Intriguing Properties of Adversarial ML Attacks in the Problem Space

Fabio Pierazzi*[1], Feargus Pendlebury*[1,2,3], Jacopo Cortellazzi[1], Lorenzo Cavallaro[1]

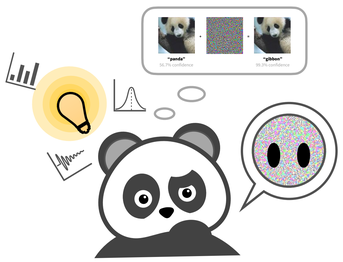
Based on the conference paper presented at IEEE Security and Privacy, 2020

[1] King's College London, UK
[2] Royal Holloway, University of London, UK
[3] The Alan Turing Institute, UK
https://s2lab.kcl.ac.uk/intriguing

Overview

Machine Learning (ML) classifiers have demonstrated impressive performance in various domains, particularly in discriminating between **malicious** and **benign** behavior in security-sensitive settings (e.g., malware detection, anomaly detection, code attribution, platform abuse). However, it has been shown that adversaries can **attack** classifiers by carefully altering input data in order to manipulate their outputs.

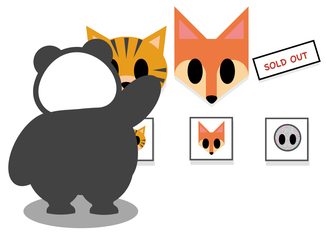
When there's a need to evade detection…



…feature-space perturbations make a good disguise…

A well-studied example of an adversarial ML attack is the **evasion attack.** Using a gradient-driven methodology, it's possible to calculate an ideal perturbation δ^* to apply to the original object x which will result in the target classifier misidentifying it as a different class.

However, in many settings it is **not possible** to convert this ideal feature vector back into a real problem-space object due to the **inverse feature mapping problem**. In these cases, the ideal transformations required to induce δ^* in x are simply **not available** because of various constraints that exist only in the problem space (e.g., plausibility).



…but in the problem space, the ideal transformations might not be available.

In this work we clarify the relationship between feature-space and problem-space and propose a **general formalization** for problem-space attacks, including a comprehensive set of constraints to consider. This allows us to highlight the strengths and weaknesses of different approaches and better formulate novel attacks.

Problem-Space Constraints

In order to formally express realizable attacks, we identify four main sets of constraints common to all problem-space manipulations:

$\mathcal{T}$ **Available transformations:** the viable modifications which can be performed in the problem space by the attacker (e.g., only addition and not removal).

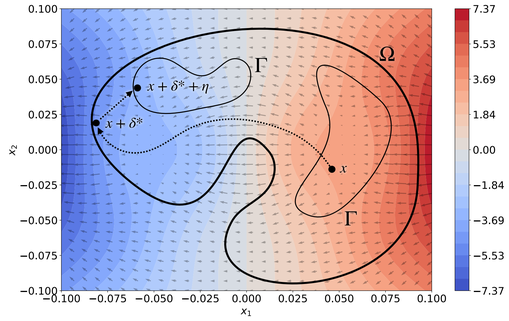
Π **Plausibility:** how to determine if the generated example is realistic upon manual inspection (e.g., an adversarial image looks like a valid image from the training distribution).

Υ **Preserved semantics:** behaviour that should remain during mutation, w.r.t. specific feature abstractions the attacker aims to be resilient against (e.g., in programs, the same dynamic call traces). Semantics may also be preserved by construction.

Λ **Robustness to preprocessing:** robustness against non-ML techniques that could trivially defeat the attack (e.g., filtering in images, dead code elimination in programs).

The Nature of Side-Effects

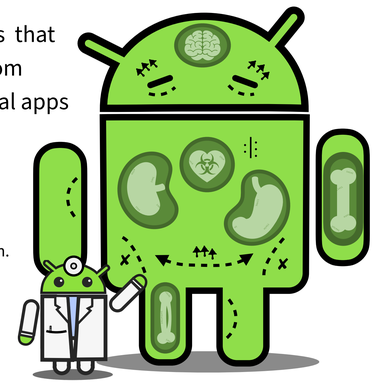
Satisfying problem-space constraints often produces side-effect features which can prevent optimal gradient-driven attacks.



After following the gradient-based attack $x + \delta^*$ derived from x on the *feasible feature space* Ω, a necessary projection to fit into the *feasible problem space* Γ results in additional features η which may have positive or negative effects on the classification of the attack point.

Evading Android Malware Detectors

With this formalization, we can design a new attack to evade Android classifiers that overcomes limitations of past solutions in this domain. We borrow methods from ***automated software transplantation*** to transplant benign code slices from real apps to a malicious host and trick the detector.



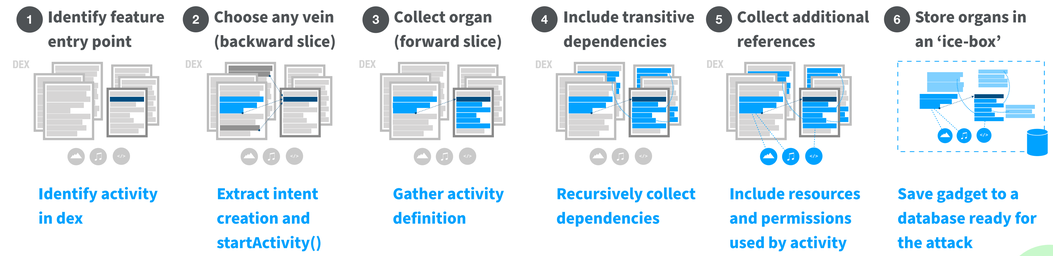
$\mathcal{T}$ Code addition through automated software transplantation.

Π Only functional code is injected rather than orphaned urls, api calls, etc. Statistical footprint (e.g, code size) remains close to the benign distribution.

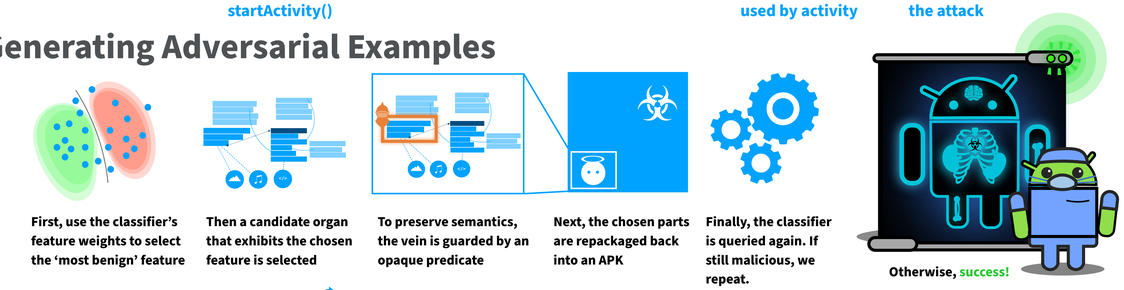
Υ Malicious semantics are preserved by construction using opaque predicates (new benign behaviour is never executed at runtime).

Λ Robust to: removal of redundant code, undeclared variables, unlinked resources, undefined references, naming conflicts, no-op instructions.

Harvesting Benign Gadgets



Generating Adversarial Examples



Research partially funded by EPSRC grants EP/L022710/2 and EP/P009301/1