How to Train your Antivirus: RL-based Hardening through the Problem Space

Ilias Tsingenopoulos*
DistriNet, KU Leuven
Belgium

Jacopo Cortellazzi*
King's College London
University College London
United Kingdom

Branislav Bošanský
Gen Digital
Czech Republic

Simone Aonzo
Eurecom
France

Davy Preuveneers
DistriNet, KU Leuven
Belgium

Wouter Joosen
DistriNet, KU Leuven
Belgium

Fabio Pierazzi
King's College London
United Kingdom

Lorenzo Cavallaro
University College London
United Kingdom

Abstract

ML-based malware detection on dynamic analysis reports is vulnerable to both evasion and spurious correlations. In this work, we investigate a specific ML architecture employed in the pipeline of a widely-known commercial antivirus, with the goal to harden it against adversarial malware. Adversarial training, the most reliable defensive technique that can confer empirical robustness, is not applicable out of the box in this domain, for the principal reason that gradient-based perturbations rarely map back to feasible problem-space programs. We introduce a novel Reinforcement Learning approach for constructing adversarial examples, a constituent part of adversarially training a model against evasion. Our approach comes with multiple advantages. It performs modifications that are feasible in the problem-space, and only those; thus it circumvents the inverse mapping problem. It also makes it possible to provide theoretical guarantees on the robustness of the model against a well-defined set of adversarial capabilities. Our empirical exploration validates our theoretical insights, where we can consistently reach 0% Attack Success Rate after a few adversarial retraining iterations.

CCS Concepts

• **Security and privacy → Malware and its mitigation**; • **Computing methodologies → Reinforcement learning**; **Adversarial learning**.

ACM Reference Format:
Ilias Tsingenopoulos, Jacopo Cortellazzi, Branislav Bošanský, Simone Aonzo, Davy Preuveneers, Wouter Joosen, Fabio Pierazzi, and Lorenzo Cavallaro. 2024. How to Train your Antivirus: RL-based Hardening through the Problem Space. In *The 27th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2024), September 30-October 02, 2024, Padua, Italy*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3678890.3678912

*Equal contribution

RAID 2024, September 30-October 02, 2024, Padua, Italy

ACM ISBN 979-8-4007-0959-3/24/09
https://doi.org/10.1145/3678890.3678912

1 Introduction

Adversarial examples have been extensively explored in numerous domains. While the fundamental challenge they pose to learning is fascinating on its own scientific accord, they also represent a significant threat to the widespread adoption of machine learning solutions in security-critical scenarios. At the same time, there is an eagerness by many cybersecurity vendors to adopt deep-learning approaches for malware detection and classification, for two main reasons. First, adversarial examples have existed in security domains long before they emerged in artificial intelligence (AI), as straightforward ways for adversaries to adapt to signature- and rule-based decision making—for instance variants from the same malware family. Secondly, while rule-based decisions can claim near zero false positives, they perform poorly in distributional shifts [69], which are further intensified under adversarial agency. Hence, the promise that AI brings is to learn general and ideally robust rules of inference in the presence of distributional shifts, both naturally and adversarially induced.

Adversarial malware examples have been predominantly explored in the feature space and on static analysis features. With the prevalence of obfuscation, polymorphic strains, and packing, static analysis is becoming less relevant for malware detection [2]. Behavioral analysis can be a very effective tool as it circumvents the challenges of obfuscated and packed binaries by exposing their runtime behavior, considered a reliable fingerprint of malicious intent. The representation of this behavior can range from API call sequences [60, 65], to fine-grained modeling of process behavior [16], and fully textual reports [47]. As dynamic analysis tools gain traction and widespread adoption, novel vulnerabilities are introduced: adversarial malware behavior that evades detection while preserving the intended functionality. Notably, this behavior can be learned as a direct consequence of dynamic analysis itself, the very process that is supposed to thwart it.

In other domains, for example image classification, the state-of-the-art defense against evasion is adversarial training [45, 66]. We can think of adversarial training as a process of generating counterfactuals to the data the model is training on: if these features were different under certain constraints, would that still be the same example? While the most effective approach for model hardening, adversarial training remains largely empirical: it is still intractable to provide guarantees on the operational robustness for any but the most trivial models. Furthermore, it is not directly applicable in domains that exhibit a large problem- to feature-space gap [55]. In dynamic analysis-based malware detection, there are several degrees of separation from the original program to the decision of the model. Not only it is challenging to map any gradient-based perturbations computed on the feature representation back to *viable* programs, but as we show in this work it is also irrelevant when defending against real-world attackers.

The imperceptible perturbation model of adversarial examples bears little resemblance to adversarial malware, which always manifest in the problem space and under a very different set of constraints. As adversarial examples are essentially out-of-distribution, it is impractical to proactively harden a model against *all* possible variations—especially in domains where they are not constrained by similarity to the original ones. It is well established that adversarial training can reduce the performance on clean samples; a lesser known property is that it can also reduce the robustness to attacks *other* than the one it was performed with [31]. As hardened models are not universally robust but instead biased towards specific kinds of distributional shifts, adversarial training has the potential to harm robustness in real-world settings. It then follows that in the domain of malware detection, adversarial training is important to be performed with the *full range* of attacks that are *possible* through the problem space—and that the threat analysis has rendered as such—and possibly *only* against those.

In this paper, we make the following contributions:

- We first obtain a wide range of Windows binaries spanning from 2012 to 2020, from various malware families to verified goodware; then we execute them in CAPEv2 to generate dynamic analysis reports. To foster research on dynamic analysis, we release our dataset in the following URL: https://paperswithcode.com/dataset/autorobust
- We justify theoretically how, under certain conditions, adversarial training through the problem-space is advantageous to gradient-based approaches like PGD [45], and define the concise set of transformations that are available for behavioral representations.
- We introduce AutoRobust, a novel RL-driven methodology for performing adversarial training and observe that, within several attack iterations followed by retraining, the model consistently moves from entirely vulnerable (~100% ASR) to fully robust (~0% ASR). Notably, AutoRobust does not assume anything about the underlying model or the adversarial capabilities apart from them being functionality-preserving. Thus, it is applicable to *any* problem-space transformations that related works investigate.
- The key takeaways of our work are the ability of our approach to bypass an ML component of a commercial antivirus, as well as the importance of incorporating adversarial training through the problem-space. To enable reproducibility and follow-up work, we open-source our codebase: https://github.com/s2labres/AutoRobust

2 Background & Related Work

In this section we discuss prior work on the domain and identify open problems and research gaps, setting in this way the ground for our approach.

2.1 Malware detection

Prior research has employed a wide range of classifiers to identify malicious software, in diverse contexts [1]. On Windows, the context we scope to, related work has operated on representations ranging from the whole binary [57], to control flow graphs [3], and static analysis [15]. Static analysis however has clear limitations in malware detection and often dynamic analysis is employed as a complementary approach. Recently, it has been demonstrated that ML models built solely on static analysis are not able to distinguish between benign and malicious samples that are packed [2]. This work reviews a wide range of ML models built on static analysis and concludes that they detect packing rather than malice. Given encryption, polymorphic strains, and packing, the efficacy of static analysis or signature-based detection deteriorates.

To capture the actual behavior of a program, researchers and analysts have relied on dynamic analysis [17]. Today there are multiple vendors that provide sandboxes for dynamic analysis; as enterprise or cloud solutions but also open source like CAPEv2 [10, 20, 56], used both in academia and in industry. Sandboxes typically generate a structured textual description of the full program behaviour, from files accessed and API sequences [52, 59, 60] to any malicious or evasive techniques used [27, 46, 67]. Malware detection has historically relied heavily on signatures and indicators of compromise, with the evident shortcoming in the inability to generalize to new strains or zero-day malware. To thwart such threats, analysts require models that can extrapolate from the observed behavior of existing malware to new samples.

However, going through dynamic analysis reports is a highly involving and time-consuming process, so given the immense amount of data that can be generated there is a great incentive to automate this process. Learning directly from raw data inputs by obviating the need for feature engineering has been very effective in numerous applications; manual feature engineering is not only labor-intensive, but also prone to introducing human bias. By processing the raw input this occurs, ML models have the capacity to construct important features for inference and provide further insights to experts, beyond any a priori notions on what constitutes malware behavior. To this end, recent work has introduced an approach that fully automates this process, named Hierarchical Multiple Instance Learning (HMIL) [47, 54]. Its relevance and significance lie in the fact that it can be directly applied to the results of dynamic analysis, as it renders structured text reports of arbitrary size end-to-end differentiable. This makes it possible to be integrated with any ML pipeline, by enabling learning on JSON data in their raw form that

can span hundreds of thousands of tokens, a challenge even for the largest of LLMs.

2.2 Problem-Space Attacks

While adversarial attacks have been investigated in a multitude of domains [6], on malware they were initially explored in the feature-space [32, 34, 63]. However there is a growing body of works that performs attacks on the full processing pipeline, even up to perturbing the original binary [4, 27, 40, 51, 55, 68], using approaches like process chopping [36], return-oriented programming [50], bytecode modifications [8], and opcode n-grams [43]. Furthermore, adversarial malware can be generated on other models and still transfer to (that is successfully evade) a malware classifier under consideration [24]. These works illustrate how a brittle feature representation can render a model vulnerable to evasion, especially when the model does not process the actual program behavior. Performing attacks through the problem space and the ensuing constraints that this imposes is an accurate representation of the actual level of threat; an important step as only a realistic assessment of that threat can build effective defenses.

In domains like malware, adversarial attacks are much more difficult to carry out, precisely because of the problem-feature space gap. In real-world settings, problem and feature spaces are well apart: any approach that computes perturbations in the feature space is always confounded by the task of mapping them back to the problem space in a feasible manner [62]. In the malware domain specifically, the non-invertibility of the mapping from features to programs, the preservation of functionality, and side-effects on the feature representation when valid perturbations are made in the problem space, make this task non-trivial. To that end, several previous works focus on the problem-space generation and verification of functionality-preserving adversarial malware [22, 23, 39]. In constructing adversarial malware it would therefore be advantageous if the agency that generated them was situated *directly* on the problem space.

2.3 Defenses & Mitigations

While ML-based malware detection is typically one of the various modules for detection a commercial antivirus solution employs [7], the defensive aspect has received relatively less attention compared to the offensive one. Approaches range from feature-space hardening [28], to combining distillation with adversarial training [58], to defenses tailored to API call-based models [59]. In the defensive literature several important principles have been investigated, from using model ensembles to the necessity of adversarial training [42], however such defenses are predominantly performed on the feature representations that become inputs for the ML models, or without consideration to problem space constraints.

In the long history of proposed and eventually broken defenses against adversarial examples [66], mainly one has stood the test of time: adversarial training [45]. The objective of adversarial training is to generate adversarial examples during training: given dataset $D = (x_i, y_i)_{i=1}^{n}$ with classes C where $x_i \in \mathbb{R}^d$ is a clean example and $y_i \in 1, ..., C$ is the associated label, the goal is to solve the following *min-max* optimization problem:

$$\min_{\phi} \mathbb{E}_{x_i, y_i \sim D} \max_{\|\delta_i\|_{L_p} \leq \epsilon} \mathcal{L}(h_\phi(x_i + \delta_i), y_i) \tag{1}$$

where $x_i + \delta_i$ is an adversarial example of x_i, $h_\phi : \mathbb{R} \to \mathbb{R}^{\mathbb{C}}$ is a hypothesis function and $\mathcal{L}(h_\phi(x_i + \delta_i), y_i)$ is the loss function for the adversarial example $x_i + \delta_i$. The inner maximization loop finds an adversarial example of x_i with label y_i for a given L_p-norm (with $L_p \in \{0, 1, 2, \inf\}$), such that $\|\delta_i\|_l \leq \epsilon$ and $h_\phi(x_i + \delta_i) \neq y_i$. The outer loop is the standard minimization task typically solved with stochastic gradient descent.

A widely accepted principle in robustness to distributional shift literature [30, 41] is that the lack of robustness can be largely attributed to spurious correlations, that is the model latching onto superficial information and artifacts manifesting as causal links between the dependent and independent variables. Analogously to computer vision, in the malware domain this information can appear unintuitive or downright irrelevant to us, for instance the specific filenames used. Yet such artifacts can still prove useful for generalization *within* the independent and identically distributed (IID) settings, as these are defined by the currently available data.

Adversarial training is, in essence, an exploration on out-of-distribution (OOD) variations of these data, under a set of constraints. As such, if this exploration is performed without regard to what variations are possible and which constraints are there, it is difficult to ascertain what the effects on the clean and robust accuracy will be. Previous works have shown that undisciplined adversarial training can provide limited robustness or even make the model more vulnerable to other transformations or constraints [25, 33]. If we look beyond ℓ_p norm perturbations and take a broader view on robustness, it becomes clear that one cannot claim that adversarially trained models are robust *in general*, but rather biased towards *specific* distortions as these are generated by specific adversarial capabilities. It is entirely conceivable then that adversarial training can harm not only performance but robustness too in real-world settings.

If adversarial training has lackluster performance or even the potential to diminish robustness when performed irrespective of transformations, then it should be performed under those and *only* those that one wants to defend against. In the context of dynamic analysis, this would elicit an exhaustive search to identify the concise set of transformations an adversary can make in the problem space. As dynamic analysis documents program behavior as it is expressed over time, this set of transformations can result to a very diverse set of possible adversarial examples, especially if one does not constrain the perturbation amount.

2.4 Research Gap

While adversarial training remains the most promising path to explore in defenses, the meaningfulness of L_p-norms and δ perturbations as constraints applies mostly in the computer vision domain, where visual similarity is important and where the majority of adversarial attacks and defenses were initially researched and performed. Every other domain, including computer vision but in real-world settings [61], will have slightly different to completely unrelated set of constraints. Generating adversarial malware in particular, demonstrates two key idiosyncrasies compared to computer

vision. Modifications are more difficult to perform, as there are hard constraints stemming from the preservation of the semantics and the original program functionality. However, only soft constraints apply on the total amount of modifications, as there is no strict requirement of resemblance to the original binary. Previous work has already explored adversarial training on malware classifiers and under such constraints [44]. They however focus on hardening models that operate on the raw-byte level of a binary, specifically the MalConv and AvastNet architectures.

In this work we focus on hardening a model that operates on reports generated by executing the malware in a sandbox and generating a dynamic analysis report. In this context then, the natural form that data occur in is textual; for this representation adversarial training, as is conventionally performed, can not be applied out of the box due to the following reasons:

- White box techniques, like FGSM and PGD, maximize the inner loop of the loss function (1) during training: this often results in feature-space perturbations that do not map back to valid program behavior.
- Dynamic analysis reports exhibit a huge variance in size (from ten tokens to hundreds of thousands), and the possible modifications on them are in theory unbounded. As not all possible adversarial examples are interesting or useful, it is better to reground the problem and make the problem tractable by focusing on those adversarial examples that are *feasible* through the problem space.
- Adversarial training remains an empirical hardening technique. It would be preferable if through a rigorous threat analysis we could show that against a specific and representative set of problem-space modifications, we could achieve a guaranteed level of robustness.

3 Threat Model

The methodology we conceive and develop in this work is a general framework for hardening ML-based classifiers $\mathcal{M}$ against adversaries that employ well-defined problem-space capabilities C for evasion. Under this formulation, it is independent of the underlying model $\mathcal{M}$ and capabilities C, and can thus accommodate a wide variety of them. In this section, we explicit our assumptions before we proceed to the concrete details of the design and our empirical evaluation. The threat model we consider focuses on the dynamic analysis report representation and is delineated as follows:

- **Assets:** Trained and deployed model $\mathcal{M}$ with corresponding weights w.
- **Adversaries:** Malware coders or malware-as-a-service (MaaS) actors.
- **Knowledge:** Given training data $\mathcal{D}$, feature representation $\mathcal{F}$, learning algorithm g, model $\mathcal{M}$, and explainer $\mathcal{E}$, we assume a Perfect Knowledge attacker that knows all $\theta_{PK} = (\mathcal{D}, \mathcal{F}, g, \mathcal{M}, \mathcal{E})$ [6].
- **Goal:** Generate adversarial malware, that is render the malware binary evasive with respect to the model $\mathcal{M}$ while preserving its original functionality.
- **Capabilities**: Make transformations on the binary that are reflected in the dynamic analysis report, as these are described in subsection 5.3. No dataset poisoning capabilities.
- **Defenses:** Use our framework to generate report variants for adversarially training the model $\mathcal{M}$.

In our threat model, assuming the perspective of the adversary, the objective is to produce an execution trace that would create a dynamic analysis report that is not detected as malicious by the target model $\mathcal{M}$. While this can be achieved by modifying the source code of the program in ways that preserve the original functionality [48], we consider this capability out of scope. A more representative and realistic threat is the attacker modifying the compiled binaries directly; we elaborate on how this can be achieved in Section 6. On the other hand, the defender's objective is to increase the robustness of model $\mathcal{M}$ to evasion, but similarly they have no access to the source code so all transformations are assumed to be performed on the compiled binary. This implies that all possible transformations Λ on the original binary will map to a subset of all the possible transformations in the feature representation Φ (see Fig. 1); the adversary cannot modify the representation in the input space of the model $\mathcal{M}$ at will.

4 AutoRobust

In this section we present AutoRobust, our general framework and methodology to harden ML-based models against problem-space attacks. Given known adversarial capabilities, we theoretically justify how our approach is preferable to gradient-based white-box hardening, and also demonstrate it empirically.

4.1 Approach

Let Ω be the latent space of a model $\mathcal{M}$ where inputs are mapped, for example the last layer before output. During training, the model learns such a latent space implicitly, with benign and malicious areas being well separated. Gradient-based attacks compute perturbations by defining an adversarial loss on the model and then backpropagating through it. With $\Phi \subset \Omega$ we represent all the possible representations in the *feature-space*. While every input sample x maps to a point in Φ, not every point in Φ maps to a valid input sample: the mapping that model $\mathcal{M}$ performs is injective but not surjective. We denote the space of valid input samples as $\Lambda \subset \Phi$. Consequently, following gradient-based perturbations will inadvertently result to points that are not in Λ and thus invalid, something that will require more computation to correct by mapping them back to Λ, and often deviating from the gradient direction itself.

Consider now an alternative to gradient-based perturbations: problem-space modifications from a capability set C that by definition produce only valid programs. Then the resulting reports are *guaranteed* to fall within Λ. If we denote as S the embedding in Λ of an original malware sample, then making a transformation $T \in C$ will produce a valid sample S' that is always in Λ. Then we can construe the adversarial task as a Markov Decision Process (MDP) to be solved: if we take the latent subspace Λ as the state space, and the transformations α_t as the action space, the goal is to sequentially apply transformations until the model $\mathcal{M}$ is evaded. Notably, this process can be optimised in gradient-based manner too [64], while always staying on the valid program space. An intuitive illustration of our approach is included in Figure 1.

In the domain of adversarial malware, the transition from gradient-based methodologies like PGD [45] or C&W [11] to problem-space

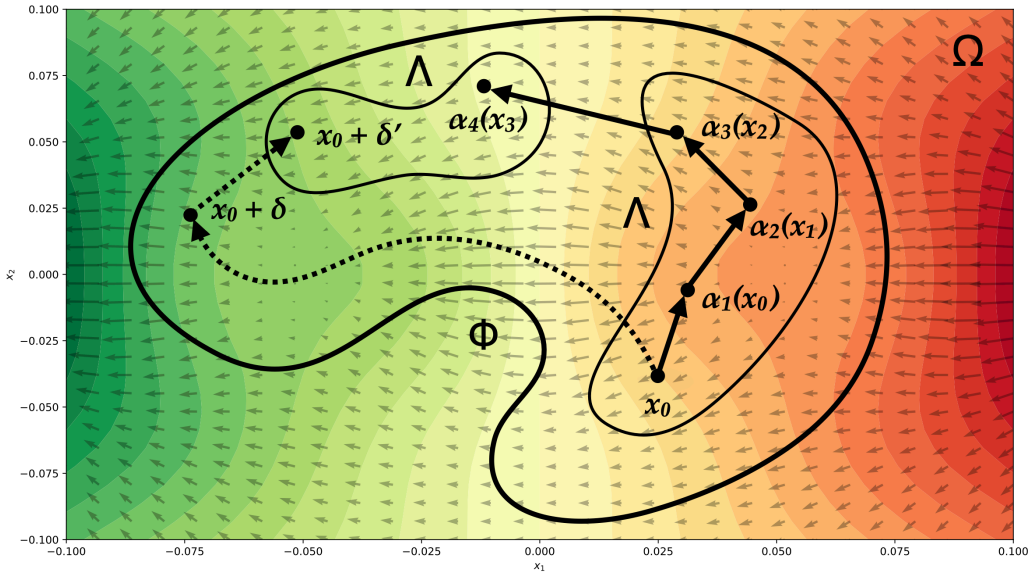


Figure 1: Comparison between traditional gradient-based attacks and AutoRobust. The dotted path shows a typical gradient-based attack: first perturbing to $x + \delta$ then projecting to $x + \delta'$ in the *feasible* problem-space Λ. Our approach (dense path) that employs transformations α_t in succession, moves by definition *only* within the feasible problem space Λ. The background displays a gradient field over the value of the discriminant function $h(x)$, with negative values (green) for the target class. The thick solid area Φ represents the feasible feature-space, while the areas denoted by Λ represent the feasible problem-space mapped to Φ.

attacks like GAMMA [22] and AIMED-RL [39] guarantees the preservation of semantics and functionality. AutoRobust, which as a framework is independent of the underlying modifications performed, comes with an additional benefit: if an agent with capability set $\mathcal{C}$ cannot find a feasible path to the adversarial side of the decision boundary of $\mathcal{M}$, then this state S^* is either unreachable under $\mathcal{C}$, or does not exist, and thus the agent is unable to evade the model. In this manner we can show that for a number of samples with starting states s_0, the model $\mathcal{M}$ is robust to the specific $\mathcal{C}$ with probability p, denoted as p-Robust.

Theorem 4.1 (Problem-Space p-Robustness). *Given model $\mathcal{M}$ and adversary $\mathcal{V}$ with problem-space capabilities $\mathcal{C}$, $\mathcal{M}$ is robust to problem-space evasion with probability p if and only if the expected reward of the optimal policy $\pi^*(a|s)$ in the corresponding MDP is $1 - p$.*

Proof. Let us assume a scalar reward for the adversary: 1 if they evade at time step $t > 0$ with the episode terminating afterwards, or 0 if they cannot evade up to a maximum time horizon $\mathcal{T}$. The objective of the adversary is to maximize the reward of their policy $\pi_\theta^{\mathcal{V}}(a|s)$, parameterized by θ, over a number of episodes N. Then the expected value of the reward $\mathcal{R}$ is:

$$\mathbb{E}_{\pi_\theta^{\mathcal{V}}}[\mathcal{R}] = \frac{1}{N}\sum_{i=1}^{N}\mathcal{R} \tag{2}$$

The optimal policy $\pi^*(a|s)$ is by definition the one that maximizes this expectation, and from the Policy Gradient Theorem [64] we can compute it in a gradient-based manner. If we bijectively map actions $a \in \mathcal{A}$ to transformations T in the capability set $\mathcal{C}$, and states $s \in \mathcal{S}$ to the latent space Λ of $\mathcal{M}$, then it is sound to construe this adversarial task as the corresponding MDP: transformations on current state s, deterministically lead to only one successor s'. Additionally, the Markov Property is preserved as this transition depends solely on s and transformation $\alpha_t \in T$. Now if we assume the value of the expectation is $1 - p$, and given that $\mathcal{R} \in \{0, 1\}$, from (2) we can show that:

$$\frac{1}{N}\sum_{i=1}^{N}\mathcal{R} = \frac{1}{N}\sum_{i=1}^{N}(0 \cdot P(ben) + 1 \cdot P(adv)) = P(adv) = 1 - p \tag{3}$$

It follows that the probability $P(ben)$ of the model $\mathcal{M}$ being robust to the adversary is p. The converse is also straightforward to show: if we assume that a model is p-Robust, then for $\mathcal{V}^*$ an optimal adversary, $P(ben) = p$, and from (2) the expectation is $1 - p$. □

We have shown how one can use RL to find optimal transformation policies in the problem-space. Afterwards, we proceed with the outer loop of Eq. (1) unchanged: the newly discovered adversarial examples together with the original, clean ones are used to retrain the model. This cycle is then repeated, where it becomes successively more difficult for the RL agent to construct adversarial examples. While the degree of robustness is estimated on the validation set of the generated adversarial examples, these are generated under the same agent policy which means they will fall under the same distribution; thus they cannot be a representative measurement of the actual level of robustness. The proper way to

assess that is by training the agent anew in a hold-out test set. Then given sufficient horizon to converge, the resulting optimal policy will indicate the level of p-Robustness, for this specific model $\mathcal{M}$ and this specific capability set C.

Generality. It should be noted that Theorem 4.1 does not assume anything about the discriminative model $\mathcal{M}$, while for the capability set C the only assumption we make is that transformations in this set are functionality preserving. Then their application and composition will result in valid programs whose embeddings that are *always* in Λ. This indicates that our methodology is applicable to *any* functionality preserving transformations in the problem-space; for instance to transformations directly on the binary that a growing literature investigates and performs [22, 23, 39, 40].

Hardening. We assert that in order to defend against realistic adversarial attacks, i.e. those that can be carried out from the problem-space, it is insufficient to perform adversarial training in the feature-space and then show a degree of empirical robustness for some arbitrary perturbation budget. This is because, unlike images, dynamic analysis reports can be of arbitrary size: adversarial training that does not take into account what an adversary *can* do will inadvertently explore inconsequential spaces around the decision boundary that do not affect evasion. Instead, as we will also demonstrate in our evaluation (Section 7), it is preferable to define the specific modifications that are feasible in the problem-space – a result of thorough threat modeling – and then construct adversarial examples with them.

Our observation is further corroborated through the results of Dyrmyshi et al. [25], where they discover that in domains where a problem-feature space representation gap exists, realistic attacks can be much more effective in hardening models against evasion. More concerningly, adversarial training with feature-space, unrealistic attacks has the capacity to actually make a model *more* vulnerable to adversarial attacks, as Hendrycks et al. have demonstrated [33]. In conjunction, these two properties indicate an important implication for all security-critical domains: adversarial training should be performed with *all* those and *only* those transformations that are feasible, threatening, and can *actually* be carried out through the problem-space.

4.2 Explanation-guided Hardening

Conceptually, our approach can be viewed as a practical and automated method for identifying useful but non-robust features [35]. Until now this was not straightforward to perform in domains where a distinct gap exists between problem and feature-space, as this identification requires an expert in-the-loop. Such useful but brittle features are commonly described as shortcuts or spurious correlations [29]. By passing to the RL agent the feasible modifications and **only** those, the agent will optimize towards changing the decision of the model, in essence discovering counterfactuals to what the model has learned: from low order features, e.g. specific filenames, to higher order concepts, e.g. relative size between features [37].

This exploration is intractable to be exhaustively performed in continuous or unconstrained in size domains; it can however become more efficient when some bias is introduced, for instance towards the most important features for classification [12]. Our task in this work is to harden against evasion an end-to-end differentiable ML model, known as HMIL [54], which ingests and classifies raw dynamic analysis reports. This model is accompanied by an explainability tool $\mathcal{E}$, specifically built for classifiers trained on raw hierarchical structured data [53]. The explanations returned by this tool consist of the minimal subset of input, in our case a sub-tree of the report, that receives the same classification as the complete input sample. The process of acquiring an explanation, given a sample to classify, consists of two phases: sub-tree ranking and sub-tree searching. First, all sub-trees are heuristically ranked based on their importance for the final classification. Then, a minimal sub-tree of the sample is searched such that its evaluation by the model remains within a threshold τ of the original. The end goal is to identify and distill the essential parts of the sample that influence the classification decision.

After this process concludes, a minimal set of entries from the dynamic analysis report is returned, something that is instrumental in our counterfactual investigation. As the number of entries in these reports can be arbitrarily large (in the hundreds of thousands), we are interested in identifying the most relevant and influential ones. Then a policy for modifying these reports can be enhanced through these explanations by precisely pointing towards what, according to the model under attack, is important for its decision. Note however that an entry pointed to by the explainer might not be modifiable in a way that it preserves the original functionality of the binary; AutoRobust performs only those modifications that preserve the functionality. After a sample has evaded detection, or a modification limit has been reached, it is added in the pool of adversarial examples. From there, to make the model robust to evasion and spurious correlations, we introduce these counterfactual examples to the training set – with their correct, original label – and together with the clean samples we perform adversarial retraining of the model.

5 Methodology

In this section we describe in detail our methodology and implementation for evaluating AutoRobust. This involves the dataset and binaries used, the dynamic analysis performed, the representation of that data, as well as the concrete transformations used on these representations.

5.1 Dataset

Unlike signature-based or rule-based detection, to build discriminative models for classifying between malicious and benign programs, we require samples from both. The paradox being that while malware are fairly easy to find, goodware are much more difficult to systematically procure. We source our goodware as they do in Dambra et al. [21]: through the community-maintained packages of Chocolatey [13] they create a dataset that spans 2012 to 2020. Regarding malware, VirusTotal [70] provided us with a dataset of Portable Executable from 2017–2020 that they released for academic purposes. Since we could not execute malware with network connectivity for safety and ethical concerns, we selected well-known representative families that would exhibit their malicious behavior under dynamic analysis even in the absence of connectivity: virlock, vobfus, and shipup. We verified that these families are sufficiently

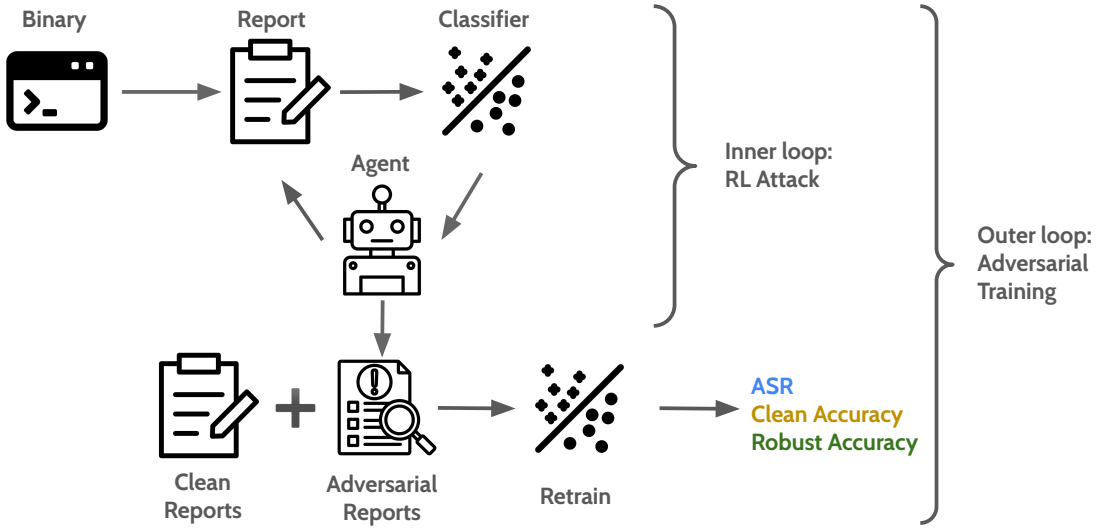


Figure 2: Schematic depiction of the AutoRobust pipeline. In the inner loop, the RL agent attacks the model and generates adversarial reports. In the outer loop, the model is retrained with standard minibatch gradient descent on clean *and* adversarial reports.

different from each other with vhash (resulting in a maximum overlap of 2,5%), while we preserve their distinct labels in order to evaluate also in a multi-class scenario, which was deemed of high interest by our industrial partner. In total, the dataset we managed to assemble contains 26,200 PE samples: 8,600 (33%) goodware and 17,600 (67%) malware.

All samples were executed in a series of Windows 7 VMware virtual machines, each with 4GB RAM, 4 vCPU, 128GB HDD, without network connectivity due to policy/ethical constraints. These were orchestrated through CAPEv2 [10], a maintained open-source successor of Cuckoo [20]. This framework is widely used for analyzing and detecting potentially malicious binaries by executing them in an isolated environment and observing their behavior without risking the security of the host system. Every sample was executed for 150 seconds, an empirical lower bound on the time required to gather the full behavior of samples [38].

As is considered best practice in the literature [46], we tried to mitigate evasive checks by running the VMwareCloak [19] script to remove well-known artifacts introduced by VMware. Moreover, we populated the filesystem with documents and other common types of files, to resemble a legitimate desktop workstation that malware may identify as a valuable target [49]. The dynamic analysis output is a detailed report with information on the syscalls[1] invoked by the binary and all the relative flags and arguments, as well as all interactions with the file system, registry, network, and other key elements of the operating system. Malware analysis and all other experiments ran on an infrastructure with 32 cores Intel(R) Xeon(R) Gold 6140 CPU @ 2.30GHz and 256GB of RAM.

[1]Without loss of generality, we use Windows APIs, Windows Native APIs, system calls, and syscalls interchangeably.

5.2 Representation

While the reports generated by CAPEv2 analysis consist of detailed information on the execution behavior of the binary, they vary considerably in size and may contain redundant or uninformative parts. As input space for the HMIL models that we train, we use the summary part of these reports. This summary is comprised of a concise description of all interactions the binary has with the system at the system call granularity level, and is composed of the following 13 categories:

- **Files / Read Files / Write Files / Delete Files**, containing the sets of all, read, written, and deleted files during execution respectively.
- **Keys / Read Keys / Write Keys / Delete Keys**, containing the sets of all, read, written, and deleted registry keys during execution respectively.
- **Executed Commands**, being the set of commands executed.
- **Resolved APIs**, being the set of external API calls that have been resolved.
- **Mutexes**, being the set of mutex names that have been invoked.
- **Created Services / Started Services**, being the sets of service names that have been created and started respectively.

This representation is delivered in JSON format, which includes a distinct key for each category described, with the corresponding values consisting of unique entries. We should note here that in the summary report of CAPEv2, in each category individual entries are listed only once.

5.3 Transformations

The explicit goal of our methodology is to identify counterfactuals on the input space of the model that indicate spurious correlations and brittle features that can be used for evasion, otherwise known

as *shortcuts* [30]. For example, our intuition indicates that a specific filename should be an irrelevant feature as it could be replaced by any other. As we are interested in defending against realistic threats, we need to exhaustively determine all the perturbations on the feature representation $\mathcal{F}$ of a sample that can be performed by the modifications that are permissible in the problem space and on the original program [55].

By permissible here we mean transformations that preserve the original functionality of the program. Addition is allowed across all categories as it is possible to introduce new entries without affecting the malicious functionality. Replacement is also feasible for all categories, with the exception of replacing APIs as it represents a potential risk for breaking the functionality. We consider feature removal in all categories a non-viable transformation; it is theoretically feasible, but not without refactoring or rewriting the source code, something we deem out of our threat model and our scope in this work. The set of transformations that are feasible through the problem-space is summarized in Table 1.

All report modifications are performed in a consistent manner and properly reflected where necessary. The structured report that represents the binary behavior contains interdependencies among the features categories that have to be respected when modifications take place; otherwise these modifications would result in a dynamic analysis report that is unrepresentative of an actual executing binary. For instance, when modifying a Mutex for writing a file, used as an argument in the NtCreateFile syscall and followed by the execution of the written file, we also modify that file everywhere else in the report. Similarly, changes made Files and Keys categories are reflected accordingly; for further details on how the feature-space transformations we employ are functionality-preserving in the problem-space, please refer to Appendix A.

While we have shown how the modifications we consider are a) functionality preserving, and b) representative of the full extent of adversarial capabilities C in the problem-space, we still lack a concise methodology for applying them. Next to *what* manner to modify with, we additionally have the questions of *where* and *how*; we therefore require a comprehensive modification policy that can perform all of the above concurrently. We grant these adversarial capabilities to our reinforcement learning (RL) agent by defining a multidiscrete[2] policy. This is a policy that at each time step selects three actions simultaneously: one action for each of the three aforementioned discrete categories. Each action represents a different choice on the modifications aspect of the JSON report: what to do, where to do it, and how to do it. Concretely, the three action categories are composed of the following discrete options to choose from:

(1) **What**: Add, Edit, X-Edit
(2) **Where**: One of the 13 categories described in 5.2.
(3) **How**: Random String, English Vocabulary, Target Vocabulary, Random Choice.

In the first category, **Add** always selects one of the entries from the target class, based on their frequency of occurrence. **Edit** replaces the entries in the report section that the second action points to (where), while keeping an incremental pointer for each section to avoid overwriting already edited entries. **X-Edit** replaces the entry or entries that the explainer (subsection 4.2) returns as the most important. The **Where** action points to the report section that the previously selected action will be performed.

As for the four manners of replacement: **Random String** constructs a string of ASCII characters selected at random, **English Vocabulary** selects an English word at random, **Target Vocabulary** selects word from the vocabulary of the target class weighted by its frequency, while **Random Choice** selects one of the three previous choices at random for each step of building a longer string, e.g. a filepath. The composition of these three actions results in a very capable policy that can modify reports of arbitrarily large size, with the added benefit that any modifications will always result to viable reports, as these would be generated from the problem-space and actual binaries. Ideally, one could investigate an even more granular modification policy, one that is able to modify reports on the character rather than the word or token level as we currently do. The reports in our dataset are on average 70K characters long, something that renders learning a character level policy on such a context length computationally intractable.

Category	Addition	Replacement	Removal
- /Write/Read/Delete Files	✔	✔	✘
- /Write/Read/Delete Keys	✔	✔	✘
Executed Commands	✔	✔	✘
Resolved APIs	✔	✘	✘
Mutexes	✔	✔	✘
Started/Created Services	✔	✔	✘

Table 1: Feasible transformations by feature category

6 Adversarial Binaries

Our objective in this work is defensive: we are interested in generating adversarial malware in order to adversarially train the HMIL model. Without loss of generality, we perform the functionality-preserving transformations in the feature space, while also introducing a proof of concept on how such adversarial variants of the malware can be constructed on the binary level. We manually verified that modifications are consistent and functionality-preserving when performed on the feature space.

The modification policy of AutoRobust is constantly updated through interaction with the classifier. This means that at each time step, we need to generate a new adversarial malware, then generate its dynamic analysis report, then classify it and repeat. Besides the engineering effort required to do this in real-time, the computational overhead of this process would be inordinate. The wall time formula in hours would thus be: $T = m * i * \frac{150}{3600}$, where m is the modifications performed per iteration, i is the number of iterations, and 150 are the seconds that each binary is executed for. In our evaluation $m = [50000, 200000]$ and $i = 15$, therefore $T = [31250, 125000]$. Thus if performed on the problem space, a full adversarial training session would take on average 7 years to complete, on our equipment and research prototype. In the feature space, as is currently performed, it takes on average 45 hours; a reduction by a factor of 1360. In this context, the only feasible solution is to apply the transformations directly on the dynamic

[2]https://www.gymlibrary.dev/api/spaces/

analysis reports, while preserving the problem space constraints and reducing the computational cost considerably. AutoRobust then allows one to harden a malware classifier against evasion without incurring excessive computational costs. We note here that whereas attackers *must* generate realistic adversarial binaries, defenders do *not* have to do that; rather, they merely have to generate feature spaces that correspond to realistic transformations. This is an asymmetry that favors defenders as it makes adversarial training much more practical.

The problem-space modifications can occur either on the source code or the compiled binary. For the former, as semantic equivalence is an undecidable problem, there are potentially unbounded ways to achieve the same functionality [5]; we therefore consider source code modifications out of scope. Focusing on adversarial binaries instead, the capability we want to bestow to them is adaptive behavior, so that during their execution they can alter this behavior on demand, as it is observed by the dynamic analysis environment. This can be achieved by the capacity to interrupt the functionality that their semantics dictate in order to interject an arbitrary number of randomly or deliberately selected API calls, or modify the arguments of the calls. On Windows and the Portable Executable (PE) format this can be achieved via hooking the Import Address Table (IAT), which contains pointers to function addresses that can be called by the binary; the goal is to overwrite these addresses to point to adversary defined functions which execute the original plus additional functionality that they define.

Related work has demonstrated how to achieve adversarial behavior on the binary level [26, 60]. The binary can packed with wrapper code that hooks all APIs and a configuration file that holds the modification logic—for AutoRobust that would be the adversarial policy $\pi_\theta^{\mathcal{V}}(a|s)$. Alternatively, external calls to runtime libraries can be funneled through an indirect jump to the address specified in the IAT. That jump points to the adversarial code consisted of the wrapped APIs, and each wrapped API calls the original API and before returning the return value, it applies the appropriate modifications. In both cases to remain stealthy and preserve functionality, the additional calls should be provided with valid inputs and make sure there are no state changes on the memory and registers.

7 Evaluation

In this section, we elaborate on the practical details of our approach that aims to harden the HMIL model against evasion, which has been trained on the reports produced by dynamic analysis. To acquire adversarial malware to train with, we operate directly on these reports after having rigorously defined the permitted transformations that enforce all necessary constraints in the problem-space.

As elaborated in Section 3, we do not generate new variants of the binaries themselves, however we do test and confirm that these transformations are possible to perform by modifying the CAPEv2 monitor library, capemon [9]. Specifically, we modified its source code adding the logic to successfully perform the proposed transformations via intercepting the syscalls of interest, e.g. for files *NtCreateFile*, *NtOpenFile*, *NtReadFile*, etc. In the case of a substitution operation, we replace the argument of the intercepted syscall, while for addition we invoke new system calls, depending on the intended feature to add. In the first scenario, in order to keep track of all the changes and do not affect the original functionality (further elaborated through an example in Appendix A) we use a shared memory structure that stores the mappings between original and modified entries. Every time that a system call is invoked, we check if one of the arguments matches with the entries of our mapping and, if this is the case we subsequently propagate the modification.

7.1 Evaluation Setup

We assume an initial HMIL model non-adversarially trained on dynamic analysis reports. The explicit goal of the adversary then is to find the optimal policy that, given the current state of the report, will do those modifications that lead to evasion. Optimal here is meant in a twofold way: in terms of a successful evasion *and* in terms of the minimal amount of transformations performed. This procedure is performed in an episodic manner, by first selecting a report belonging to the origin class and subsequently modifying it until it evades detection or the maximum number of modifications (the perturbation budget) is reached. When this happens the RL agent proceeds to a new report, and continues for a predefined total number of training steps; after that, all the resulting adversarial reports are appended to the adversarial dataset.

Subsequently, one epoch of adversarial retraining is performed by sampling from *both* the original and the adversarial dataset. The above steps constitute a *single* iteration in the AutoRobust methodology, with 15 of them performed in total. The schematically depict the full pipeline of our experimental evaluation approach in Figure 2, while more information on the HMIL model, the RL agents, and all hyperparameters are included in Appendix B.

7.2 Metrics

To properly measure and assess the performance of our approach, we monitor 4 primary metrics:

- **Score**: the probability that the HMIL model assigns to the *origin* class at the end of each episode, averaged over all episodes.
- **ASR**: the Attack Success Rate is the percentage of episodes in one iteration that the malware successfully evades; $ASR = \frac{s}{n} \times 100\%$, with $s =$ **Score** ≤ 0.5 and n the total number of episodes. ASR is a very relevant metric to assess how capable a policy is in turning malware reports adversarial.
- **C.Acc**: the model accuracy on clean samples from the hold-out set, measured in %.
- **R.Acc**: the model accuracy on adversarial samples from the hold-out set, that is *after* retraining the model.

Between adversarial training iterations, the modification policy can widely vary, and thus the distribution of modifications and adversarial reports will vary accordingly. Therefore, **R.Acc** *is not* a true indication of the level of robustness: while **R.Acc** is indeed evaluated on the hold-out test set of adversarial examples, these are still generated from the same policy and will therefore come from the *same* distribution.

The only way to acquire a representative level of robustness is through training a new adversarial policy from scratch, to properly assess if the model has any "blindspots" left. This process is repeated until convergence; at its end, a final more extensive round of training is performed with a hold-out set, and the resulting **ASR** is

Table 2: Performance of AutoRobust (AR), gradient-based (GB) adversarial training, and the original vanilla HMIL model considering the two perturbation budgets, 1K and 2K. The metrics displayed are: Score (of the origin class), Attack Success Rate (ASR), Clean Accuracy (C.Acc), and Robust Accuracy (R.Acc), all reported on the same hold-out test set.

Model	Adv. Training	Score		ASR (%)		R.Acc (%)		C.Acc (%)
		1K	2K	1K	2K	1K	2K	
Binary	None	0.31	0.23	73.8	80.2	26.2	19.8	99.0
	GB	0.32	0.30	72.2	71.8	27.8	28.2	99.1
	AR	**1.00**	**0.98**	**0.0**	**1.6**	**100**	**98.4**	99.1
Multiclass	None	0.11	0.25	100	99.8	0.0	0.2	98.8
	GB	0.12	0.12	99.5	99.8	0.5	0.2	98.8
	AR	**1.00**	**0.99**	**0.0**	**0.5**	**100**	**99.5**	98.8

the one reported. As a baseline to compare our approach to, we additionally perform and evaluate *gradient-based* adversarial training, considered as the state-of-the-art defensive approach for hardening models against evasion, irrespective of domain [45].

7.3 Results

In our experiments we evaluate the robustness of two types of classification models: one binary to decide between malware and goodware, and one multiclass to decide between malware family. Our adversaries operate under 2 perturbation budgets, namely 1K and 2K, while each entry modification, be it addition or replacement, counts as one perturbation. Given that the average size of a report in our dataset is ∼1K entries, these can be considered very generous constraints. For the binary and multiclass settings, Table 2 reports the performance of the original model as well as the two different kinds of adversarial training we evaluate: Gradient-Based (GB) and our approach AutoRobust (AR). From our empirical results we can make the following observations:

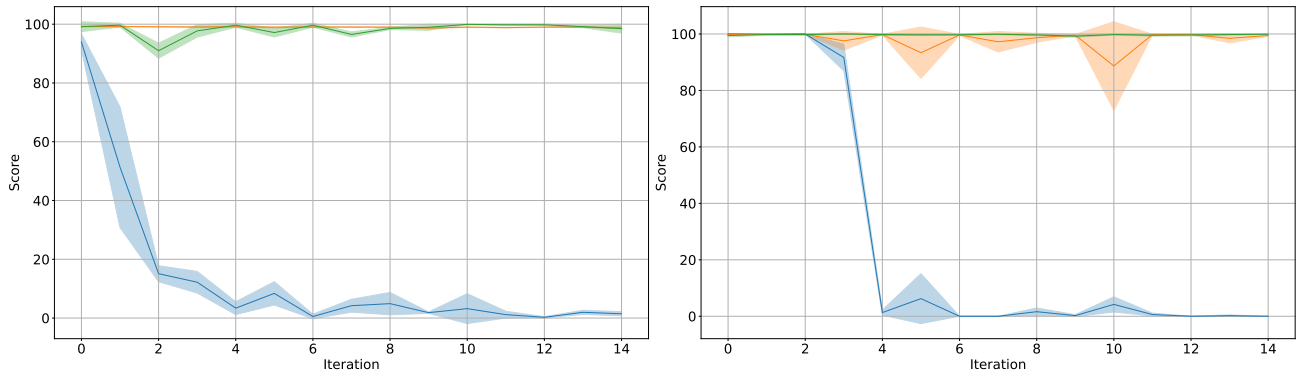
- While the initial non-adversarially trained model achieves excellent clean accuracy, it is excessively vulnerable to evasive attacks, often with as few as 10 modifications. This is a good indication that while the model initially picks up many useful discriminative features, these are brittle and decidedly not robust [35].
- Notably, gradient-based adversarial training shows little potential in defending against problem-space adversaries. This can be intuitively explained as perturbations based on the gradient of the model will frequently map to either invalid or uninformative areas of the feature-space that map to problem-space changes the adversary will be either unable or uninterested to do. In such domains it is therefore strictly preferable to generate adversarial examples using capabilities you want to defend against.
- During training we can observe that the ASR consistently drops over few iterations. After all 15 iterations are finished, the ASR in the hold-out test set is always 0, demonstrating that for this model and under these adversarial capabilities, it is conceivable to fully defend against the latter.
- Furthermore, we confirm that our approach is robust to hyperparameter selection, which is described in Appendix B. Fig. 3 plots the mean and variance of ASR, C.Acc, and R.Acc for the 15 iterations and over multiple runs with different hyperparameters. While both C.Acc and R.Acc consistently remain near 100%, ASR gradually depletes to near 0.
- Surprisingly, adversarial training with GB and AR slightly *increases* the C.Acc of the binary model. This might appear counter-intuitive at first, as robustness and performance are typically in trade-off; note however that we do 15 more iterations of training compared to the vanilla model, where adversarial training can also function as a form of regularization.
- Finally, the model's transition from being outright vulnerable to fully robust indicates that it is imperative to perform adversarial training in any security critical domain where adversarial agency should be considered a given.

7.4 Explanations

While deep learning obviates the laborious process of feature engineering and holds great promise in generalizing beyond its training data, we have to be judicious and inquisitive about what the model actually learns. Spurious correlations are not only erroneous biases that a model can pick up when trained end-to-end on raw data, they additionally can be exploited by an attacker to evade the model. It is thus crucial to remove such attack vectors at the time of training. With AutoRobust we automate this demanding, human-in-the-loop process, by substituting the requirement for expert input on every potential shortcut feature, with asking a simple question: what are *all* the feasible ways something could be different and still be the same thing?

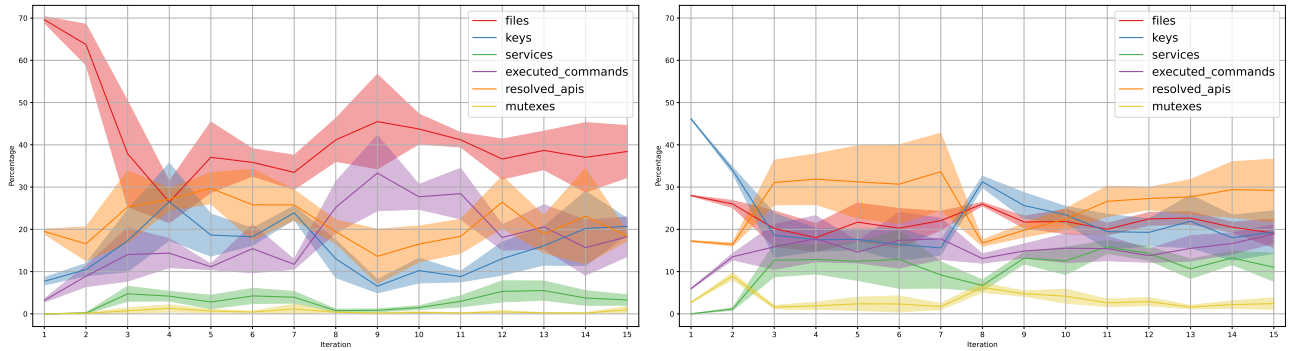
To get a deeper understanding of how the feature importance for a model shifts as the latter is retrained, we collect the explanations within a single iteration – that contains multiple episodes – and evaluate how these explanations evolve over multiple iterations. Figure 4 plots this progression. We remind the reader that the explanations returned are not scored by importance, but instead consist of a discrete, variable size set with entries from the dynamic analysis report.

In the binary model, we observe that initially a disproportionate focus is put on files that later decreases, while registry keys and resolved APIs increase and remain important. *Mutexes* and *services* represent a very small part of the explanations, unsurprisingly as they both have on average very few entries; what is surprising



(a) Binary classification **(b) Multiclass classification**

Figure 3: Progression of Attack Success Rate (blue), Clean Accuracy (yellow), and Robust Accuracy (green) over the 15 iterations of adversarial training. For each metric, the mean with one standard deviation are displayed, as we do multiple runs where hyperparameters and random seeds for selecting samples vary.



(a) Explanations in binary evaluations. **(b) Explanations in multiclass evaluations.**

Figure 4: The most frequent explanations as returned by the HMIL model explainer, over the 15 iterations. As for each iteration the number of episodes varies, we normalize explanations to a percentage over the whole iteration; mean values with one standard deviation are plotted.

however is that *executed commands* consistently rise in importance. In the multiclass model we observe comparable patterns, with the notable exception that for adversarially retrained models *resolved apis* are the most important explanations. The latter is in line with our intuition that as the only category where entries cannot be replaced, it should be more informative when discriminating between malicious and benign binaries.

In conclusion, AutoRobust proves exceptionally effective in hardening a ML model against problem space attacks, and it can be viewed as an approach for first forcing and subsequently learning on the possible distribution shifts. By changing in a report what could be different – and *only* what could *realistically* be different – without the label changing, it is a methodology for performing a counterfactual investigation on what is an essential feature and what is an artifact of the dataset. In that way, AutoRobust shares similarities with (and inspiration from) dataset enhancement techniques, where transformations on the original dataset help the model generalise [18].

8 Discussion

When considering hardening ML-based models and adversarial training in general, gradient-based approaches are state-of-practice. In our work we demonstrate that while this might be true for specific domains, it is not necessarily a general principle; in the dynamic analysis based malware detection we consider in this work this appears to not be the case. Instead, our novel approach has proven exceedingly effective in hardening a model and defending against problem-space attacks – attacks that are pervasive and realistic in real-world settings – while gradient-based adversarial training showed very limited effectiveness. This does not come as a complete surprise and has an intuitive explanation: in domains where the input space is unconstrained in size, but otherwise structured and constrained in adversarial capabilities, it is better to adversarially train by utilizing the latter two. One can think of our approach as confining the adversarial search within the manifold that corresponds to realizable programs.

While in the binary setting the performance metrics follow a well-defined trend, the multiclass case shows a few aberrations. As these co-occur with a drop in clean accuracy, we can partly attribute them to the ratio of clean and adversarial examples used for retraining and catastrophic forgetting. This issue became much less prominent when this ratio was tweaked, while subsequent iterations showed conclusively that the model adapted successfully and has the capacity to learn both clean and adversarial versions of reports. Overall, our evaluation demonstrates that AutoRobust is capable of zeroing out the success rate of attacks and is robust to hyperparameter selection; remarkably, it slightly improves the performance on clean data too.

Limitations. Our intention in this work was to focus on a real-world setting, hence we collaborated with an industrial partner that provided the ML model used as a component of their widely-known commercial antivirus. This makes our results very relevant for real-world settings, but naturally our evaluation is carried out on the HMIL model. We have however supplemented this empirical part with an extensive theoretical analysis that demonstrates the generality of our approach for any model or context that exhibits a problem/feature-space gap; AutoRobust is agnostic to the specific problem-space capabilities that are used to generate adversarial variants.

Another potential limitation is that due to stringent testing protocols, our malware analysis has been conducted without any network connectivity. Due to our compliance with safety guidelines, it is plausible that some of the analyzed binaries displayed a narrower extent of their full behavior due to lack of connectivity. While our methodology does not involve direct modification of the binaries themselves, our transformations preserve functionality by design. Nonetheless, we have verified their correctness on a handful of adversarial executions traces. Specifically, in the proof of concept we developed (cf. Appendix A) we manually verify these transformations are correctly performed on binaries under dynamic analysis and do not break the malware behavior or functionality. Finally, due to the computational cost on our research infrastructure and the transformations being functionality-preserving, the AutoRobust pipeline is executed on the feature space. This naturally limits our capacity to address or modify aspects of binary behavior that lie outside of this specific feature representation.

Future Work. Our AutoRobust framework, together with our p-Robustness formulation [4.1] are pertinent to any domain that has a distinct gap between problem and feature spaces [25]. It is thus a very promising approach to apply to other real-world settings and environments. Additionally, learning modification policies at the character level proved a considerable technical challenge, especially when some reports span millions of characters. Thus in our work we focused on more coarse controls, which when composed are nevertheless very powerful. In the future, a more granular policy could control modifications at the character level. Large Language Models (LLMs) are good candidates as they can implicitly learn the structure of a valid report, while their generation policy can be adapted to generate adversarial examples through RL feedback [14].

9 Conclusion

With ML-based detection of Windows malware on dynamic analysis environments becoming more prominent, significant efforts have been dedicated to assessing the resilience and effectiveness of the ML classification against adversarial malware, that is variants that can evade detection. In this work, we justify theoretically and empirically why adversarial attacks on malware should be performed through the problem space, and introduce *AutoRobust*, a novel methodology based on RL for hardening detection models against malware evasion. As the information contained in dynamic analysis reports will most certainly include artifacts, *AutoRobust* automates the identification and removal of spurious correlations and brittle features in the input space of the model which can be used for evasion: namely to generate adversarial malware. By optimizing with RL the process of discovering counterfactuals to what a naively trained model has learned, we can guarantee a probabilistic level of robustness against well-defined adversaries. Notably, given a ML-based malware detection module of a commercial antivirus, we show how vulnerable it can be to adversarial malware, as well as how *AutoRobust* can grant near perfect robustness to the model *without* adversely affecting and even improving its performance on clean inputs.

Acknowledgments

This research is partially funded and supported by: the Research Fund KU Leuven; the Cybersecurity Research Program Flanders; the EU funded project KINAITICS (Grant Agreement Number 101070176); a Google ASPIRE research award; EPSRC Grant EP/X015971/1.

References

[1] Adel Abusitta, Miles Q Li, and Benjamin CM Fung. 2021. Malware classification and composition analysis: A survey of recent developments. *Journal of Information Security and Applications* 59 (2021), 102828.

[2] Hojjat Aghakhani, Fabio Gritti, Francesco Mecca, Martina Lindorfer, Stefano Ortolani, Davide Balzarotti, Giovanni Vigna, and Christopher Kruegel. 2020. When malware is packin'heat; limits of machine learning classifiers based on static analysis features. In *Network and Distributed Systems Security (NDSS) Symposium 2020*.

[3] Hisham Alasmary, Ahmed Abusnaina, Rhongho Jang, Mohammed Abuhamad, Afsah Anwar, DaeHun Nyang, and David Mohaisen. 2020. Soteria: Detecting adversarial examples in control flow graph-based malware classifiers. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 888–898.

[4] Hyrum S Anderson, Anant Kharkar, Bobby Filar, David Evans, and Phil Roth. 2018. Learning to evade static PE machine learning malware models via reinforcement learning. *arXiv preprint arXiv:1801.08917* (2018).

[5] Paolo Baldan, Francesco Ranzato, Linpeng Zhang, et al. 2021. A Rice's theorem for abstract semantics. *LEIBNIZ INTERNATIONAL PROCEEDINGS IN INFORMATICS* 198 (2021), 1–19.

[6] Battista Biggio and Fabio Roli. 2018. Wild patterns: Ten years after the rise of adversarial machine learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2154–2156.

[7] Marcus Botacin, Felipe Duarte Domingues, Fabrício Ceschin, Raphael Machnicki, Marco Antonio Zanata Alves, Paulo Lício de Geus, and André Grégio. 2022. Antiviruses under the microscope: A hands-on perspective. *Computers & Security* 112 (2022), 102500.

[8] Justin Burr and Shengjie Xu. 2021. Improving adversarial attacks against executable raw byte classifiers. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 1–2.

[9] CAPEMON. [n. d.]. *CAPEMON*. https://github.com/kevoreilly/capemon.

[10] CAPEv2. [n. d.]. *CAPEv2 sandbox*. https://github.com/kevoreilly/CAPEv2.

[11] Nicholas Carlini and David Wagner. 2017. Towards evaluating the robustness of neural networks. In *2017 ieee symposium on security and privacy (sp)*. Ieee, 39–57.

[12] Stephen Casper, Max Nadeau, Dylan Hadfield-Menell, and Gabriel Kreiman. 2022. Robust feature-level adversaries are interpretability tools. *Advances in Neural Information Processing Systems* 35 (2022), 33093–33106.

[13] Chocolately. [n. d.]. *Chocolately*. https://chocolatey.org/.

[14] Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. 2017. Deep reinforcement learning from human preferences. *Advances in neural information processing systems* 30 (2017).

[15] Mihai Christodorescu and Somesh Jha. 2003. Static analysis of executables to detect malicious patterns. In *12th USENIX Security Symposium (USENIX Security 03)*.

[16] Andrea Continella, Alessandro Guagnelli, Giovanni Zingaro, Giulio De Pasquale, Alessandro Barenghi, Stefano Zanero, and Federico Maggi. 2016. Shieldfs: a self-healing, ransomware-aware filesystem. In *Proceedings of the 32nd annual conference on computer security applications*. 336–347.

[17] Bas Cornelissen, Andy Zaidman, Arie Van Deursen, Leon Moonen, and Rainer Koschke. 2009. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering* 35, 5 (2009), 684–702.

[18] Ekin D Cubuk, Barret Zoph, Dandelion Mane, Vijay Vasudevan, and Quoc V Le. 2018. Autoaugment: Learning augmentation policies from data. *arXiv preprint arXiv:1805.09501* (2018).

[19] Kyle Cucci. [n. d.]. *VMwareCloak*. https://github.com/d4rksystem/VMwareCloak.

[20] Cuckoo. [n. d.]. *Cuckoo sandbox*. https://github.com/cuckoosandbox/cuckoo.

[21] Savino Dambra, Yufei Han, Simone Aonzo, Platon Kotzias, Antonino Vitale, Juan Caballero, Davide Balzarotti, and Leyla Bilge. 2023. Decoding the Secrets of Machine Learning in Malware Classification: A Deep Dive into Datasets, Feature Extraction, and Model Performance. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 60–74.

[22] Luca Demetrio, Battista Biggio, Giovanni Lagorio, Fabio Roli, and Alessandro Armando. 2021. Functionality-preserving black-box optimization of adversarial windows malware. *IEEE Transactions on Information Forensics and Security* 16 (2021), 3469–3478.

[23] Luca Demetrio, Scott E Coull, Battista Biggio, Giovanni Lagorio, Alessandro Armando, and Fabio Roli. 2021. Adversarial exemples: A survey and experimental evaluation of practical attacks on machine learning for windows malware detection. *ACM Transactions on Privacy and Security (TOPS)* 24, 4 (2021), 1–31.

[24] Ambra Demontis, Marco Melis, Maura Pintor, Matthew Jagielski, Battista Biggio, Alina Oprea, Cristina Nita-Rotaru, and Fabio Roli. 2019. Why do adversarial attacks transfer? explaining transferability of evasion and poisoning attacks. In *28th USENIX security symposium (USENIX security 19)*. 321–338.

[25] Salijona Dyrmishi, Salah Ghamizi, Thibault Simonetto, Yves Le Traon, and Maxime Cordy. 2023. On the empirical effectiveness of unrealistic adversarial hardening against realistic adversarial attacks. In *2023 IEEE symposium on security and privacy (SP)*. IEEE, 1384–1400.

[26] Fenil Fadadu, Anand Handa, Nitesh Kumar, and Sandeep Kumar Shukla. 2019. Evading API call sequence based malware classifiers. In *International Conference on Information and Communications Security*. Springer, 18–33.

[27] Nicola Galloro, Mario Polino, Michele Carminati, Andrea Continella, and Stefano Zanero. 2022. A Systematical and longitudinal study of evasive behaviors in windows malware. *Computers & Security* 113 (2022), 102550.

[28] Marek Galovic, Branislav Bosansky, and Viliam Lisy. 2021. Improving robustness of malware classifiers using adversarial strings generated from perturbed latent representations. *arXiv preprint arXiv:2110.11987* (2021).

[29] Robert Geirhos, Jörn-Henrik Jacobsen, Claudio Michaelis, Richard Zemel, Wieland Brendel, Matthias Bethge, and Felix A Wichmann. 2020. Shortcut learning in deep neural networks. *Nature Machine Intelligence* 2, 11 (2020), 665–673.

[30] Robert Geirhos, Carlos RM Temme, Jonas Rauber, Heiko H Schütt, Matthias Bethge, and Felix A Wichmann. 2018. Generalisation in humans and deep neural networks. *Advances in neural information processing systems* 31 (2018).

[31] Justin Gilmer and Dan Hendrycks. 2019. A Discussion of 'Adversarial Examples Are Not Bugs, They Are Features': Adversarial Example Researchers Need to Expand What is Meant by 'Robustness'. *Distill* (2019). https://doi.org/10.23915/distill.00019.1 https://distill.pub/2019/advex-bugs-discussion/response-1.

[32] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. 2017. Adversarial examples for malware detection. In *Computer Security–ESORICS 2017: 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part II 22*. Springer, 62–79.

[33] Dan Hendrycks and Thomas Dietterich. 2019. Benchmarking neural network robustness to common corruptions and perturbations. *arXiv preprint arXiv:1903.12261* (2019).

[34] Weiwei Hu and Ying Tan. 2017. Generating adversarial malware examples for black-box attacks based on GAN. *arXiv preprint arXiv:1702.05983* (2017).

[35] Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Logan Engstrom, Brandon Tran, and Aleksander Madry. 2019. Adversarial examples are not bugs, they are features. *Advances in neural information processing systems* 32 (2019).

[36] Kyriakos K Ispoglou and Mathias Payer. 2016. {malWASH}: washing malware to evade dynamic analysis. In *10th USENIX workshop on offensive technologies (WOOT 16)*.

[37] Been Kim, Martin Wattenberg, Justin Gilmer, Carrie Cai, James Wexler, Fernanda Viegas, et al. 2018. Interpretability beyond feature attribution: Quantitative testing with concept activation vectors (tcav). In *International conference on machine learning*. PMLR, 2668–2677.

[38] Alexander Küchler, Alessandro Mantovani, Yufei Han, Leyla Bilge, and Davide Balzarotti. 2021. Does Every Second Count? Time-based Evolution of Malware Behavior in Sandboxes.. In *NDSS*.

[39] Raphael Labaca-Castro, Sebastian Franz, and Gabi Dreo Rodosek. 2021. AIMED-RL: Exploring adversarial malware examples with reinforcement learning. In *Machine Learning and Knowledge Discovery in Databases. Applied Data Science Track: European Conference, ECML PKDD 2021, Bilbao, Spain, September 13–17, 2021, Proceedings, Part IV 21*. Springer, 37–52.

[40] Raphael Labaca-Castro, Luis Muñoz-González, Feargus Pendlebury, Gabi Dreo Rodosek, Fabio Pierazzi, and Lorenzo Cavallaro. 2021. Realizable universal adversarial perturbations for malware. *arXiv preprint arXiv:2102.06747* (2021).

[41] Gert RG Lanckriet, Laurent El Ghaoui, Chiranjib Bhattacharyya, and Michael I Jordan. 2002. A robust minimax approach to classification. *Journal of Machine Learning Research* 3, Dec (2002), 555–582.

[42] Deqiang Li, Qianmu Li, Yanfang Ye, and Shouhuai Xu. 2021. A framework for enhancing deep neural networks against adversarial malware. *IEEE Transactions on Network Science and Engineering* 8, 1 (2021), 736–750.

[43] Xiang Li, Kefan Qiu, Cheng Qian, and Gang Zhao. 2020. An adversarial machine learning method based on opcode n-grams feature in malware detection. In *2020 IEEE Fifth International Conference on Data Science in Cyberspace (DSC)*. IEEE, 380–387.

[44] Keane Lucas, Samruddhi Pai, Weiran Lin, Lujo Bauer, Michael K Reiter, and Mahmood Sharif. 2023. Adversarial training for {Raw-Binary} malware classifiers. In *32nd USENIX Security Symposium (USENIX Security 23)*. 1163–1180.

[45] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 2017. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083* (2017).

[46] Lorenzo Maffia, Dario Nisi, Platon Kotzias, Giovanni Lagorio, Simone Aonzo, and Davide Balzarotti. 2021. Longitudinal Study of the Prevalence of Malware Evasive Techniques. *arXiv preprint arXiv:2112.11289* (2021).

[47] Šimon Mandlík, Matěj Račinskỳ, Viliam Lisỳ, and Tomáš Pevnỳ. 2022. JsonGrinder. jl: automated differentiable neural architecture for embedding arbitrary JSON data. *The Journal of Machine Learning Research* 23, 1 (2022), 13508–13512.

[48] Jiang Ming, Zhi Xin, Pengwei Lan, Dinghao Wu, Peng Liu, and Bing Mao. 2017. Impeding behavior-based malware analysis via replacement attacks to malware specifications. *Journal of Computer Virology and Hacking Techniques* 13 (2017), 193–207.

[49] Najmeh Miramirkhani, Mahathi Priya Appini, Nick Nikiforakis, and Michalis Polychronakis. 2017. Spotless sandboxes: Evading malware analysis systems using wear-and-tear artifacts. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1009–1024.

[50] Christoforos Ntantogian, Georgios Poulios, Georgios Karopoulos, and Christos Xenakis. 2019. Transforming malicious code to ROP gadgets for antivirus evasion. *IET Information Security* 13, 6 (2019), 570–578.

[51] Daniel Park and Bülent Yener. 2020. A survey on practical adversarial examples for malware classifiers. In *Reversing and Offensive-oriented Trends Symposium*. 23–35.

[52] Jeffrey Pennington, Richard Socher, and Christopher D Manning. 2014. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 1532–1543.

[53] Tomáš Pevnỳ, Viliam Lisỳ, Branislav Bošanskỳ, Petr Somol, and Michal Pěchouček. 2022. Explaining Classifiers Trained on Raw Hierarchical Multiple-Instance Data. *arXiv preprint arXiv:2208.02694* (2022).

[54] Tomáš Pevnỳ and Petr Somol. 2017. Using neural network formalism to solve multiple-instance problems. In *Advances in Neural Networks-ISNN 2017: 14th International Symposium, ISNN 2017, Sapporo, Hakodate, and Muroran, Hokkaido, Japan, June 21–26, 2017, Proceedings, Part I 14*. Springer, 135–142.

[55] Fabio Pierazzi, Feargus Pendlebury, Jacopo Cortellazzi, and Lorenzo Cavallaro. 2020. Intriguing properties of adversarial ml attacks in the problem space. In *2020 IEEE symposium on security and privacy (SP)*. IEEE, 1332–1349.

[56] CERT Polska. [n. d.]. *DRAKVUF sandbox*. https://github.com/CERT-Polska/drakvuf-sandbox.

[57] Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles K Nicholas. 2018. Malware detection by eating a whole exe. In *Workshops at the thirty-second AAAI conference on artificial intelligence*.

[58] Hemant Rathore, Adithya Samavedhi, Sanjay K Sahay, and Mohit Sewak. 2021. Robust malware detection models: learning from adversarial attacks and defenses. *Forensic Science International: Digital Investigation* 37 (2021), 301183.

[59] Ishai Rosenberg, Asaf Shabtai, Yuval Elovici, and Lior Rokach. 2021. Sequence Squeezing: A Defense Method Against Adversarial Examples for API Call-Based RNN Variants. In *2021 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1–10.

[60] Ishai Rosenberg, Asaf Shabtai, Lior Rokach, and Yuval Elovici. 2018. Generic black-box end-to-end attack against state of the art API call based malware classifiers. In *Research in Attacks, Intrusions, and Defenses: 21st International Symposium, RAID 2018, Heraklion, Crete, Greece, September 10-12, 2018, Proceedings 21*. Springer, 490–510.

[61] Mahmood Sharif, Sruti Bhagavatula, Lujo Bauer, and Michael K Reiter. 2019. A general framework for adversarial examples with objectives. *ACM Transactions on Privacy and Security (TOPS)* 22, 3 (2019), 1–30.

[62] Ryan Sheatsley, Nicolas Papernot, Michael Weisman, Gunjan Verma, and Patrick McDaniel. 2020. Adversarial examples in constrained domains. *arXiv preprint arXiv:2011.01183* (2020).

[63] Jack W Stokes, De Wang, Mady Marinescu, Marc Marino, and Brian Bussone. 2018. Attack and defense of dynamic analysis-based, adversarial neural malware detection models. In *MILCOM 2018-2018 IEEE Military Communications Conference (MILCOM)*. IEEE, 1–8.

[64] Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. 1999. Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems* 12 (1999).

[65] Ronghua Tian, Rafiqul Islam, Lynn Batten, and Steve Versteeg. 2010. Differentiating malware from cleanware using behavioural analysis. In *2010 5th international conference on malicious and unwanted software*. Ieee, 23–30.

[66] Florian Tramer, Nicholas Carlini, Wieland Brendel, and Aleksander Madry. 2020. On adaptive attacks to adversarial example defenses. *Advances in neural information processing systems* 33 (2020), 1633–1645.

[67] Philipp Trinius, Carsten Willems, Thorsten Holz, and Konrad Rieck. 2009. A malware instruction set for behavior-based analysis. *None* (2009).

[68] Ilias Tsingenopoulos, Ali Mohammad Shafiei, Lieven Desmet, Davy Preuveneers, and Wouter Joosen. 2022. Adaptive Malware Control: Decision-Based Attacks in the Problem Space of Dynamic Analysis. In *Proceedings of the 1st Workshop on Robust Malware Analysis*. 3–14.

[69] Sohini Upadhyay, Shalmali Joshi, and Himabindu Lakkaraju. 2021. Towards robust and reliable algorithmic recourse. *Advances in Neural Information Processing Systems* 34 (2021), 16926–16937.

[70] VirusTotal. [n. d.]. *VirusTotal*. https://www.virustotal.com.

A Problem-space transformations

In Section 7 we presented the full range of functionality-preserving transformations on the specific feature space we are investigating in this work. As generating new binary variants for every modification performed is a considerable engineering effort and time consuming process, we developed a proof of concept (PoC) that covers the basic usage of every Windows API intercepted by capemon.

- In this PoC we implement the same exact logic that AutoRobust applies in the feature space.
- We create a shared memory table that keeps track of all the name changes across program execution, so whenever one of the APIs of interest is called its argument was replaced accordingly as indicated by the modification.
- This entry is then added to the shared table as a stage 1 modification; if there are successive modifications we do track them all for completeness.
- From that moment on, whenever another API call is invoked, we do check if the original name is in one of the arguments and replace it according to our substitution history.
- Depending on the API we employ different strategies for argument substitution. As we do not want to impact the original program behavior, we have to conform to the viable transformations for each API. For instance, creating a mutex allows for completely arbitrary names, for file generation we have to provide valid filepaths, and in command execution we need to ensure all previous modifications are reflected without introducing new random entries which could make the program crash. We preserve this reasoning in the feature space, as illustrated in subsection A.2.

Adapting our PoC to runtime modifications could include the aforementioned logic in a dll wrapped with the binary, as discussed in Section 6. Under the specifics of dynamic analysis and our use-case, given access to the decisions of the model $\mathcal{M}$ attackers can generate adversarial binaries in the problem space with considerable automation, but with less granularity than AutoRobust: whatever modifications they make, they have to be packaged in batch to a new binary that will be subsequently analyzed by the sandbox. As the entries in the summary of the dynamic analysis report are not ordered, the placement of functions in adversarial binaries does not matter; as long as they return to the main functionality of the program with its internal state unaffected, these functions are allowed to interact with each other.

A.1 Functionality Preservation in AutoRobust

Here we outline in detail the modification procedure, which keeps track of the changed entries, and how it ensures that all changes propagate correctly throughout a dynamic analysis report. In order to do it, after the agent has performed a modification on the report, we seek for other occurrences of the same entry in different categories. In case we do, we perform the same modification on it and we do this for all the dynamic report categories subsection 5.2. In that way the modifications performed will reflect both a viable report variant in the feature-space, and a binary with its functionality preserved in the problem-space. For this detailed explanation, we consider a malicious program that wants to modify the firewall rules. To do that, it needs to invoke **netsh** Windows utility, something we expect to appear in the category *executed commands*:

```
netsh firewall add allowedprogram
    ↪ "C:\Users\John\AppData\Roaming\malicious.exe"
    ↪ "malicious.exe" ENABLE
```

Changing the command itself or its could inadvertently affect the original malware behavior. However, renaming the file from "malicious.exe" to another name, provided the new name is consistently referenced, would not affect the malware's functionality. An initial modification could be:

```
netsh firewall add allowedprogram
    ↪ "C:\Users\John\AppData\Roaming\hello.exe"
    ↪ "hello.exe" ENABLE
```

Following a modification on the executed commands, our transformation logic performs a sweep of the entire report to identify all places that it is referenced; subsequently, every occurrence of "malicious.exe" is substituted with "hello.exe", for example corresponding entries within both the *Files* and *Write Files* sections. Such entries indicate not only the generation of the file but also the activities related to content modification associated with it and in order to be consistent we change those entries as well. Additionally, our investigation unveils instances within the *executed commands* section that indicate subsequent efforts to execute the binary after adjustments had been made to the firewall settings. Our transformations ensure the modification of the filename in this case as well, in that way preserving the original functionality.

This example helps to demonstrate how modifications are propagated around the executed commands category. The same reasoning is applied to the performed modifications that need to be reflected in other categories; for example changes in *Write Keys* are reflected the *Keys* section. Because we do not modify the binaries themselves, we can guarantee that for the granularity provided by our current feature space the transformations are coherent, working at the same abstraction level.

A.2 Dynamic Analysis Reports

To illustrate the feature space we are working with, and also demonstrate how effortless evading the HMIL model initially is, we include an example dynamic analysis report as well as its adversarial variant. Listing 1 shows the original report as generated through execution in our sandbox.

Listing 1: Original report entries


```
{
  "summary": {
    "files": [
      "C:\\Users\\John\\AppData\\Local\\Temp\\\\xb0\\xa1
          ↪ \\xb0\\xa1\\xb0\\xa1\\xb0\\xa1\\xb0\\xa1\\
          ↪ xb0\\xa1\\xb0\\xa1\\xb0\\xa1\\xb0\\xa1\\xb0
          ↪ \\xa1\\xb0\\xa1\\xb0\\xa1\\xb0\\xa1\\xb0\\
          ↪ xa1\\xb0\\xa1\\xb0\\xa1\\xb0\\xa1\\xb0\\xa1
          ↪ \\xb0\\xa1\\xb0\\xa1\\xb0\\xa1\\xb0\\xa1\\
          ↪ xb0\\xa1\\xb0\\xa1\\xb0\\xa1\\xb0\\xa1\\xb0
          ↪ \\xa1\\xb0\\xa1\\xb0\\xa1\\xb0\\xa1\\xb0\\
          ↪ xa1\\xb0\\xa1\\xb0\\xa1",
      "C:\\Users\\John\\AppData\\Local\\Temp\\03cb16b5aa2
          ↪ 1f4b0a10a8e3e.exe",
      "C:\\vbxjf.exe"
    ],
    "read_files": [
      "C:\\Users\\John\\AppData\\Local\\Temp\\03cb16b5aa2
          ↪ 1f4b0a10a8e3e.exe"
```

```
    ],
    "write_files": [
      "C:\\vbxjf.exe"
    ],
    "delete_files": [],
    "keys": [
      "DisableUserModeCallbackFilter",
      "HKEY_LOCAL_MACHINE\\Software\\Microsoft\\Windows
          ↪ NT\\CurrentVersion\\GRE_Initialize",
      "HKEY_LOCAL_MACHINE\\SOFTWARE\\MICROSOFT\\WINDOWS
          ↪ NT\\CURRENTVERSION\\GRE_Initialize\\
          ↪ DisableMetaFiles"
    ],
    "read_keys": [
      "DisableUserModeCallbackFilter",
      "HKEY_LOCAL_MACHINE\\SOFTWARE\\MICROSOFT\\WINDOWS
          ↪ NT\\CURRENTVERSION\\GRE_Initialize\\
          ↪ DisableMetaFiles"
    ],
    "write_keys": [],
    "delete_keys": [],
    "executed_commands": [
      "c:\\vbxjf.exe"
    ],
    "resolved_apis": [
       "kernel32.dll.VirtualAlloc",
      "kernel32.dll.VirtualProtect",
      "kernel32.dll.VirtualFree",
      "kernel32.dll.LoadLibraryA",
      "kernel32.dll.GetProcAddress",
      "kernel32.dll.ExitProcess",
      "msvcrt.dll.atoi",
      "shlwapi.dll.PathFileExistsA",
      "user32.dll.wsprintfA",
      "kernel32.dll.OpenProcess",
      "kernel32.dll.VirtualAllocEx",
      "kernel32.dll.WriteProcessMemory",
      "kernel32.dll.WaitForSingleObject",
      "kernel32.dll.VirtualFreeEx",
      "kernel32.dll.GetProcessHeap",
      "kernel32.dll.GetModuleHandleA",
      "kernel32.dll.HeapAlloc",
      "kernel32.dll.HeapFree",
      "kernel32.dll.IsBadReadPtr",
      "kernel32.dll.DeleteFileA",
      "kernel32.dll.GetModuleFileNameA",
      "kernel32.dll.CloseHandle",
      "kernel32.dll.ReadFile",
      "kernel32.dll.GetFileSize",
      "kernel32.dll.CreateFileA",
      "kernel32.dll.WriteFile",
      "kernel32.dll.CreateProcessA",
      "kernel32.dll.GetStartupInfoA",
      "kernel32.dll.Sleep",
      "kernel32.dll.FreeLibrary",
      "msvcrt.dll.strchr",
      "msvcrt.dll._CIfmod",
      "user32.dll.TranslateMessage",
      "user32.dll.DispatchMessageA",
      "user32.dll.GetMessageA",
      "user32.dll.MessageBoxA",
      "user32.dll.PeekMessageA",
      "user32.dll.GetWindowThreadProcessId"
    ],
    "mutexes": [],
    "created_services": [],
    "started_services": []
  }
}
```


On the other hand, Listing 2 shows the same report after enough transformations have been applied so that it has successfully evaded the target classifier, i.e. classified as goodware. For simplicity, we report only the entries that have been modified. We can observe that our transformations change consistently the name of the files and the directory paths in which these are saved, propagating also the modification to *executed commands* section. Then, our RL agent also adds a mutex, which has no impact on the actual behavior of the original binary.

Listing 2: Modified report entries


```
{
  "summary": {
    "files": [
      "C:\\Users\\John\\AppData\\Local\\Saxon\\Temp\\
          ↪ sorage\\shrinal\\
          podostemad\\hired\\oxyphenyl\\demonocracy\\
          ↪ Lapsana\\Kamasin\\
          menoschesis\\spinnerular\\symptomless\\
          ↪ unknelled\\Cladocera\\lame\\
          suspensively\\uncomparably\\Utopianize\\
          ↪ demigroat\\ovistic\\equalize\\
          lounger\\staghunt\\Cascadia\\head\\overrule\\
          ↪ chichimecan\\
          Edestosaurus\\berrigan\\booger\\cinurous\\
          ↪ unhurt\\gollar\\uraniid\\
          blousing\\lunge\\recrease\\rage\\limy\\
          ↪ predefense\\spadger\\
          Wykehamist\\taxidermist\\outcrier\\patchwork\\
          ↪ specular\\huffishness\\
          tarnal\\cabin\\tenementary\\rectalgia\\
          ↪ cataplasis\\flavor\\
          sprank\\rigor\\unwrinkle\\partitionary\\
          ↪ dancery\\demihigh\\
          warrantee\\sherifate\\thereout\\fourteenthly\\
          ↪ mousefish\\
          unfairylike\\tenaille\\iodobromite\\octoglot\\
          ↪ Cartist\\
          heartscald\\wellsite\\triphony\\picturely\\
          ↪ zoning\\deal\\pyelectasis\\
          unselfish\\marshite\\truckful\\beworn\\
          ↪ thriller\\divaricately\\Earnie",
      "C:\\Users\\horseback\\zeuglodont\\supping\\aphasic
          ↪ \\institor.exe",
      "C:\\surgeproof\\marionette.exe"

    ],
    "read_files": [
      "C:\\Users\\horseback\\zeuglodont\\supping\\aphasic
          ↪ \\institor.exe"
    ],
    "write_files": [
      "C:\\surgeproof\\marionette.exe"
    ],
    "delete_files": [],
    "keys": [
      ...
    ],
    "read_keys": [
      ...
    ],
    "write_keys": [],
    "delete_keys": [],
    "executed_commands": [
      C:\\surgeproof\\marionette.exe}
    ],
    "resolved_apis": [
      ...
    ],
    ("mutexes": ["Asjk"],
    "created_services": [],
    "started_services": []
  }
}
```


B HMIL & RL Hyperparameters

The model is retrained for a total of 15 iterations on the new adversarial examples generated, where in each iteration agents are trained from scratch. We cold-start new agents because agents from previous iterations often failed to evade the retrained model; this strongly indicates that knowing the capability set C is not sufficient to defend, as multiple different policies – and thus multiple

different distributions – can result from it. We emphasize that robust accuracy is not a true indication of actual robustness; while evaluated on a test set, this comes from the *same* distribution due to being generated by the same agent. The realistic level of robustness is evaluated only through a final round of attacks, with starting reports neither the model nor the agent have seen before. As is best practice in adversarial machine learning (AML), candidate reports are only those that are correctly classified by the model.

States & Rewards. For all agents, the state representation s for a sample x is its 32-dimensional embedding on the last layer before the fully connected of HMIL model. The reward functions we evaluate with are straightforward: we reward the score decrease in the source class and its increase in the target class, while optionally penalizing the total number of steps and any disallowed moves – which if selected, are never allowed to go through. Table 3 reports on all hyperparameters – or ranges – that we experimented with. Throughout our experiments, we observe that results are robust to hyperparameter selection.

Table 3: Hyperparameters.

Hyperparameter	Binary	Multiclass
learning rate	e-3 – 3e-3	e-3 – 3e-3
steps per episode	1000 – 2000	1000 – 2000
steps per iteration	5e4 – 3e5	2e4 – 2e5
iterations	15	15
batch size RL	32	32
batch size HMIL	128	128