Exposing Weaknesses of Malware Detectors with Explainability-Guided Evasion Attacks

Wei Wang  
University of Adelaide  
Australia

Ruoxi Sun  
University of Adelaide  
Australia

Tian Dong  
Shanghai Jiao Tong University  
China

Shaofeng Li  
Shanghai Jiao Tong University  
China

Minhui Xue  
University of Adelaide  
Australia

Gareth Tyson  
Queen Mary University of London  
United Kingdom

Haojin Zhu  
Shanghai Jiao Tong University  
China

ABSTRACT
Numerous open-source and commercial malware detectors are available. However, the efficacy of these tools has been threatened by new adversarial attacks, whereby malware attempts to evade detection using, for example, machine learning techniques. In this work, we design an adversarial evasion attack that relies on both feature-space and problem-space manipulation. It uses explainability-guided feature selection to maximize evasion by identifying the most critical features that impact detection. We then use this attack as a benchmark to evaluate several state-of-the-art malware detectors. We find that (i) state-of-the-art malware detectors are vulnerable to even simple evasion strategies, and they can easily be tricked using off-the-shelf techniques; (ii) feature-space manipulation and problem-space obfuscation can be combined to enable evasion without needing white-box understanding of the detector; (iii) we can use explainability approaches (e.g., SHAP) to guide the feature manipulation and explain how attacks can transfer across multiple detectors. Our findings shed light on the weaknesses of current malware detectors, as well as how they can be improved.

1 INTRODUCTION
Malware continues to be one of the most pressing security issues that users face today. Recent research has shown that the total number of malware infections has been rising for the last decade (2009 to 2018) [5]. In 2018, the number of malware infections was 812.6 million, while during the first nine months of 2019, at least 7.2 billion malware attacks and 151.9 million ransomware attacks were reported. Furthermore, the attack rate has hit a new high during the COVID-19 pandemic [16]. These figures suggest that traditional signature-based methods cannot keep up with the rampant growth of novel malware. Hence, commercial antivirus companies have started using machine learning [1, 59] to enable detection without the need for signatures. However, research has demonstrated that attackers can evade machine learning-based detectors by manipulating the malware features that such detectors use [32, 33, 49, 50, 63, 64]. Because of this, commercial antivirus systems are susceptible to adversarial attacks [55]. Although there has been a several works [27, 35, 36] looking at adversarial attacks in computer vision (where adversaries change specific pixels), adversarial attacks on malware are far less understood.

For the purposes of this paper, we divide such attacks into two broad categories. The first group relies on problem-space obfuscation. Here we consider the problem space as a domain containing real-world objects (e.g., malware code, images, audio). Obfuscations in the problem-space change the semantic meanings of code snippets and further obfuscate the malicious signatures or patterns, thereby fooling rule-based malware detectors. Researchers have proposed a variety of such obfuscation techniques to generate adversarial malware that can evade detection by manipulating this domain [9, 13, 18, 21, 24, 41]. These include approaches such as hiding the control flow, inserting dummy code, and manipulating variable names.

The second group of adversarial attack relies on feature-space manipulation. This is performed on feature vectors that a detector induces from the problem-space. For example, a malware detector may induce a feature vector representing the control flow of malware code. This, however, means an attacker must know exactly how to change the problem-space (e.g., code) to result in a specific change to the feature space. Such attacks are becoming more prominent because machine learning-based detectors have reduced the efficacy of problem-space attacks. This occurs when a problem-space obfuscation does not influence the projected feature space, thereby negating its impact on the malware detector.

Despite this, feature-space manipulation is more difficult than obfuscation. This is because, after manipulating the feature space, it is necessary to map the modification onto the malware’s code. However, a single byte change can break the program or damage the malware’s original purpose. As a result, adversaries usually cannot directly modify the raw bytes of the program file. Instead, feature-space manipulation requires finding the correct action(s) on the problem-space that will influence the feature values (but without changing run-time functionality). These actions could be, for example, adding a redundant section (e.g., adding a new code section without linking its address in the section table) or injecting dead code that is unreachable (e.g., adding a file I/O request under an always-false condition, so that the dummy code will never be executed). Note, we are not the first to explore this topic. Similar
techniques have been implemented by Demetrio et al. [31] in a black-box optimization of adversarial Windows malware. However, they focus on the problem-space, instead of feature-space manipulation.

With the above challenge in mind, we focus on exploring how to guide feature-space manipulations and how to invert them back to the problem space. We do this with the explicit goal of evaluating weaknesses of state-of-the-art malware detectors to identify malware with specific manipulations. Since most commercial malware detectors are not open-source, this must be done in a detector-agnostic manner (i.e., decoupling the attack strategy from the specifics of the detector). With this in mind, we design a detector-agnostic adversarial attack which combines feature-space manipulation and problem-space obfuscation. We then evaluate it against various state-of-the-art malware detectors. In contrast to prior research, we further propose a novel method to explain the root cause of an attack’s ability to work across different detectors (i.e., its “transferability”). Our research helps security researchers to better understand adversarial attacks and provides insights on how to improve malware defence strategies. The main contributions of this paper are four-fold:

- **We propose an explainability-guided & model-agnostic evasion attack.** Our attack generates adversarial malware while preserving the malicious functions of the malware. We exploit SHapley Additive exPlanations (SHAP) and introduce the concept of Accrued Malicious Magnitude (AMM) to guide the feature selection approach for feature-space manipulation. We further combine feature-space manipulation and problem-space obfuscation to improve evasion across detectors.

- **We use our attack as a benchmark to evaluate the ability of state-of-the-art malware detectors to protect against adversarial attacks.** We show that commercial antivirus engines are vulnerable to even simple evasion strategies. Experimental results indicate that our approach has significant evasion capability: It decreases the detection rate of six malware detectors by 52.5%, and bypasses an average of 52.3 of the 64 antivirus engines in VirusTotal (VT).

- **We further explore the effectiveness of feature-space manipulation vs. problem-space obfuscation in ablation studies.** Through our explainability-guided (SHAP) approach, we explain how manipulations trained on one detector can work on another detector (i.e., transferability). We show that the combination of both explainability-guided feature-space manipulation and problem-space obfuscation presents the best evasion results. We show that this can be used to enable attacks without requiring white-box understanding of the detectors.

- **In our evaluation, we expose weaknesses in several state-of-the-art malware detectors and commercial antivirus engines.** We find that some malware detectors are easy to compromise, e.g., being misled by adding one empty line. This may be because they do not focus on malicious functionalities or truly suspicious feature patterns. Considering that the evasion attacks under assessment only utilize off-the-shelf tools without any optimization, evasion attacks in the wild could be far more dangerous.

To the best of our knowledge, this is the first paper to systematically evaluate and understand weaknesses of malware detectors in a way that combines feature-space and problem-space with semantic explainability.

## 2 RELATED WORK & MOTIVATION

In this section, we introduce state-of-the-art research in malware detection and machine learning model explanation domains, followed by an motivating example.

### 2.1 Related Work and Background

**Malware detectors.** Modern antivirus engines utilize rule-based analysis, such as signature matching, static unpacking, heuristics matching, and emulation techniques [7, 44]. However, rule-based antivirus engines rely heavily on expert knowledge. With the advantage of feature extraction derived from machine learning techniques, there is a flurry of work that integrates machine learning models into malware detectors [25, 43, 44, 61, 62]. We focus our evaluation on detectors that use static features due to their prevalence in providing pre-execution detection and prevention for many commercial endpoint protection solutions, such as Kaspersky [8], Avast [4], and ESET [3].

A few studies explore the effect of obfuscations on anti-malware products, utilizing off-the-shelf tools. Maiorca et al. [49] and Pommila [51] evaluated several anti-malware products using code obfuscation by a single tool. Hammad et al. [39] conducted a large-scale empirical study that evaluates the effectiveness of the top anti-malware products, including 7 open-source, academic, and commercial obfuscation tools. Several studies [28, 54] have evaluated machine learning-based malware classifier models with the adversarial samples generated by generative adversarial networks (GANs) or automated poisoning attacks. Yizheng et al. [29] studied machine learning classifiers with global robustness properties. Deqiang et al. [46] conducted an empirical study to detect dataset shift and adversarial examples in Android malware detectors. Compared to the evaluation conducted in our paper, the scope of these studies only covers either the rule-based products or the machine learning-based models in isolation (rather than both).

**Evasion attacks against malware detectors.** The goal of the evasion attacks is to generate a small perturbation for a given malware sample that results in it being misclassified. This type of attack has been extensively explored in computer vision, and previous research efforts have also investigated the applicability of such techniques to malware classification. Xu et al. [63] proposed a genetic programming-based approach to perform a directed search for evasive variants for PDF malware. Demetrio et al. [30] demonstrated that genetic programming based adversarial attacks are applicable to portable executable (PE) malware classifier. Two recent works [23, 57] also applied deep reinforcement learning to generate adversarial samples for Windows PE malware to bypass machine learning models.

**SHapley Additive exPlanations (SHAP).** Research into explainable machine learning has proposed multiple systems to interpret the predictions of complex models. In this paper, we rely on SHAP [47] (based on the coalitional game theory concept of Shapley values). Hence, we briefly describe its operation. The SHAP framework subsumes several earlier model explanation techniques together, including LIME [52] and Integrated Gradients [58]. SHAP has the objective of explaining the final value of a prediction by attributing a value to each feature based on its contribution to the
To motivate our adversarial attack, we analyze a sequence of source code decompiled from an Android malware, which is tagged as malicious by 39 (out of 67) detectors from VirusTotal (VT) [20]. From the source code, we find a snippet of malicious code shown in Figure 1. As shown in lines 3 and 4, the malware executes a native scripts via root permissions by the /su -c . /script1 command.

In order to bypass machine learning-based detectors, we should perturb its feature-space towards 'benign'. Hence, we insert several function calls with always-false condition closure (i.e., time<0) to ensure they are unreachable during run-time, preserving the original (malicious) functionality. These function calls are randomly selected from benign features, which involve a list of function calls extracted from benign apps, provided by an Android dataset, Drebin [25]. The inserted code is marked as blue in the middle part of Figure 1. After rebuilding the source code, the modified binary executes a native scripts via root permissions by the /su -c . /script1 command.

In order to understand why these modifications evade the malware detectors, the remainder of this paper develops an explainability-guided feature-space manipulation and problem-space obfuscation tool. We apply this to multiple datasets and evaluate its efficacy.

### 3 THREAT MODEL & PROBLEM DEFINITION

In this section, we define the threat model and take a deep dive into our research problem.

#### 3.1 Threat Model

We follow the methodology by Carlini et al. [26] and describe the threat model of evasion attacks against malware detectors from three aspects: the adversary’s goals, capabilities, and knowledge.

**Adversary goal.** The adversary’s goal is to manipulate malware samples to evade the detection of malware detectors, including commercial antivirus engines and machine learning-based detectors. The types of malware we consider in this study are Android APKs, Windows PE files, and Unix-like ELF files. In the evaluation, we only use binary detectors which determine if the software under
Figure 2: The overview of our evaluation framework.

test is benign or malicious. The goal of attackers in this work is to cause the malicious samples to be misclassified as benign.

**Adversary capability.** We assume that the adversary does not have access to the training phase or the model of the machine learning-based detectors, nor to the source code of antivirus engines. For instance, the adversary cannot inject poisoned data in the training dataset or manipulate any code or output of the antivirus engines.

Further, in this research, the modification of the malware samples is limited. Concretely, the adversary cannot arbitrarily change the input data. We only consider a manipulation successful if the malware continues to act maliciously as intended by the designer.

**Adversary knowledge.** In this work, we assume a black-box attack scenario in which the adversary has limited knowledge about the detector, and can only obtain the classification results through a limited number of queries. A classification result can be a score or simply a label. The internals of the deployed detectors are agnostic to an attacker, e.g., the attackers do not know whether the detector is a machine learning-based or rule-based and the structure of detectors.

However, they will still have some basic knowledge about learning-based detectors, e.g., the access to open-source datasets, features extraction methods [53], or off-the-shelf machine-learning detectors. We argue this is both practical and reasonable.

### 3.2 Problem Definition

Our goal is to evaluate the efficacy of evasion attacks against malware detectors using generated adversarial samples. Considering a malware detector mapping a software sample $x \in X$ to a classification label $l \in \{0, 1\}$ (where 0 represents benign and 1 represents malicious), the goal of evasion attack can be summarized as:

$$F(x) = 1, x_a = Gen(x), F(x_a) = 0, \quad (3)$$

where $F$ could be either a trained machine learning model or an antivirus engine. $x$ is the original malware sample, and $Gen$ is the sample generator that is able to generate adversarial sample $x_a$ while keeping its malware functionality the same as $x$.

To ensure the reproducibility and coverage of our evaluation, we have several criteria on the selection of evasion attack and adversarial sample generation strategies:

- **Easy-to-obtain.** We only utilize open-source and off-the-shelf tools, instead of proposing any new attack technique ourselves.
- **Compatible.** We combine multiple evasion attacks, which we believe will put greater stress on the detectors and make the evaluation as comprehensive as possible.
- **Explainable.** An explainable approach is preferred as it will help us to analyze the evaluation results and find out potential weaknesses in malware detectors.

To establish such an evaluation strategy, we consider generating adversarial samples through perturbation in both the feature space (e.g., manipulating values in feature vectors) and the problem space (e.g., modifying malware source code). To achieve our goal, the problem can be split into two sub-problems: (i) generating adversarial samples and (ii) evaluating them against malware detectors, both of which are detailed in Section 4.

### 3.3 Ethical Considerations

Our research is concentrated on the **defence** scope that explains the adversarial evasion attacks and determines the potential weaknesses of current malware detection methodologies. Hence, we declare: (i) the motivating example we presented is only a code snippet without actual functionality; (ii) all tools and datasets involved in our experiment are publicly available, and we also anonymize the antivirus engines with a simple serial number label; (iii) considering the potential security issues, we will not release the source code of the proposed attack and any adversarial samples, as well as the information of commercial antivirus involved in our evaluation, except for academic uses approved by the ethical committee.
4 METHODOLOGY

Our research methodology consists of three key components (see Figure 2): (i) explainability-guided feature selection, to select the feature manipulation; (ii) an adversarial sample generator, to generate the evasive samples; and (iii) an evasion attack evaluation, to evaluate the proposed evasion attacks against more than 60 malware detectors and to explain their transferability. Here, transferability refers to the the performance of the evasion attack when generating adversarial samples for one machine-learning model and then applying them to different detectors.

4.1 Explainability-Guided Feature Selection

In the first step of our methodology, we utilize SHapley Additive exPlanations (SHAP) to create an explainability-guided attack. Adversarial attacks can be applied either in a model-specific or model-agnostic manner under SHAP. The workflow of the explainability-guided feature selection is illustrated in Algorithm 1. For a set of seed malware, S, we aim to generate a corresponding adversarial sample set, A, such that they evade the target model only by adding or modifying features.

Pre-processing. We first extract features from the training samples X of a trained machine learning model m (line 2). Then both vectorized samples, X’, and the model are input to shap() to calculate the SHAP value matrix M (line 3). The matrix is then used to select the most evasive features and the most benign-oriented values.

Feature selection. To select the feature that has largest malicious magnitude, we propose the concept of Accrued Malicious Magnitude (AMM). The AMM is defined as the product of the magnitude of SHAP values in each feature and the number of samples that have malicious-oriented values in feature space. Specifically, starting from the getRange() line 5, we first calculate the range of SHAP values in each feature and store the results in a one-dimension vector D. D indicates the potential magnitude we can modify on each feature, i.e., each di ∈ D presents the difference between the maximum value and the minimum value of feature fi. Next, for each feature, we count how many samples have SHAP value larger than the mean SHAP value of that feature (the countLarge(M) in line 6). Note that in our experiments, we labeled malicious as 1 and benign as 0. This helps us to determine which feature potentially has more samples that can be modified towards benign. Therefore, a larger ci ∈ C means that, for feature fi, there are more samples that have a SHAP value towards malicious (larger than the medium), such that more samples can be manipulated towards benign. Therefore, we select the most evasive feature according to the AMM values, denoting the dot product of the range of SHAP values (D) and the number of SHAP values greater than mean (C) (line 7). By calculating AMM values, denoting the dot product of the range of SHAP values in each feature and store the results in a one-dimension vectorized samples, adversarial attacks can be applied either in a model-specific or model-agnostic manner. The workflow of the explainability-guided attack is illustrated in Algorithm 1. For a set of seed malware, S, we aim to generate a corresponding adversarial sample set, A, such that they evade the target model only by adding or modifying features.

Greedy strategy. After obtaining feature-value pairs, we conduct a greedy strategy, removing samples that have the same value, v, for feature f from the dataset (lines 12 to 16). We do this to make sure that the same feature-value pair will not be selected again. The procedure repeats until we find N feature-value pairs. These N pairs are then used in the next stage to generate the adversarial malware samples.

4.2 Adversarial Sample Generator

The adversarial sample generator implements two strategies: feature-space manipulation and problem-space obfuscation. Explainability-guided feature-space manipulation involves changing features selected by Algorithm 1 to mislead the detector. Obfuscation transforms source code into a more complicated form for humans, and possibly machines, to read and understand. Note that we later experiment with running these two stages in different sequences i.e., explainability-guided first (E-O) or obfuscation first (O-E).

Feature-space manipulation. To train a machine learning model, the first step is to convert input data into vectors of features (i.e., the feature extraction process). In an evasion attack, we manipulate features to induce misclassifications. However, not every feature has equal influence on the result of the detector, so the question becomes: how can we gain insight into a model’s decision in a generic, model-agnostic way? Thus, we rely on SHAP to understand which features drive the model towards a benign classification. Guided by SHAP, we can manipulate the malware sample and

Algorithm 1: Explainability-Guided Feature-Space Selection

| Input: | Machine learning model m, dataset X, and the number of features to be selected N. |
|---|---|
| Output: | Feature patch P. |
| 1 | P = map(Feature, Value). |
| 2 | X’ = vectorize(X). |
| 3 | M = shap(X’, m). |
| 4 | while size(P) < N do |
| 5 | D = getRange(M). |
| 6 | C = countLarge(M). |
| 7 | AMM = D · C. |
| 8 | f = arg max(AMM). |
| 9 | v = arg min(M[f]). |
| 10 | if isManipulatable(f) then |
| 11 | P = P ∪ (f, v). |
| 12 | for each x’ ∈ X’ do |
| 13 | if x’[f] ≠ v then |
| 14 | idx = getIndex(x’, x’); |
| 15 | M = M \ [M[idx]; |
| 16 | X’ = X’ \ x’; |
| 17 | return P. |

Update feature patch. After obtaining a pair of (f, v), if the selected feature f is manipulable, we add the pair into map P as the Feature Patch to be used in the feature-space manipulation (line 11). Although, the SHAP framework can find features that impact the decision boundary, some of them cannot be manipulated directly. For example, considering the feature that counts the size of a binary, when we modify the value of another feature, the former one will be modified indirectly. Therefore, the features and values we select to be manipulated follow two principles employed by the previous literature [37, 38, 53] on adversarial evasion attacks against malware detectors. These principles are: (i) features are manipulable in the original problem space; (ii) selected features have no dependencies or cannot be affected by other features.
cause a misclassification. Importantly, we must ensure the malware functionality is preserved. Equation 4 summarizes the feature-space manipulation:

\[
x' = \text{vectorize}(x), \\
a' = \text{manipulateFeature}(x', P), \\
\text{Gen}'(x) = \text{buildSample}(a', x),
\]

where \(x'\) is the result of applying feature extraction on sample \(x\) using vectorize(). The SHAP value matrix, \(M\), is obtained through shap(). The SHAP algorithm. \(m\) represents the machine learning model. manipulateFeature() manipulates the sample in feature-space guided by SHAP. Note that, the sample generator \(\text{Gen}'()\) will take the manipulated feature-space sample \(a'\) and the original sample \(x\) as input, and implement the changes in feature-space back to problem-space to generate the adversarial sample, while keeping its malware functionality.

Note that, due to the strong semantic restrictions of the binaries, we cannot simply choose any arbitrary pairs of feature and values for our evasion attack. Instead, we restrict the feature-space manipulation to only features and values that are independent and can be modified with original functionalities preserved. Therefore, we design a binary builder to implement the inverting of features, and mapping the manipulation back to the problem space.

Concretely, for each seed malware, we apply the feature extractor to convert it into the feature space as \(\text{seed}'\) and then manipulate its features according to the map \(P\) we collected previously in Algorithm 1. With the manipulated feature-space sample \(a'\), we conduct buildSample() to generate a problem-space adversarial sample. The specific procedure is detailed in Section 5.4.

**Problem-space obfuscation.** Considering that many antivirus malware detectors are not based on machine learning models, the capability of a feature-space manipulation could be limited. Therefore, we also apply problem-space obfuscation on the adversarial sample generated by \(\text{Gen}'(x)\) to evade the antivirus engines. As the sample has been generated in the problem-space after previous feature-space manipulation, we can directly input it to the obfuscation tool \(\text{obfus}()\) as shown in Equation 5, where \(s\) presents the obfuscation strategy, and \(\text{Gen}()\) is the adversarial sample generator that generates \(x_a\) in Equation 3.

\[
\text{Gen}(x) = \text{obfus}(\text{Gen}'(x), s), \\
x_a = \text{Gen}(x).
\]

In our study, we adopt the following obfuscation techniques:

- **Control-flow Graph Alteration (CFG).** A control-flow graph (CFG) is a representation, using graph notation, of all paths that might be traversed through a program during its execution. This technique modifies the CFG without impacting the code semantics [24]. This strategy can bypass some CFG and syntactic analysis-based antivirus scanners.

- **Dead-Code Insertion (DCI).** Dead-code insertion adds ineffective instructions, e.g., NOP, JMP or debugging information to a program. These instructions do not impact the functionality while altering the sequence of instruction execution.

- **Instruction Substitution (IS).** This technique is to replace specific instructions with equivalent ones. In the Android system, we leverage reflection APIs to replace and envelop original function calls to make them difficult to be detected by bytecode analysis.

- **Encryption (ENC).** The obfuscator encrypts sensitive resources, e.g., URLs, malicious code sections and embedded native libraries, in the binary. Encryption in this transformation allows antivirus engines that rely on specific text strings and bytecode to be evaded. On WinPE or ELF binaries, this technique encrypts section tables and moves section contents to bypass antivirus engines.

We will discuss how the problem-space obfuscation is implemented in particular in Section 5.5.

**Execution.** Considering that both problem-space obfuscation and feature-space manipulation modify the structure and instructions of a binary, we propose two sequences of integrating both strategies: (i) explainability-guided first and then obfuscation-guided (E-O) approach; and (ii) obfuscation-guided first and then explainability-guided (O-E) approach.

### 4.3 Evasion Attack Evaluation

In the final phase of our research methodology, we evaluate the evasion performance of our proposed strategy against commercial and open source antivirus engines, learning-based detectors and VirusTotal [20]. The detectors are summarized in Table 1.

| Name | Description |
|------|-------------|
| AV1  | An open-source rule-based antivirus engine. |
| AV2  | A commercial hybrid antivirus engine. |
| AV3  | A commercial hybrid antivirus engine. |
| SVM  | A linear support vector machine classifier. |
| GBM  | LightGBM, a tree-based classifier. |
| NN   | A feed-forward neural network with 3 hidden layers. |
| VT   | VirusTotal, a free online service that integrates over 70 antivirus detectors. We use 64 of them. |

AV1 is an open-source rule-based antivirus engine developed and actively maintained by a historic technology corporation. AV2 and AV3 are two historic and world-class commercial antivirus engines available for more than 30 years. AV2 and AV3 claim that they are hybrid antivirus engines which adopt both rule-based and machine learning-based techniques. Further, we use VirusTotal in our evaluation to enlarge the scope. VirusTotal is an online service that provides over 70 antivirus scanners to detect malicious files and URLs. Our experiment found that 64 scanners are always available in malware detection while others are not stable, i.e., sometimes available and sometimes not. Therefore, we leverage these 64 scanners as a benchmark. Researchers and users can upload their files and obtain detection results via their API. To follow the convention of the prior studies [25, 53], we also select three off-the-shelf machine learning-based malware detectors. Support Vector Machines (SVMs) are a set of supervised learning methods used for classification, regression and outliers detection. LightGBM [42] (GBM) is a free and open source distributed gradient boosting framework, based on the decision tree algorithm, originally developed by Microsoft. In addition, we introduce a simple-structured feed-forward neural network (NN), containing five layers: an input layer; three fully-connected hidden layers that use ReLU activation functions;
and the last one ends with a Sigmoid activation (a standard choice for binary classification).

5 EXPERIMENT SETUP

In this section, we describe the setup of our experiment including the experiment environment, the datasets and model training, how we extract features, how we implement the binary builder, and how we obfuscate the sample.

5.1 Experimental Environment

Our experimental environment is a PC workstation with 64GB RAM, AMD Ryzen 3750X 8-core CPU and Linux Mint 20.1 Cinnamon installed. We also utilise Windows 10 Pro 20H2 version running in the VirtualBox [12] for WinPE obfuscation and the antivirus engine evaluation. To reduce mutual influence between antivirus engines, we make a snapshot of a pure Windows system with the embedded Microsoft Defender [11] disabled. After each individual experiment on each dataset and antivirus engine completes, we restore the snapshot to rollback the system back to the status when the system is initially installed.

5.2 Datasets and Model Training

In our experiments, we rely on the three datasets listed in Table 2, covering major operating systems: Windows, Unix-like, and Android.

SOREL-20M (WinPE). The Portable Executable (PE) format is the standard file format for executable files, object code, and Dynamic Link Libraries (DLLs) used in 32- and 64-bit versions of the Windows operating systems. We use SOREL-20M [40] as the WinPE dataset in our experiment. SOREL-20M is a representative public dataset of malicious and benign WinPE samples used for malware classification, consisting of extracted features of 9,470,626 benign and 9,919,251 malicious samples, as well as corresponding malicious samples in the dataset. We use these malicious APKs to train an SVM model which was adopted from Drebin using Androguard [2]. Here, we describe how we perform feature extraction.

Androguard. Androguard [2] is a python tool to analyze and manipulate Android files. It disassembles an APK file and converts its byte code and resource files into a readable and structured format. We leverage Androguard to extract features.

From the Drebin dataset, Androguard extracts over 545,000 features. These features are divided into 8 logical subsets representing hardware components, requested permissions, app components, filtered intents, restricted API calls, permissions used in the source codes, suspicious API calls, and network addresses, respectively. We further extract the first four types of features from the manifest file and the rest four from the Dalvik Executable (dex) file, and then we use these features in the detection of learning-based detectors.

5.4 Binary Builder Implementation

In this section, we introduce how we map (and implement) feature-space manipulations back to the problem space.

Binary builder for WinPE & ELF. The features in WinPE and ELF are divided into three types: (i) can be manipulated directly; (ii) can be manipulated indirectly by modifying other features; and (iii) cannot be manipulated.

Features that can be manipulated directly include static information values, such as major_linker_version in WinPE and machine_type in ELF. Features that can be manipulated indirectly include statistical information, e.g., sizeof_sections and num_strings (the number of constant text strings in the binary). Features that cannot be modified are parts of hash vectors and histograms. We leverage LIEF to manipulate features that can be modified directly and indirectly for both WinPE and ELF files.
We next detail the obfuscation strategies and tools used in our approach, as summarized in Table 3.

### 5.5 Obfuscation Strategies

To demonstrate how to manipulate the feature space, we explain an example where we change the timestamp and increase sizeof_sections and numstrs in Android. First, we modify timestamp directly in the PE header. Then, we generate a new section whose content is a byte vector of random text strings. The section name is set as a data, indicating read-only constant strings. Finally, we insert the section into the binary, add the section header into the Section Table and modify size of sections in the Optional Header.

We do not modify or remove existing sections for the purpose of preserving the functionality of a sample; instead, we modify general information, e.g., timestamp and major_linker_version, and add new sections that are not executable. We leverage the LIEF framework to manipulate features that can be modified directly or specifically affected, and rebuild a new adversarial sample.

**Binary builder for APK.** APK features can be manipulated on byte code in the dex files and the manifest file. To ensure no loss of functionality is inadvertently introduced as a side effect of feature manipulation, we only add unreachable API calls into the byte code and inject items into the manifest file so that these changes will never be executed during run-time. Specifically, since features are a vector of boolean values representing the existence of a feature, proposed by Drebin, the feature value could only be modified from 0 (absence) to 1 (presence). If a feature manipulation cannot be implemented in this way, we skip it and continue with the next most important feature.

For instance, assume that the feature-space manipulation requires us to modify the value of feature api_call::java.io.File.delete and permission::android.permission.SYSTEM_ALERT_WINDOW from 0 to 1. To achieve this, we first add java.io.File.delete() into APK byte code. We create an instance of java.io.File with an empty string parameter and invoke its delete() function under an if (false) closure to make it unreachable. Then, we add an item, <uses-permission android:name="android.permission.SYSTEM_ALERT_WINDOW" />, into AndroidManifest.xml.

### RelocBonus for WinPE

To conduct binary obfuscation on WinPE files in problem space, we leverage an open source obfuscation project, RelocBonus [13], which instruments the Windows Loader into acting as an unpacking engine. RelocBonus implements the Encryption strategy of obfuscation mentioned in Table 3. This obfuscation method starts with multiple ASLR Preselection attacks that force binary mapping at a predictable address. It then encrypts the entry section and embeds a new Relocations Table that, when paired with a preselected base address, causes the loader to reconstruct the PE and execute it with ease. Its commercial counterparts, e.g., VMProject [21] and Virbox Protector [18], are not used since they are close-sourced and their obfuscation principles are not publicly available.

### Obfuscapk for APK

An Android application can be easily compiled into smali [15] code, an assembly language that supports the full functionality of the dex format. We leverage Obfuscapk [24], an opensource obfuscation tool that is actively maintained, to obfuscate smali code, then rebuild the code into a variant sample. Obfuscapk offers 12 different obfuscation strategies (listed in Table 6 in appendix) covering all three categories of binary obfuscation techniques (i.e., CFG alteration, Dummy Code Insertion and Encryption) mentioned in Section 4.2. Note that, multiple strategies can be used as a combination in practice.

However, we are not going to adopt all of the strategy combinations onto each feature-space manipulated adversarial samples, as there could be $2^{12}$ combinations. Instead, we conduct a strategy selection experiment to find the most effective combinations of strategies. We first randomly select 20 malicious APK samples as seeds and generate 1 sample for each combination of strategy. Then we submit all the 81,920 obfuscated samples (4,096 samples per seed) to VirusTotal and record the average reduction on detection rate across antivirus engines to select the most evasive strategy. Finally, we identify a combinations of five strategies that have the best evasion results and apply it in the problem-space obfuscation (as detailed in Section A in Appendix).

### Kiteshield for ELF

We choose a state-of-the-art and off-the-shelf tool, Kiteshield [9], to conduct the obfuscation process on ELF files. Kiteshield implements the Encryption strategy in Table 3. It wraps ELF binaries with multiple layers of encryption and injects them with loader code that decrypts, maps, and executes the packed binary entirely in userspace. In contrast, its counterparts, e.g., UPX [17] and ELF Encrypter [6], either simply compress executable binaries or are outdated.

### 6 EVALUATION & RESULTS

We next employ our pipeline to evaluate the efficacy of our evasion techniques, as well as test the susceptibility of the detectors to adversarial attacks. Specifically, we compare the detection rates between

### Table 2: Datasets and models training

| Name             | Type     | # Samples in Datasets Benign | Malicious | Model    | Training Set | Validation Set | Testing Set | Feature Extractor | # Features | # Seeds | # Adversarial Samples |
|------------------|----------|------------------------------|-----------|----------|-------------|----------------|-------------|-------------------|------------|--------|----------------------|
| DREBIN           | APK      | 123,453                      | 5,560     | SVM      | 5,560       | 2,224          | 3,336       | Androguard        | 545,000    | 2,000  | 4,000                |
| SOREL-20M        | WinPE   | 9,470,626                    | 9,919,251 | LightGBM | N/A²        | N/A           | N/A         | LIEF              | 2,381      | 2,000  | 4,000                |
| VirusShare ELF   | ELF      | 6,439                        | 45,533¹   | NN       | 6,439       | 2,574          | 3,863       | LIEF              | 2,448      | 2,000  | 4,000                |

¹: We use its pre-trained model; ²: ELF benign binaries are obtained from Linux Mint; ³: ELF malicious binaries are obtained from VirusShare.
We start by evaluating our proposed evasion attack against learning-
ori-
we focus on the difference in detection rates between the
APKs, as shown in Figure 3(b). The detection rates for AV2 and AV3
AV1, AV2, AV3 and VT. The detection rate of VT is calculated as
(E-O and O-E). Again, engines in VT performs poorly, detecting
(E-O) and 7.2% (O-E) adversarial samples; AV3 performs similar
(ii)
against adversarial samples with the E-O strategy for both WinPE
gies) and 85.1% (both strategies). That said, the detection rate of
the three learning based detectors: SVM, GBM, and NN. These follow the configurations shown in Table 2.

6.1 Performance of Adversarial Attacks
We start by evaluating our proposed evasion attack against learning-
biased detectors and antivirus engines. As our performance metric,
we focus on the difference in detection rates between the original
samples and the adversarial samples generated by our proposed strategy. We generate two types of adversarial samples: (i) explainability-guided first and then obfuscation-guided approach (E-O); and (ii) obfuscation-guided first and then explainability-guided approach (O-E). We then perform an ablation study to explain the contribution of each component in our proposed strategy by comparing explainability-only (E) and obfuscation-only (O) samples.

Performance of AV engines. We test several antivirus engines,
AV1, AV2, AV3 and VT. The detection rate of VT is calculated as
the average detection rate across the 64 engines.

Figure 3(a) shows that the three antivirus engines and VT do not identify a large portion of our adversarial samples with either E-O and O-E strategies. Specifically, AV1 cannot detect any adversarial samples with both E-O and O-E strategies; AV2 identifies 7.5% (E-O) and 7.2% (O-E) adversarial samples; AV3 performs similar to AV2, detecting 6.2% (E-O) and 6.1% (O-E) adversarial samples. Remarkably, the detection rate of antivirus engines in VirusTotal decreases from 53.1% to 13.5% (E-O) and 10.2% (O-E).

In contrast, the AVs tend to perform better against adversarial APKs, as shown in Figure 3(b). The detection rates for AV2 and AV3 decrease only slightly from 99.8% and 98.2% to 93.4% (both strategies) and 85.1% (both strategies). That said, the detection rate of AV1 decreases considerably from 72.5% (original samples) to 32.7% (E-O and O-E). Again, engines in VT performs poorly, detecting just 23.2% of adversarial samples with both strategies (compared to 52.7% of original samples). Worryingly, the AVs perform worst on the ELF dataset, as shown in Figure 3(c), where no adversarial samples can be detected.

Performance of ML detectors. We next test the three machine learning-based detectors: SVM, GBM, and NN. Figure 3(a) shows that the three learning based detectors perform extremely well against adversarial samples with the E-O strategy for both WinPE and ELF malware. Despite this, they cannot identify a notable portion of adversarial samples generated using the O-E strategy. For example, the SVM can only detect 52.5% of malware generated using the O-E strategy. This compares to 93.2% for the original samples. The GBM detector performs worst: the detection rates of type O-E is just 7.5% compared to 99.1% for the original samples. Similarly, the detection rate on the NN is 13.2% (O-E) compare to 99.9% for the original ones. This confirms that attackers who apply feature-space manipulations after problem-space obfuscation will evade more detectors.

APK malware exhibits markedly different trends though. In Figure 3(b), we see that the detection rates for SVM, GBM, and NN are all lower than 5.2% when compared against their original samples (that attain over 99%). In contrast to both ELF and WinPE malware, both E-O and O-E strategies have the same detection rates on APKs. To put differently, the sequence of adopting explainability-guided and obfuscation-guided strategy does not affect the evasion performance of APKs.

Summary. From the experimental results above, antivirus engines and learning-based detectors can both be evaded by our adversarial generation strategies. Antivirus engines and VirusTotal perform the worst on ELF adversarial samples, where the detection rate decreases from 99% to 0%. Learning-based detectors can also be evaded by APK adversarial samples, (the rate decreases from 99.5% to as low as 2.7%). We find that the sequence of explainability-guided and obfuscation-guided approaches does not impact the attack performance for APK malware. However, there is a noticeable impact for WinPE and ELF malware, particularly for learning-based detectors. We will discuss the reason in Sections 6.2 and 6.3.

Takeaway 1: Both antivirus engines and learning-based detectors can be effectively evaded by our proposed adversarial generation strategies.

6.2 Ablation Study
To explore which part of the adversarial generation contributes most to the efficacy of the evasion attacks, we next perform an ablation study. For this, we generate samples using explainability-only and obfuscation-only approaches. We then test if they can be correctly identified as malware by the detectors. Figure 4 presents the results.

Performance of Antivirus engines. Overall, antivirus engines and VirusTotal detect more explainability-only samples than obfuscation-only samples. For WinPE malware (Figure 4(a)), the detection rate of obfuscation-only samples against AV1, AV2, and


In contrast, detection rates for APKs are rather different to either WinPE or ELF malware. In Figure 4(b), we see the detection rates of explainability-only and obfuscation-only samples are actually very similar (differences between 1.2% and 6.6%).

Performance of ML detectors. The detection rate of the learning-based detectors are very different to the AVs. Here, the explainability-only strategy performs better than the obfuscation-only approach. The models sustain almost 100% detection rates for WinPE, APK, and ELF malware when using obfuscation alone.

This shows that an attacker must rely on explainability-only attacks for subvert learning-based detectors. In Figure 4(a), we see that the detection rates of explainability-only WinPE samples are just 32.7% (SVM), 6.0% (GBM) and 1.4% (NN). We see similar trends in Figure 4(b) for APKs. That said, the learning-based detectors are far more robust against ELF malware. For example, Figure 4(c) shows that the SVM model can still detect 65.3% of explainability-only samples (although GBM and NN models perform much worse).

To ascertain the reason why learning-based models perform much better than AVs on obfuscated-only malicious samples, we conducted an additional experiment which inputs the obfuscated benign samples to models. We found that the obfuscation strategy on WinPE and ELF datasets can significantly challenge the accuracy of learning-based detectors as all obfuscated benign samples are misclassified as malicious.

Summary. The above shows that learning-based detectors can be evaded by adversarial samples adopting the explainability-guided approach; whereas obfuscation tends to be more effective when evading AVs. Obfuscation alone can significantly challenge the accuracy of learning-based detectors. A case is further studied to explain such a challenge in Section 7.2.

Takeaway 2: The obfuscation strategy contributes more to the evasion attack on antivirus engines and challenges the accuracy of learning-based detectors, whereas the explainability-guided approach performs better against learning-based detectors.

6.3 Transferability Analysis
Transferability is the ability for an attack to be effective against multiple learning-based detectors. To study this, we next train the three learning-based detectors with each dataset to analyze the transferability of our proposed evasion attack. Here, we seek to understand how a manipulation guided by one detector performs against the other detectors.

From the earlier results in Figure 3, we observe that the explainability-guided approach consistently evades detection across all three learning-based detectors. Recall that, according to Algorithm 1, we select the features with highest AMM values as the most evasive features to conduct feature-space manipulation.

Feature overlap. To explore the reason why our proposed attack can transfer across detectors, we present the AMM values of the top features across each dataset in a heatmap, shown in Figure 5. As the number of features differs across each dataset, we only display...
the top 1024 features that have the highest AMM values in the original detector models (SVM model for APK dataset, GBM model for WinPE dataset, and NN model of ELF dataset). In each subplot, we present the 1024 features as 32 rows by 32 columns of dots (normalized to [0, 1]), where the darker dots indicate higher values of AMM (which indicates a greater possibility to be selected a feature to be manipulated). Further, we sort the features in the original dataset according to the AMM values in descending order. Therefore, darker dots scattered in the upper zone of the nine subplots indicate that there are more features having been selected across detectors.

From the heatmap we can observe that (i) the features in the APK dataset generated from the SVM have a number of overlaps with the counterparts from GBM and NN; (ii) the AMM values of WinPE from GBM and ELF from NN have less overlaps with SVM. The overlaps resonate with the main study results — the transferability of APK samples outperforms the transferability of WinPE and ELF samples, as the overlap of features with high AMM values in APK is larger than the overlap in WinPE and ELF. Thus, the overlaps explains why the evasion attack can transfer across learning-based detectors. Simply put, if we manipulate enough features across different learning-based models (i.e., feature overlaps), the evasion attack can be transferred.

Detection rate. We can also evaluate transferability of adversarial samples generated by inspecting the detection rates. Figure 6 shows the detection rates of all models, across all datasets. The y-axis represents each dataset with three generating models, and the x-axis shows the target detectors. Figure 6(a) is generated from adversarial samples with first explainability-guided and then obfuscation-guided (E-O) sequence while Figure 6(b) shows the contrary (O-E) sequence. In the figure, a darker color indicates a higher detection rate (representing lower transferability).

As shown in Figure 6(a), the transferability of the E-O strategy on WinPE and ELF is impacted heavily by the obfuscation, as discussed in Section 6.2. From Figure 6(b), we see APK samples have better transferability performance than WinPE and ELF samples on learning-based models (16.5% vs. 47.7% and 71.5%, averagely), which aligns with the feature overlaps shown in Figure 5. Antivirus engines have poor performance on ELF and WinPE adversarial samples with both strategies: under 7.5% of WinPE samples and no ELF samples are detected by all three AVs.

Regarding the APK dataset, both E-O and O-E have similar detection rates by all detectors. APK adversarial samples have limited evasion performance on AV2 and AV3, while under 35.8% of samples are detected by AV1. Adversarial samples generated from SVM have the best transferability to other learning-based detectors: under 5.2% samples generated from SVM are detected by SVM, GBM and NN. This result resonates with the overlaps in the heatmap shown in Figure 5, where feature vector components from SVM overlap most of the counterparts of GBM and NN.

Takeaway 3: The evasion attack transferability depends on the overlaps of features with large Accrued Malicious Magnitude (AMM) values between different learning-based models.

7 CASE STUDIES
In the previous section, we are left with two issues: (i) our adversarial generation strategy cannot evade learning-based detection for some malware samples; (ii) adversarial samples generated with a different sequence of feature-space manipulation and problem-space obfuscation result in different detection rates. To explore these issues, we present two case studies.

7.1 A Case Study on Evasion Capability
We have shown that some adversarial samples can evade their generation learning-based models, but some cannot. The reason could be either that the number of manipulated features is not enough to invert the prediction, or that the manipulated features have a limited impact on the prediction. To explore the reason, we choose two original APK samples, Sample 1 and Sample 2, to generate their adversarial samples. We then test their evasion capability. The adversarial sample of Sample 1 inverts its prediction as benign, and that of Sample 2 remains malicious.

First we use Sample 1 and Sample 2 to manipulate different size of features (guided by Algorithm 1) and compare their decision function values. The decision function (DF) is the support vector of the SVM. A negative DF value represents benign and positive represents malicious. Initially, the DF value of Sample 1 and Sample 2 are 0.231 and 0.247. When selecting N = 75 features to manipulate, the DF value of Sample 1 turns to -0.013 (i.e., benign). In contrast, the DF value of Sample 2 remains positive (i.e., malicious) at 0.068.

We further generate SHAP values of the original and adversarial samples (with N = 75 features selected) of Sample 1 and Sample 2 to analyze impact of the manipulated features. The results are shown in Figure 7. The subfigures labeled (A) and (B) indicate the original and adversarial samples of Sample 1, and (C) and (D) are for Sample 2. The X-axis indicates the prediction value; the Y-axis is the feature ID. $f(x)$ and $f(x_a)$ are the prediction value of the original and adversarial samples. We use red bars to indicate positive SHAP values, and blue bars to highlight negative SHAP values of each feature ($\phi_j$ in Equation 1). Feature IDs shown in the figure are parts of manipulated features that had the greatest impact on Sample 1.

Figure 7(b) shows that the SHAP values of manipulated features change significantly towards negative (blue bars), thereby pushing the output, $f(x_a)$, towards negative. In contrast, the SHAP values of the features of Sample 2, shown in Figure 7(d), change far less. Specifically, only features 19853, 20201 and 20886 are manipulated towards negative while others are not. This means that we cannot manipulate enough features to force the decision making towards benign for Sample 2.

Moreover, Sample 2 has more positive SHAP values than Sample 1 (see Figure 7(c)). This means it requires more negative features to manipulate. This result indicates that the manipulated features have limited impact on Sample 2 to invert the result from malicious to benign. Arguably, if we increase the range of selecting feature to 290, Sample 2 is then identified as benign. Therefore, the detection result depends on how many features have malicious-oriented values that we can manipulate in the sample. However, infinitely increasing the number of selecting features would also lead to heavily computational load and decrease the efficiency of attack.
After generating the adversarial samples, we extract their features. Therefore, the obfuscated samples can mislead the detection of whether a sample is benign or malicious.

| Feature                     | Strategies          |
|-----------------------------|---------------------|
| File size                   | Seed E O E-O O-E    |
| Start of program headers    | 175.488 1.338.088   |
| Size of program headers     | 64 1.331.200 64    |
| Number of program headers   | 56 116 56 56 116    |
| Start of section headers    | 3 123 2 62 122     |
| Size of section headers     | 146.112 1.290.936   |
| Number of section headers   | 64 124 64 64 124   |

Takeaway 4: The evasion capability of a manipulated sample depends on the number of its features with malicious-oriented values that can be manipulated.

7.2 A Case Study on Sequence of Feature-Space and Problem-Space Modification

From Figure 3(c), we found that adversarial samples that are generated by running a different sequence of feature-space manipulation and problem-space obfuscation leads to different detection rates. This was particularly the case for learning-based detectors when applied to the WinPE and ELF datasets. Specifically, regarding WinPE and ELF, the average detection rate of adversarial samples with E-O strategy is 100% while the counterparts of O-E strategy is 24.4% (WinPE) and 43.1% (ELF). To ascertain this, we conduct a case study on an ELF malicious sample, which performs worst with the explainability-first-obfuscation-last (E-O) sequence.

We generate four adversarial samples from this ELF binary with explainability-only (E), obfuscation-only (O), explainability-first-obfuscation-last (E-O) strategy and the contrary (O-E) strategy. After generating the adversarial samples, we extract their features and submit them to NN, which identify samples with E and O-E as benign and O and E-O as malicious.

We then compare their features, and the key differences are shown in Table 4. From the table, we see that after adopting obfuscation, the features of both the O and E-O samples are similar to that of the original seed sample. By contrast, the features of samples with O-E are similar to E but significantly different from the original sample, which means that feature-space manipulation is maintained by O-E strategy, making the O-E sample evade the detection. For instance, the start of program headers, size of program and section headers of both O and E-O are 64 bytes, 56 bytes and 64 bytes, remaining the same with seed. Since the O and E-O samples along with the original sample are identified as malicious while E and O-E are benign, we confirm that the obfuscation strategy encrypting section data in ELF’s binary can affect the feature space and worsen the evasion performance.

We further obfuscated 1000 benign samples from Linux system-installed binaries, and the results are all malicious. By reviewing the feature space of obfuscated samples, the key features listed in Table 4 are modified to the values that are similar to malware. Therefore, the obfuscated samples can mislead the detection of learning-based detectors to malicious ignoring whether the original sample is benign or malicious.

Takeaway 4: The evasion capability of a manipulated sample depends on the number of its features with malicious-oriented values that can be manipulated.

8 DISCUSSION

This section discusses challenges to our work, as well as limitations and how they might be addressed in the future.

Learning models. Our experiments are conducted on three existing models followed by the prior research [25, 53], including SVM, LightGBM and a feed-forward neural network. These models, however, are trained with the default configuration and parameters from each dataset. This does not preclude the possibility that alternative models and configurations may gain superior performance.

Adaptive defenses. Existing defenses against adversarial generation, for example, adversarial training [48] and differential privacy (DP) [22, 34, 45], may not be effective against our proposed evasion attack. For adversarial training, it is because that our attack is based on the transferability of important features among different learning-based detectors, in lieu of optimization tricks, e.g., FGSM [36]. On the other hand, DP-based robust machine learning techniques cannot defend against our attacks, because unbounded random perturbations may break the generated samples’ functionality. One possible adaptive defense is to analyze feature distribution of queries before we carry out de facto malware detection. For example, Slack et al. [56] propose an adversarial classification approach to fool explanation methods, e.g., SHAP and LIME. They offer a different classifier, e.g., an extremely biased one, along with the original one to the input according to the perturbation distribution. However, this classification approach can be an attack target and cannot always boost the detection accuracy.

Dynamic detection. Dynamic feature detection can be a practical defense against our evasion attack since we only insert static unreachable instructions into the malware. Feature-space manipulation and problem-space obfuscation rely on static syntactic and structural modification. These modifications can bypass static machine learning-based detectors and rule-based antivirus engines. However, the malicious behaviors will still be exposed during runtime and identified by the detectors that adopt dynamic analysis. That said, dynamic feature detection consuming more resources to monitor this approach may be impractical on a large scale.

9 CONCLUSION

This paper has developed and evaluated a new adversarial malware attack, where the goal is to evade detection by antivirus engines. We have proposed a mix of techniques, including explainability-guided (SHAP) feature-space and problem-space manipulation. Using this attack as a benchmark, we confirm the weaknesses of state-of-the-art malware detectors to adversarial attack. Our research includes the following key findings: (i) Attackers can effectively evade antivirus engines and learning-based detectors by concatenating explainability-guided feature manipulation and obfuscation techniques; (ii) Combining these two strategies attains better results than using them in isolation, exposing weaknesses in both antivirus engines and learning-based malware detectors; and (iii) Through our use of SHAP, we have explained how these attacks are transferable across detectors. Exploring the latter constitutes our key line of future work. Through this research, we hope to contribute to the broader discussion on the importance of explainability-guided feature manipulation and obfuscation strategies.
of future work, as we believe this could prompt a new approach to defending against evasion attacks.

REFERENCES
[1] [n.d.]. AI and machine learning. https://www.avast.com/en-us/technology/ai-and-machine-learning
[2] [n.d.]. Androguard documentation. https://androguard.readthedocs.io/en/latest/
[3] [n.d.]. Antivirus and Internet Security. ESET. https://www.eset.com/
[4] [n.d.]. Avast. https://www.avast.com
[5] [n.d.]. Cyber security statistics. https://www.pulsesec.us/resources/cyber-security-statistics/ 
[6] [n.d.]. ELF Encypter. http://elfencyper.sourceforge.net
[7] [n.d.]. How anti-virus software works. https://cs.stanford.edu/people/eroberts/cs201/projects/viruses/anti-virus.html
[8] [n.d.]. Kaspersky cyber security solutions for home & business. https://www.kaspersky.com/au/
[9] [n.d.]. Kiteshield. https://github.com/GunshipPenguin/kiteshield
[10] [n.d.]. Linux Mint. https://linuxmint.com/
[11] [n.d.]. Microsoft defender antivirus. https://www.microsoft.com/en-au/windows/security/ 
[12] [n.d.]. Oracle virtualbox. https://www.virtualbox.org/
[13] [n.d.]. ReLoC bonus. https://github.com/nickcann/ReloCBonus
[14] [n.d.]. Repository - Linux Mint. http://packages.linuxmint.com/
[15] [n.d.]. Smali. https://github.com/JesusFreke/smali
[16] [n.d.]. Sonicwall research malware attacks 2019. https://www.msspartner.com/cybersecurity-research/sonicwall-research-malware-attacks-2019
[17] [n.d.]. UPX: The ultimate packer for executables. https://upx.github.io/
[18] [n.d.]. Oracle virtualbox. https://www.virtualbox.org/
[19] [n.d.]. VirusTotal. https://www.virustotal.com
[20] [n.d.]. VirusTotal. https://www.virustotal.com
[21] [n.d.]. VirusTotal. https://www.virusTotal.com
[22] [n.d.]. VMProject software protection. https://vmsoft.com
[23] [n.d.]. Martin Abadi, Andy Chu, Ian Goodfellow, H Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. 2016. Deep learning with differential privacy. In ACM SIGSAC Conference on Computer and Communications Security.
[24] [n.d.]. Hyrum S Anderson, Anant Kulkarni, Bobby Filar, David Evans, and Phil Roth. 2018. Learning to evade state PE machine learning malware models via reinforcement learning. arXiv preprint arXiv:1803.08917 (2018).
[25] [n.d.]. Simone Aonzo, Gabriel Claudiu Georgiu, Luca Verderame, and Alessio Merlo. 2020. Obfuscap: An open-source black-box obfuscation tool for Android apps. SoftwareX (2020).
[26] [n.d.]. Daniel Arp, Michael Spreitenbarth, Malte Hupner, Hugo Gascon, and Konrad Rieck. 2014. Drebin: Effective and explainable detection of Android malware in your pocket. In Network and Distributed System Security Symposium (NDSS).
[27] [n.d.]. Nicholas Carlini, Anish Athalye, Nicholas Papernot, Wieland Brendel, Jonas Rauber, Dimitris Tsipras, Ian Goodfellow, Aleksander Madry, and Alexey Kurakin. 2019. On evaluating adversarial robustness. arXiv preprint arXiv:1902.06705 (2019).
[28] [n.d.]. Nicholas Carlini and David Wagner. 2017. Towards evaluating the robustness of neural networks. In IEEE Symposium on Security and Privacy (SP).
[29] [n.d.]. Sen Chen, Minhui Xue, Lingling Fan, Lei Ma, and Lihua Xu. 2019. How can we craft large-scale Android malware? An automated poisoning attack. In International Workshop on Artificial Intelligence for Mobile (AI4Mobile).
[30] [n.d.]. Yuheng Chen, Shiqi Wang, Yue Qin, Xiaojing Liao, Suman Jana, and David Wagner. 2021. Learning security classifiers with verified global robustness properties. In ACM SIGSAC Conference on Computer and Communications Security (2021).
[31] [n.d.]. Luca Demetrio, Battista Biggio, Giovanni Lagorio, Fabio Roli, and Alessandro Armando. 2020. Efficient black-box optimization of adversarial windows malware with constrained manipulations. arXiv preprint arXiv:2003.13520 (2020).
[32] [n.d.]. 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 (2021).
[33] [n.d.]. Ambra Demontis, Marco Melis, Alina Oprea, Cristina Nita-Rotaru, and Fabio Roli. 2019. Why do adversarial attacks transfer? Explaining transferability of evasion and poisoning attacks. In USENIX Security Symposium (USENIX Security).
[34] [n.d.]. Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam D. Smith. 2006. Calibrating noise to sensitivity in private data analysis. In Theory of Cryptography; Third Theory of Cryptography Conference (TCC).
[35] [n.d.]. Kevin Eykholt, Ivan Evtimov, Earlene Fernandez, Bo Li, Amir Rahmati, Chao Wei Xiao, Atul Prakash, Tadayoshi Kohno, and Dawn Song. 2018. Robust physical-world attacks on deep learning visual classification. In IEEE Conference on Computer Vision and Pattern Recognition.
[36] [n.d.]. Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and harnessing adversarial examples. International Conference on Learning Representations (2015).
[37] [n.d.]. Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. 2016. Adversarial perturbations against deep neural networks for malware classification. arXiv preprint arXiv:1607.04435 (2016).
[38] [n.d.]. Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. 2017. Adversarial examples for malware detection. In European Symposium on Research in Computer Security.
[39] [n.d.]. Mahmoud Hammad, Joshua Garcia, and Sam Malek. 2018. A large-scale empirical study on the effects of code obfuscations on Android apps and anti-malware products. In International Conference on Software Engineering. 421–431.
[40] [n.d.]. Richard Harang and Ethan M. Ruddl. 2020. SOREL-20M: A large scale benchmark dataset for malicious PE detection. arXiv preprint arXiv:2012.07834 (2020).
[41] [n.d.]. Jinbo Jung, Chanil Jeon, Max Wolotsky, Insu Yun, and Taesoo Kim. 2017. AVPASS: Leaking and bypassing antivirus detection model automatically. In Black Hat USA Briefings. (Black Hat USA).
[42] [n.d.]. Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qweei Ye, and Tie-Yan Liu. 2017. LightGBM: A highly efficient gradient boosting decision tree. Advances in Neural Information Processing Systems (2017).
[43] [n.d.]. Tae-Guen Kim, Bo-Joon Kang, Min Hoo, Sukir Sezer, and Eul Gyu Im. 2018. A multimodal deep learning method for Android malware detection using various features. IEEE Transactions on Information Forensics and Security (2018).
[44] [n.d.]. Dhillung Kirat and Giovanni Vigna. 2015. MalGene: Automatic extraction of malware of future work, as we believe this could prompt a new approach to defending against evasion attacks.

Figure 7: SHAP values of two APK samples before and after adversarial generation. (a) and (b) are the original and adversarial samples of Sample 1; (c) and (d) are of Sample 2. Details of features can be found in Table 5 of Appendix Section A.
Then we rank the summary of delta values (e.g., CallIndirection, AssetEncryption, LibEncryption, Goto and ArithmeticBranch) of VirusTotal (2019). We randomly choose 20 Android seed samples that are scored above 30 by VirusTotal, and generate 4,096 combinations of obfuscation options making 4096 combinations of strategies. Rather than trying every permutation, we conduct a setup experiment to make the problem tractable (considering the resource limitations of VirusTotal).

We randomly choose 20 Android seed samples that are scored above 30 by VirusTotal, and generate 4,096 combinations of obfuscation options for each seed. After submitting 81,920 obfuscated samples to the VirusTotal, we calculate the score differences between the seed files and obfuscated files by option combinations. Then we rank the summary of delta values (e.g., average score dropping on VirusTotal) among different seeds. The result of this setup experiment is listed in Table 7.

According to the result of the top evasive combinations of strategies in Table 7, the combination of CFHJL (i.e., CallIndirection, AssetEncryption, LibEncryption, Goto and ArithmeticBranch) is the most effective one. All of the samples generated by this strategy evaded AV2 and AV3, and only one of the samples was detected by AV1.

In APK machine-learning model training, the feature vector involves a sequence of 1 and 0, representing whether a specific feature exists or not. To find how SHAP-guided features affect the evasion result, we conducted another experiment to compare the detection rate of manipulating 10, 25, 50, 75, 100, 125 and 150 features on 20 sets of 100 seed samples, 2,000 samples in total. The result is shown in Figure 8. By manipulating more features, the detection rate decreases. The turning point of the trending comes to 75 features with 5.1% adversarial samples detected; by contrast, the rates of 100 features comes only 2.1% lower than that of 75 features. Therefore, we choose N as 75 in Algorithm 1 for APK binaries.

### A Hyperparameters

This section introduces how we choose the most evasive obfuscation strategy and the amount of features manipulated in explainability-guided sample generation for APK binaries. Obfuscapck provides 12 obfuscation options making 4096 combinations of strategies. Rather than trying every permutation, we conduct a setup experiment to make the problem tractable (considering the resource limitations of VirusTotal).

We randomly choose 20 Android seed samples that are scored above 30 by VirusTotal, and generate 4,096 combinations of obfuscation options for each seed. After submitting 81,920 obfuscated samples to the VirusTotal, we calculate the score differences between the seed files and obfuscated files by option combinations. Then we rank the summary of delta values (e.g., average score dropping on VirusTotal) among different seeds. The result of this setup experiment is listed in Table 7.

According to the result of the top evasive combinations of strategies in Table 7, the combination of CFHJL (i.e., CallIndirection, AssetEncryption, LibEncryption, Goto and ArithmeticBranch) is the most effective one. All of the samples generated by this strategy evaded AV2 and AV3, and only one of the samples was detected by AV1.

In APK machine-learning model training, the feature vector involves a sequence of 1 and 0, representing whether a specific feature exists or not. To find how SHAP-guided features affect the evasion result, we conducted another experiment to compare the detection rate of manipulating 10, 25, 50, 75, 100, 125 and 150 features on 20 sets of 100 seed samples, 2,000 samples in total. The result is shown in Figure 8. By manipulating more features, the detection rate decreases. The turning point of the trending comes to 75 features with 5.1% adversarial samples detected; by contrast, the rates of 100 features comes only 2.1% lower than that of 75 features. Therefore, we choose N as 75 in Algorithm 1 for APK binaries.

### B Structures of WinPE and ELF Binaries

Figure 9(a) illustrates the structure of WinPE binary. Specifically, a WinPE binary involves five parts:

- **DOS Header** is a legacy header from DOS era to maintain compatibility with legacy Windows systems.
- **PE Header** involves general information including the target architecture, number of sections and symbols, timestamp, and the header size.

| Type   | Strategy          | Index |
|--------|-------------------|-------|
| IS     | AdvancedReflection| A     |
|        | Reflection         | D     |
| CFGA   | CallIndirection    | C     |
|        | Reorder            | E     |
| ENC    | AssetEncryption    | F     |
|        | ConstStringEncryption| G    |
|        | LibEncryption      | H     |
|        | ResStringEncryption| I     |
| DCI    | Goto              | J     |
|        | Nop               | K     |
|        | ArithmeticBranch  | L     |

Figure 8: Detection rates of manipulating different sizes of feature maps on APK.
Figure 10: Manipulable raw features extracted from an ELF binary.

Table 7: Setup result of obfuscation strategies. Each strategy combination is applied to 20 seed files.

| Strategies  | Detection Rate |
|-------------|----------------|
|             | AV1  | AV2  | AV3  | VT  |
| Original    | 100% | 100% | 100% | -   |
| ABFG        | 45.0%| 60.0%| 45.0%| 7   |
| CFHJL       | 5.0% | 15.0%| 5.0% | 10  |
| CHL         | 15.0%| 30.0%| 30.0%| 8   |
| BHJK        | 15.0%| 30.0%| 30.0%| 8   |

1 Referring to Table 6
2 Average detection rate of 64 engines on VirusTotal.

• Optional Header contains detailed information required by the system to load, such as the entry-point address, dll characteristics, size of file, and version information.

• Section Table is a list of section headers storing the section name, address, relocations, and other general information.

• Sections contains contiguous chunks of bytes hosting the real content of the executable. For example, .text stores the code, .data stores global variables, and .rdata stores read-only constants and counting.

Figure 9: Structure of WinPE, APK and ELF files.

Figure 9(b) illustrates the structure of ELF binary. An ELF file has two views: the program (segment) header indicates the segments used at run-time, whereas the section header lists the set of sections of the binary. Each ELF file comprises one ELF header, followed by file data which includes:

• Program header table, describing zero or more memory segments.

• Section header table, describing zero or more sections.

• Sections referred to by entries in the program header table or section header table. Similar to WinPE, .text, .data, and .rdata store the code, global variables, and read-only constants, separately.

Not all features in an ELF file can be modified directly. Figure 10 shows the raw features that can be modified directly or indirectly.