follow the gradient towards the transformation that most likely leads to a successful evasion [38]

In problemspace settings such as software, the function φ is typically not invertible and not differentiable, so the search for transforming z to perform the attack cannot be purely gradient-based.

In the general case where the feature mapping φ is neither invertible nor differentiable, the adversary must perform a search in the problem-space that approximately follows the negative gradient in the feature space.

If Theorem 2 is satisfied only on a subset of feature dimensions Xi in X , which collectively create a subspace Xeq ⊂ X , the attacker can restrict the search space to Xeq, for which they know that an equivalent problem/feature-space manipulation exists

## Describing problem-space attacks in different domains

Yang et al [75] do not specify which preprocessing they are robust against, and their approach may significantly alter the semantics of the program—which may account for the high failure rate they observe in the mutated apps.

This inspired us to propose a novel attack that overcomes such limitations

## ATTACK ON ANDROID

Our formalization of problem-space attacks has allowed for the identification of weaknesses in prior approaches to malware evasion applicable to Android [60, 75].

We propose—through our formalization—a novel problem-space attack in this domain that overcomes these limitations, especially in terms of preserved semantics and preprocessing robustness

## Threat Model

The threat model must be defined in terms of attacker knowledge and capability, as in related literature [11, 19, 65].

While the attacker knowledge is represented in the same way as in the traditional feature-space attacks, their capability .

We represent the knowledge as a set θ ∈ Θ which may contain (i) training data D, (ii) the feature set X , (iii) the learning algorithm g, along with the loss function L minimized during training, (iv) the model parameters/hyperparameters w.

Zero Knowledge (ZK) black-box attacks, where the attacker has no information on the target system, but has some information on which kind of feature extraction is performed.

More details on the threat models can be found in [11, 65]

## Available Transformations

We use automated software transplantation [10] to extract slices of bytecode from benign donor applications and inject them into a malicious host, to mimic the appearance of benign apps and induce the learning algorithm to misclassify the malicious host as benign. An advantage of this process is that we avoid relying on a hardcoded set of transformations [e.g., 58]; this ensures adaptability across different application types and time periods.

We use automated software transplantation [10] to extract slices of bytecode from benign donor applications and inject them into a malicious host, to mimic the appearance of benign apps and induce the learning algorithm to misclassify the malicious host as benign..

We use automated software transplantation [10] to extract slices of bytecode from benign donor applications and inject them into a malicious host, to mimic the appearance of benign apps and induce the learning algorithm to misclassify the malicious host as benign.1

An advantage of this process is that we avoid relying on a hardcoded set of transformations [e.g., 58]; this ensures adaptability across different application types and time periods.

Malicious semantics preserved by construction through use of AST-based transplantation.

(i) Code is realistic by construction through use of automated software transplantation.

The host entry point LH is inserted into a randomly chosen function among those of the selected class, to avoid creating a pattern that might be identified by an analyst

## Preserved Semantics

Given an application z and its modified version z , we aim to ensure that z and z lead to the same dynamic execution, i.e., the malicious behavior of the application is preserved.

We enforce this by construction by wrapping the newly injected execution paths in conditional statements that always return False.

This guarantees the newly inserted code is never executed at runtime—so users will not notice anything odd while using the modified app.

An intent-filter could declare the app as an eligible option for reading PDF files; whenever attempting to open a PDF file, the user would be able to choose the host app, which would trigger an Activity defined in the transplanted benign bytecode—violating our constraint of preserving dynamic functionality

## Robustness to Preprocessing

Program analysis techniques that perform redundant code elimination would remove unreachable code.

Our evasion attack relies on features associated with the transplanted code, and to preserve semantics we need conditional statements that always resolve to False at runtime; so, we must subvert static analysis techniques that may identify that this code is never executed.

We achieve this by relying on opaque predicates [51], i.e., carefully constructed obfuscated conditions where the outcome is always known at design time, but the actual truth value is difficult or impossible to determine during a static analysis.

We refer the reader to Appendix D for a detailed description of how we generate strong opaque predicates and make them look legitimate

## Plausibility

An example is satisfactorily plausible if it resembles a real, functioning Android application.

Our methodology aims to maximize the plausibility of each generated object by injecting full slices of bytecode from real benign applications.

We can conclude that plausibility is guaranteed by construction thanks to the use of automated software transplantation [10].

This contrasts with other approaches that inject standalone API calls and URLs or no-op operations [e.g., 60] that are completely orphaned and unsupported by the rest of the bytecode.

We practically assess that each mutated app still functions properly after modification by installing and running it on an Android emulator.

We are unable to thoroughly explore every path of the app in this automated manner, it suffices as a smoke test to ensure that we have not fundamentally damaged the structure of the app

## Search Strategy

We propose a gradient-driven search strategy based on a greedy algorithm, which aims to follow the gradient direction by transplanting a gadget with benign features into the malicious host.

There are two main phases: Initialization (Ice-Box Creation) and Attack (Adversarial Program Generation).

The main reason for this, instead of looking for gadgets on-the-fly, is to have an immediate estimate of the side-effect features when each gadget is considered for transplantation.

We sort the gadgets in order of decreasing negative contribution, which ideally leads to a faster convergence of z’s score to a benign value

We filter this candidate list to include gadgets only if they satisfy some practical feasibility criteria.

## Experimental Settings

We create a prototype of our novel problemspace attack (§III) using a combination of Python for the ML functionality and Java for the program analysis operations; in particular, to perform transplantations in the problem-space we rely on FlowDroid [9], which is based on Soot [68].

As defined in the threat model (§III-A), we consider the DREBIN classifier [8], based on a binary feature space and a linear SVM, and its recently proposed hardened variant, Sec-SVM [23], which requires the attacker to modify more features to perform an evasion.

We perform a random split of the dataset to simulate absence of concept drift [55]; this represents the most challenging scenario for an attacker, as they aim to mutate a test object coming from the same distribution as the training dataset.

Since these errors are related on implementation limitations of the FlowDroid research prototype, and not conceptual errors, the success rates in the remainder of this section refer only to applications that did not throw FlowDroid exceptions during the transplantation phase

## Evaluation

We analyze the performance of our Android problem-space attack in terms of runtime cost and successful evasion rate.

An attack is successful if an app z, originally classified as malware, is mutated into an app z that is classified as goodware and satisfies the problem-space constraints.

While the plausibility problem-space constraint is satisfied by design by transplanting only realistic existing code, it is informative to analyze how the statistics of the evasive malware relate to the corresponding distributions in benign apps.

The low runtime cost suggests that it is feasible to perform this attack at scale and reinforces the need for new defenses in this domain

## ON ATTACK AND RESULTS

We provide some deeper discussion on the results of our novel problem-space attack. Android Attack Effectiveness.

We conclude that it is practically feasible to evade the state-of-the-art Android malware classifier DREBIN [8] and its hardened variant, Sec-SVM [23], and that we are able to automatically generate realistic and inconspicuous evasive adversarial applications, often in less than 2 minutes.

This shows for the first time that it is possible to create realistic adversarial applications at scale.

Our Android evasion attack (§III) demonstrates for the first time that it is feasible to evade feature-space defenses such as Sec-SVM in the problemspace—and to do so en masse

## RELATED WORK

Many works on problem-space attacks have been explored on different domains: text [3, 43], PDFs [22, 41, 45, 46, 74], Windows binaries [38, 59, 60], Android apps [23, 31, 75], NIDS [6, 7, 20, 28], ICS [76], and Javascript source code [58]

Each of these studies has been conducted empirically and followed some inferred best practices: while they share many commonalities, it has been unclear how to compare them and what are the most relevant characteristics that should be taken into account while designing such attacks.

Our attack does not require hardcoding and by design is resilient against traditional non-ML program analysis techniques

## VIII\_ CONCLUSIONS

Since the seminal work that evidenced intriguing properties of neural networks [66], the community has become more widely aware of the brittleness of machine learning in adversarial settings [11].

To better understand real-world implications across different application domains, we propose a novel formalization of problem-space attacks as we know them today, that enables comparison between different proposals and lays the foundation for more principled designs in subsequent work.

We uncover new relationships between feature space and problem space, and provide necessary and sufficient conditions for the existence of problem-space attacks.

Our novel problem-space attack shows that automated generation of adversarial malware at scale is a realistic threat—taking on average less than 2 minutes to mutate a given malware example into a variant that can evade a hardened state-of-the-art classifier

## Theorem Proofs

We need to prove that, with Equation 12, the condition is sufficient for the attacker to find a problem-space transformation that misclassifies the object

Another way to write Equation 12 is to consider that the attacker knows transformations that affect individual features only.

For any object z ∈ Z with features φ(z) = x ∈ X , for any feature-space dimension Xi of X , and for any value v ∈ domain(Xi), let us assume the attacker knows a valid problem-space transformation sequence T : T(z) |= Γ, φ(T(z)) = x , such that: xi = xi + v, xi ∈ x, xi ∈ x (13)

These two equations refer to the existence of a problem-space transformation T that affects only one feature Xi in X by any amount v ∈ domain(Xi).

Equation 12 must be valid for all possible perturbations within the considered feature space

## Opaque Predicates Generation

We use opaque predicates [4] as inconspicuous conditional statements always resolving to False to preserve dynamic semantics of the Android applications.

To consistently generate NP-Hard k-SAT problems we use Random k-SAT [61] in which there are 3 parameters: the number of variables n, the number of clauses m, and the number of literals per clause k.

To construct a 3-SAT formula, m clauses of length 3 are generated by randomly choosing a set of 3 variables from the n available, and negating each with probability 50%.

An empirical study by Selman et al [61] showed that n should be at least 40 to ensure the formulas are hard to resolve

They show that formulas with too few clauses are under-constrained while formulas with too many clauses are over-constrained, both of which reduce the search time.

An example of a generated opaque predicate is shown in Listing 1

## DREBIN and Sec-SVM Implementation Details

We have access to a working Python implementation of DREBIN based on sklearn, androguard, and aapt, and we rely on LinearSVC classifier with C=1.

To train the Sec-SVM, we perform an extensive hyperparameter grid-search: with Adam and Stochastic Gradient Descent (SGD) optimizers; training epochs of 5 to 100; batch sizes from 20 to 212; learning rate from 100 to 10−5.

We start from k = wmax, where wmax = maxi(wi) for all features i; we continue reducing k until we reach a weight distribution similar to that reported in [23], while allowing a maximum performance loss of 2% in AUROC.

In this way, we identify the best value for our setting as k = 0.2.

We have verified with Demontis et al [23] the correctness of our SecSVM implementation and its performance, for the analysis performed in this work

## Attack Algorithms

Algorithm 1 and Algorithm 2 describe in detail the two main phases of our search strategy: organ harvesting and adversarial program generation.

For the sake of simplicity, we describe a low-confidence attack, i.e., the attack is considered successful as soon as the classification score is below zero.

It is immediate to consider high-confidence variations.

Note that when using the minimal injection host zmin to calculate the features that will be induced by a gadget, features in the corresponding feature vector xmin should be noted and dealt with .

In our case xmin contained the following three features: { "intents::android\_intent\_action\_MAIN":1, "intents::android\_intent\_category\_LAUNCHER":1, "activities::\_MainActivity":1}

## Findings

A recent promising direction by Incer et al [34] studies the use of monotonic classifiers, where adding features can only increase the decision score; such classifiers require non-negligible time towards manual feature selection, and—at least in the context of Windows malware [34]—they suffer from high false positives and an average reduction in detection rate of 13%

## FlowDroid Errors

We performed extensive troubleshooting of FlowDroid [9] to reduce the number of transplantation failures, and the transplantations without FlowDroid errors in the different configurations are as follows: 89.5% for SVM (L), 85% for SVM (H), 80.4% for Sec-SVM (L), 73.3% for Sec-SVM (H)

These failures are only related to bugs and corner cases of the research prototype of FlowDroid, and do not pose any theoretical limitation on the attacks.

Some examples of the errors encountered include: inability to output large APKs when the app’s SDK version is less than 21; a bug triggered in AXmlWriter, the third party component used by Algorithm 1: Initialization (Ice-Box Creation).

Output: Ice-box of harvested organs with feature vectors.

While length(ice-box[i])< nd do zj ← Randomly sample a benign app with feature xi = 1.

17 return ice-box; Algorithm 2: Attack (Adv. Program Generation) malware if h(x) > 0, otherwise as goodware.

28 else return Failure; FlowDroid, when modifying app Manifests; and FlowDroid injecting system libraries found on the classpath when they should be excluded