Intriguing Properties of Adversarial ML Attacks in the Problem Space

Fabio Pierazzi*†, Feargus Pendlebury*†‡, Jacopo Cortellazzi†, Lorenzo Cavallaro†
† King's College London, UK, ‡ Royal Holloway, University of London, UK
{*fabio.pierazzi, feargus.pendlebury, jacopo.cortellazzi, lorenzo.cavallaro*}*@kcl.ac.uk*

***Abstract*—Recent research efforts on adversarial ML have investigated problem-space attacks, focusing on the generation of real evasive objects in domains where, unlike images, there is no clear inverse mapping to the feature space (e.g., software). However, the design, comparison, and real-world implications of problem-space attacks remain underexplored.**

This paper makes two major contributions. First, we propose a general formalization for adversarial ML evasion attacks in the problem-space, which includes the definition of a comprehensive set of constraints on available transformations, preserved semantics, absent artifacts, and plausibility. We shed light on the relationship between feature space and problem space, and we introduce the concept of *side-effect features* as the by-product of the inverse feature-mapping problem. This enables us to define and prove necessary and sufficient conditions for the existence of problem-space attacks. We further demonstrate the expressive power of our formalization by using it to describe several attacks from related literature across different domains.

Second, building on our general formalization, we propose a novel problem-space attack on Android malware that overcomes past limitations in terms of semantics and artifacts. Experiments on a dataset with 170K Android apps from 2017 and 2018 show the practical feasibility of evading a state-of-the-art malware classifier, DREBIN [5], along with its hardened version, SecSVM [18]. Our results demonstrate that "adversarial-malware as a service' is a realistic threat, as we automatically generate thousands of realistic and inconspicuous adversarial applications at scale, where on average it takes only a few minutes to generate an adversarial app. Yet, out of the 1300+ papers on adversarial ML published in the past six years, roughly 35 focus on malware [12]—and many remain only in the feature space.

Our formalization of problem-space attacks paves the way to more principled research in this domain. We responsibly release the code and dataset of our novel attack to other researchers, to encourage future work on defenses in the problem space.

***Index Terms*—adversarial machine learning; problem space; input space; malware; program analysis; evasion.**

I. Introduction

Adversarial ML attacks are being extensively studied in multiple domains [8] and pose a major threat to the large-scale deployment of machine learning solutions in security-critical 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 in order to correspond to an adversarial feature vector. The main challenge resides in the *inverse feature-mapping problem* [9, 10, 26, 40, 41, 51] since in many settings it is not possible to convert a feature vector into a problem-space object because the feature-mapping function is neither invertible nor differentiable. In addition, the modified problem-space object needs to be a valid, inconspicuous member of the considered domain, and robust to non-ML detection attempts (e.g., program analysis). Existing work investigated problem-space attacks on text [3, 37], malicious PDFs [9, 34, 35, 39, 40, 64], Android malware [18, 53, 65], Windows malware [32], ICS [66], source code attribution [51], malicious Javascript [22], and eyeglass frames [55]. However, while there is a good understanding on how to perform feature-space attacks [13], it is less clear what the requirements are for an attack in the problem space, and how to compare strengths and weaknesses of existing solutions in a principled way.

In this paper, we propose a novel general formalization of problem-space attacks, which identifies key requirements and commonalities among different domains. We identify four major categories of constraints to be defined at design time: which problem-space *transformations* are available to be performed automatically while looking for an adversarial variant; which object *semantics* must be preserved between the original and its adversarial variant; which transformation *artifacts* must be prevented by construction, so that adversarial objects cannot be easily detected through non-ML techniques (e.g., dead code elimination in software); and how to ensure that the generated object is a *plausible* member of the input distribution, especially upon manual inspection. We introduce the concept of *side-effect features* as the by-product of trying to generate a problem-space transformation that perturbs the feature space in a certain direction. Defining side-effect features allows us to shed more light on the relationships between feature space and problem space: we define and prove necessary and sufficient conditions for the existence of problem-space attacks, and identify two main types of search strategies (gradient-driven and problem-driven) for generating adversarial objects in the problem space.

We further use our formalization to describe several interesting attacks proposed in both problem space and feature space. This analysis evidenced that promising prior problem-space attacks in the Android malware domain [25, 53, 65] suffer from some limitations, especially in terms of semantics and artifacts. Grosse et al. [25] only add individual features to the Manifest, which preserves semantics but may leave artifacts (e.g., unused permissions); moreover, they are constrained bya maximum feature-space perturbation, which we show is less relevant for problem-space attacks. Rosenberg et al. [53] leave artifacts during the app transformation which are easily

*Equal contribution.

detected through lightweight non-ML techniques. Yang et al. [65] may significantly alter the semantics of the program (which may account for the high failure rate observed in their mutated apps), and do not specify which artifacts they avoid. Quiring et al. [51] propose another closely related approach using adversarial ML for misleading authorship attribution; although not focused on malware detection, it performs code transformations in the source code, where the authors hardcode simple transformations that change the program stylometric properties (e.g., from *for* cycle to *while* loop). Although promising, this is insufficient in the context where one aims to evade a malware classifier that captures program semantics. These inspires us to propose, through our formalization, a novel problem-space attack in the Android malware domain that overcomes limitations of existing solutions.

In summary, this paper has two major contributions:

- We propose a novel general formalization of problem-space attacks (§II) which defines key requirements and commonalities of different domains, proves necessary and sufficient conditions for problem-space 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. 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 transplantations [7] and overcomes limitations of prior work in terms of semantics and artifacts. We experimentally demonstrate (§IV) on a dataset of 170K apps from 2017-2018 that it is feasible for an attacker to evade a state-of-the-art malware classifier, DREBIN [5], and its hardened version, SecSVM [18]. The time required to generate an adversarial example is in the order of minutes, thus demonstrating that the "adversarial-malware as a service" scenario is a realistic threat, and existing defenses are not sufficient.

To foster future research on this topic, we responsibly release the code and data of our novel attack to other researchers through email (§VII).

II. Problem-Space Adversarial ML Attacks

We focus on *evasion attacks* [9, 13, 26], where the adversary does not have access to the training data and modifies objects at test time to induce targeted misclassifications. We provide background from related literature on *feature-space* attacks (§II-A), and then introduce a novel formalization of *problem-space* attacks (§II-B). Finally, we highlight the main parameters of our formalization—where we demonstrate its generality by instantiating it on both traditional feature-space and more recent problem-space attacks from related works in several domains (§II-C). Threat modeling based on attacker knowledge and capability is the same as in related work [8, 16, 58], and is reported in Appendix B for completeness. To ease readability, Appendix A reports a symbol table.

A. Feature-Space Attacks

We remark that all definitions of feature-space attacks (§II-A) have already been consolidated in related work [8, 13, 17, 18, 25, 27, 38, 59]; we report them for completeness and as a basis for identifying relationships between feature-space and problem-space attacks in the following subsections.

We consider a *problem space* $\mathcal{Z}$ (also referred to as *input space*) that contains objects of a considered domain (e.g., images [13], audio [14], programs [51], PDFs [39]). We assume that each object $z \in \mathcal{Z}$ is associated with a ground-truth label $y \in \mathcal{Y}$, where $\mathcal{Y}$ is the space of possible labels. Machine learning algorithms mostly work on numerical vector data [11], hence the objects in $\mathcal{Z}$ must be transformed into a suitable format for ML processing.

Definition 1 (Feature Mapping)**.** A *feature mapping* is a function $\varphi : \mathcal{Z} \longrightarrow \mathcal{X} \subseteq \mathbb{R}^n$ that, given a problem-space object $z \in \mathcal{Z}$, generates an n-dimensional feature vector $\boldsymbol{x} \in \mathcal{X}$, such that $\varphi(z) = \boldsymbol{x}$. This definition includes also *implicit/latent* mappings, where the features are not observable in input but are instead implicitly computed by the model (e.g., deep learning [23]).

Definition 2 (Discriminant Function)**.** Given an m-class machine learning classifier $g : \mathcal{X} \longrightarrow \mathcal{Y}$, a *discriminant function* $h_i : \mathcal{X} \times \mathcal{Y} \longrightarrow \mathbb{R}$ outputs a real number $h_i(\boldsymbol{x})$ that represents the fitness of object $\boldsymbol{x}$ to class $i \in \mathcal{Y}$. Higher outputs of the discriminant function h_i represent better fitness to class i. In particular, the predicted label of an object $\boldsymbol{x}$ is $g(\boldsymbol{x}) = \hat{y} = \arg\max_{i \in \mathcal{Y}} h_i(\boldsymbol{x})$.

The purpose of a *targeted* feature-space attack is to modify an object $\boldsymbol{x} \in \mathcal{X}$ with assigned label $y \in \mathcal{Y}$ to an object $\boldsymbol{x}'$ that is classified to a target class $t \in \mathcal{Y}$, $t \neq y$ (i.e., to modify $\boldsymbol{x}$ so that it is misclassified as a target class t). The attacker can identify a perturbation $\boldsymbol{\delta}$ to modify $\boldsymbol{x}$ so that $g(\boldsymbol{x}+\boldsymbol{\delta}) = t$ by optimizing a carefully-crafted *attack objective function*. We refer to the definition of attack objective function in Carlini and Wagner [13] and in Biggio and Roli [8], which takes into account *high-confidence* attacks and multi-class settings.

Definition 3 (Attack Objective Function)**.** Given an object $\boldsymbol{x} \in \mathcal{X}$ and a target label $t \in \mathcal{Y}$, an *attack objective function* $f_t : \mathcal{X} \times \mathcal{Y} \longrightarrow \mathbb{R}$ is defined as follows:

$$f_t(\boldsymbol{x}) = \max_{i \neq t} \{h_i(\boldsymbol{x})\} - h_t(\boldsymbol{x}) \tag{1}$$

In particular, $\boldsymbol{x}$ is classified as a member of t if and only if $f_t(\boldsymbol{x}) < 0$. Optionally, an adversary can enforce a *desired attack confidence* $\kappa \in \mathbb{R}$ such that the attack is considered successful if and only if $f_t(\boldsymbol{x}) < -\kappa$.

The intuition to perform an adversarial attack in the feature space is to minimize f_t by modifying $\boldsymbol{x}$ in directions that follow the negative gradient of f_t, i.e., to get $\boldsymbol{x}$ closer to the target class t.

In addition to the attack objective function, a considered problem-space domain may also come with constraints on the modification of the feature vectors. For example, in the image domain the value of pixels must be bounded between 0 and 255 [13]; in software, some features in $\boldsymbol{x}$ may only be added but not removed (e.g., API calls [18]).

Definition 4 (Feature-Space Constraints)**.** We define Ω as the set of *feature-space constraints*, i.e., a set of constraints on the possible feature-space modifications. The set Ω reflects the requirements of realistic problem-space objects. Given an object $\boldsymbol{x} \in \mathcal{X}$, any modification of its feature values can be represented as a *perturbation vector* $\boldsymbol{\delta} \in \mathbb{R}^n$; if $\boldsymbol{\delta}$ satisfies Ω, we write $\boldsymbol{\delta} \models \Omega$.

As examples of feature-space constraints, in the image domain [e.g., 8, 13] the perturbation $\boldsymbol{\delta}$ is subject to an upper bound based on l_p norms ($||\boldsymbol{\delta}||_p \leq \delta_{max}$), to preserve similarity to the original object; in the software domain [e.g., 18, 25], only some features of $\boldsymbol{x}$ may be modified, such that $\boldsymbol{\delta}_{lb} \preceq \boldsymbol{\delta} \preceq \boldsymbol{\delta}_{ub}$ (where $\boldsymbol{\delta}_1 \preceq \boldsymbol{\delta}_2$ implies that each element of $\boldsymbol{\delta}_1$ is $\leq$ the corresponding i-th element in $\boldsymbol{\delta}_2$).

We can now formalize the traditional feature-space attack as in related work [8, 9, 13, 18, 46].

Definition 5 (Feature-Space Attack)**.** Given a machine learning classifier g, an object $\boldsymbol{x} \in \mathcal{X}$ with label $y \in \mathcal{Y}$, and a target label $t \in \mathcal{Y}$, $t \neq y$, the adversary aims to identify a perturbation vector $\boldsymbol{\delta} \in \mathbb{R}^n$ such that $g(\boldsymbol{x} + \boldsymbol{\delta}) = t$. The desired perturbation can be achieved by solving the following optimization problem:

$$\boldsymbol{\delta}^* = \arg\min_{\boldsymbol{\delta} \in \mathbb{R}^n} \quad f_t(\boldsymbol{x} + \boldsymbol{\delta}) \tag{2}$$

$$\text{subject to:} \quad \boldsymbol{\delta} \models \Omega \tag{3}$$

A feature-space attack is successful if $f_t(\boldsymbol{x} + \boldsymbol{\delta}^*) < 0$ (or less than $-\kappa$, if a desired attack confidence is enforced).

Without loss of generality, we observe that the feature-space attacks definition can be extended to ensure that the adversarial example is closer to the training data points (e.g., through the λ-parameter tuning in [9]).

B. Problem-Space Attacks

This section presents a novel formalization of problem-space attacks and introduces insights into the relationship between feature space and problem space.

Inverse Feature-Mapping Problem. The major challenge that complicates (and, in most cases, prevents) the direct applicability of gradient-driven feature-space attacks to find problem-space adversarial examples is the so-called *inverse feature-mapping problem* [9, 10, 26, 40, 41, 51]. As an extension, Quiring et al. [51] discuss the *feature-problem space dilemma*, which highlights the difficulty of moving in both directions: from feature space to problem space, and from problem space to feature space. In most cases, the feature mapping function φ is not bijective, i.e., *not invertible* and *not surjective*. This means that given $z \in \mathcal{Z}$ with features $\boldsymbol{x}$, and a feature-space perturbation $\boldsymbol{\delta}^*$, there is no one-to-one mapping that allows going from $\boldsymbol{x} + \boldsymbol{\delta}^*$ to an adversarial problem-space object z'. Nevertheless, there are two additional scenarios. If φ is not invertible but is *differentiable*, then it is possible to backpropagate the gradient of $f_t(\boldsymbol{x})$ from $\mathcal{X}$ to $\mathcal{Z}$ to derive how the input can be changed in order to follow the negative gradient (e.g., to know which input pixels to perturbate to follow the gradient in the deep-learning latent feature space). If φ is not invertible and not differentiable, then the challenge becomes finding a way to map the adversarial feature vector $\boldsymbol{x}' \in \mathcal{X}$ to an adversarial object $z' \in \mathcal{Z}$, by applying a transformation to z in order to produce z' such that $\varphi(z')$ is "as close as possible" to $\boldsymbol{x}'$; i.e., to follow the gradient towards the transformation that likely achieves success of the adversarial attack [32]. In problem-space settings such as programs, 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 this section, we consider the general case in which the feature mapping φ is not differentiable and not invertible (i.e., the most challenging setting), and we refer to this context to formalize problem-space evasion attacks.

First, we define a *problem-space transformation* operator through which we can alter problem-space objects. Due to their generality, we adapt the code transformation definitions from the *compiler engineering* literature [1, 51] to formalize general problem-space transformations.

Definition 6 (Problem-Space Transformation)**.** A problem-space transformation $T : \mathcal{Z} \longrightarrow \mathcal{Z}, z \mapsto z'$ takes as input a problem-space object $z \in \mathcal{Z}$ and modifies it to $z' \in \mathcal{Z}$. We refer to the following notation: $T(z) = z'$.

The possible problem-space transformations are either *addition*, *removal*, or *modification* (i.e., combination of addition and removal). In the case of programs, *obfuscation* is a special case of modification.

Definition 7 (Transformation Sequence)**.** A transformation sequence $\mathbf{T} = T_1 \circ T_2 \circ \cdots \circ T_n$ is the subsequent application of multiple problem-space transformations to a problem-space object z.

Intuitively, given a problem-space object $z \in \mathcal{Z}$ with label $y \in \mathcal{Y}$, the purpose of the adversary is to find a transformation sequence $\mathbf{T}$ such that the transformed object $\mathbf{T}(z)$ is classified into any target class t chosen by the adversary ($t \in \mathcal{Y}$, $t \neq y$). A possible way to achieve such a transformation would be to first compute a feature-space perturbation $\boldsymbol{\delta}^*$, and then modify the problem-space object z so that features corresponding to $\boldsymbol{\delta}^*$ are carefully altered. However, 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. However, this search is not unconstrained, because the adversarial problem-space object $\mathbf{T}(z)$ must be realistic.

Problem-Space Constraints. Given a problem-space object $z \in \mathcal{Z}$, a transformation sequence $\mathbf{T}$ must lead to an

object $z' = \mathbf{T}(z)$ that is valid and realistic. To express this formally, we identify four main types of constraints common to any problem-space attack:

1) *Available transformations*, which describe which modifications can be performed in the problem-space by the attacker (e.g., only addition and not removal).
2) *Preserved semantics*, which describe which semantics are to be preserved while mutating z to z', with respect to specific feature abstractions which the attacker aims to be resilient against (e.g., in programs, the transformed object may need to produce the same dynamic call traces). Semantics may also be preserved by construction [e.g., 51].
3) *Plausibility* (or *Inconspicuousness*), which describes which (qualitative) properties must be preserved in mutating z to z', so that z appears realistic upon manual inspection. For example, often an adversarial image must look like a valid image from the training distribution [13]; a program's source code must look manually written and not artificially or inconsistently altered [51]. In the general case, verification of plausibility may not be automated, and may require human analysis.
4) *Absent artifacts*, which is the set of artifacts that are prevented *by design* while transforming z to z'. In other words, this type of constraint determines robustness against non-ML techniques that could defeat the attack (e.g., filtering in images, dead code removal in programs).

These constraints have been sparsely mentioned in prior literature [8, 9, 51, 64], but have never been identified together as an exhaustive set for problem-space attacks. When designing a novel problem-space attack, it is fundamental to explicitly define these four types of constraints, to clarify strengths and weaknesses of a proposed attack strategy.

We now introduce formal definitions for the problem-space constraints. First, similarly to [8, 18], we define the space of available transformations.

Definition 8 (Available Transformations)**.** We define $\mathcal{T}$ as the space of *available transformations*, which determines which types of automated problem-space transformations T can be carried out by the attacker. In general, it determines if and how the attacker can add, remove, or edit parts of the original object $z \in \mathcal{Z}$ to obtain a new object $z' \in \mathcal{Z}$. We write $\mathbf{T} \in \mathcal{T}$ if a transformation sequence consists of available transformations.

For example, the pixels of an image may be modified only if they remain within the range 0 to 255 [e.g., 13]; in programs, an adversary may only add valid no-op API calls to ensure that any modifications preserve functionality [e.g., 53].

Moreover, the attacker needs to ensure that some semantics are preserved during the transformation of z, according to some feature abstractions. Semantic equivalence is known to be generally undecidable [7, 51]; hence, as in [7], we formalize semantic equivalence through *testing*, by borrowing notation from *denotational semantics* [50].

Definition 9 (Preserved Semantics)**.** Let us consider two problem-space objects z and z' ($z \mapsto z'$), and a suite of automated tests Υ to verify preserved semantics. We define z and z' to be *semantically equivalent* with respect to Υ if they satisfy all its tests $\tau \in \Upsilon$. In particular, we denote semantics equivalence with respect to a test suite Υ as follows:

$$[\![z]\!]^\tau = [\![z']\!]^\tau, \forall \tau \in \Upsilon \tag{4}$$

Informally, Υ consists of automated tests that are aimed at evaluating whether z and z' (or parts of them) lead to the same abstract representations in a certain feature space. In other words, the tests in Υ model preserved semantics. For example, in programs a typical test aims to verify that malicious functionality is preserved (i.e., the malware still performs its malicious purpose); this is done through tests where, given a certain test input, the program produces exactly the same output [7]. Additionally, the attacker may want to be robust against dynamic analysis by ensuring that an adversarial program (z') leads to the same instruction trace as its benign version (z)—so as not to raise suspicion in feature abstractions derived from dynamic analysis.

Plausibility is more subjective than semantic equivalence, but in many scenarios it is critical that an adversarial object is inconspicuous when manually audited by a human. In order to be plausible, a human analyst must believe that the adversarial object is a valid member of the problem-space distribution.

Definition 10 (Plausibility)**.** We define Π as the set of manual tests to verify *plausibility*, which check whether an object in $\mathcal{Z}$ looks like a valid member of the data distribution to a human being. We write that a test $\pi(z)$, $\pi \in \Pi$, returns 1 if the test is *true* (i.e., z is plausible), and 0 otherwise.

Plausibility is often hard to verify automatically; previous work has often relied on user studies with domain experts to judge the plausibility of the generated objects (e.g., program plausibility in [51], realistic eyeglass frames in [55]). Plausibility in software-related domains may also be enforced by construction during the transformation process, e.g., by relying on automated software transplantation [7, 65].

In addition to problem-space constraints and semantic equivalence, the adversarial problem-space objects need to be resilient to non-ML automated techniques that could identify and remove residual *artifacts*, otherwise the attack could be trivially compromised.

Definition 11 (Absent Artifacts)**.** We define Λ as the set of artifacts that must be absent in an object $z' = \mathbf{T}(z)$. In general, artifacts may reveal that an adversarial object is not valid, or may remove parts of the object that correspond to features required to carry the adversarial attack. To verify that the artifacts in Λ are absent, we define $\mathbf{A}_\Lambda : \mathcal{Z} \longrightarrow \mathcal{Z}$ as the *artifacts removal operator*, which removes artifacts in Λ from the problem-space object z'; we say that z' does not contain artifacts if $\mathbf{A}_\Lambda(\mathbf{T}(z)) = \mathbf{T}(z)$.

For the sake of simplicity, it is reasonable to consider only artifacts which may be introduced by the attacker transfor-

mations, and not artifacts that may have already been present in the original application. Examples of artifacts in Λ may include pixel artifacts that could be easily removed with some filter (in images); dead or redundant code, undeclared variables, or missing references (in programs).

We can now define an exhaustive set that consists of all the constraint elements defined in the previous definitions.

Definition 12 (Problem-Space Constraints)**.** We define the *problem-space constraints* as the set of all constraints satisfying $\mathcal{T}, \Upsilon, \Pi, \Lambda$. More formally, we denote it as $\Gamma = \{\mathcal{T}, \Upsilon, \Pi, \Lambda\}$. We write $\mathbf{T}(z) \models \Gamma$ if a transformation sequence applied to object $z \in \mathcal{Z}$ satisfies all the problem-space constraints, and we refer to this as a *valid* transformation sequence. The problem-space constraints Γ determine the feature-space constraints Ω, and we denote this relationship as $\Gamma \vdash \Omega$ (i.e., Γ determines Ω); with a slight abuse of notation, we can also write that $\Omega \subseteq \Gamma$, because some constraints may be specific to the problem space (e.g., program size similar to that of benign applications) and may not be possible to enforce in the feature space $\mathcal{X}$.

Side-Effect Features. Satisfying the problem-space constraints Γ further complicates the inverse feature mapping, as Γ is a superset of Ω. Moreover, enforcing Γ may require substantially altering an object z to ensure satisfaction of all constraints during mutations. Let us focus on an example in the software domain, so that z is a program with features $\boldsymbol{x}$; if we want to transform z to z' such that $\varphi(z') = \boldsymbol{x} + \boldsymbol{\delta}$, we may want to add to z a program o where $\varphi(o) = \boldsymbol{\delta}$. However, the union of z and o may have features different from $\boldsymbol{x}+\boldsymbol{\delta}$, because other consolidation operations are required (e.g., name deduplication, class declarations, resource names normalization)—which cannot be feasibly computed in advance for each possible objects in $\mathcal{Z}$. Hence, after modifying z in the hope of obtaining a problem-space object z' with certain features (e.g., close to $\boldsymbol{x} + \boldsymbol{\delta}$), the attacker-modified object may have some additional features that are not related to the intended transformation (e.g., adding one API which maps to a feature in $\boldsymbol{\delta}$), but that are required to satisfy all the problem-space constraints in Γ (e.g., inserting valid parameters for the API call, and importing dependencies for its invocation). We call *side-effect features* $\boldsymbol{\varepsilon}$ the features that are altered in $z \mapsto z'$ specifically for the satisfaction of problem-space constraints. We observe that these features do not follow any particular direction of the gradient, and hence they could have both a positive or negative impact on the classification score.

Analogy with Projection. Figure 1 presents the analogy between side-effect features $\boldsymbol{\varepsilon}$ and the notion of *projection* in numerical optimization [11], which helps explain the nature and impact of $\boldsymbol{\varepsilon}$ in problem-space attacks. In the example of Figure 1, the background is a heatmap where the right-half (red colors) corresponds to higher values of a discriminant function $h(x)$ to classify an object into the malicious class, whereas the left part (blue colors) corresponds to lower values. The purpose of the adversary is to conduct a *maximum confidence attack* that has an object misclassified as benign. The vertical central

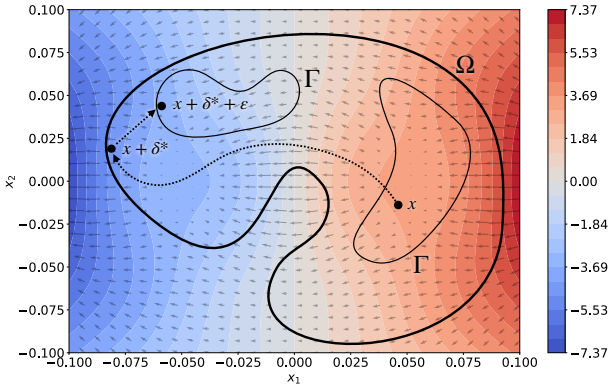


Fig. 1. Example of projection of the feature-space attack vector $\boldsymbol{x} + \boldsymbol{\delta}^*$ in the *feasible* problem space, resulting in side-effect features $\boldsymbol{\varepsilon}$. The background displays the value of the discriminant function for the malicious class, $h(\boldsymbol{x})$, where positive values indicate that the object is malicious. Small arrows represent directions of the negative gradient. The thick solid line represents the *feasible* feature space determined by Ω, and the thin solid line that determined by Γ (which is more restrictive). The dotted arrow represents the gradient-based attack $\boldsymbol{x} + \boldsymbol{\delta}^*$ derived from $\boldsymbol{x}$, which is then projected into $\boldsymbol{x} + \boldsymbol{\delta}^* + \boldsymbol{\varepsilon}$ to fit into the feasible problem space.

curve (where the heatmap value is equal to zero) represents the decision boundary: objects on the left-half of this figure are classified as benign, and objects on the right-half as malicious. The thick solid line represents the *feasible feature space* determined by constraints Ω, and the thin solid line the *feasible problem space* determined by Γ (which corresponds to two unconnected areas). We assume that the initial object $\boldsymbol{x} \in \mathcal{X}$ is always within the feasible problem space. In this example, the attacker first conducts a gradient-based attack in the feature space on object $\boldsymbol{x}$, which results in a feature vector $\boldsymbol{x} + \boldsymbol{\delta}^*$, which is classified with high-confidence as benign. However, we can observe that the point is not included in the feasibility space of constraints Γ, which is more restrictive than that of Ω. Hence, the attacker needs to find a *projection* that maps $\boldsymbol{x} + \boldsymbol{\delta}^*$ back to the feasible problem-space regions, which leads to the addition of a side-effect feature vector $\boldsymbol{\varepsilon}$. We observe that in most cases this projection cannot be found analytically since problem-space constraints Γ cannot be expressed in closed-form, hence the attacker needs to find a transformation sequence $\mathbf{T}$ such that $\varphi(\mathbf{T}(z)) = \varphi(z')$ is within the feasibility region of problem-space constraints Γ.

Definition 13 (Side-Effect Feature Vector)**.** We define $\boldsymbol{\varepsilon}$ as the *side-effect feature vector* that results from enforcing Γ while choosing a sequence of transformations $\mathbf{T}$ such that $\mathbf{T}(z) \models \Gamma$. In other words, $\boldsymbol{\varepsilon}$ are the features derived from the *projection* of a feature-space attack onto a feasibility region that satisfies problem-space constraints Γ.

It is relevant to observe that, in the general case, if an object z_o is added to (or removed from) two different objects z_1 and z_2, it is possible that the resulting side-effect feature vectors $\boldsymbol{\varepsilon}_1$ and $\boldsymbol{\varepsilon}_2$ are different (e.g., in the software domain [51]). Hence, in settings where the feature mapping φ is neither differentiable nor invertible, and where the problem-space representation is very different from the feature-space representation

(e.g., unlike in images or audio), it is generally infeasible or impossible to compute the exact impact of these side-effect features on the objective function in advance—because the set of problem-space constraints cannot be expressed analytically in closed-form.

Considerations on Attack Confidence. There are some important characteristics of the impact of the side-effect features $\boldsymbol{\varepsilon}$ on the attack objective function. If the attacker performs a *maximum-confidence attack* in the feature space under constraints Ω, then the confidence of the problem-space attack will always be *lower or equal* than the one in the feature-space attack. This is intuitively represented in Figure 1, where the point is moved in the maximum-confidence attack area within Ω, and the attack confidence is reduced after projection to the feasibility space of the problem space, induced by Γ. In general, the confidence of the feature- and problem-space attacks could also be equal, depending on the constraints Ω and Γ, and on the shape of the discriminant function h, which is also not necessarily convex (e.g., in deep learning [23]). On the other hand, in the case of *low-confidence* feature-space attacks, projecting into the problem-space feasibility constraint may result in a positive or negative impact (not known a priori) on the value of the discriminant function. This can be seen from Figure 1, where the object $\boldsymbol{x} + \boldsymbol{\delta}^*$ would be found close to the center of the plot, where $h(\boldsymbol{x}) = 0$.

Problem-Space Attack. We now have all the components required to formalize a problem-space attack.

Definition 14 (Problem-Space Attack)**.** We define a *problem-space attack* as the problem of finding the sequence of valid transformations $\mathbf{T}$ for which the object $z \in \mathcal{Z}$ with label $y \in \mathcal{Y}$ is misclassified to a target class $t \in \mathcal{Y}$ as follows:

$$\begin{aligned} \operatorname{argmin}_{\mathbf{T}\in\mathcal{T}} \quad & f_t(\varphi(\mathbf{T}(z))) = f_t(\boldsymbol{x} + \boldsymbol{\delta}^* + \boldsymbol{\varepsilon}) && (5)\\ \text{subject to:} \quad & [\![z]\!]^\tau = [\![\mathbf{T}(z)]\!]^\tau, \quad \forall \tau \in \Upsilon && (6)\\ & \pi(\mathbf{T}(z)) = 1, \quad \forall \pi \in \Pi && (7)\\ & \mathbf{A}_\Lambda(\mathbf{T}(z)) = \mathbf{T}(z) && (8) \end{aligned}$$

where $\boldsymbol{\varepsilon}$ is a side-effect feature vector that separates the feature vector generated by $\mathbf{T}(z)$ from the theoretical feature-space attack $\boldsymbol{x} + \boldsymbol{\delta}^*$ (under constraints Ω). An equivalent, more compact, formulation is as follows:

$$\begin{aligned} \operatorname{argmin}_{\mathbf{T}\in\mathcal{T}} \quad & f_t(\varphi(\mathbf{T}(z))) = f_t(\boldsymbol{x} + \boldsymbol{\delta}^* + \boldsymbol{\varepsilon}) && (9)\\ \text{subject to:} \quad & \mathbf{T}(z) \models \Gamma && (10) \end{aligned}$$

Search Strategy. The typical search strategy for adversarial perturbations in feature-space attacks is based on following the negative gradient of the objective function through some numerical optimization algorithm, such as stochastic gradient descent [8, 13, 14]. However, it is not possible to directly apply gradient descent in the general case of problem-space attacks, when the feature space is not invertible nor differentiable [8, 51]; and it is even more complicated if a transformation sequence $\mathbf{T}$ produces side-effect features $\boldsymbol{\varepsilon} \neq \mathbf{0}$. In the problem space, we identify two main types of search strategy: *problem-driven* and *gradient-driven*. In the problem-driven approach, the search of the optimal $\mathbf{T}$ proceeds heuristically by beginning with random mutations of the object z, and then learning from experience how to appropriately mutate it further in order to misclassify it to the target class (e.g., using Genetic Programming [64] or variants of Monte Carlo tree search [51]). We observe that this problem-driven approach iteratively uses local approximations of the negative gradient to mutate the objects. The gradient-driven approach attempts to identify mutations that follow the negative gradient by relying on an approximate inverse feature mapping (e.g., in PDF malware [40], in Android malware [65]). If a search strategy equally makes extensive use of both problem-driven and gradient-driven methods, we call it *hybrid-driven*. We note that search strategies may have different trade-offs in terms of *effectiveness* and *costs*, depending on the time and resources they require, that warrants further investigation as future work.

Feature-space attacks can still give us some useful information: before searching for a problem-space attack, we can verify whether a feature-space attack exists, which is a necessary condition for realizing the problem-space attack.

Theorem 1 (Necessary Condition for Problem-Space Attacks)**.** Given a problem-space object $z \in \mathcal{Z}$ of class $y \in \mathcal{Y}$, with features $\varphi(z) = \boldsymbol{x}$, and a target class $t \in \mathcal{Y}$, $t \neq y$, an attacker will find a transformation sequence $\mathbf{T}$ that causes $\mathbf{T}(z)$ to be classified as t *only if* there is a solution for the feature-space attack under constraints Ω. More formally, only if:

$$\exists \boldsymbol{\delta}^* = \arg \min_{\boldsymbol{\delta}\in\mathbb{R}^n : \boldsymbol{\delta} \models \Omega} f_t(\boldsymbol{x} + \boldsymbol{\delta}) : f_t(\boldsymbol{x} + \boldsymbol{\delta}^*) < 0 \qquad (11)$$

The proof of Theorem 1 is in Appendix C. We observe that Theorem 1 is necessary but *not sufficient* because of the inverse feature-mapping problem and the side-effect features. In other words, if there exists an optimal feature-space perturbation $\boldsymbol{\delta}^*$, there may not exist a problem-space transformation sequence $\mathbf{T}$ that alters the feature space of $\mathbf{T}(z)$ exactly so that $\varphi(\mathbf{T}(z)) = \boldsymbol{x} + \boldsymbol{\delta}^*$. This is because, in practice, given a target feature-space perturbation $\boldsymbol{\delta}^*$, a problem-space transformation will generate a vector $\varphi(\mathbf{T}(z)) = \boldsymbol{x} + \boldsymbol{\delta}^* + \boldsymbol{\varepsilon}^*$, where $\boldsymbol{\varepsilon}^* \neq \mathbf{0}$ (i.e., where there may exist at least one i for which $\varepsilon_i \neq 0$).

Theorem 2 (Necessary and Sufficient Condition for Problem-Space Attacks)**.** Given a problem-space object $z \in \mathcal{Z}$ of class $y \in \mathcal{Y}$, with features $\varphi(z) = \boldsymbol{x}$, and a target class $t \in \mathcal{Y}$, $t \neq y$, an attacker will find a transformation sequence $\mathbf{T}$ that causes $\boldsymbol{x}$ to be misclassified as t *if and only if*:

$$\exists \boldsymbol{\delta}^* = \arg \min_{\boldsymbol{\delta}\in\mathbb{R}^n : \boldsymbol{\delta} \models \Omega} f_t(\boldsymbol{x} + \boldsymbol{\delta}) : f_t(\boldsymbol{x} + \boldsymbol{\delta}^*) < 0 \qquad (12)$$

$$\forall \boldsymbol{\delta} \in \mathbb{R}^n : \boldsymbol{\delta} \models \Omega, \quad \exists \mathbf{T} : \mathbf{T}(z) \models \Gamma, \varphi(\mathbf{T}(z)) = \boldsymbol{x} + \boldsymbol{\delta} \qquad (13)$$

Informally, an attacker is always able to find a problem-space attack if a feature-space attack exists (necessary condition) and they know problem-space transformations that can modify any feature by any value (sufficient condition).

The proof of Theorem 2 is in Appendix C. In the general case, to obtain a perturbation $\boldsymbol{\delta}$ the transformation $\mathbf{T}$ will also introduce some side-effect features $\boldsymbol{\varepsilon}$ (due to the requirement

that problem-space constraints Γ must be satisfied), which would prevent easily finding a problem-space transformation that follows the negative gradient. We also remark that Theorem 2 does not require the feature mapping to be invertible or differentiable, although some sort of "mapping" between problem- and feature-space perturbations needs to be known by the attacker.

Corollary 2.1. If Theorem 2 is satisfied only on a subset of feature dimensions X_i in $\mathcal{X}$, which collectively create a subspace $\mathcal{X}_{eq} \subset \mathcal{X}$, then the attacker can restrict the search space to $\mathcal{X}_{eq}$, for which they know that an equivalent problem/feature-space manipulation exists.

C. Describing problem-space attacks in different domains

We clarify by example what are the main parameters that need to be explicitly defined while designing problem-space attacks. We consider a representative set of adversarial attacks in different domains: PDFs [64], images [13], face recognition [55], Javascript [22], code attribution [51], Windows binaries [32], and three problem-space attacks on Android, two from the literature [53, 65] and ours proposed in §III.

Table I reports the details of these attacks according to our framework; the cells with gray background denote that the parameter is either not specified or not relevant to the proposal in that domain. This table shows the expressiveness power of our formalization, and how it is able to unveil strengths and weaknesses of different proposals. In particular, we identify some major limitations in the two recent problem-space attacks on Android [53, 65]. Rosenberg et al. [53] leave artifacts during the app transformation which are easily detected without the use of machine learning (see §VI for details), and relies on no-op APIs which could be removed through dynamic analysis. Yang et al. [65] does not specify which artifacts it prevents, and their approach may significantly alter the semantics of the program—which may account for a high failure rate they observe in the mutated apps. This inspired us to propose a novel attack that overcomes such limitations.

III. Attack on Android

Our formalization of problem-space attacks has allowed for the identification of weaknesses in prior approaches for Android malware evasion [53, 65]. Hence, we propose—through our formalization—a novel problem-space attack in this domain that overcomes the limitations of past literature [53, 65], especially in terms of preserved semantics and absence of artifacts (see §II-C and §VI for a detailed comparison).

A. Threat Model

We assume an attacker with *perfect knowledge* $\theta_{PK} = (\mathcal{D}, \mathcal{X}, g, \boldsymbol{w})$. This follows Kerckhoffs' principle [31] and ensures a defense does not rely on "security by obscurity" by unreasonably assuming some properties of the defense can be kept secret [16]. Although deep learning has been extensively studied in adversarial attacks, recent research [e.g., 49] has shown that—if retrained frequently—the DREBIN classifier [5] achieves state of the art performance for Android malware detection. DREBIN relies on a linear SVM, and embeds apps in a *binary* feature-space $\mathcal{X}$ which captures the presence/absence of components in Android applications in $\mathcal{Z}$ (such as permissions, URLs, Activities, Services, strings). We assume to know classifier g and feature-space $\mathcal{X}$, and train the parameters $\boldsymbol{w}$ with SVM hyperparameter $C = 1$, as in the original DREBIN paper [5]. We also evaluate the effectiveness of our problem-space attack against a hardened version of the SVM algorithm, namely Sec-SVM, which has been recently proposed by Demontis et al. [18]. Sec-SVM enforces more evenly distributed feature weights, which require an attacker to modify more features to evade detection.

We consider an attacker intending to evade detection based on *static analysis*, without relying on code obfuscation as it may increase suspiciousness of the apps [60, 62] (see §V).

B. Available Transformations

We use *automated software transplantation* [7] to extract slices of bytecode (i.e., *gadgets*) from benign *donor* applications and inject them into a malicious *host*, to induce the learning algorithm to misclassify the malicious host as a benign app.[1] An advantage of this process is that we avoid relying on a hardcoded set of transformations [e.g., 51]; this ensures adaptability across different application types and time periods. In this work, we consider only *addition* of bytecode to the malware—which ensures that we do not hinder the malicious functionality.

Organ Harvesting. In order to augment a malicious host with a given *benign feature* X_i, we must first extract a bytecode gadget ρ corresponding to X_i from some donor app. As we intend to produce realistic examples, we use *program slicing* [63] to extract a functional set of statements that includes a reference to X_i. The final gadget consists of the this target reference (*entry point* L_o), a forward slice (*organ* o), and a backward slice (*vein* v). We first search for L_o, corresponding to an appearance of code corresponding to the desired feature in the donor. Then, to obtain o, we perform a context-insensitive forward traversal over the donor's System Dependency Graph (SDG), starting at the entry point, transitively including all of the functions called by any function whose definition is reached. Finally, we extract v, containing all statements needed to construct the parameters at the entry point. To do this, we compute a backward slice by traversing the SDG in reverse. Note that while there is only one organ, there are usually multiple veins to choose from, but only one is necessary for the transplantation. When traversing the SDG, class definitions that will certainly be already present in the host are excluded (e.g., system packages such as `android` and `java`). For example, for an Activity feature where the variable `intent` references the target Activity of interest, we might extract the invocation `startActivity(intent)` (entry point

[1] Our approach is generic and it would be immediate to do the opposite, i.e., transplant malicious code into a benign app. However, this would require a dataset with *annotated* lines of malicious code. For this practical reason and for the sake of clarity of this section, we consider only the scenario of adding benign code parts to a malicious app.

TABLE I

PROBLEM-SPACE EVASION ATTACKS IN DIFFERENT SETTINGS AND DOMAINS, MODELED WITH OUR FORMALIZATION.

		DOMAINS									
PARAMETERS		**PDF** [64]	**Android Transplantation** [65]	**Android RNN** [53]	**Image Classification** [13]	**Face Recognition** [55]	**Code Attribution** [**?**]	**Audio** [14]	**Javascript** [22]	**Windows** [32]	**Our Android Attack** (see §III)
THREAT MODEL	Knowledge θ	ZK	ZK	ZK	PK	PK	ZK	PK	ZK	PK	PK
	Feature mapping φ	Invertible: no. Differentiable: no.	Invertible: no. Differentiable: no.	Invertible: no. Differentiable: no.	Invertible: no. Differentiable: yes.	Invertible: no. Differentiable: yes.	Invertible: no. Differentiable: no.	Invertible: no. Differentiable: yes.	Invertible: no. Differentiable: no.	Invertible: no. Differentiable: no.	Invertible: no. Differentiable: no.
	Feature space $\mathcal{X}$	Static (metadata, object keywords and properties, structural).	Static analysis (RTLD model [65]).	Dynamic API sequences, and static printable strings (also in latent feature space).	Latent feature space of pixels.	Latent feature space of pixels.	Syntactic and lexical static features.	Latent feature space of audio stream.	Static syntactic, based on AST, PDG, CFG.	Feature mapping of MalConv [52].	Lightweight static analysis (binary features).
	Problem space $\mathcal{Z}$	PDF	Software (bytecode)	Software (bytecode)	Image (pixels)	Printed image (pixels)	Software (source code)	Audio (signal)	Software (source code)	Software (binary)	Software (bytecode)
	Classifier g	SVM-RBF (Hidost [57]), RF (PDFRate [56]).	kNN, DT, SVM (and VirusTotal [24]).	RNN/LSTM variants, and transferability to traditional classifiers (e.g., RF, SVM).	Deep learning.	Deep learning.	Any classifier.	Deep learning.	Any classifier.	Deep learning (*MalConv* [52]).	Linear SVM (DREBIN [5]) and its hardened version (Sec-SVM [**?**]).
PROBLEM-SPACE CONSTRAINTS	Available Transformations $\mathcal{T}$	Addition/Removal of elements in the PDF tree structure.	Code addition and modification (within the same program) through *automated software transplantation*.	(i) Addition of no-op API calls with valid parameters. (ii) Repacking of the input malware.	(i) Modification of pixel values. (ii) Pixel values between 0 and 255 ($\boldsymbol{x} + \boldsymbol{\delta} \in [0, 1]^n$).	(i) Modification of pixel values. (ii) Pixel values between 0 and 255 ($\boldsymbol{x} + \boldsymbol{\delta} \in [0, 1]^n$). (iii) Pixels are printable. (iv) Robust to 3D rotations.	(i) Pre-defined set of semantics-preserving code transformations (i.e., modifications). (ii) No changes to the layout of the code.	(i) Addition of audio noise. (ii) Audio values bounded (i.e., $\boldsymbol{x} + \boldsymbol{\delta} \in [-M, +M]$).	Transplantation of semantically-equivalent benign ASTs.	Addition of carefully-crafted bytes at the end of the binary.	Code addition through *automated software transplantation*.
	Preserved Semantics Υ	Malicious network functionality is still present (verification with Cuckoo Sandbox).	(i) The application still installs and executes. (ii) The malicious functionality is preserved.	Same API sequences with same function return values (verification with Cuckoo Monitor).			By construction through use of semantics-preserving transformations.		By construction through use of AST-based transplantation that preserves the malicious semantics.		By construction with *opaque predicates* (newly inserted code is not executed at runtime).
	Absent Artifacts Λ		Unclear. A component takes care of the consolidation after transplantation.	Redundant code, undeclared variables, unlinked resources, undefined references, name conflicts					Name inconsistencies of functions and variables.		Redundant code, undeclared variables, unlinked resources, undefined references, name conflicts, no-op instructions, repacking artifacts
	Plausibility Π		Code is realistic by construction through use of automated software transplantation.	The added no-op API calls do not raise errors.	Perturbation constraint $\|\|\boldsymbol{\delta}\|\|_p \leq \delta_{max}$, to ensure the changes are imperceptible to a human.	(i) Perturbation constraint $\|\|\boldsymbol{\delta}\|\|_p \leq \delta_{max}$, (ii) Smooth pixel transitions so the eyeglass frames look legitimate with plausible deniability.	The code does not look suspicious and seems written by a human (survey with developers).	Perturbation constrained ($dB_x(\boldsymbol{\delta}) \leq dB_{max}$), so that added noise resembles white background noise and new audio sounds similar (to a human).	By construction through use of automated AST transplantation.		(i) Code is realistic by construction through use of automated software transplantation. (ii) The mutated app installs and starts correctly on an emulator.
Search Strategy		**Problem-driven.** Genetic Programming.	**Gradient-driven.** Prioritizing mutations that will affect features typical of malware evolution (e.g., phylogenetic trees) and those present in both malware and goodware.	**Hybrid-driven.** Greedy algorithm selects API calls in order to minimize the difference between the current and previous iterations w.r.t. the direction of the Jacobian.	**Gradient-driven.** Stochastic Gradient Descent in the feature space.	**Gradient-driven.** Stochastic Gradient Descent in the feature space.	**Problem-driven.** New Monte-Carlo Search algorithm, applied to the problem space.	**Gradient-driven.** Adam optimizer with learning rate 10 and 5,000 max iterations.	**Problem-driven.** Search of isomorphic sub-AST graphs in benign samples that are equivalent to malicious sub-ASTs	**Gradient-driven.** Although the feature mapping is not invertible and not differentiable, the authors devise an algorithm to project byte padding on the line of the negative gradient.	**Gradient-driven.** We define an approximate inverse of the feature mapping, and then devise a greedy algorithm in the problem space that follows the negative gradient.
Side-effect features $\boldsymbol{\epsilon}$		$\boldsymbol{\epsilon} \simeq \mathbf{0}$	$\boldsymbol{\epsilon} \neq \mathbf{0}$	$\boldsymbol{\epsilon} \simeq \mathbf{0}$	$\boldsymbol{\epsilon} = \mathbf{0}$	$\boldsymbol{\epsilon} = \mathbf{0}$	$\boldsymbol{\epsilon} \simeq \mathbf{0}$	$\boldsymbol{\epsilon} = \mathbf{0}$	$\boldsymbol{\epsilon} \neq \mathbf{0}$	$\boldsymbol{\epsilon} = \mathbf{0}$	$\boldsymbol{\epsilon} \neq \mathbf{0}$

L_o), the class implementation of the Activity itself along with any referenced classes (organ o), and all statements necessary to construct `intent` with its parameters (vein v). There is a special case for Activities which have no corresponding vein in the bytecode (e.g., a `MainActivity` or an Activity triggered by an intent filter declared in the Manifest); here, we provide an *adapted vein*, a minimal Intent creation and `startActivity()` call adapted from a previously mined benign app that will trigger the Activity. Note that organs with original veins are always prioritized above those without.

Organ Implantation. In order to implant some gadget ρ into a host, it is necessary to identify an injection point L_H where v should be inserted. Implantation at L_H should fulfill two criteria: firstly, it should maintain the syntactic validity of the host; secondly, it has to be as inconspicuous as possible. To maximize the probability of fulfilling the first criterion, we restrict L_H to be between two statements of a class definition in a non-system package. For the second criterion, we take a heuristic approach by using *Cyclomatic Complexity* (CC)—a software metric that quantifies the code complexity of components within the host—and choosing L_H such that we maintain existing homogeneity of CC across all components. Specifically, we randomly select classes from the host until we identify a candidate for which the class CC, including the gadget ρ, is *close* to the average CC of all components, i.e., within $\mu \pm 3\sigma$ (where μ is the mean CC of the components and σ is the CC standard deviation). Finally, the host entry point L_H is inserted into a *randomly chosen* function among those of the selected class, to avoid leaving a systematic artifact that might be identified by an analyst.

C. Preserved Semantics

Given an application z and its modified (adversarial) 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 within conditional statements that always return `False`. This guarantees that the newly inserted code is never executed at runtime—so that users will not notice anything odd while using the modified application. In §III-D, we describe how we generate such conditionals without leaving artifacts.

To further preserve semantics, we also decide to omit `intent-filter` elements as transplantation candidates. This is because, for example, an `intent-filter` could declare the app as an eligible option for reading PDF files; consequently, whenever attempting to open a PDF file, the user would be able to choose the host app, which (if selected) would trigger an Activity defined in the transplanted benign bytecode—and this would violate our constraint of preserving dynamic functionality.

D. Absent Artifacts

Unreachable code is an artifact which would be trivial to remove through program analysis techniques that perform redundant code elimination. 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* [45], i.e., carefully constructed obfuscated conditions where the outcome is always known at design time (in our case, `False`), but the actual truth value is difficult or impossible to determine using static analysis techniques. We refer the reader to Appendix D for a detailed description of how we generate strong opaque predicates and make them look legitimate.

E. Plausibility

In our model, an example is satisfactorily plausible if it resembles a real, functioning Android application (i.e., is a valid member of the problem-space $\mathcal{Z}$). Our methodology aims to maximize the plausibility of each generated object by injecting full slices of bytecode from *real* benign applications. There is only one case in which we inject artificial code: the opaque predicates that guard the entry point of each gadget (see Appendix D for an example). In general, we can conclude that plausibility is guaranteed *by construction* thanks to the use of automated software transplantation [7]. This contrasts with other approaches that inject *standalone* API calls and URLs or *no-op* operations [e.g., 53] that are completely orphaned and unsupported by the rest of the bytecode (e.g., an API call result that is never used).

Finally, although plausibility is typically checked manually, we also practically assess that the each mutated app still functions properly after modification by installing and running it on an Android emulator. Although 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.

F. 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 algorithm is greedy as it harvests a set of candidate gadgets at initialization time, which are then used at attack time. This section offers an overview of the proposed search strategy, and the detailed steps of the two phases are reported in Appendix F.

Initialization Phase (Ice-Box Creation). We first harvest gadgets from potential donors and collect them in an *ice-box* G, which is used for transplantation at attack time. The main reason for performing this initialization, instead of looking for gadgets at runtime, is to have an immediate estimate of the *side-effect features* when each gadget is considered for transplantation. Looking for gadgets on-the-fly is a possibility, but may lead to less optimal solutions and uncertain runtimes.

For the initialization we aim to gather gadgets that move the score of an object towards the benign class (i.e., negative

score), hence we consider the classifier's top n_f benign features (i.e., with negative weight). For each of the top-n_f features, we extract n_d candidate gadgets for the attack phase, excluding those that lead to an overall positive score (i.e., malicious gadget score). We recall that this may happen even for benign features since the context extracted through forward and backward slicing may contain many other features that are indicative of maliciousness. We empirically verify that with $n_f = 500$ and $n_d = 5$ we are able to create a successfully evasive app for all the malware in our experiments. To estimate the side-effect feature vectors for the gadgets, we inject each into a *minimal app*, i.e., an Android app we developed with minimal functionality (see Appendix F). It is important to observe that the ice-box can be expanded over time, as long as the target classifier does not change its weights significantly. Algorithm 1 in Appendix F reports the detailed steps of the initialization phase.

Attack Phase. We aim to automatically mutate z into z' so that it is misclassified as goodware, i.e., $h(\varphi(z')) < 0$, by transplanting harvested gadgets from the ice-box G. First we search for the list of ice-box gadgets that should be injected into z. Each gadget ρ_j in the ice-box G has feature vector $\boldsymbol{r}_j$, which includes the desired feature and side-effect features. We consider the actual feature-space contribution of gadget i to the malicious host z with features $\boldsymbol{x}$ by performing the set difference of the two binary vectors, $\boldsymbol{r}_j \wedge \neg \boldsymbol{x}$. We then sort the gadgets in order of decreasing negative contribution, which ideally leads to a faster convergence of z's score to a benign value. Next we filter this candidate list to include gadgets *only if* they satisfy some practical feasibility criteria. In particular, we define a *check_feasibility* function which implements some heuristics to limit the excessive increase of certain statistics which would raise suspiciousness of the app. Preliminary experiments of our search strategy revealed a tendency to add too many permissions in the Android Manifest. Hence, we empirically enforce that candidate gadgets add no more than 1 new permission to the host app. Moreover, we do not allow addition of permissions listed as *dangerous* in the Android documentation.[2] The other app statistics remained reasonably within the distribution of benign apps (more discussion in §IV), and so we decide not to forcibly limit them during the attack. The remaining candidate gadgets are iterated over and for each candidate ρ_j, we combine the gadget feature vector $\boldsymbol{r}_j$ with the input malware feature vector $\boldsymbol{x}$, such that $\boldsymbol{x}' = \boldsymbol{x} \vee \boldsymbol{r}_j$. We repeat this procedure until the updated $\boldsymbol{x}'$ is classified as goodware (for low-confidence attacks) or until an attacker-defined confidence level is achieved (for high-confidence attacks). Finally, we inject all the candidate gadgets at once through automated software transplantation, and check that problem-space constraints are verified and that the app is still classified as goodware. Algorithm 2 in Appendix F reports the detailed steps of the attack phase.

[2]https://developer.android.com/guide/topics/permissions/overview#dangerous_permissions

IV. Experimental Evaluation

We evaluate the effectiveness of our novel problem-space Android attack, in terms of success rate and required time—also when in presence of feature-space defenses.

A. Experimental Settings

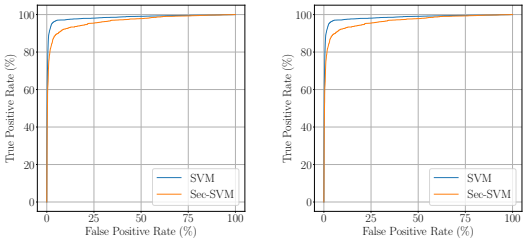
Prototype. We create a prototype of our novel problem-space attack (§III) using a combination of Python for the ML functionalities and Java for the program analysis operations; in particular, to perform transplantations in the problem-space we rely on FlowDroid [6], which is based on Soot [61]. We release the code of our prototype to other academic researchers (see §VII). We ran all experiments on an Ubuntu VM with 48 vCPUs and 290GB of RAM and NVIDIA Tesla K40 GPU.

Classifiers. As defined in the threat model (§III-A), we consider the DREBIN classifier [5], based on a binary feature space and a linear SVM, and its recently proposed hardened variant, Sec-SVM [18], which requires the attacker to modify more features to perform an evasion. We use hyperparameter C=1 for the linear SVM as in [5], and identify the optimal Sec-SVM parameter $k = 0.25$ (i.e., the maximum feature weight) in our setting by enforcing a maximum performance loss of 2% AUC. See Appendix E for implementation details.

Attack Confidence. We consider two attack settings: *low-confidence* (**L**) and *high-confidence* (**H**). The (L) attack merely overcomes the decision boundary (so that $h(\boldsymbol{x}) < 0$). The (H) attack maximizes distance from the hyperplane into the goodware region; while generally this distance is unconstrained, here we set it to be $\leq$ than the first quartile of benign apps. This avoids adding too many unnecessary modifications to a program, which may only increase suspiciousness or chances of transplantation errors.

Dataset. We collect apps from AndroZoo [2], a large-scale dataset with timestamped Android apps crawled from different stores, and with VirusTotal summary reports. We use the same labeling criteria as Tesseract [49] (which is derived from Miller et al. [43]): an app is considered *goodware* if it has 0 VirusTotal detections, as *malware* if it has 4+ VirusTotal detections, and is discarded as *grayware* if it has between 1 and 3 VirusTotal detections. For the dataset composition, we the spatial constraint of Tesseract of using an average of 10% malware [49]. The final dataset contains ~170K recent Android applications, dated between Jan 2017 and Dec 2018, specifically 152,632 goodware and 17,625 malware.

Dataset Split. Tesseract [49] has demonstrated that, in non-stationary contexts (as Android malware is), if time-aware splits are not considered, then the results may be inflated because of *concept drift* (i.e., changes in the data distribution). However, here we aim to specifically evaluate the effectiveness of an adversarial attack. Although it likely exists, the relationship between adversarial and concept drift is still unknown, and unveiling it is outside the scope of this work. If we were to perform a time-aware split, it would be impossible to determine whether the success rate of our ML-driven adversarial attack was due to an intrinsic weakness of the classifier or due to natural concept drift of malware (e.g.,



(a) ROC (b) ROC (Zoom)

Fig. 2. Performance of SVM and Sec-SVM in absence of adversarial attacks.

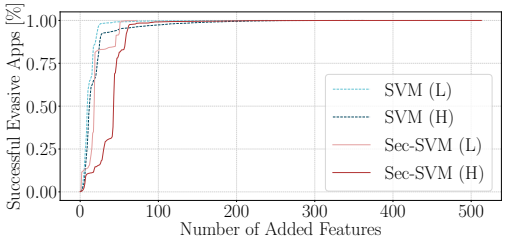


Fig. 3. Cumulative distribution of features added to adversarial malware.

due to novel non-ML techniques on which malware developers rely on to evade detection). Hence, we choose to perform a *random split* of the dataset to simulate *absence of concept drift* [49]; this also 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 (on which the classifier likely has higher confidence). In particular, we consider a 66% training and 34% testing random split.[3]

Testing. The test set contains a total of 5,952 malware. The statistics reported in the remainder of this section refer only to *true positive* malware (5,330 for SVM and 4,108 for Sec-SVM), i.e., we create adversarial variants only if the app is detected as malware by the classifier under evaluation. Intuitively, it is not required to make an adversarial example of a malware application that is already misclassified as goodware; hence, we avoid results inflation by removing false negative objects from the dataset. During the transplantation phase of our problem-space attack some errors occur due to bugs and corner-case errors in the FlowDroid framework [6]. 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 (see Appendix G for details).

B. 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.

Figure 2 reports the AUROC of SVM and Sec-SVM on the DREBIN feature space in absence of attacks. As expected [18], Sec-SVM sacrifices some detection performance in return for greater feature-space adversarial robustness.

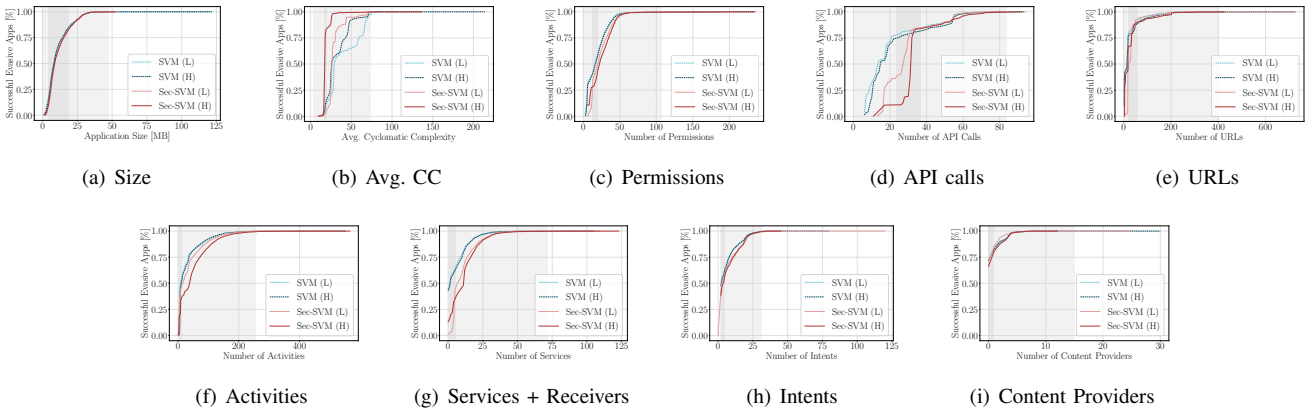
Attack Success Rate. We perform our problem-space attack on *true positive* malware in the testing set, i.e., for all malware objects correctly classified as malware. In particular, we consider four settings depending on the defense algorithm and the attack confidence: SVM (L), SVM (H), Sec-SVM (L), and Sec-SVM (H). In absence of FlowDroid exceptions (see Appendix G), we are able to create an evasive variant for each malware in all four configurations. In other words, we achieve a misclassification rate of 100.0% on the successfully generated apps, where the problem-space constraints are satisfied by construction (as defined in §III). Figure 3 reports the cumulative distribution of features added when generating evasive apps for the four different configurations. As expected, Sec-SVM requires the attacker to modify more features, but here we are no longer interested in the feature-space properties, since we are performing a problem-space attack.

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. Figure 4 reports the cumulative distribution of app statistics: the X-axis reports the statistics values, whereas the Y-axis reports the cumulative percentage of evasive malware apps. Each plot reports four main lines, for SVM and Sec-SVM, in low-confidence (L) and high-confidence (H) settings. We also shade two gray areas: a *dark gray area* spanning between first quartile q_1 and third quartile q_3 of the statistics for the benign applications; the *light gray area* refers to the 3σ rule and reports the area within the 0.15% and 99.85% of the benign apps distribution.

Figure 4 shows that while evading Sec-SVM tends to cause a shift towards the higher percentiles of each statistic, the vast majority of apps falls within the gray regions in all configurations. We note that this is just a qualitative analysis to verify that the statistics of the evasive apps roughly align with those of benign apps; it is not sufficient to have an anomaly in one of these statistics to determine that an app is malicious (otherwise, very trivial rules could be used for malware detection itself, and this is not the case). We also observe that there is little difference between the statistics generated by Sec-SVM and by traditional SVM; this means that greater feature-space perturbations do not necessarily correspond to greater perturbations in the problem-space, reinforcing the feasibility and practicality of evading Sec-SVM.

Runtime Overhead. A reader may wonder how much time is required to perform our problem-space attack. The time to perform the search strategy that happens in the feature space is almost negligible, and the most demanding operation is in the actual code modification. Figure 5 shows *violin plots*

[3]We consider only one split due to the overall time required to run the experiments. Including some prototype overhead, it requires about one month to run all configurations.



(a) Size (b) Avg. CC (c) Permissions (d) API calls (e) URLs

(f) Activities (g) Services + Receivers (h) Intents (i) Content Providers

Fig. 4. Statistics of the evasive malware variants, compared with statistics of benign apps. The dark gray background highlights the area between first and third quartile of benign applications; the light gray background is based on the 3σ rule and highlights values benign statistics between 0.15% and 99.85% of the distribution (i.e., spanning 99.7% of the distribution).

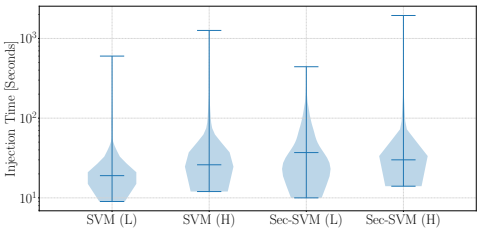


Fig. 5. Violin plots of injection times per adversarial app.

depicting the distribution of injection times for the malware in our test set. Injection time is the most expensive operation in our approach while the rest is mostly pipeline overhead. The time spent per app is low: in most cases, less than 100 seconds, and always less than 2,000 seconds (~33 mins). The low runtime cost suggests that it is feasible to perform this attack at scale, and demands for new defenses in this domain.

V. Discussion 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 [5] and its hardened variant, Sec-SVM [18], 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.

Obfuscation. It could be argued that traditional obfuscation methods can be used to simply hide malicious functionality. The novel problem-space attack in this work evaluates the feasibility of an "adversarial-malware as a service" scenario, where the use of mass obfuscation may raise the suspicions of the defender; for example, antivirus companies often classify samples as malicious simply because they utilize obfuscation or packing [60, 62]. Moreover, some other analysis methods combine static and dynamic analysis to prioritize evaluation of code areas that are likely obfuscated [e.g., 36]. On the contrary, our transformations aim to be fully inconspicuous by adding only legitimate benign code and, to the best of our knowledge, we do not leave any relevant artifact in the process.

Defenses. Unlike settings where feature and problem space are closely related (e.g., images and audio), limitations on feature-space l_p perturbations are often insufficient to determine the risk and feasibility of an attack in the real world. Our novel problem-space formalization (§II) paves the way to the study of *practical* defenses that can be effective in settings which lack an inverse feature mapping. Simulating and evaluating attacker capabilities in the problem space helps define realistic threat models with more constrained modifications in the feature space—which may lead to more robust classifier design. 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 problem-space—and to do so *en masse*. A recent promising direction by Incer et al. [28] studies the use of *monotonic classifiers*, where adding features can only increase the decision score (i.e., an attacker cannot rely on adding more features to evade detection); however, such classifiers require non-negligible time towards manual feature selection (i.e., on features that are harder for an attacker to change), and—at least in the context of Windows malware [28]—they suffer from high false positives and an average reduction in detection rate of 13%. Moreover, we remark that we decide to add goodware parts to malware for practical reasons: the opposite transplantation would be immediate to do if a dataset with *annotated* malicious bytecode segments were available. As part of future work we aim to investigate whether it would still be possible to evade monotonic classifiers by adding only a minimal number of malicious slices to a benign application.

VI. Related Work

Adversarial Machine Learning. Adversarial ML attacks have been studied for more than a decade [8]. These attacks aim to modify objects either at training time (*poisoning* [58]) or at test time (*evasion* [9]) to compromise the confidentiality,

integrity, or availability of a machine learning model. Lots of frameworks and formalizations have been proposed in the literature to describe the feature-space attacks, either as optimization problems [9, 13] (see also §II-A for a detailed description) or game theoretic frameworks [17].

Problem-Space Attacks. Recently, research on adversarial ML has moved towards domains in which the feature mapping is not invertible or not differentiable. Here, the adversary needs to modify the objects in the problem space (i.e., input space) without knowing exactly how this will affect the feature space. This is known as the *inverse feature-mapping* problem [9, 26, 51]. Many works on problem-space attacks have been explored on different domains: text [3, 37], PDFs [34, 35, 39, 40, 64], Windows binaries [32, 52], Android apps [18, 25, 53, 65], ICS [66], and Javascript source code [51]. However, 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 formalization (§II) aims to close this gap, and we demonstrate its generality by using it to describe representative feature-space and problem-space attacks from the literature (§II-C).

Adversarial Android Malware. This paper also proposes a novel problem-space attack in the Android domain (§III) to generate adversarial malware; our attack overcomes limitations of existing proposals, which are evidenced through our formalization. The most related approaches to our novel attack are on attribution [51], and on adversarial malware generation [25, 53, 65]. Quiring et al. [51] do *not* consider malware detection, but misleading authorship attribution. They design a set of simple mutations to change the programming style of an application to match the style of a target developer (e.g., replacing *for* loops with *while* loops). This strategy is effective for attribution, but is insufficient for malware detection as changing only stylometric properties would not affect the outcome of a malware classifier which tries to capture program semantics. Moreover, it is not feasible to define a hardcoded set of transformations for all possible semantics—which may also leave unwanted artifacts in the mutated code. Conversely, our attack relies on automated software transplantation to ensure plausibility of the generated code and avoids hardcoded code mutation artifacts.

Grosse et al. [25] aim to perform minimal modifications that preserve semantics, and decide to modify only single lines of code in the Manifest; this may leave artifacts such as unused permissions or undeclared classes. Moreover, they limit their adversarial perturbation to 20 features, whereas our problem-space constraints represent a more realistic threat model.

Yang et al. [65] propose a method for adversarial Android malware generation. Similarly to us, they rely on *automated software transplantation* [7] and evaluate their adversarial attack against the DREBIN classifier [5]. However, they do not formally define which semantics are preserved by their transformation, and their approach is extremely unstable, breaking the majority of apps they mutate (e.g., they report failures after 10+ modifications on average—which means they would likely not be able to evade Sec-SVM [18] which on average requires modifications of 50+ features). Moreover, their paper does not contain any clear descriptions of possible artifacts in mutated programs. Conversely, our attack is resilient to the insertion of a large number of features (§IV), it preserves dynamic app semantics through opaque predicates (§III-C), and it avoids a large set of artifacts (§III-D).

Rosenberg et al. [53] propose a black-box adversarial attack that works against Android malware classifiers relying on API sequence call analysis. In addition to the limited focus on API-based sequence features, their problem-space transformation leaves two major artifacts which could be detected by traditional program analysis methods: the addition of no-operation instructions (*no-ops*), and patching of the import address table (IAT). Firstly, the inserted API calls need to be executed at runtime and so contain individual no-ops hardcoded by the authors following a practice of "security by obscurity", which is known to be ineffective [16, 31]; intuitively, they could be detected and removed by identifying the tricks used by attackers to perform no-op API calls (e.g., reading 0 bytes), or by filtering the "dead" API calls (i.e., which did not perform any real task) from the dynamic execution sequence before feeding it to the classifier. Secondly, to avoid requiring access to the source code, the new API calls are inserted and called using IAT patching. However, all of the new APIs must be included in a separate segment of the binary and, as IAT patching is a known malicious strategy used by malware authors [20], IAT calls to non-standard dynamic linkers or multiple jumps from the IAT to an internal segment of the binary would immediately be identified as suspicious. Conversely, our attack does not require hardcoding and by design avoids such artifacts that would be easily removed using non-ML program analysis techniques.

VII. Availability

We release the code and data of our approach to other researchers by responsibly sharing it. Please send an e-mail to all authors with subject "*Code Access Request*".

VIII. Conclusions

Since the seminal work that evidenced intriguing properties of neural networks [59], the community has become aware of the brittleness of machine learning in adversarial settings [8].

To better understand real-world implications across different application domains, we propose a novel general formalization of problem-space attacks that enables comparison between different proposals and offers a more principled design framework for future 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.

REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles,Techniques, and Tools (2nd Edition)*. Addison Wesley, 2007.

[2] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon. Androzoo: Collecting Millions of Android Apps for the Research Community. In *ACM Mining Software Repositories (MSR)*, 2016.

[3] M. Alzantot, Y. Sharma, A. Elgohary, B.-J. Ho, M. Srivastava, and K.-W. Chang. Generating natural language adversarial examples. In *Empirical Methods in Natural Language Processing (EMNLP*, 2018.

[4] E. K. Andreas Moser, Christopher Kruegel. Limits of static analysis for malware detection. 2007.

[5] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In *NDSS*, 2014.

[6] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. D. McDaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *PLDI*. ACM, 2014.

[7] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke. Automated software transplantation. In *ISSTA*. ACM, 2015.

[8] B. Biggio and F. Roli. Wild patterns: Ten years after the rise of adversarial machine learning. *Pattern Recognition*, 2018.

[9] B. Biggio, I. Corona, D. Maiorca, B. Nelson, N. Šrndić, P. Laskov, G. Giacinto, and F. Roli. Evasion attacks against machine learning at test time. In *ECML-PKDD*. Springer, 2013.

[10] B. Biggio, G. Fumera, and F. Roli. Security evaluation of pattern classifiers under attack. *IEEE TKDE*, 2013.

[11] C. M. Bishop. *Pattern Recognition and Machine Learning*. 2006.

[12] N. Carlini. List of Adversarial ML Papers, 2019. URL https://nicholas.carlini.com/writing/2019/all-adversarial-example-papers.html.

[13] N. Carlini and D. Wagner. Towards evaluating the robustness of neural networks. In *IEEE Symp. S&P*, 2017.

[14] N. Carlini and D. Wagner. Audio adversarial examples: Targeted attacks on speech-to-text. In *Deep Learning for Security (DLS) Workshop*. IEEE, 2018.

[15] N. Carlini and D. A. Wagner. Adversarial examples are not easily detected: Bypassing ten detection methods. In *AISec@CCS*, pages 3–14. ACM, 2017.

[16] N. Carlini, A. Athalye, N. Papernot, W. Brendel, J. Rauber, D. Tsipras, I. Goodfellow, and A. Madry. On evaluating adversarial robustness. *arXiv preprint arXiv:1902.06705*, 2019.

[17] N. Dalvi, P. Domingos, S. Sanghai, D. Verma, et al. Adversarial classification. In *KDD*. ACM, 2004.

[18] A. Demontis, M. Melis, B. Biggio, D. Maiorca, D. Arp, K. Rieck, I. Corona, G. Giacinto, and F. Roli. Yes, machine learning can be more secure! a case study on android malware detection. *IEEE Transactions on Dependable and Secure Computing*, 2017.

[19] W. F. Dowling and J. H. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *J. Log. Program.*, 1(3): 267–284, 1984.

[20] S. Eresheim, R. Luh, and S. Schrittwieser. The evolution of process hiding techniques in malware-current threats and possible countermeasures. *Journal of Information Processing*, 2017.

[21] R. Fan, K. Chang, C. Hsieh, X. Wang, and C. Lin. LIBLINEAR: A library for large linear classification. *J. Mach. Learn. Res.*, 9:1871–1874, 2008.

[22] A. Fass, M. Backes, and B. Stock. HideNoSeek: Camouflaging Malicious JavaScript in Benign ASTs. In *ACM CCS*, 2019.

[23] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT press, 2016.

[24] Google. VirusTotal, 2004. URL https://www.virustotal.com.

[25] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel. Adversarial examples for malware detection. In *ESORICS*. Springer, 2017.

[26] L. Huang, A. D. Joseph, B. Nelson, B. I. Rubinstein, and J. Tygar. Adversarial machine learning. In *AISec*. ACM, 2011.

[27] L. Huang, A. D. Joseph, B. Nelson, B. I. Rubinstein, and J. Tygar. Adversarial machine learning. In *Proceedings of the 4th ACM workshop on Security and artificial intelligence*, pages 43–58. ACM, 2011.

[28] I. Incer, M. Theodorides, S. Afroz, and D. Wagner. Adversarially robust malware detection using monotonic classification. In *Proc. Int. Workshop on Security and Privacy Analytics*. ACM, 2018.

[29] J. Jeon, X. Qiu, J. S. Foster, and A. Solar-Lezama. Jsketch: sketching for java. In *ESEC/SIGSOFT FSE*, pages 934–937. ACM, 2015.

[30] A. Kamath, R. Motwani, K. V. Palem, and P. G. Spirakis. Tail bounds for occupancy and the satisfiability threshold conjecture. In *FOCS*, pages 592–603. IEEE Computer Society, 1994.

[31] A. Kerckhoffs. La cryptographie militaire. In *Journal des sciences militaires*, 1883.

[32] B. Kolosnjaji, A. Demontis, B. Biggio, D. Maiorca, G. Giacinto, C. Eckert, and F. Roli. Adversarial malware binaries: Evading deep learning for malware detection in executables. In *EUSIPCO*. IEEE, 2018.

[33] T. Larrabee. Test pattern generation using boolean satisfiability. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 11(1):4–15, 1992.

[34] P. Laskov and N. Šrndić. Static Detection of Malicious JavaScript-Bearing PDF Documents. In *ACSAC*. ACM, 2011.

[35] P. Laskov et al. Practical evasion of a learning-based classifier: A case study. In *2014 IEEE symposium on security and privacy*, pages 197–211. IEEE, 2014.

[36] M. Leslous, V. V. T. Tong, J.-F. Lalande, and T. Genet. Gpfinder: tracking the invisible in android malware. In *MALWARE*. IEEE, 2017.

[37] J. Li, S. Ji, T. Du, B. Li, and T. Wang. Textbugger: Generating adversarial text against real-world applications. *NDSS*, 2019.

[38] D. Lowd and C. Meek. Good word attacks on statistical spam filters. In *CEAS*, volume 2005, 2005.

[39] D. Maiorca, G. Giacinto, and I. Corona. A Pattern Recognition System for Malicious PDF Files Detection. In *Intl. Workshop on Machine Learning and Data Mining in Pattern Recognition*. Springer, 2012.

[40] D. Maiorca, I. Corona, and G. Giacinto. Looking at the bag is not enough to find the bomb: an evasion of structural methods for malicious pdf files detection. In *ASIACCS*. ACM, 2013.

[41] D. Maiorca, B. Biggio, and G. Giacinto. Towards robust detection of adversarial infection vectors: Lessons learned in pdf malware. *arXiv preprint*, 2019.

[42] M. Melis, D. Maiorca, B. Biggio, G. Giacinto, and F. Roli. Explaining black-box android malware detection. In *EUSIPCO*. IEEE, 2018.

[43] B. Miller, A. Kantchelian, M. C. Tschantz, S. Afroz, R. Bachwani, R. Faizullabhoy, L. Huang, V. Shankar, T. Wu, G. Yiu, et al. Reviewer Integration and Performance Measurement for Malware Detection. In *DIMVA*. Springer, 2016.

[44] D. Mitchell, B. Selman, and H. Levesque. Hard and easy distributions of sat problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, AAAI'92, pages 459–465. AAAI Press, 1992. ISBN 0-262-51063-4. URL http://dl.acm.org/citation.cfm?id=1867135.1867206.

[45] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *ACSAC*, 2007.

[46] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami. The limitations of deep learning in adversarial settings. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 372–387. IEEE, 2016.

[47] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*, 2017.

[48] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-Learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[49] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro. TESSERACT: Eliminating Experimental Bias in Malware Classification across Space and Time. In *28th USENIX Security Symposium*, Santa Clara, CA, 2019. USENIX Association. USENIX Sec.

[50] B. C. Pierce and C. Benjamin. *Types and programming languages*. MIT press, 2002.

[51] E. Quiring, A. Maier, and K. Rieck. Misleading authorship attribution of source code using adversarial learning. *USENIX Security Symposium*, 2019.

[52] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. K. Nicholas. Malware detection by eating a whole exe. In *AAAI Workshops*, 2018.

[53] I. Rosenberg, A. Shabtai, L. Rokach, and Y. Elovici. Generic black-box end-to-end attack against state of the art API call based malware classifiers. In *RAID*. Springer, 2018.

[54] B. Selman, D. G. Mitchell, and H. J. Levesque. Generating hard satisfiability problems. *Artif. Intell.*, 81(1-2):17–29, 1996. doi: 10.1016/0004-3702(95)00045-3. URL https://doi.org/10.1016/0004-3702(95)

00045-3.
[55] M. Sharif, S. Bhagavatula, L. Bauer, and M. K. Reiter. Accessorize to a crime: Real and stealthy attacks on state-of-the-art face recognition. In *ACM CCS*. ACM, 2016.
[56] C. Smutz and A. Stavrou. Malicious pdf detection using metadata and structural features. In *ACSAC*. ACM, 2012.
[57] N. Šrndic and P. Laskov. Detection of malicious pdf files based on hierarchical document structure. In *NDSS*, 2013.
[58] O. Suciu, R. Mărginean, Y. Kaya, H. Daumé III, and T. Dumitraş. When Does Machine Learning FAIL? Generalized Transferability for Evasion and Poisoning Attacks. *USENIX Security Symposium*, 2018.
[59] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing properties of neural networks. *ICLR*, 2014.
[60] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas. Sok: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In *IEEE Symposium on Security and Privacy*, 2015.
[61] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*. IBM Corp., 2010.
[62] G. Vigna and D. Balzarotti. When malware is packin' heat. In *USENIX ENIGMA*, 2018.
[63] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449. IEEE Press, 1981. URL http://dl.acm.org/citation.cfm?id=800078.802557.
[64] W. Xu, Y. Qi, and D. Evans. Automatically evading classifiers. In *NDSS*, 2016.
[65] W. Yang, D. Kong, T. Xie, and C. A. Gunter. Malware detection in adversarial settings: Exploiting feature evolutions and confusions in android apps. In *ACSAC*. ACM, 2017.
[66] G. Zizzo, C. Hankin, S. Maffeis, and K. Jones. Adversarial machine learning beyond the image domain. In *ACM DAC*, 2019.

APPENDIX

A. Symbol Table

Table II reports the major symbols used throughout the paper. The readers can use this table as a reference for the notation.

TABLE II
TABLE OF SYMBOLS.

SYMBOL	DESCRIPTION
$\mathcal{Z}$	Problem space (i.e., input space).
$\mathcal{X}$	Feature space $\mathcal{X} \subseteq \mathbb{R}^n$.
$\mathcal{Y}$	Label space.
φ	Feature mapping function $\varphi : \mathcal{Z} \longrightarrow \mathcal{X}$.
h_i	Discriminant function $h_i : \mathcal{X} \longrightarrow \mathbb{R}$ that assigns object $\boldsymbol{x} \in \mathcal{X}$ to discriminant score in $\mathbb{R}$ (e.g., distance from hyperplane).
g	Classifier $g : \mathcal{X} \longrightarrow \mathcal{Y}$ that assigns object $\boldsymbol{x} \in \mathcal{X}$ to class $y \in \mathcal{Y}$. Also known as *decision function*. It is defined based on the output of the discriminant functions $h_i, \forall i \in \mathcal{Y}$.
$\mathcal{L}_y$	Loss function $\mathcal{L}_y : \mathcal{X} \times \mathcal{Y} \longrightarrow \mathbb{R}$ of object $\boldsymbol{x} \in \mathcal{X}$ with respect to class $y \in \mathcal{Y}$.
$f_{y,\kappa}$	Attack objective function $f_{y,\kappa} : \mathcal{X} \times \mathcal{Y} \times \mathbb{R} \longrightarrow \mathbb{R}$ of object $\boldsymbol{x} \in \mathcal{X}$ with respect to class $y \in \mathcal{Y}$ with maximum confidence $\kappa \in \mathbb{R}$.
f_y	Compact notation for $f_{y,0}$.
Ω	Feature-space constraints.
$\boldsymbol{\delta}$	$\boldsymbol{\delta} \in \mathbb{R}^n$ is a symbol used to denote a feature-space perturbation vector.
$\boldsymbol{\varepsilon}$	Side-effect feature vector.
T	Transformation $T : \mathcal{Z} \longrightarrow \mathcal{Z}$.
$\mathbf{T}$	Transformation sequence $\mathbf{T} = T_1 \circ T_2 \circ \cdots \circ T_n$.
$\mathcal{T}$	Space of available transformations.
Υ	Suite of automated tests $\tau \in \Upsilon$ to verify preserved semantics.
Π	Suite of manual tests $\pi \in \Pi$ to verify plausibility. In particular, $\pi(z) = 1$ if $z \in \mathcal{Z}$ is plausible, else $\pi(z) = 0$.
Λ	Set of absent artifacts from object $z \in \mathcal{Z}$.
$\mathbf{A}_\Lambda$	Artifact removal operator $\mathbf{A}_\Lambda : \mathcal{Z} \longrightarrow \mathcal{Z}$.
Γ	Problem-space constraints Γ, consisting of $\{\Pi, \Upsilon, \mathcal{T}, \Lambda\}$.
$\mathcal{D}$	Training dataset.
$\boldsymbol{w}$	Model hyper-parameters.
Θ	Knowledge space.
θ	Threat model assumptions $\theta \in \Theta$; more specifically, $\theta = (\mathcal{D}, \mathcal{X}, g, \boldsymbol{w})$. A *hat* symbol is used if only estimates of parameters are known.

B. Threat Model

The threat model must be defined in terms of attacker *knowledge* and *capability*, as in related literature [8, 16, 58]. While the attacker knowledge is represented in the same way as in the traditional feature-space attacks, their capability also includes the problem-space constraints Γ. For completeness, we report the threat model formalization proposed in Biggio and Roli [8].

Attacker Knowledge. We represent the knowledge as a set $\theta \in \Theta$ which may contain (i) training data $\mathcal{D}$, (ii) the feature set $\mathcal{X}$, (iii) the learning algorithm g, along with the loss function $\mathcal{L}$ minimized during training, (iv) the model parameters/hyperparameters $\boldsymbol{w}$. A parameter is marked with a *hat* symbol if the attacker knowledge of it is limited or only an estimate (i.e., $\hat{\mathcal{D}}$, $\hat{\mathcal{X}}$, $\hat{g}$, $\hat{\boldsymbol{w}}$). There are three major scenarios [8]:

- *Perfect Knowledge (PK) white-box attacks*, in which the attacker knows all parameters and $\theta_{PK} = (\mathcal{D}, \mathcal{X}, g, \boldsymbol{w})$.
- *Limited Knowledge (LK) gray-box attacks*, in which the attacker has some knowledge on the target system. Two common settings are LK with Surrogate Data (LK-SD), where $\theta_{LK-SD} = (\hat{\mathcal{D}}, \mathcal{X}, g, \hat{\boldsymbol{w}})$, and LK with Surrogate Learners, where $\theta_{LK-SL} = (\hat{\mathcal{D}}, \mathcal{X}, \hat{g}, \hat{\boldsymbol{w}})$.
- *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 (e.g., only static analysis in programs, or structural features in PDFs). In this case, $\theta_{LK} = (\hat{\mathcal{D}}, \hat{\mathcal{X}}, \hat{g}, \hat{\boldsymbol{w}})$.

Note that θ_{PK} and θ_{LK} imply knowledge of any defenses used to secure the target system against adversarial examples,

depending on the degree to which each element is known [15].

Attacker Capability. The capability of an attacker is expressed in terms of his ability to modify feature space and problem space, i.e., the attacker capability is described through feature-space constraints Ω and problem-space constraints Γ.

We observe that the attacker's knowledge and capability can also be expressed according to the FAIL [58] model as follows: knowledge of *Features* $\mathcal{X}$ (F), the learning *Algorithm* g (A), *Instances* in training $\mathcal{D}$ (I), *Leverage* on feature space and problem space with Ω and Γ (L).

More details on the threat models can be found in [8, 58].

C. Theorem Proofs

Proof of Theorem 1. We proceed with a proof by contradiction. Let us consider a problem-space object $z \in \mathcal{Z}$ with features $\boldsymbol{x} \in \mathcal{X}$, which we want to misclassify to a target class $t \in \mathcal{Y}$. Without loss of generality, we consider a low-confidence attack, with desired attack confidence $\kappa = 0$ (see Equation 3). We assume by contradiction that there is no solution to the feature-space attack; more formally, that there is no solution $\boldsymbol{\delta}^* = \arg\min_{\boldsymbol{\delta} \in \mathbb{R}^n : \boldsymbol{\delta} \models \Omega} f_t(\boldsymbol{x} + \boldsymbol{\delta})$ that satisfies $f_t(\boldsymbol{x} + \boldsymbol{\delta}^*) < 0$. We now try to find a transformation sequence $\mathbf{T}$ such that $f_t(\varphi(\mathbf{T}(z))) < 0$. Let us assume that $\mathbf{T}^*$ is a transformation sequence that corresponds to a successful problem-space attack. By definition, $\mathbf{T}^*$ is composed by individual transformations: a first transformation T_1, such that $\varphi(T_1(z)) = \boldsymbol{x} + \boldsymbol{\delta}_1$; a second transformation T_2 such that $\varphi(T_2(T_1(z)) = \boldsymbol{x} + \boldsymbol{\delta}_1 + \boldsymbol{\delta}_2$; a k-th transformation $\varphi(T_k(\cdots T_2(T_1(z)))) = \boldsymbol{x} + \sum_k \boldsymbol{\delta}_k$. We recall that the feature-space constraints are determined by the problem-space constraints, i.e., $\Gamma \vdash \Omega$, and that, with slight abuse of notation, we can write that $\Omega \subseteq \Gamma$; this means that the search space allowed by Γ is smaller or equal than that allowed by Ω. Let us now replace $\sum_k \boldsymbol{\delta}_k$ with $\boldsymbol{\delta}^\dagger$, which is a feature-space perturbation corresponding to the problem-space transformation sequence $\mathbf{T}$, such that $f_t(\boldsymbol{x} + \boldsymbol{\delta}^\dagger) < 0$ (i.e., the sample is misclassified). However, since the constraints imposed by Γ are stricter or equal than those imposed by Ω, this means that $\boldsymbol{\delta}^\dagger$ must be a solution to $\arg\min_{\boldsymbol{\delta} \in \Omega} f_t(\boldsymbol{x} + \boldsymbol{\delta})$ such that $f_t(\boldsymbol{x} + \boldsymbol{\delta}^\dagger) < 0$. However, this is impossible, because we hypothesized that there was no solution for the feature-space attack under the constraints Ω. Hence, having a solution in the feature-space attack is a *necessary condition* for finding a solution for the problem-space attack.

Proof of Theorem 2. The existence of a feature-space attack (Equation 12) is the necessary condition, which has been already proved for Theorem 1. Here, we need to prove that, with Equation 13, the condition is also sufficient for the attacker to find a problem-space transformation that misclassifies the object. Another way to write Equation 13 is to consider that the attacker knows transformations that affect individual features only (modifying more than one feature will result as a composition of such transformations). Formally, for any object $z \in \mathcal{Z}$ with features $\varphi(z) = \boldsymbol{x} \in \mathcal{X}$, for any feature-space dimension X_i of $\mathcal{X}$, and for any value $v \in domain(X_i)$, let us assume the attacker knows a valid problem-space transformation sequence $\mathbf{T} : \mathbf{T}(z) \models \Gamma, \varphi(\mathbf{T}(z)) = \boldsymbol{x}'$, such that:

$$x'_i = x_i + v, \qquad x_i \in \boldsymbol{x}, x'_i \in \boldsymbol{x}' \tag{14}$$

$$x'_j = x_j, \qquad \forall j \neq i, x_j \in \boldsymbol{x}, x'_j \in \boldsymbol{x}' \tag{15}$$

Intuitively, these two equations refer to the existence of a problem-space transformation $\mathbf{T}$ that affects only one feature X_i in $\mathcal{X}$ by any amount $v \in domain(X_i)$. In this way, given *any* adversarial feature-space perturbation $\boldsymbol{\delta}^*$, the attacker is sure to find a transformation sequence that modifies each individual feature step-by-step. In particular, let us consider $idx_0, \ldots, idx_{q-1}$ corresponding to the $q > 0$ values in $\boldsymbol{\delta}^*$ that are different from 0 (i.e., values corresponding to an actual feature-space perturbation). Then, a transformation sequence $\mathbf{T} : \mathbf{T}(z) \models \Gamma, \mathbf{T} = \mathbf{T}^{idx_0} \circ \mathbf{T}^{idx_1} \circ \cdots \circ \mathbf{T}^{idx_q}$ can always be constructed by the attacker to satisfy $\varphi(\mathbf{T}(z)) = \boldsymbol{x} + \boldsymbol{\delta}^*$. We highlight that we do not consider the existence of a specific transformation in $\mathcal{Z}$ that maps to $\boldsymbol{x} + \boldsymbol{\delta}^*$ because that may not be known by the attacker; hence, the attacker may never learn such a specific transformation. Thus, Equation 13 must be valid for all possible perturbations within the considered feature space.

D. 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 ensure the intractability of such an analysis, we follow the work of Moser et al. [45] and build opaque predicates using a formulation of the 3-SAT problem such that resolving the truth value of the predicate is equivalent to solving the NP-complete 3-SAT problem.

The k-satisfiability (k-SAT) problem asks whether the variables of a Boolean logic formula can be consistently replaced with `True` or `False` in such a way that the entire formula evaluates to `True`; if so the formula is said to be *satisfiable*. Such a formula is easily expressed in its conjunctive normal form

$$\bigwedge_{i=1}^{m} (V_{i1} \vee V_{i2} \vee ... \vee V_{ik})$$

where $V_{ij} \in \{v_1, v_2, ..., v_n\}$ are Boolean variables and k is the number of variables per clause.

Importantly, when $k = 3$, formulas are only NP-Hard in the worst case—30% of 3-SAT problems are in P [54]. This baseline guarantee is not sufficient as our injected code should never execute. Additionally, we require a large number of random predicates to reduce commonality between the synthetic portions of our generated examples.

To consistently generate NP-Hard k-SAT problems we use *Random k-SAT* [54] 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. [54] showed that n should be

Listing 1. Simplified example of an opaque predicate generated by JSketch. The opaque predicate wraps an *adapted vein* that calls a class containing benign features. Note that while we render the equivalent Java here for clarity, the actual transplantation occurs at a lower level of abstraction (Dalvik bytecode). The Random k-SAT parameters shown are our ideal parameters; in practice they are modulated around these values as part of the JSketch synthesis in order to avoid them becoming fingerprintable (e.g., having common length boolean arrays and loops between all predicates).

```
void opaque() {
  Random random = new Random();
  this();
  boolean[] arrayOfBoolean = new boolean[40];
  byte b1;
  for (b1 = 0; b1 < arrayOfBoolean.length; b1++)
    arrayOfBoolean[b1] = random.nextBoolean();
  b1 = 1;
  for (byte b2 = 0; b2 < 184.0D; b2++) {
    boolean bool = false;
    for (byte b = 0; b < 3; b++)
      bool |= arrayOfBoolean[random.nextInt(
          arrayOfBoolean.length)];
    if (!bool)
      b1 = 0;
  }
  if (b1 != 0) {

    // Beginning of adapted vein
    Context context = ((Context)this).
        getApplicationContext();
    Intent intent = new Intent();
    this(this, h.a(this, cxim.qngg.TEhr.sFiQa.class));
    intent.putExtra("1", h.p(this));
    intent.addFlags(268435456);
    startActivity(intent);
    h.x(this);
    return;
    // End of adapted vein

  }
}
```

at least 40 in order to ensure the formulas are hard to resolve. Additionally, 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. These experiments led to the following conjecture.

Threshold Conjecture [54]. Let us define c^* as the threshold at which 50% of the formulas are satisfiable. For $m/n < c^*$, as $n \to \infty$, the formula is satisfiable with probability 100%, and for $m/n > c^*$, as $n \to \infty$, the formula is unsatisfiable with probability 100%.

The current state-of-the-art for c^* is $3.42 < c^* \approx 4.3 < 4.51$ for 3-SAT [30, 44, 54]. We use this conjecture to ensure that the formulas used for predicates are unsatisfiable with high probability, i.e., that the predicate is likely a contradiction and will always evalute to `False`.

Additionally we discard any generated formulas that fall into two special cases of 3-SAT that are polynomially solvable:

- **2-SAT:** The construction may be 2-SAT if it can be expressed as a logically equivalent 2CNF formula [33].
- **Horn-SAT:** If at most one literal in a clause is positive, it is a *Horn clause*. If all clauses are Horn clauses, the formula is Horn-SAT and solvable in linear time [19].

We tested 100M Random 3-SAT trials using the fixed clause-length model with parameters $n \simeq 40, m \simeq 184, c^* \simeq 4.6$. All (100%) of the generated constructions were unsatisfiable (and evaluated to `False` at runtime) which aligns with the findings of Selman et al. [54]. This probability is sufficient to prevent execution with near certainty.

To further reduce artifacts introduced by reusing the same predicate, we use *JSketch* [29], a sketch-based program synthesis tool, to randomly generate new predicates prior to injection with some variation while maintaining the required properties. Post-transplantation, we verify for each adversarial example that Soot's program optimizations have not been able to recognize and eliminate them. An example of a generated opaque predicate (rendered in equivalent Java rather than Dalvik bytecode) is shown in Listing 1.

E. 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.

We now describe the details of our implementation of the Sec-SVM approach [18]. To have have full control of the training procedure, we approximate the linear SVM as a *single-layer* neural network (NN) using PyTorch [47]. We recall that the main intuition behind Sec-SVM is that classifier weights are distributed more evenly in order to force an attacker to modify more features to evade detection. Hence, we modify the training procedure so that the Sec-SVM weights are bounded by a *maximum weight value* k at each training optimization step. Similarly to Demontis et al. [18], we perform feature selection for computational efficiency, since PyTorch does not support sparse vectors. We use an l_2 (Ridge) regularizer to select the top 40,000 features which reduces AUROC by only 2%. This performance retention follows from recent results that shows SVM tends to overemphasize a subset of features [42]. To train the Sec-SVM, we perform an extensive hyperparameter grid-search: with Adam and Stochastic Gradient Descent (SGD) optimizers; training epochs of 5, 10, 20, 40, and 80; batch sizes from 2^0 to 2^{12}; learning rate from 10^0 to 10^{-5}. We identify the best single-layer NN configuration for our training data to have the following parameters: Stochastic Gradient Descent (SGD), batch size 1024, learning rate 10^{-3}, and 10 training epochs. We then perform a grid-search of the Sec-SVM hyperparameter k (i.e., the maximum weight absolute value [18]) by clipping weights during training iterations. We start from $k = w_{max}$, where $w_{max} = \max_i(w_i)$ for all features i; we then continue reducing k until we reach a weight distribution similar to that reported in [18], while allowing a maximum performance loss of 2% in AUROC. In this way, we identify the best hyperparameter value for our setting as $k = 0.25$.

In §IV, Figure 2 reported the AUROC for the DREBIN classifier [5] in SVM and Sec-SVM modes. The SVM mode has been evaluated using the `LinearSVC` class of scikit-learn [48] that utilizes the LIBLINEAR library [21]; as in the DREBIN paper [5], we use hyperparameter C=1. The performance degradation of the Sec-SVM compared to the baseline SVM shown in Figure 2 is in part related to the defense itself (as detailed in [18]), and in part due to minor convergence issues (since our single-layer NN converges less effectively

Algorithm 1: Initialization (Ice-Box Creation)

Input: Discriminant function $h(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x} + b$, which classifies $\boldsymbol{x}$ as malware if $h(\boldsymbol{x}) > 0$, otherwise as goodware. Minimal app $z_{min} \in \mathcal{Z}$ with features $\varphi(z_{min}) = \boldsymbol{x}_{min}$.
Parameters: Number of features to consider n_f; number of donors per-feature n_d.
Output: Ice-box of harvested organs with feature vectors.

ice-box $\leftarrow \{\}$ ▷ Empty key-value dictionary.
$L \leftarrow$ List of pairs (w_i, i), sorted by increasing value of w_i.
$L' \leftarrow$ First n_f elements of L, then remove any entry with $w_i \geq 0$.
for (w_i, i) *in* L' **do**
  ice-box[i] $\leftarrow$ [] ▷ Empty list for gadgets with feature i.
  while length(ice-box[i])$< n_d$ **do**
    $z_j \leftarrow$ Randomly sample a benign app with feature $x_i = 1$.
    Extract gadget $\rho_j \in \mathcal{Z}$ with feature $x_i = 1$ from z_j.
    $s \leftarrow$ Software stats of ρ_j
    $z' \leftarrow$ Inject gadget ρ_j in app z_{min}.
    $(\boldsymbol{x}_{min} \vee \boldsymbol{e}_i \vee \boldsymbol{\varepsilon}_j) \leftarrow \varphi(z')$ ▷ $\boldsymbol{e}_i$ is a one-hot vector.
    $\boldsymbol{r}_j \leftarrow (\boldsymbol{e}_i \vee \boldsymbol{\varepsilon}_j) \leftarrow \varphi(z') \wedge \neg\boldsymbol{x}_{min}$ ▷ Gadget features obtained through set difference.
    if $h(\boldsymbol{r}_j) > 0$ **then**
      Discard the gadget;
    else
      Append $(\rho_j, \boldsymbol{r}_j, s)$ to ice-box[i]. ▷ Store gadget
return ice-box;

than the LIBLINEAR implementation of scikit-learn). We have verified with Demontis et al. [18] the correctness of our Sec-SVM implementation and its performance, for the analysis performed in this work.

F. 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 (as we evaluate in §IV).

Note that when using the minimal injection host z_{min} to calculate the features that will be induced by a gadget, features in the corresponding feature vector $\boldsymbol{x}_{min}$ should be noted and dealt with accordingly (i.e., discounted). In our case x_{min} contained the following three features:

```
{ "intents::android_intent_action_MAIN":1,
  "intents::android_intent_category_LAUNCHER":1,
  "activities::_MainActivity":1}
```

G. FlowDroid Errors

We performed extensive troubleshooting of FlowDroid [6] 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 FlowDroid, when modifying app Manifests; and FlowDroid injecting system libraries found on the classpath when they should be excluded.

Algorithm 2: Attack (Adv. Program Generation)

Input: Discriminant function $h(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x} + b$, which classifies $\boldsymbol{x}$ as malware if $h(\boldsymbol{x}) > 0$, otherwise as goodware. Malware app $z \in \mathcal{Z}$. Ice-box G.
Parameters: Problem-space constraints.
Output: Adversarial app $z' \in \mathcal{Z}$ such that $h(\varphi(z')) < 0$.

$\mathcal{T} \leftarrow$ Transplantation through gadget addition.
$\Upsilon \leftarrow$ Smoke test through app installation and execution in emulator.
$\Pi \leftarrow$ Plausibility by-design through code consolidation.
$\Lambda \leftarrow$ Artifacts from last column of Table I.
$\Gamma \leftarrow \{\mathcal{T}, \Upsilon, \Pi, \Lambda\}$
$s_z \leftarrow$ Software stats of z
$\boldsymbol{x} \leftarrow \varphi(z)$
$L \leftarrow$ [] ▷ Empty list.
$\mathbf{T}(z) \leftarrow$ Empty sequence of problem-space transformations.
for $(\rho_j, \boldsymbol{r}_j, s)$ *in* G **do**
  $\boldsymbol{d}_j \leftarrow \boldsymbol{r}_j \wedge \neg\boldsymbol{x}$ ▷ Feature-space contribution of gadget j.
  $score_j \leftarrow h(\boldsymbol{d}_j)$ ▷ Impact on decision score.
  Append the pair $(score_j, i, j)$ to L ▷ Feature i, Gadget j.
$L' \leftarrow$ Sort L by increasing $score_j$ ▷ Negative scores first.
for $(score_j, i, j)$ *in* L' **do**
  if z *has* $x_i = 1$ **then**
    Do nothing; ▷ Feature i already present.
  else if z *has* $x_i = 0$ **then**
    $(\rho_j, \boldsymbol{r}_j, s) \leftarrow$ element j in ice-box G
    if check_feasibility(s_z, s) *is True* **then**
      $\boldsymbol{x} \leftarrow (\boldsymbol{x} \vee \boldsymbol{e}_i \vee \boldsymbol{\varepsilon}_j)$ ▷ Update features of z.
      Append transplantation $T \in \mathcal{T}$ of gadget ρ_j in $\mathbf{T}(z)$.
      if $h(\boldsymbol{x}) < 0$ **then**
        Exit from cycle; ▷ Attack gadgets found.
$z' \leftarrow$ Apply transformation sequence $\mathbf{T}(z)$ ▷ Inject chosen gadgets.
if $h(\varphi(z')) < 0$ **and** $\mathbf{T}(z) \models \Gamma$ **then**
  return z';
else
  return Failure;