



# Bypassing NGAV for Fun and Profit

Ishai Rosenberg and Shai Meir

November 30, 2020

Deep Instinct Ltd  
{ishair, shaim}@deepinstinct.com

## Abstract

In this paper, we demonstrate the first methodological approach to "reverse engineer" a NGAV model and features without reversing the product, and generate a PE malware that bypasses next generation anti-virus (NGAV) products.

Previous such attacks against such machine learning based malware classifiers only add new features and do not modify existing features to avoid harming the modified malware executable's functionality, making such executables easier to detect.


In contrast, we split the adversarial example generation task into two parts: First, we find the importance of all features for a specific sample using explainability algorithms. Then, we conduct a feature-specific modification (e.g., checksums, timestamp, IAT, etc.), feature-by-feature.

In order to apply our attack to NGAV with unknown classifier architecture, we leverage the concept of transferability, i.e., different classifiers using different features subsets and trained on different datasets still have similar subset of important features. Using this concept, we attack a publicly available classifier and generate malware PE files that evade not only that classifier, but also commercial NGAV.

We also demonstrate additional techniques, such as the sliding window approach to understand the most important features in the attacked classifier.

## 1 Introduction

In this paper, we demonstrate the ability of an adversary to bypass commercial next-generation anti virus (NGAV) products using explainability of multi-feature types malware classifiers algorithms to produce the most important features for a known, white-box model, instead of reversing the attacked NGAV. Then, due to the concept of transferable explanations, the same important features are relevant to the target NGAV, whom used features are unknown.



Therefore, by modifying existing malware's important features by the white-box model's explanation to fool it, not only the white-box model would be fooled, but also the target NGAV.

In the cyber security domain there is a unique challenge which is not addressed by previous research: Malware classifiers (which gets an executable file as input and predict the labels of benign or malicious for each file) often use more than a single feature type (see Section 1.1.2). Thus, adversaries who want to subvert those systems should consider modifying more than a single feature type. Some feature types are easier to modify without harming the executable functionality than others (see Section 1.1.2). In addition, even the same feature type might be modified differently depending on the sample format. For instance, modifying a printable string inside a PE file might be more challenging than modifying a word within the content of an email, although the feature type is the same. This means that we should not only take into account the impact of a feature on the prediction, but also the difficulty of modifying this feature type [1]. In addition, some features are dependent on other features, meaning that modifying one feature requires modifying other features for the executable to continue functioning. For instance, modifying the `AddressOfEntryPoint` requires modifying the `VirtualAddress` of the code section, where `AddressOfEntryPoint` exists.

The end result is that when adversaries want to modify a PE file without harming its functionality, the feature modification must be done manually. In this way, only features that are easy to modify, not dependent on other features who are challenging to modify (which is feature specific) would be modified. Thus, we would want to modify the smallest numbers of features, because each feature's modification requires a manual effort. Moreover, each modified feature can create a feature distribution anomaly that could be detected by anomaly detection algorithms (e.g., [2]). Therefore, the adversary aims to modify as little features as possible, even if he/she could modify all features automatically. In order to achieve this goal, the adversary would like to get the list of most impactful features for a specific sample (the malware which tries to bypass the malware classifier) and manually select the features that are the easiest to modify. This is the approach we take in this paper, as opposed to previous adversarial attacks, that try to do both in the same algorithm and thus try to modify all features in the same manner, resulting in generating malware executables that don't run.

In order to select the most indicative features for a sample, the adversary can use explainability algorithms. However, the adversary is not familiar with the architecture of the target malware classifier. In order to resolve this issue, adversaries can train their own malware classifier and use its features instead. Those features are likely to have a high impact even in the target classifier, due to the concept of transferability of explainability.



## 1.1 The Challenges in End-To-End Adversarial Examples of Malware Executables

Most published adversarial attacks, including those that were published at academic cyber security conferences have focused on the computer vision domain, e.g., generating a cat image that would be classified as a dog by the classifier. However, the cyber security domain – and particularly the malware detection task - seems a much more relevant domain for adversarial attacks, because unlike the computer vision domain, the cyber security domain is an “inherently adversarial” domain: there are actual adversaries with clear targeted goals. Examples include ransomware developers who depend on the ability of their ransomware to evade anti-malware products that would prevent both its execution and the developers from collecting the ransom money, and other types of malware that need to steal user information (e.g., keyloggers), spread across the network (worms) or perform any other malicious functionality while remaining undetected.

Given the obvious relevance of the cyber security domain to adversarial attacks, why do most adversarial learning researchers focus on computer vision? Besides the fact that image recognition is a popular machine learning research topic, another major reason is that performing an end-to-end adversarial attack in the cyber security domain is more difficult than performing such an attack in the computer vision domain. The differences between adversarial attacks performed in those two domains and the challenges that arise in the cyber security domain are discussed in the subsections that follow.


### 1.1.1 Executable (Malicious) Functionality

Any adversarial executable must preserve its malicious functionality after the sample’s modification. This might be the main difference between the image classification and malware detection domains, and pose the greatest challenge. In the image recognition domain, the adversary can change every pixel’s color (to a different valid color) without creating an “invalid picture” as part of the attack. However, in the cyber security domain, modifying an API call or arbitrary executable’s content byte value might cause the modified executable to perform a different functionality (e.g., modifying a `WriteFile()` call to `ReadFile()` ) or even crash (if you change an arbitrary byte in an opcode to an invalid opcode that would cause an exception).

In order to address this challenge, adversaries in the cyber security domain must implement their own methods (which are usually feature-specific) to modify features in a way that will not break the functionality of the executable. For instance, the adversarial attack used in Rosenberg et al. [3] modifies API call traces in a functionality preserving manner.

### 1.1.2 There are Many Feature Types

In the cyber security domain, classifiers usually use more than a single feature type as input (e.g., malware detection using both PE header metadata and byte



entropy in Saxe et al. [4]). Some feature types are easier to modify without harming the executable functionality than others. For instance, in the adversarial attack used in Rosenberg et al. [3], appending printable strings to the end of file is much easier than adding API calls using a dedicated framework built for this purpose. In contrast, in an image adversarial attack, modifying each pixel has the same level of difficulty.

### 1.1.3 Executables are More Complex than Images

An image used as input to an image classifier (usually a convolutional neural network, CNN) is represented as a fixed size matrix of pixel colors. If the actual image has different dimensions than the input matrix, the picture will usually be resized, clipped, or padded to fit the dimension limits.

An executable, on the other hand, has a variable length: executables can range in size from several KB to several GB. It's also unreasonable to expect a clipped executable to keep its original classification. Let's assume we have a 100MB benign executable into which we inject a shellcode at a function near the end-of-file. If the shellcode is clipped in order to fit the malware classifier's dimensions, there is no reason that the file would be classified as malicious, because its benign variant would be clipped to the exact same form.

In addition, the code execution path of an executable may depend on the input, and thus, the adversarial perturbation should support any possible input that the malware may encounter when executed in the target machine.

While this is a challenge for malware classifier implementation, it also affects adversarial attacks against malware classifiers. Attacks in which you have a fixed input dimension, (e.g., a 28\*28 matrix for MNIST images), are much easier to implement than attacks in which you need to consider the variable file size.

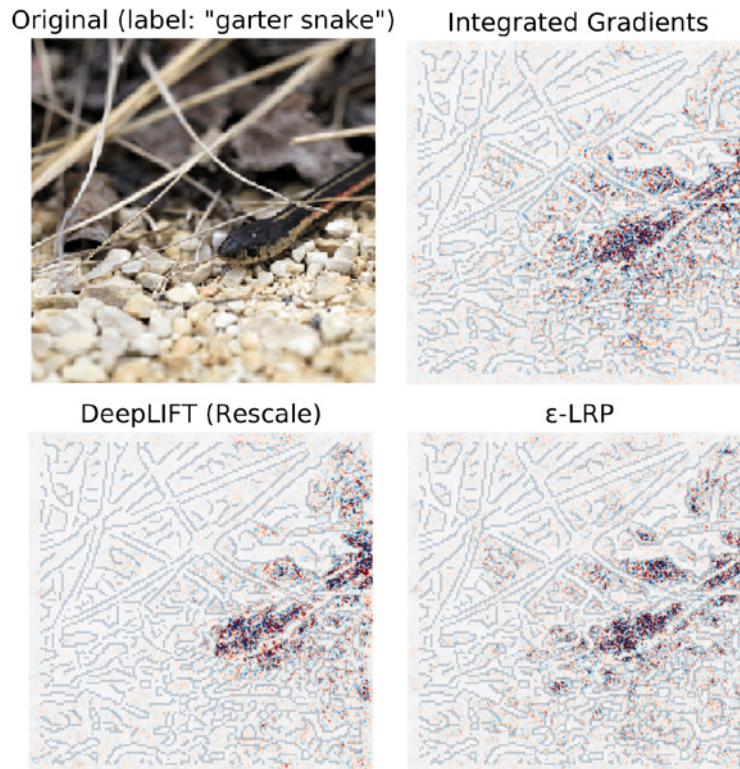


Figure 1: Explainability Algorithms in the Image Recognition Domain (Taken from [5])

## 1.2 Paper’s Contribution

The main contributions of this paper are as follows:

1. This is the first attack that eliviates the need to reverse the attacked NGAV (as done in [6]) or to be familiar with its features ([7]). Our attack only requires a non-empty subset of modifiable features common to the attacked NGAV and the surrogate model.
2. This is the first paper to discuss the usage of explainability by adversaries, as well as our novel technique, called sliding window attack, in order to both choose specific features to perturb and maintain the minimal perturbation (that is, number of perturbed features).

## 2 High-Level Attack Overview

### 2.1 Threat Model

The adversary’s goal is to modify a malware executable for it to bypass a multi-feature types malware classifier without harming the executable’s functionality (Section 1.1.1), that is, generating an end-to-end malware adversarial example. In this paper we limit ourselves to static features, that is, features that can be extracted from the file without running it. Static features can be raw features, e.g., the malware file’s binary content. They can also be properties (e.g., the PE file’s section names) parsed from the PE, termed PE structural features. Raw-byte features require a very long training process and current state-of-the-art GPU hardware usually limits the file size which can be classified using such classifier (e.g. [8]), making this a non realistic use case. Using dynamic features (e.g., executed API calls in Rosenberg et al. [3]) is also less common use case in real-life scenarios, since it requires a sandbox environment in order to avoid running a malware on the computer we want to protect, which might harm it. We therefore decided to focus on PE structural features, which are used by real-world classifiers. We assume the adversary has no knowledge or access to the attacked malware classifier, e.g., the classifier type, architecture or training set - only to the confidence score of the attacked classifier (a gray-box attack). We do assume the adversary can figure out **some** of the features used by the attacked malware classifier, but not all of them. This is a common case in cyber security, especially with static features, where many classifiers are using similar PE structural features, (e.g., [4, 7, 9]) but the exact subset of features is unknown.

### 2.2 Attack Overview

In order to evade detection by the malware classifier, the adversary is using the method depicted in Figure 2.



Figure 2: The Proposed Attack Flow

This method is detailed in Algorithm 1.

In this method, we use the following definition: An explainable machine learning algorithm  $A(m, \mathbf{v})$  takes as arguments a machine learning model  $m$  and a sample’s vector  $\mathbf{v}$  and returns a vector of  $length(\mathbf{v})$  values which represent the weights of impact of the features in  $\mathbf{v}$ , such that a higher weight indicates a more impactful feature for classifying the vector  $\mathbf{v}$  by the model  $m$ . An example of the usage of explainability can be seen in Figure 1. We see that three different explainable algorithms (Integrated Gradients [10], DeepLIFT [11] and Layer-

---

**Algorithm 1** End-to-End PE Structural Features-Based Adversarial Example Generation

---

1. Use explainable machine learning algorithm (Section 3.2.1) to get a list of features importance for the classification of the malware on the surrogate model (see Section 2.3) and fine-tune this list using the sliding window attack (Section 3.2.2).
  2. Perform an analysis on a data set and features proven to accurately represent the attacked malware classifier (Section 3.3) by the previous step.
  3. From those features, choose those easier to modify (Section 3.4).
  4. Modify each “easily modifiable” feature using the list of predefined feature values (Section 3.4), selecting the value that result in the lowest confidence score. Repeat bullets 2 and 3 until a benign classification is achieved by the attacked malware classifier.
- 


wise Relevance Propagation (LRP) [12]) highlight similar features as the most important to classify a gartner snake image (mainly, pixels in the snake’s head area).

This method is relying on two assumptions, evaluated in the following subsections:

- (1) The most important features in the attacked malware classifier would be similar to those of the surrogate model, and they would also be found by an explainability algorithm. Thus, modifying these features in the method mentioned above would affect the attacked malware classifier as well. Detailed in Section 2.3, and (2) The adversary can modify the malware binary without harming its functionality. Detailed in Section 3.4.

## 2.3 Transferability of Explainability

The concept of transferability of explainability is defined as follows: Given two different models,  $m_1$  and  $m_2$  with different classifier type and architecture trained on a similar dataset and input features list, the output of an explainable machine learning algorithm (see Definition 2.2) would be similar for  $m_1$  and  $m_2$ . Transferability of explainability means that the feature group indicated to have a high impact on a specific sample classification on one model would be similar to the list of the same explainability algorithm on another model. We argue that this holds true regardless of the classifier type, architecture, training set or even explainable algorithm. The only requirement is that the features used by both classifiers need to be similar enough (otherwise impactful features in one model are meaningless in the other model) - but not identical. An example of the usage of transferability of explainability can be found in Rosenberg et al. [13], where the authors used one classifier (a deep neural network) as a surrogate model to attack another classifier (a gradient boosted decision tree).



This concept is especially important for multi-feature types malware classifiers: On the one hand, the adversary is unaware of the attacked classifier architecture, so using transferability is essential. On the other hand, modifying too many features might cause the adversarial example to be caught by anomaly detectors (e.g., [2]). Therefore, a small perturbation (that is, modifying a small amount of features) is desired.

## 3 Low-Level Attack Implementation

### 3.1 Dataset and Classifiers

We used the Ember dataset. It is thoroughly described in [9], and is the state-of-the-art dataset of 1M malware and benign-ware, equally distributed. We split the dataset into a training-set of 300K malware and 300K benignware and a test set of 200K malware and 200K benignware.

As a surrogate model, trained by the adversary, we used the gradient boosted decision tree (GBDT) classifier used in [9], which outperformed state-of-the-art raw features model [8]. This classifier input is a vector of 2381 Ember’s PE structural features and its output is a binary classification: malicious or benign file. It is trained using **LightGBM** with 100 trees and 31 leaves per tree.

As the target classifier, we will use a commercial NGAV, which, in this paper, we will call  $NGAV_1$ .

### 3.2 Feature Explainability

#### 3.2.1 Feature Explainability Using SHAP

We wish to find the features with the highest impact on our target classifier and change them. Since the target classifier is not exposed to us and we can only arbitrarily query for the confidence score, e.g., a gray-box attack. In our approach we are using a surrogate model Ember [9] trained on independently chosen dataset of malicious and benign PE samples.

In our attack we decided to leverage SHAP [14] in order to extract the features that are the most significant in the model for the specific sample classification. Note that by leveraging SHAP we can query the surrogate model for feature importance on a specific sample which may differ from feature importance of another sample. This helps us in prioritizing and therefor optimizing the attack we conduct on the features.

In Figure 3 we can see the first 15 most important features (of the surrogate model) according to SHAP for the specific sample we want to evade detection. A main challenge is what features can we perturbate/modify without harming the PE functionality? We split the feature types by the difficulty of controlling each feature. We highlighted in green features that we can entirely control with little effort and have no implication on the PE functionality. Orange



describes features we can partially control; at times adding can be achieved with little effort, but removing or modifying may require greater effort. In purple we highlighted features that we have indirect control of. The best example are character distribution features, where inserting new characters to the PE eventually changes the entire character distribution of the PE file. It is worth noticing that usually it is possible for an attacker to control character distribution features of the entire PE by appending a corrective distribution buffer to the overlay of the file or even insert corrective distribution buffer to one or more new sections. However, the latter may be more difficult to achieve due to additional corrections that are required for the PE file to function afterwards. Finally, the red highlight describes either features that are impossible to modify or that they require significant effort beyond the scope of this work.

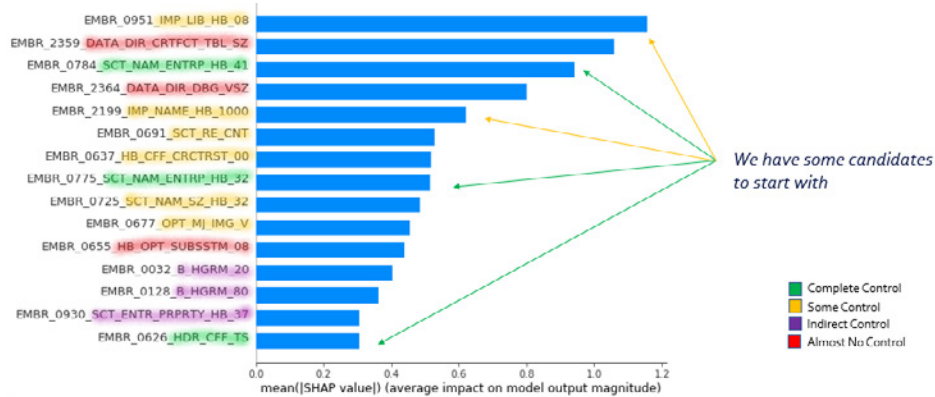



Figure 3: SHAP’s feature importance of the attacked PE sample and the surrogate model

### 3.2.2 Sliding Window Attack

In the sliding window attack, we are attacking the target classifier by deliberately modifying parts (windows) of the PE file. In this way we can see how the score changes when we hide specific parts of the original PE file. We have developed two major methods of modifying the window:

- Completely zero out the window
- Only change the strings found in the window

We can also control the type of string obfuscation, e.g., a simple scramble, xor with a known key, replace with specific characters etc. For string detection we used the following regular expression: “( $\backslash w \backslash . \{ 5, \}$ )” and similar ones for wide-character strings and Unicode strings. The size of the window could be



controlled and adapted to the size of the file that we are attacking, therefore we could limit the maximal number of attacks (queries) per file.

Once a significant change in the score is detected we investigate in which part of the file it occurred. For instance: code section, overlay, import table etc. We also apply a divide-and-conquer approach for the specific window, to reduce and pinpoint the exact location. For example, if the original window size was 64KBytes, we would divide it into two separate windows of 32KBytes and apply the technique on both windows, continuing this process until we are either satisfied with the discovery or reached a minimal window size, in our case: 512 bytes.

Since the divide-and-conquer approach was applied only to parts of the file with a high impact on the attacked classifier's score, the entire process is quite fast. Furthermore, this technique can be done in parallel and improved further if required. There are however, two major drawbacks for this technique:

- It is difficult to apply to the PE header (for a classifier that parses the PE header)
- The findings are not always straight forward

Attacking the PE header requires a specific implementation, in order to create a “fuzzer”-like attack method. This will no longer be a sliding window type of attack, due to the need of alternating each field specifically. The PE header itself contains hundreds of fields, where usually each one becomes a feature, either directly or through a hashing technique. We focus only on modifying specific features in the PE header and avoid implementing a generic fuzzer, due to the total number of feature value combinations required and their mutual effect on each other.

As we will show, it is possible to considerably limit the number of possible values per feature, for example, for the COFF File Header TimeDateStamp it is sufficient to test 180 values (out of a possible  $2^{32}$  values) if we allow only values up to 15 years ago and splitting the interval for 12 months per year, thus  $12 * 15 = 180$  different values. This has proved to be sufficient and at times could change the score by 0.7 and even higher values in the  $[-1, 1]$  score interval of the target classifier.

Another challenge in fuzzing a PE header is that some attacks may result in a non-functioning PE file. Other parts of the file are not trivially modifiable. For example, detecting some part of the code section, overlay, resource section etc. Although one can devise a solution for each such challenge, it might be very difficult to achieve without resorting into packing the entire PE. For these reasons we leave the PE header fuzzer for future work.

All that said, by using such a technique, we do get a significant understanding of what the classifier deems as important in the specific sample. In Figure 4 we show an illustration of the Sliding Window Attack.

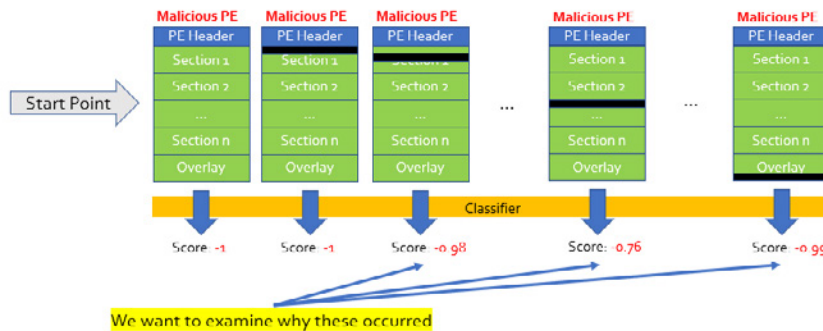


Figure 4: An illustration of the sliding window attack. The black square represents the progress of the window across the file.

In figure 5 we can see the small effect the sliding window had when “passing over” the imports section and how it also agrees with SHAP’s feature importance.

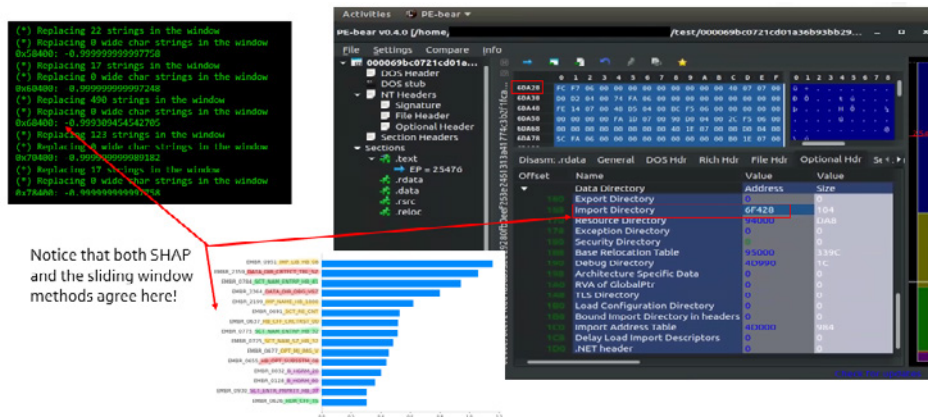


Figure 5: A screenshot of the sliding window attack, using 32KBytes window size and string obfuscation only.

### 3.3 Dataset Imports Analysis

One of the feature types we’re going to modify is the imports, which are both easy to add and have big effect on the classifier’s prediction (see Figure 3). We rely on the fact that many static analysis engines are eventually forced to

extract information from such features and we will utilize this list when we attack the import table. Interesting insights regarding imports are related to the PE structure and the underlying OS.

In order to maximize the efficiency of the imports attack we extracted the imports list of 21K PE files with malicious and benign labels split 50/50. Although one can obtain a large amount of samples, from many online sources, s.a., VirusShare, we consider Ember dataset [9], released in 2018, to be the preferred choice, since this dataset is highly reliable with respect to the labels.

The next step is to create two (lib\_name:import\_name, frequency) tuple lists, one for the malicious samples and one for the benign samples. The idea behind such a list is to offer a very simplified statistical representation of the impact a specific import existence might have on the model. From both lists we produce a single percentage difference list with a score for each import library-function pair ranging from 1 to -1, where 1 means it is exclusively found in the benign samples and -1 means it is exclusively found in the malicious samples, as shown in Figure 6.

One should note that the appearance of a library-function pair in the import table is only a “circumstantial evidence”, as the use of the function cannot be enforced or predicted in any way. An example is a false condition wrapping a function call. On the other hand, all used and necessary library-function pairs do not have to appear in the import table since they can also be loaded dynamically, using the string representation of the library-function call pair to request the address. This property is also reflected in Figure 6, where the API used to load library-function addresses dynamically, namely LoadLibraryA, is more common in malicious files than in benign files.

Library-Function Pair	Percentage Diff
msvcrt.dll:free	0.239202558
msvcrt.dll:malloc	0.238521566
msvcrt.dll:_initterm	0.217461439
kernel32.dll:LoadLibraryA	-0.264094599
kernel32.dll:GetModuleHandleA	-0.269134553
kernel32.dll:ExitProcess	-0.368820338

Figure 6: In this example we listed the top three import library-function associated with benign population and the top three import library-function pairs associated with malicious population (negative values). It is important to understand that the library-function pairs are not malicious and the numbers simply represent how many observations we had of that pair in each population. These are widely used APIs of the Win32API library.

### 3.4 Selecting Modifiable Features

In this section we will list the features that we chose to manipulate in the PE file with a short description of each feature.


Table 1: Adversarial Examples Success Rate and Average Number of Modified Features

Property	Description
Checksum	Has no impact on the functionality unless it is a driver or a critical dll ( <b>PE spec</b> )
TimeDateStamp	Has no impact on the functionality
New Sections	Inserting new section with different characteristics and predetermined entropies or sections extracted from benign files. Should be done carefully – usually possible
Entry Point Trampoline	Existing code section if enough slack space found otherwise in a new section
New Imports	Choose wisely from the list we established before
Rename Sections	Hold a list of section names mostly found in benign files
And more	Linker version, Min/Maj OS version - <b>TinyPE</b> is a good source for ideas

We chose to modify features listed in table 1. It is important to note that as we mentioned in Section 3.3, “circumstantial evidence” with respect to imported functions, the same also holds for other feature types of the PE file such as the ones listed here. In other words, these features have little to no contribution to the actual purpose of the underlying PE file activity but their distribution is learned by classifiers therefor enabling attacker to have some degree of control on the score of the classifier. It is extremely difficult, and arguably impossible, to accurately and correctly describe the functionality of a PE file by means of static analysis, let alone with the runtime constraints, and we rely on this fact to aid us in changing the initial classifier’s prediction for the target PE file from malicious to benign. Below we will discuss the type of modification each feature has undergone.

#### 3.4.1 Checksum

The modification of this feature is straight forward: Once all changes were complete and a new PE file was built (with **LIEF**) its checksum was recalculated and both versions of the PE were tested. The one with the lower score was discarded (our score range is  $[-1, 1]$ ). Since the feature was at times



very dominant it was necessary to recalculate it before moving on to the next attack phase (feature modification). Interestingly enough, while this feature is only checked for drivers and critical dlls, it has significant impact on non critical PE files in the target classifier.

### 3.4.2 TimeDateStamp

Attacking this field seems as a major challenge since, being 4 bytes long, it has many possible values. Thus, we decided to limit the modified values to not go further than 15 years ago. We calculated how many seconds are in 15 years and divided that by 180, resulting in approximately a TimeDateStamp for every month in the past 15 years ( $12 * 15$ ). This proved to be very a effective approach as we expected the classifier to produce a lower score for files older than 15 years. The exact threshold (instead of 15 years) can be determined automatically and we leave that for future work.

### 3.4.3 New Sections

New sections were added to the file with different attributes such as:

- IMAGE\_SCN\_CNT\_CODE - The section contains executable code
- IMAGE\_SCN\_MEM\_WRITE - The section can be written to.
- IMAGE\_SCN\_MEM\_READ - The section can be read.
- IMAGE\_SCN\_MEM\_EXECUTE - The section can be executed as code.
- IMAGE\_SCN\_CNT\_UNINITIALIZED\_DATA - The section contains uninitialized data.

For each section we added we can control the entropy that will be calculated for it by choosing the character buffer stream that will be incorporated in the actual section. Sections were 64KB long. For high entropy section we would simply choose a random character until we reached the section size. For sections with low entropy, we would first choose one character at random out of the set  $[0x00, 0xCC, 0xFF]$  and use it to adjust the entropy as we build the buffer. If the entropy was higher than the requested entropy we would insert the pre-chosen character otherwise a random character. In addition we also added the feature of inserting binary blobs extracted from other files as the content of the new section. More on that in Section 3.4.4. In our experiments, this type of modification had little “positive” impact on the classifier’s score.

### 3.4.4 Entry Point Trampoline

The trampoline we chose to insert was “pasted” right before a rip of a code section containing the entry point of a very well known benign file from the

OS vendor. The rip was of size 64KBytes and the trampoline length was deducted from the tail of it. The code for the trampoline was standard “GoTo OEP” (Original Entry Point) but we chose to use a dynamic always false condition that will branch to both the ripped code section and the OEP. Below is an example of the template.

---

**Algorithm 2** Trampoline to OEP

---

```
mov edi, edi ; 0x8BFF
push ebp ; 0x55
mov ebp, esp ; 0x8BEC
push ecx ; 0x51
mov ecx, 0 ; 0xB900000000
xor eax, ecx ; 0x33C9
pop ecx ; 0x59
pop ebp ; 0x5D
push OEP ; 0x68[OEP]
pop eax, ; 0x58
jnz $+2 ; 0x7502
jmp eax ; 0xFFE0
[Paste of ripped benign entry point]
```

---

We are aware that this was a blunt use of a ripped binary code but we are also aware that our goal was to evade a classifier and not a reverse engineer or analyst. It will be very apparent for a reverse engineer for example that the RVAs used in the blob are incorrect but less so for a classifier as it usually relies on extracting statistical features. Even more so, many feature extraction techniques from the code section actually normalize the assembly code, register names, RVAs, VAs, immediates and so on, sometime leaving only the normalized opcodes or a statistical representation of the opcodes (average, standard deviation, etc...). There are two major obstacles for extracting robust features from the code section:

1. Operand noise - Noise that is created when the same compiler setup (version, flags) produces new code and certain decisions, s.a. register allocation, may change due to small changes in the code that will result in different register allocations. In a sense it is a subset of the next item.
2. Compiler noise - Major changes that occur to the generated code due to use of different compiler flags, optimizations and compiler vendors.

Therefore, it is sometimes necessary to ignore the “noise” in the data in order to make the malware classifier more robust to compiled variants, etc. However, this leaves a way for adversaries to affect the more statistical features with a suitable weapon, for example, in the form of code blobs extracted from highly benign files usually from the OS vendor.



### 3.4.5 New Imports

Inserting new imports (and therefor sometime also new library dependency) to a PE file can easily result in a non-functioning PE file. The imports we chose to insert were selected from the previously prepared list of imports (see Section 3.3). The total size of the list was more than 100K (imports) library function pairs and we had three methods of **selecting** the next import to insert:

1. Go over the list and insert an import one by one, if the score did not improve the import was discarded.
2. Insert imports in batches of size  $n$ .
3. Insert imports in batches of size  $n$  out of a total of  $m$  where  $n < m$ . If a specific batch was not improving the score it was discarded. In this way we could control granularity better and avoid imports that have negative affect on each other.

As for **inserting** the imports list into the PE, we had four main strategies:


1. Insert all selected imports to a new section. This method often resulted in a non-functioning PE due to missing dependencies.
2. Insert all imported function from libraries already imported by the PE and the rest to be dumped as strings into a new section or the overlay. This also had a small chance to generate a non-functioning PE file, mostly because of new functions that are present in newer versions of the dlls that may not be present in the target operating system environment.
3. Insert all import as null terminated strings to the overlay of the file. The overlay is the byte offset beyond the last byte of highest section in the PE.
4. Insert all imports as null terminated strings to a new (data) section which is not used.

This type of attack was very successful and eventually resolved into appending the strings only without modifying the import table of the PE. The success of this stage is attributed to the inability to determine by means of static analysis if there is any part of the code that tries to resolve those strings into an actual function address and use it.

### 3.4.6 Rename Sections

In this method we kept a list of section names commonly used in benign files. The list can be obtained in a similar approach from the bank of malicious and benign files used for the imports (see Section 3.3). However, in the specific case we present there was no gain in altering the section names, as the existing section names were all present in our preferred benign section names list.





### 3.4.7 Additional Perturbations

There is an abundance of ideas and additional features that can be altered. To name some examples: Linker version, Min/Maj OS version, certain flags in PE header characteristics, alter SizeOfHeaders, and more. **TinyPE** contains many additional ideas that can be evaluated in future work.

## 4 Bypassing a Commercial NGAV: A Concrete Example

### 4.1 The Perturbed Malware

We used a very known malware, with the SHA-256 value:

0x000069bc0721cd01a36b93bb29280fb0eef263e2461313a41774c3b2f1fca7d9.

It is being detected by about 60 security products in VirusTotal, and is identified by Microsoft as: 'TrojanDownloader:Win32/Silcon!rfn'. Additional details of this malware can be found in VirusTotal at:

<https://www.virustotal.com/gui/file/000069bc0721cd01a36b93bb29280fb0eef263e2461313a41774c3b2f1fca7d9>

### 4.2 Attack Analysis for

We will describe the steps and results of our attack on the target classifier in Table 2. Notice that we avoided using steps that harm the functionality of the original PE, e.g., adding a new import section. The same holds for adding the actual functions to the import table: we may be able to finish the process quicker or with fewer steps, but the end result may not be functional anymore.

Table 2: Attack stages for the perturbed malware sample


Step	Change in Score	Classifier Score	Classification	File Size
-	-	-0.999999999997758	Malicious	598KB
Insert 10k import names to a new section + Checksum correction	0.001013504302047	-0.998986495695711	Malicious	
TimeStamp attack + Checksum correction	0.003757372367677	-0.995229123328034	Malicious	
Trampoline in new section + 10k imports into overlay (same as before in this case)	0.519168000280035	-0.476061123047999	Malicious	
20k imports into overlay, this time in 1k batches, dropping batches that do not improve score	1.280170733720289	0.80410961067229	Benign	
Timestamp + Checksum	0.012015640716198	0.816125251388488	Benign	1.41MB

We started with a score near to a perfect -1 and a precision of  $10^{-11}$ .

First we discuss the checksum and TimeDateStamp attacks. Whenever we modify and build a new PE we test it against the target classifier and in addition we again attack the TimeDateStamp feature and perform a checksum correction. These are two attacks that are relatively easy to perform and if they do not improve the score we ignore them and continue with the previous PE build.

The next step we took was to insert 10K import strings composed of the library name followed by its function names, as null terminated strings, to a new data section in the file. The idea behind this was to have a significant impact in terms of order of magnitude precision. After fixing the checksum of the resulting PE file the score changed to  $-0.99898\dots$ . This is  $10^{-2}$  precision as opposed to the previous  $10^{-11}$ . In plain terms, this tells us that once before, our target classifier  $NGAV_1$  showed high confidence regarding the samples class, and now it is orders of magnitude less confident.

Next we perform an attack on the TimeDateStamp feature and correct the checksum. Since this is done in every stage as described before we will not



elaborate on each occasion it was performed. This attack resulted in a 0.0037 improvement over the previous score.

The next step of the attack is to introduce a trampoline (in a new code sections in this case) to the OEP, and append the same 10K imports from before but this time to the overlay. The added trampoline contributed only 0.1 to the score but the import strings appended to the overlay added an additional 0.42 improvement to the score, together totaling more than a 0.5 change.

For the next attack we continued our imports approach, but this time we used a batch size of 1K from a subset of 20K out of the total 120K import pairs extracted from the files. In each iteration, the batch was discarded if there was no improvement to the score. We required 5 additional 1K batches to be introduced into the overlay, where the final batch of 1K import function pairs produced a whopping 0.8 jump in the score. The total change in score for this step was  $> 1.28$ . Essentially, we can conclude the attack at this point since the class of the sample was changed while we preserve the functionality. Nevertheless we conducted an additional TimeDateStamp attack with checksum fixing to gain an additional 0.012 to the score.

### 4.3 Practical Tips for Adversarial Learning Practitioners

Although there are probably more than one way to achieve a successful attack on the  $NGAV_1$  we've found that at times the order of operation (or modification) mattered.


For instance, applying the trampoline before inserting a new section with import strings, resulted in a very different flow and scores. This might be due to the order of the sections, their addresses or any other feature value the classifier created to describe the "meta-difference" between the two orders of operations. This added "value" is irrelevant to the underlying and final functionality of the built PE.

At other times, achieving an order-of-magnitude difference is very hard, but in most times, once it is achieved it becomes easier to achieve greater and greater impacts on the classifier's score (after you got the snowball rolling..).

Note that such an attack is significantly simpler for an adversary that holds the source code of the perturbed malware. This will remove a significant overhead in such an attack and may also allow perturbation of features more difficult to accomplish without the source code. However, this assumption would obviously makes our attack less usable in real-life scenario, so we avoided using it.

### 4.4 Our Attack Results

Finally, here we present the results as captured from [VirusTotal website](#). Before the attack, our malware is detected by 57 of the anti malware products



in VirusTotal (Figure 7). After the attack we uploaded the modified PE to VirusTotal again and it was detected only by 30 anti malware products (Figure 8). Thus, besides  $NGAV_1$ , which was clearly bypassed, our attack also bypassed a significant amount of other AV products. Among those, there are additional NGAV vendors that were not directly targeted here. This demonstrate the transferability of explainability property, mentioned in Section 2.3. We start by results on the original sample followed by the results of the perturbed sample.

57 / 68

57 engines detected this file

000069bc0721cd01a36b93bb29280fb0eef263e2461313a41774c3b2f11ca7d9

SkinSharp For VC++

598.00 KB Size

2020-10-12 08:35:34 UTC 12 days ago

direct-cpu-clock-access peee runtime-modules

DETECTION	DETAILS	RELATIONS	BEHAVIOR	COMMUNITY
Acronis	Suspicious	Ad-Aware	Gen:Variant.Barys.54394	
AegisLab	Trojan.Win32.Delike.4lc	AhnLab-V3	Malware/Win32.Generic.C1452407	
Alibaba	Trojan.Win32.Kryptik.e835d36b	ALYac	Gen:Variant.Barys.54394	
Antiy-AVL	Trojan.Win32.SGeneric	SecureAge APEX	Malicious	
Arcabit	Trojan.Barys.DD47A	Avast	Win32:Evo-gen [Susp]	
AVG	FileRep/Malware	Avira (no cloud)	HEUR/AGEN.1114459	
Baidu	Win32.Trojan.Kryptik.aep	BitDefender	Gen:Variant.Barys.54394	
BitDefenderTheta	Gen:NN.Zexif.34298.Lu0@u4OwQBpi	CAT-QuickHeal	Trojan.Silcon.A5	
ClamAV	Win.Malware.Nymaim.4403	Comodo	TrojWare.Win32.Regisup.DG@edd0q3	
CrowdStrike Falcon	Win/malicious_confidence_100% (W)	Cybereason	Malicious.e6740d	
Cylance	Unsafe	Cynet	Malicious (score: 100)	
Cyren	W32/S-39d426e1Eldorado	DrWeb	Trojan.Inject.2.31002	
eGambit	Unsafe.AI_Score_70%	Emsisoft	Gen:Variant.Barys.54394 (B)	
eScan	Gen:Variant.Barys.54394	ESET-NOD32	A Variant Of Win32/Kryptik.EWVD	
F-Secure	Heuristic:HEUR/AGEN.1114459	FireEye	Generic.mg.bf54061e6740d5c0	
Fortinet	W32/Kryptik.EYDHtr	GData	Gen:Variant.Barys.54394	
Ikarus	Trojan.Crypt	Jiangmin	Trojan.Generic.aakkn	
F-Secure	Heuristic:HEUR/AGEN.1114459	FireEye	Generic.mg.bf54061e6740d5c0	
Fortinet	W32/Kryptik.EYDHtr	GData	Gen:Variant.Barys.54394	
Ikarus	Trojan.Crypt	Jiangmin	Trojan.Generic.aakkn	
K7AntiVirus	Trojan ( 004ef0321)	K7GW	Trojan ( 004ef0321)	
Kaspersky	HEUR:Trojan.Win32.Generic	MAX	Malware (ai Score=100)	
McAfee	Trojan-Goznym/BF54061E6740	McAfee-GW-Edtion	BehavesLike/Win32/Dropper.hc	
NANO-Antivirus	Trojan.Win32.Kryptik.fcdnpq	Palo Alto Networks	Generic.ml	
Panda	Troj/Generic.gen	Qihoo-360	Generic/HEUR/QVM20.1.2613.Malware.Gen	
Rising	Malware.UndefinedR.C.(TFE:2jPlcJXmz7...	Sangfor Engine Zero	Malware	
SentinelOne (Static ML)	DFI - Malicious PE	Sophos AV	Mal/Generic-S	
Sophos ML	Mal/Generic-S	Symantec	Trojan.Gen	
TrendMicro	TROJ_NYMAIM.GQA	TrendMicro-HouseCall	TROJ_NYMAIM.GQA	
VBA32	BScope.Trojan.Inject	VIPRE	Trojan.Win32.Generic.BT	
Webroot	W32.Trojan.Gen	Yandex	Trojan.Delike!	
ZoneAlarm by Check Point	HEUR:Trojan.Win32.Generic	Bkav	Undetected	
CMC	Undetected	Elastic	Undetected	
Kingsoft	Undetected	Malwarebytes	Undetected	
MaxSecure	Undetected	SUPERAntiSpyware	Undetected	
TACHYON	Undetected	ViRobot	Undetected	
Zillya	Undetected	Zoner	Undetected	
Avast-Mobile	Unable to process file type	Symantec Mobile Insight	Unable to process file type	

21  
Figure 7: Our perturbed malware, before our modifications

DETECTION	DETAILS	RELATIONS	BEHAVIOR	COMMUNITY
Ad-Aware		Gen:Variant.Barys.54394	ALYac	Gen:Variant.Barys.54394
SecureAge APEX		Malicious	Arcabit	Trojan.Barys.DD47A
Baidu		Win32.Trojan.Kryptik.aep	BitDefender	Gen:Variant.Barys.54394
BitDefenderTheta		Gen:NN.Zexaf.34570.AD3@a8ho0ai	Comodo	TrojWare.Win32.RegSup.DG@6dd0q3
CrowdStrike Falcon		Win/malicious_confidence_80% (D)	Cybereason	Malicious.1281a
Cynet		Malicious (score: 100)	DrWeb	Trojan.Inject2.31002
Elastic		Malicious (high Confidence)	Emsisoft	Gen:Variant.Barys.54394 (B)
eScan		Gen:Variant.Barys.54394	FireEye	Generic.mg.2300fc5128e1a264
GData		Gen:Variant.Barys.54394	Ikarus	Trojan.Crypt
K7GW		Trojan (700001211)	Kaspersky	HEUR:Trojan.Win32.Generic
MAX		Malware (ai Score=84)	McAfee	Trojan-Goznym91718DFD37C
McAfee-GW-Edition		BehavesLike.Win32.Ramnit.th	Qihoo-360	HEUR/QVM20.1A9F7/Malware.Gen
Rising		Malware.Undefined8.C (TFE:5tjuPwS8D...	SentinelOne (Static ML)	DFI - Suspicious PE
Symantec		SMG.Heur/gen	Tencent	Malware.Win32.Gencirc.10b3a1f9
VBA32		RScope.Trojan.Inject	ZoneAlarm by Check Point	HEUR:Trojan.Win32.Generic
Acronis		Undetected	AegisLab	Undetected
AhnLab-V3		Undetected	Alibaba	Undetected
Acronis		Undetected	AegisLab	Undetected
AhnLab-V3		Undetected	Alibaba	Undetected
Antiy-AVL		Undetected	Avast	Undetected
AVG		Undetected	Avira (no cloud)	Undetected
Bkav		Undetected	CAT-QuickHeal	Undetected
ClamAV		Undetected	CMC	Undetected
Cylance		Undetected	Cyren	Undetected
eGambit		Undetected	ESET-NOD32	Undetected
F-Secure		Undetected	Fortinet	Undetected
Jiangmin		Undetected	K7AntiVirus	Undetected
Kingsoft		Undetected	Malwarebytes	Undetected
MaxSecure		Undetected	Microsoft	Undetected
NANO-Antivirus		Undetected	Palo Alto Networks	Undetected
Panda		Undetected	Sangfor Engine Zero	Undetected
Sophos AV		Undetected	Sophos ML	Undetected
SUPERAntiSpyware		Undetected	TACHYON	Undetected
TotalDefense		Undetected	TrendMicro	Undetected
TrendMicro-HouseCall		Undetected	VIPRE	Undetected
VIRobot		Undetected	Webroot	Undetected
Yandex		Undetected	Zillya	Undetected
Zoner		Undetected	Avast-Mobile	Unable to process file type

Figure 8: The perturbed malware, after the modifications (notice the different hash value)



## 5 Conclusions and Future Work

In this paper, we present a method to generate end-to-end multi-feature types adversarial examples for PE malware classifiers, using explainability algorithms and our own methods to decide which features to modify, eliminating the need to reverse the bypassed NGAV. Our method is the first to tackle the challenging task of generating end-to-end adversarial examples of PE structural features, allowing not only feature addition but also feature modification.


Our evaluation demonstrates that explainability is a dual edged sword, which can also be leveraged by adversaries. When considering the call to generate more explainable models, which decisions can be interpreted by humans [15], one should take into account its negative effects, such as making adversarial examples less challenging in certain situations, as presented in this paper.

Our future work will include improving the query-efficiency of our attack (in sense of queries to the attacked malware classifier), in order to make it useful to attack cloud-based classifiers, by using gradient-based approaches (e.g., JSMA [16]) over the surrogate model in order to find the optimal feature modification out of the initial predetermined list. We would also research the detection and defense methods against such attacks, for instance, anomaly detection classifiers that recognize anomalous PE structure.

## References

- [1] Z. Katzir and Y. Elovici, “Quantifying the resilience of machine learning classifiers used for cyber security,” *Expert Systems with Applications*, vol. 92, pp. 419 – 429, 2018. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0957417417306590>
- [2] M. Zakeri, F. F. Daneshgar, and M. Abbaspour, “A static heuristic approach to detecting malware targets,” *Security and Communication Networks*, vol. 8, no. 17, pp. 3015–3027, 2015. [Online]. Available: <https://doi.org/10.1002/sec.1228>
- [3] I. Rosenberg, A. Shabtai, L. Rokach, and Y. Elovici, “Generic black-box end-to-end attack against state of the art API call based malware classifiers,” in *Research in Attacks, Intrusions, and Defenses - 21st International Symposium, RAID 2018, Heraklion, Crete, Greece, September 10-12, 2018, Proceedings*, ser. Lecture Notes in Computer Science, M. Bailey, T. Holz, M. Stamatogiannakis, and S. Ioannidis, Eds., vol. 11050. Springer, 2018, pp. 490–510. [Online]. Available: [https://doi.org/10.1007/978-3-030-00470-5\\_23](https://doi.org/10.1007/978-3-030-00470-5_23)
- [4] J. Saxe and K. Berlin, “Deep neural network based malware detection using two dimensional binary program features,” in *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*. IEEE, oct 2015.
- [5] M. Ancona, E. Ceolini, C. Oztireli, and M. Gross, “Towards better understanding of gradient-based attribution methods for deep neural networks,” in *International Conference on Learning Representations*, 2018. [Online]. Available: <https://openreview.net/forum?id=Sy21R9JAW>
- [6] “Cylance, I Kill You!” <https://skylightcyber.com/2019/07/18/cylance-i-kill-you>, accessed: 2019-08-24.
- [7] H. S. Anderson, A. Kharkar, B. Filar, D. Evans, and P. Roth, “Learning to evade static PE machine learning malware models via reinforcement learning,” *CoRR*, vol. abs/1801.08917, 2018. [Online]. Available: <http://arxiv.org/abs/1801.08917>
- [8] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. K. Nicholas, “Malware detection by eating a whole EXE,” in *The Workshops of the The Thirty-Second AAAI Conference on Artificial Intelligence, New Orleans, Louisiana, USA, February 2-7, 2018.*, ser. AAAI Workshops, vol. WS-18. AAAI Press, 2018, pp. 268–276. [Online]. Available: <https://aaai.org/ocs/index.php/WS/AAAIW18/paper/view/16422>
- [9] H. S. Anderson and P. Roth, “EMBER: an open dataset for training static PE malware machine learning models,” *CoRR*, vol. abs/1804.04637, 2018. [Online]. Available: <http://arxiv.org/abs/1804.04637>



- 
- [10] M. Sundararajan, A. Taly, and Q. Yan, “Axiomatic attribution for deep networks,” in *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ser. ICML 17. JMLR.org, 2017, p. 33193328.
- [11] A. Shrikumar, P. Greenside, and A. Kundaje, “Learning important features through propagating activation differences,” in *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ser. ICML 17. JMLR.org, 2017, p. 31453153.
- [12] S. Bach, A. Binder, G. Montavon, F. Klauschen, K.-R. Müller, and W. Samek, “On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation,” *PLoS ONE*, vol. 10, no. 7, p. e0130140, 07 2015. [Online]. Available: <http://dx.doi.org/10.1371/journal.pone.0130140>
- [13] I. Rosenberg, S. Meir, J. Berrebi, I. Gordon, G. Sicard, and E. Omid David, “Generating end-to-end adversarial examples for malware classifiers using explainability,” in *2020 International Joint Conference on Neural Networks (IJCNN)*, 2020, pp. 1–10.
- [14] S. M. Lundberg and S.-I. Lee, “A unified approach to interpreting model predictions,” in *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds. Curran Associates, Inc., 2017, pp. 4765–4774. [Online]. Available: <http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf>
- [15] C. Rudin, “Please stop explaining black box models for high stakes decisions,” *CoRR*, vol. abs/1811.10154, 2018. [Online]. Available: <http://arxiv.org/abs/1811.10154>
- [16] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, “The limitations of deep learning in adversarial settings,” in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, mar 2016.