Deceiving Portable Executable Malware Classifiers into Targeted Misclassification with Practical Adversarial Samples

Yunus Kucuk
Department of Computer Science
Binghamton University, State University of New York
ykucuk1@binghamton.edu

Guanhua Yan
Department of Computer Science
Binghamton University, State University of New York
ghyan@binghamton.edu

ABSTRACT

Due to voluminous malware attacks in the cyberspace, machine learning has become popular for automating malware detection and classification. In this work we play devil's advocate by investigating a new type of threats aimed at deceiving multi-class Portable Executable (PE) malware classifiers into targeted misclassification with practical adversarial samples. Using a malware dataset with tens of thousands of samples, we construct three types of PE malware classifiers, the first one based on frequencies of opcodes in the disassembled malware code (*opcode classifier*), the second one the list of API functions imported by each PE sample (*API classifier*), and the third one the list of system calls observed in dynamic execution (*system call classifier*). We develop a genetic algorithm augmented with different support functions to deceive these classifiers into misclassifying a PE sample into any target family. Using an Rbot malware sample whose source code is publicly available, we are able to create practical adversarial samples that can deceive the opcode classifier into targeted misclassification with a successful rate of 75%, the API classifier with a successful rate of 83.3%, and the system call classifier with a successful rate of 91.7%.

CCS CONCEPTS

• **Security and privacy → Malware and its mitigation**; • **Theory of computation → Adversarial learning**.

KEYWORDS

Malware classification; adversarial machine learning

ACM Reference Format:
Yunus Kucuk and Guanhua Yan. 2020. Deceiving Portable Executable Malware Classifiers into Targeted Misclassification with Practical Adversarial Samples. In *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy (CODASPY '20), March 16–18, 2020, New Orleans, LA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3374664.3375741

1 INTRODUCTION

Malware are responsible for a variety of online criminal activities. AV-TEST, a computer security institute, has collected 137.47 million new malware variants in 2018, an increase of 13% over the previous year [17]. To counter against voluminous malware attacks, machine learning techniques have become popular for detecting and classifying malware variants, such as Windows PE (Portable Executable) malware [33, 40, 45, 47, 63, 65], PDF malware [43, 56, 57, 59], Android malware [11, 16, 29, 50], and Linux malware [38, 55].

The increasing use of machine learning in malware defense has raised concern about their robustness in adversarial environments [61]. There has been recent research on adversarial machine learning attacks against malware classifiers, including *evasion* of PDF malware classifiers [18, 58, 62], PE malware classifiers [14, 15, 35, 37] and Android malware classifiers [28, 64]. Most of these previous works, however, focus on development of efficient algorithms for finding adversarial samples instead of creation of real adversarial samples that can work validly in practice; the few exceptions include generation of evasive PDF malware in [58] and [62], PE malware in [15], and Android malware in [64].

Against this backdrop, we explore a new problem in the study of adversarial malware samples: *can we generate functionality-validated malware executables that can deceive a multi-class PE malware classifier into targeted misclassification*? Towards this end, we need to tackle the following unique challenges. *Firstly*, there is limited action space to modify PE malware features without breaking their functionalities. For example, when evading a malware classifier trained on opcode frequency features extracted from disassembled malware code [19, 42, 53], the attacker cannot just pad the malware file with new instructions to match the frequency histogram of a benign file, because the sample created may not be a valid PE file or execute correctly. *Secondly*, various types of features can be used to classify PE malware, including those from both static and dynamic analysis [63]. Hence, it is difficult to develop a unified methodology to evade the malware classifiers trained from different types of malware features. For example, although the work in [15] has considered using packing procedures or modifying functionality-insensitive fields in PE files to create mutated samples for evasion attacks, the adversarial malware samples created as such cannot evade the classifiers trained from features extracted from dynamic execution in malware sandboxes. *Last but not least*, our goal in this work extends the focus of existing works on creation of evasive malware samples to a more general malware deception problem, which is to deceive a multi-class malware classifier into classifying an adversarial malware sample into any target family, including the benign class. Malware deception cannot be achieved with widely used malware packing techniques, such as compression and encryption, because the samples created may not fall into the feature space of the targeted malware family.

In this work we develop a generic methodology based on genetic algorithms to search for practical adversarial malware samples.

CODASPY '20, March 16–18, 2020, New Orleans, LA, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7107-0/20/03…$15.00
https://doi.org/10.1145/3374664.3375741

Our technique can deceive a variety of PE malware classifiers into targeted misclassification with customized distance functions measuring similarity among malware samples in the feature space used by these classifiers. Our work performs obfuscation at the source code level and exploits the causality between source code modification and changes of malware features extracted from the compiled executables to generate adversarial samples. To ensure their practicality, our method validates their functionalities in a sandbox.

In a nutshell, our main contributions are summarized as follows:

- We use a malware dataset with tens of thousands of PE samples to train three multi-class classifiers, the first one based on frequencies of opcodes in the disassembled malware code (*opcode classifier*), the second one the list of API functions imported by each PE sample (*API classifier*), and the third one the list of system calls observed in dynamic execution (*system call classifier*) to classify a PE sample into its corresponding family.
- Based on bogus control flow obfuscation, we study the effects on the opcode feature values due to basic block cloning through differential analysis. To create an adversarial sample that can deceive the opcode classifier into targeted misclassification, we formulate an optimization problem that minimizes the Kullback-Leibler divergence between the opcode frequency feature vectors from the target. We develop a genetic algorithm to search practical malware variants that can deceive the opcode classifier into targeted misclassification.
- To deceive the API classifier, we use a popular API obfuscation technique to conceal imported API functions. We formulate an optimization problem that minimizes the Euclidean distance between the imported API function feature vectors from the target. We adopt a similar genetic algorithm to produce malware variants whose API feature vectors are gradually converging towards the point with a minimal Euclidean distance from the target sample in the feature space.
- As the system call classifier is trained from malware features extracted from dynamic execution, we consider adding dummy API calls on malware's execution paths to influence the list of system calls invoked. We carefully select API calls to be added to the original malware sample with predictable effects on the feature vectors. A similar genetic algorithm is used to search adversarial samples based on a fitness function that uses the Jensen-Shannon divergence to measure the distance of system call frequency feature vectors.
- The experiments reveal that our methods can mutate the Rbot sample, whose source code has been publicly available, into a functionable variant misclassified by the opcode classifier into any target family with a successful rate of 75%, by the API classifier with a successful rate of 83.3%, and by the system call classifier with a successful rate of 91.7%.

The remainder of this work is organized as follows. Section 2 presents related work. Section 3 introduces the problem formulation. Section 4 presents the malware dataset and the three malware classifiers considered in this work. In Sections 5, 6 and 7, we present the algorithms for deceiving these three classifiers into targeted misclassification, respectively. We discuss the scope of this work in Section 8 and draw concluding remarks in Section 9.

2 RELATED WORK

Machine learning-based PE malware defense. In the literature, a variety of features have been extracted from PE malware samples to train predictive machine learning models, including n-gram byte sequences in the binary executables [33, 45, 52, 63], PE header fields [46, 54, 63], opcode n-gram sequences [19, 42, 53], Windows API calls [13, 51, 65], structural features contained within control flow graphs [23, 34], and many others. In this work, we focus on creation of adversarial malware samples to deceive three specific multi-class PE malware classifiers, although in principle our methodology can be extended to deceive other types of malware classifiers. In addition to the variety of malware features used, different classification models have also been considered in these previous works, including SVM [33, 53, 63, 65], decision trees or ensemble learning based on decision trees [42, 45, 51, 53, 63, 65], Naive Bayes [33, 42, 53, 63, 65], neural networks [42, 53]. In our work, we consider only random forest as the malware classifier as it has been consistently found that ensemble learning based on decision trees tends to perform well on malware classification [33, 42, 45, 51, 53, 63, 65].

Generation of adversarial malware samples. Existing efforts dedicated to creating adversarial malware samples for evading machine learning-based malware detection systems are summarized in Table 1. The survey tells us that although it has been successful to generate real PDF malware examples that can evade machine learning-based malware detection [58, 62], efforts on adversarial attacks against PE malware classifiers focused on creation of *hypothetical* malware samples based on modification of features extracted from the malware samples. For some features, however, it may be difficult to modify their values while preserving malware's functionability. The rare exceptions include adversarial machine learning attacks against binary malware classifiers trained on static analysis features [15] and API call features from dynamic execution [48]. Different from our work, they do not attack a multi-class malware classifier and are able to evade neither the opcode classifier nor the system call classifier considered in this study.

Algorithms for searching adversarial samples. Many algorithms have been proposed to generate adversarial samples for deep learning prediction models. These methods include Fast Gradient Sign Method (FGSM) [27], the Jacobian-based Saliency Map approach (JBSM) [44], DeepFool [41], the Carlini & Wagner's attack [22], ElasticNet attacks [24], and momentum-based iterative algorithms [26]. Among the previous works aimed at generating adversarial malware samples, the algorithms used to find adversarial examples include genetic algorithms [62], reinforcement learning [15], FGSM [35, 37], generative adversarial networks [30], gradient descent algorithms [32], JBSM [28], and Carlini & Wagner's attack [37]. As this work uses random forest as the malware classifier, those methods developed for generating adversarial samples for deep learning models are not directly applicable.

3 PROBLEM FORMULATION

Consider a PE malware classifier C_{mal}, which classifies PE samples into disjoint families $F = \{f_i\}_{1 \leq i \leq |F|}$. Without loss of generality, all benign samples are assumed to belong to the same family $f_{benign} \in F$. To perform threat analysis of adversarial samples on classifier

Table 1: Existing works on creation of adversarial malware samples for deceiving malware classifiers. Being *Validated* means that the functionality of an adversarial sample created has been validated through dynamic analysis.

Article	Malware Type	Validated?	Binary or Multi-Class?	Targeted Misclassification?
Biggio *et al.* [18]	PDF malware	✗	Binary	Yes
Srndic and Laskov [58]	PDF malware	✓	Binary	Yes
Xu *et al.* [62]	PDF malware	✓	Binary	Yes
Grosse *et al.* [28]	Android malware	✗	Binary	Yes
Demontis *et al.* [25]	Android malware	✗	Binary	Yes
Yang *et al.* [64]	Android malware	✓	Binary	Yes
Liu *et al.* [37]	PE malware	✗	Binary	Yes
Kolosnjaji *et al.* [32]	Unknown	✗	Binary	Yes
Kreuk *et al.* [35]	PE malware	✗	Binary	Yes
Hu and Tan [30]	Unknown	✗	Binary	Yes
Al-Dujaili *et al.* [14]	PE malware	✗	Binary	Yes
Anderson *et al.* [15]	PE malware	✓	Binary	Yes
Rosenberg *et al.* [48]	PE malware	✓	Binary	Yes
Abusnaina *et al.* [12]	Linux malware	✗	Multi-class	Yes
Our work	**PE malware**	✓	**Multi-class**	**Yes**

C_{mal}, we play devil's advocate by obfuscating an existing PE sample PE_o. The true family of sample PE_o is assumed to be $f_o \in F$.

Our goal is to transform PE_o into a new sample PE'_o such that classifier C_{mal} misclassifies PE'_o as a target family f_d where $f_d \neq f_o$. The purpose of targeted misclassification can be:

- *Malware evasion:* The author of malware sample PE_o may want to obfuscate it so classifier C_{mal} would misclassify it as benign (i.e., $f_d = f_{benign}$), regardless of its original family f_o. This can effectively evade the detection of C_{mal} for malicious PE programs. As demonstrated in Table 1, the majority of existing works on adversarial malware samples focuses on malware evasion.
- *Malware deception:* In addition to malware evasion which requires $f_d = f_{benign}$, an attacker may also want to achieve $f_d \neq f_{benign}$. In these cases, the attacker would like to create PE samples that can deceive malware classifier C_{mal} into believing that they are from a specific family. For instance, if the attacker can morph a benign PE file into a sample that is misclassified by C_{mal} as a Stuxnet attack [8], it may cause panic to the institutions who deploy malware classifier C_{mal} or mislead their threat mitigation operations.

In this work we consider the malware deception problem, which is a generalization of the malware evasion problem because no constraint is imposed on the target family f_d.

Adversary model. In this work, we make the following assumptions about the adversary:

- *Availability of source code:* Different from existing works that directly modify malware samples (e.g., malicious PDF files) or their features to evade detection, we instead assume that the source code of the malware sample should be available. The availability of source code enlarges the action space of the attacker for deceiving malware classifier C_{mal} into targeted misclassification.
- *Malware classifier as a blackbox:* When creating adversarial samples, we assume that the attacker does not know any internal detail about malware classifier C_{mal}, such as its model parameters, except the type of features it uses for malware classification. The attacker treats C_{mal} as a blackbox, which returns the classification result with an input PE sample.
- *Anchor samples from the target family:* We assume that the attacker has access to a set of PE samples from the target family f_d, or equivalently their features with which C_{mal} would classify the samples as the target family f_d. We use A_{f_d} to denote the set of anchor samples for target family f_d.

Regarding the third assumption, one may wonder why the attacker would bother to create new adversarial samples pretending to belong to the target family if he already has real examples from it. There could be multiple reasons for this. Firstly, in practice there may be multiple defense schemes deployed against malware attacks. If the signatures of these real samples from the target family have already been known, they may be detectable by up-to-date AV (Anti-Virus) scanners. Secondly, depending on the attacker's intention, directly using the real samples from the target family may not achieve his deception goal. For example, the attacker may just want to use a benign PE sample pretending to be a Stuxnet malware for purpose of intimidation without causing any real harm.

4 MALWARE DATASET AND CLASSIFIERS

In this section we describe the PE malware dataset we use in this study, three types of malware classifiers trained on this dataset, and finally the source code of an Rbot sample used to deceive the three classifiers into targeted misclassification.

PE malware dataset. Albeit it is easy to download a large number of malware samples from public repositories (e.g., VirusShare [9]), it is not easy to find many labeled ones. We use a PE malware dataset that contains 15,721 unpacked and 9,479 packed PE malware samples belonging to 12 malware families, *Bagle, Bifrose, Hupigon, Koobface, Ldpinch, Lmir, Rbot, Sdbot, Swizzor, Vundo, Zbot,* and *Zlob.* Our

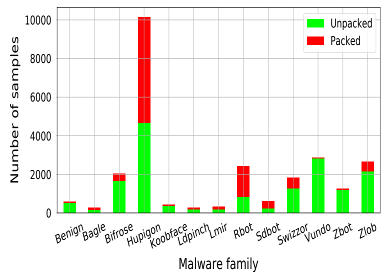

Figure 1: Distribution of family sizes in the malware dataset

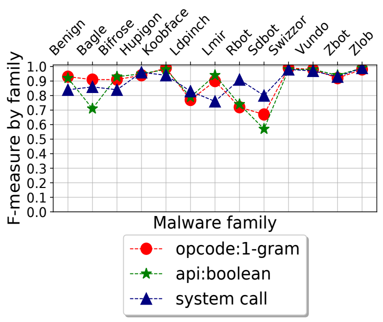

Figure 2: Classification performance of the three malware classifiers

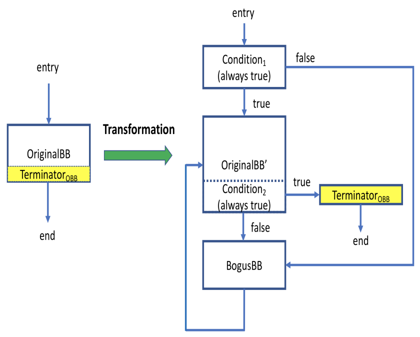

Figure 3: Illustration of bogus flow control obfuscation

malware dataset has more samples than that used in Kaggle's Microsoft malware classification competition, which consists of 10,868 labeled malware samples belonging to nine families [5]. Moreover, the Microsoft malware dataset is unsuitable for this study, as we cannot find source code for any of its nine families. In addition to our malware dataset, we use 519 *Benign* PE samples and as mentioned earlier, all these benign samples are treated as belonging to a special *Benign* family. The size distribution of these 13 malware families is given in Figure 1. The malware dataset is imbalanced across different malware families, which is consistent with the previous observations [49, 53, 63].

PE malware classifiers. We extract three types of malware features, each used to train a random forest classifier. We choose the random forest classification model because previous reports have consistently shown that ensemble learning based on decision trees perform well on the three types of malware features considered in this study [33, 42, 45, 51, 53, 63, 65]. The following discussion elaborates on how we build these three malware classifiers.

(1) Opcode classifier $C_{mal}(O^{(n)})$: random forest classifier trained on opcode features. Malware classifiers trained with opcode n-gram features have been investigated in a plethora of previous works [19, 42, 53]. To obtain opcode n-gram features, we disassemble the binary executable program of each unpacked malware in our dataset into instructions in an assembly language. Packed malware samples usually have malware instruction code obfuscated and are thus ignored in training $C_{mal}(O^{(n)})$. An assembly instruction includes both its opcode and operands. For example, using the IDA Pro tool [4], we can obtain the first nine instructions at the entry point of a Zbot malware sample:

i1: *push* ebp	i2: *mov* ebp, esp	i3: *sub* esp, 10h
i4: *push* ebx	i5: *xor* ecx, ecx	i6: *xor* b1, b1
i7: *call* sub_41C53B	i8: *test* al, al	i9: *jz* loc_41D55C

The opcode in each instruction is shown in italics in the above table. The opcode n-gram feature uses the combination of n consecutive opcodes within a sliding window as its feature name and its frequency as its feature value. In the example, there are seven 1-gram features with its feature values shown in the parentheses: *push* (2/9), *mov* (1/9), *sub* (1/9), *xor* (2/9), *call* (1/9), *test* (1/9), and *jz* (1/9); there are eight 2-gram features, each with a value of 1/8.

The instructions disassembled from a PE program can be represented as a control flow graph (CFG), where each node characterizes a basic block without instructions branching out except at its exit and each edge a control flow transition between basic blocks through jump instructions. In our implementation, we use the popular IDA Pro tool to reconstruct the CFG of each PE program but our methodology can be generalized to any CFG recovery technique.

Given the CFG $G_p^{(b)}(V_p^{(b)}, E_p^{(b)})$ constructed from program p, we obtain its opcode n-gram feature vector as follows. We use $O^{(n)}$ to denote the entire set of opcode n-gram features. For each opcode n-gram feature $x \in O^{(n)}$, let its count in a basic block of $G_p^{(b)}$, $v \in V_p^{(b)}$, be $\alpha_v(x)$. It is noted that superscript (b) is intentionally added to distinguish the binary executable of program p from its other representations, which we shall see later. The *count value* of opcode n-gram feature x in program p, where $x \in O^{(n)}$, is:

$$\beta_p(x) = \sum_{v \in V_p^{(b)}} \alpha_v(x) \tag{1}$$

The *frequency value* of opcode n-gram feature x in program p is obtained by normalizing $\beta_p(x_n)$ with $\sum_{x_n \in O_n} \beta_p(x_n)$. As discussed earlier, the frequency values of opcode n-gram features are used to train a malware classifier. We use $C_{mal}(O^{(n)})$ to denote the specific malware classifier trained with opcode n-gram features in $O^{(n)}$.

(2) API classifier $C_{mal}(A)$: random forest classifier trained on imported API function features. Another type of malware features widely used for PE malware classification is the imported Windows API functions, which can be discovered from the Import Directory Table in the `.idata` section by following the second data directory of the image optional header inside the PE file header.

We use IDA Pro [4] to extract Windows API functions imported by unpacked PE samples in our dataset. Packed samples are ignored because their imported API functions may be obfuscated. The union of all imported Windows API functions is used to create the API function feature vector, where the feature value of a Windows API function is 1 if imported by the sample, or 0 otherwise. We train a multi-class random forest classifier based on the API function features. We use $C_{mal}(A)$ to denote this malware classifier.

(3) System call classifier $C_{mal}(S)$: random forest classifier trained on system call features. System calls have been used for malware detection and classification in a number of previous works [20, 21, 38].

We use the Intel Pin tool [6] to dynamically instrument each malware sample in our dataset, either packed or unpacked, and monitor the sequence of system calls executed in a Windows 7 malware sandbox with Internet connections emulated by FakeNet [3]. We find that 4255 unpacked samples cannot be dynamically instrumented by Intel Pin and are thus ignored in our experiments. For all the malware sample dynamically instrumented by Intel Pin, we observe 463 unique system calls. They are used to construct each sample's feature vector, whose values indicate the frequency of each system call made by the malware. Using these system call features, we train a multi-class random forest classifier to classify a PE sample to its corresponding family. We use $C_{mal}(S)$ to denote this malware classifier.

Classification performance. We use 5-fold cross validation on our malware dataset to evaluate the classification performances of the malware classifiers. For each malware family, we treat the classification results as from a binary classifier: either belonging to the family or not. Precision P is defined as $T_p/(T_p + F_p)$, where T_p and F_p are the numbers of true positive and false positive samples in the results, respectively. Similarly, recall R is defined as $T_p/(T_p+F_n)$, where T_p and F_n are the numbers of true positive and false negative samples in the results, respectively. The F-measure (or F-1 score) of the binary classifier is the harmonic mean of precision and recall, $2PR/(P + R)$.

The classification performance results for each family by the three malware classifiers, $C_{mal}(O^{(1)})$, $C_{mal}(A)$ and $C_{mal}(S^{(1)})$ are shown in Figure 2. Over all the 13 families, the mean F-measures for these three malware classifiers are 0.893, 0.878, and 0.924, respectively.

Rbot source code compilation. In our work, we do not want to use the source code of a benign program because it does not allow us to evaluate the effectiveness of our method in malware evasion. To deceive a malware classifier into targeted misclassification, we also prefer working on a malware sample that belongs to one of the output families of that classifier. We add this sample to the training dataset so that the classifier can correctly classifies it into the correct family. The reason behind this extra step is to ensure that if a new variant transformed from this sample is classified into the target family, it should not be done by a *mistake* of the classifier.

Due to these concerns, we use the source code of an Rbot sample publicly available in the malware zoo on github [7]. To produce functionable PE malware samples from the Rbot source code, we have used two different compilation methods:

Compilation with MSVC++: We have compiled the Rbot source code with the MSVC++ (Microsoft Visual C++) 6.0 compiler (_MSC_VER == 1200, Visual Studio 6.0). During this process we overcame Windows version-specific challenges such as installing required runtime libraries in the system directories and installing Microsoft SDK (Software Development Kit) in the case that existing libraries are not compatible with the compiler, and configuring the correct library paths and system paths. The size of the Rbot malware sample compiled with MSVC++ is 222KB in a Windows 7 VM (Virtual Machine).

Compilation with clang++. A practical PE sample compiled with clang++ should be able to use Windows runtime libraries, which requires Visual C++ ABI (Application Binary Interface) compatibility. We have compiled the Rbot source code with clang++ version 3.4 in a Windows VM and produced the PE sample with the MSVC++ 6.0 linker. As the Visual C++ ABI changes over time, linking the clang-produced object files with the MSVC++ linker may incur compatibility issues. We overcame these challenges by providing the clang compiler with multiple flags, including *-fms-compatibility*, *-fms-extensions*, *-fdelayed-template-parsing*, and *-fmsc-version=1200* (which indicates the Microsoft compiler version to be MSVC++ 6.0). Successful compilation with clang++ also calls for slight modification of the source code, such as replacing use of the *new* operator with a call of the *malloc* function and redeclaring variables outside the *for* loops. The size of the Rbot malware sample compiled with clang++ is 265KB in a Windows 7 VM.

Malware deception goal. Our goal is to obfuscate the Rbot malware at the source code level so that the malware executable compiled with either MSVC++ or clang++ satisfies the following two criteria. First, the adversarial malware sample should deceive one of the three aforementioned classifiers into classifying it to a targeted family. It is noted that our goal is to deceive machine learning-based malware classifiers into targeted misclassification. The popular VirusTotal service [10] relies on tens of AV (Anti-Virus) scanners for malware detection. These AV scanners usually use signature-based detection and the detection results may not include the correct family labels of the malware samples. Therefore our experiments do not use the detection results of the VirusTotal service for performance evaluation.

The second criterion requires the adversarial malware example created to be functionable. We validate the functionability of an Rbot malware sample as follows. We set up a C&C server and configure each Rbot malware sample to communicate with the server after it infects a VM running Windows 7. To automate the whole process, we use the Cuckoo tool [2]. The communication between the bot and the server is done through an mIRC channel, which the botmaster uses to control the bot-infected machine through various commands. In our implementation, we emulate six different commands from the botmaster, which are summarized in Table 2. If the bot machine's responses to these commands are the same as expected, the malware sample is deemed as functionable.

5 DECEIVING OPCODE CLASSIFIER

This section describes how to deceive classifier $C_{mal}^{O^{(1)}}$.

5.1 Methodology

Bogus control flow. The building block of our method is bogus control flow that adds fake basic blocks to an existing CFG. Our implementation is based on bogus control flow in llvm-obfuscator [31] illustrated in Figure 3. The bogus flow obfuscation is performed on a basic block, *OriginalBB*, in the CFG represented at the LLVM IR level. The last instruction of *OriginalBB* is $Terminator_{OBB}$ indicating the exit of this basic block. The transformation introduces two always-true conditions, $Condition_1$ and $Condition_2$, which can be any opaque predicates used for malware obfuscation. $Condition_1$ is added before *OriginalBB*: if it is true, *OriginalBB* is executed except that its $Terminator_{OBB}$ is replaced with $Condition_2$; otherwise, the control flow is directed to a bogus basic block *BogusBB*. Moreover, if $Condition_2$ holds true, $Terminator_{OBB}$ is executed; otherwise, the control flow is also directed to the bogus basic block *BogusBB*.

Table 2: Rbot commands and the corresponding expected responses to test the functionality of samples created

Command	Response
.login *password*	[MAIN]: Password accepted.
.status	[MAIN]: Status: Ready. Bot Uptime: 0d 0h 0m.
.opencmd	[CMD]: Remote shell ready. Microsoft Windows [Version 6.1.7601] Copyright (c) 2009 Microsoft Corporation. All rights reserved. C:\Windows\system32>
.cmd dir C:\	Volume in drive C has no label. Volume Serial Number is A48C-CF74 Directory of C:\ 06/10/2009 02:42 PM 24 autoexec.bat 06/10/2009 02:42 PM 10 config.sys 07/13/2009 07:37 PM <DIR> PerfLogs 03/05/2019 03:06 PM <DIR> Program Files 10/04/2017 01:09 PM <DIR> Python27 10/04/2017 04:02 PM <DIR> Users 10/04/2017 04:22 PM <DIR> Windows 2 File(s) 34 bytes 5 Dir(s) 24,380,584,832 bytes free
.cmdstop	[CMD]: Remote shell stopped. (1 thread(s) stopped.)
.logout	[MAIN]: User *botmaster* logged out.

Generation of adversarial samples. With bogus flow obfuscation, we can add arbitrary LLVM IR instructions inside *BogusBB* as they are never executed. To obtain legitimate LLVM IR instructions, we can clone existing basic blocks represented at the LLVM IR level. It is noted, however, that classifier $C_{mal}(O^{(n)})$ is trained with the opcode features extracted from binary executables. To achieve targeted misclassification by $C_{mal}(O^{(n)})$, we must carefully insert bogus basic blocks, which, after their compilation to a PE program, can appropriately change the opcode n-gram feature values.

We use differential analysis to find an optimal combination of bogus flow obfuscations for targeted misclassification. Given the source code of malware p written in C/C++, we use *clang++* to compile it into a CFG represented at the LLVM IR level. For clarity of presentation, we use $G_p^{(IR)}(V_p^{(IR)}, E_p^{(IR)})$ to represent such a CFG constructed for program p. In each iteration, we choose a basic block in $V_p^{(IR)}$, treat it as *OriginalBB* and then clone it into *BogusBB* as seen in Figure 3. The new program p' after adding a bogus basic block is compiled to its corresponding binary executable.

For each basic block $v^{(IR)} \in V_p^{(IR)}$, the differences in the count values of opcode n-gram features in $O^{(n)}$ is given by:

$$\Delta(v^{(IR)}, x) = \beta_{p'}(x) - \beta_p(x) \quad \text{for each } x \in O^{(n)}. \tag{2}$$

For brevity of discussion, we use $\Delta(v^{(IR)}, \cdot)$ to denote the vector of changes in the count values of all the opcode n-gram features and call it the Δ-vector of basic block $v^{(IR)}$. It is possible that transformation of different basic blocks leads to the same Δ-vector. We thus choose a subset of basic blocks from $G_p^{(IR)}(V_p^{(IR)}, E_p^{(IR)})$ with distinct Δ-vectors and put them into set $S_{BB}^{(IR)}$. Similarly, we use $\Delta(\cdot, x)$ to denote the vector of count value changes for feature x over all the basic blocks in $S_{BB}^{(IR)}$.

In the next step, we determine *how many times* each basic block in $S_{BB}^{(IR)}$ should be transformed with bogus flow obfuscation illustrated in Figure 3. We let vector M, indexed in the same order as $\Delta(\cdot, x)$, include the number of times that each basic block in $S_{BB}^{(IR)}$ should be transformed. Hence, the dot product of vectors M and $\Delta(\cdot, x)$, i.e., $M \cdot \Delta(\cdot, x)$, gives the total change on the count value of feature x extracted from the final PE program p' after all the transformations are carried out.

After transformation based on vector M, the count value of opcode n-gram feature x in program p', where $x \in O^{(n)}$, becomes:

$$\beta_{p'}(x) = \beta_p(x) + M \cdot \Delta(\cdot, x), \tag{3}$$

and the normalization factor $\sum_{x \in O^{(n)}} \beta_{p'}(x)$ is used to derive its frequency value.

Recall that it is assumed that the attacker has a set of anchor samples for target family f_d, denoted by A_{f_d}. For targeted misclassification, an anchor PE sample, q, is picked from A_{f_d}. As frequencies of opcode n-gram sequences are used to train classifier $C_{mal}(O^{(n)})$, we aim to find such a vector M that minimizes the KL (Kullback-Leibler) divergence between the frequency feature vectors of programs p' and q, where the KL divergence between any two discrete probability distributions P and Q is defined as follows:

$$D_{KL}(P, Q) = -\sum_{x \in X} P(X) \log \frac{Q(x)}{P(x)}. \tag{4}$$

However, when $P(x) = 0$ for some x that is in the support of distribution Q, the KL divergence cannot be computed. We thus calculate the middle point between the frequency feature vectors of programs p' and q, and then minimize the KL divergence between it and the frequency feature vector of program q, which leads to the following optimization problem:

$$\begin{aligned} \underset{M}{\text{minimize}} \quad & D_{KL}(P, Q) \\ \text{where} \quad & X = \{x \in O^{(n)} : \beta_{p'}(x)\beta_q(x) \neq 0\} \\ & P(x) = \tfrac{1}{2}\Big(\frac{\beta_{p'}(x)}{\sum_{x \in O^{(n)}} \beta_{p'}(x)} + \frac{\beta_q(x)}{\sum_{x \in O^{(n)}} \beta_q(x)}\Big) \\ & Q(x) = \frac{\beta_q(x)}{\sum_{x \in O^{(n)}} \beta_q(x)} \end{aligned}$$

Our modification of KL divergence to address the 0-support problem has been inspired by the Jensen-Shannon (JS) divergence [36]. As seen in Section 7, the JS divergence can also be directly used in the search for adversarial malware samples.

Genetic algorithm. After testing a few optimization techniques such as simulated annealing and mixed integer programming solvers, we choose the genetic algorithm to search the solution to the aforementioned optimization problem due to its easy implementation. To use the genetic algorithm, we define a chromosome (or individual) as an instantiation of vector M in the optimization problem. Hence, the length of each chromosome is the number of basic blocks we can replicate through bogus flow obfuscation.

The pseudocode is shown in Algorithm 1. It starts with a randomly generated population H of size $|H|$, where for any $h \in H$ each of its elements is uniformly chosen among $[0, ..., 100]$ (recall that h is an instantiating of vector M). The genetic algorithm runs

Algorithm 1 Genetic algorithm for finding adversarial samples to deceive classifier C_{mal} into targeted misclassification

```
Require: feature vector $X_p$, target family $f_d$, feature vector $X_q$,
    parameter $n$, parameter $t$
Ensure: $p$ is misclassified by $C_{mal}$ as family $f_d$
  $H \leftarrow$ a randomly generated population
  for each $i \in \{1, ..., n\}$ do
      $J \leftarrow \emptyset$                                    ▷ $J$ is a map
      for each chromosome $h \in H$ do
          $X_{p'} \leftarrow$ MODIFY($X_p, h$)
          if classifier $C_{mal}$ classifies $X_{p'}$ as $f_d$ then
              Generate sample $p'$ and test its functionality
              if $p'$ is functional then
                  return sample $p'$
              end if
          end if
          $J[h] \leftarrow$ FITNESS($X_{p'}, X_q$)        ▷ Keep the fitness score of
  chromosome $h$ in map $J$
      end for
      $L \leftarrow$ top $t$ chromosomes with highest scores in map $J$
      Perform full mutation on each chromosome in $L$ and add the
  results onto list $L_{fm}$
      Perform partial mutation on each chromosome in $L$ and add
  the results onto list $L_{pm}$
      Perform crossover on chromosomes from $L$ and add the results
  onto list $L_{co}$
      Generate new population $H \leftarrow L_{fm} \cup L_{pm} \cup L_{co}$
  end for
```

Algorithm 2 Support functions for deceiving opcode classifier

```
  function MODIFY($X_p, h$)
      return $X_p + h \cdot \Delta$-vector
  end function
  function FITNESS($X_{p'}, X_q$)
      return $D_{KL}(X_{p'}, X_q)$
  end function
```

n iterations, each creating a new generation of population. In our implementation the population size $|H|$ is 1,000 and $n = 400$.

For each individual in the current population, we use it to modify the original sample based on Eq. (3); the modification function is defined by MODIFY() in Algorithm 2. The modified hypothetical sample is fed to the malware classifier to check whether it achieves targeted misclassification. If it passes the targeted misclassification test, we create the corresponding real sample through compilation with LLVM and then test its functionality. If the malware sample works as expected, the genetic algorithm returns the sample.

The fitness score of a hypothetical sample is calculated as the KL-distance between its feature vector and that of the anchor sample as described in Eq. (4). If none in the current population H succeeds in achieving targeted misclassification practically, we choose the top t chromosomes with the highest fitness scores from H and add them onto list L. In our implementation, we choose t to be 200. We define three types of genetic operations: *(1) Full mutation*: Consider the k-th element in vector M and let its highest value be $z_k^{(max)}$ in the current population. We iterate over every chromosome on L. For its k-th element for each $k \in [1, |M|]$, we update it by adding a random number r that is uniformly chosen from $[-z_k^{(max)}, z_k^{(max)}]$; if it becomes negative, it is assigned to be 0. *(2) Partial mutation:* We define three ranges in $Z = \{1, 10, 100\}$. We iterate over every chromosome on L. For each range $z \in Z$, we perform the following. For each of the $|M|$ elements of the current chromosome, it is mutated with probability 0.5. If it is chosen to mutate, we update it by adding a random number r, which is uniformly chosen among $[-z, z]$; similarly, if the new number becomes negative, it becomes 0. As there are three ranges, for each chromosome on L we produce three new variants. *(3) Crossover:* We randomly choose two chromosomes h_0 and h_1 from list L and create a new variant as follows. For each $k \in [1, |M|]$, the k-th item of the new variant is randomly chosen from h_0 and h_1 at the same place. The crossover operation is performed for 200 times.

If, for the current anchor sample q, the genetic algorithm fails to find a practical adversarial example that can deceive the classifier into targeted misclassification, we choose a different one from set A_{f_d} and rerun the genetic algorithm.

5.2 Implementation

Modification of llvm-obfuscator [31]. To implement our algorithm for generating adversarial malware samples, we modify the bogus control flow module of llvm-obfuscator as follows:

- *Removal of existing conflicting features:* Bogus control flow is implemented by llvm-obfuscator as an LLVM pass with options to control the probability of obfuscating a function (`funcBCF`) or a basic block (`boguscf-prob`). Our algorithm is deterministic and the stochasticity feature is thus unnecessary. We also remove the existing feature of adding junk instructions into a cloned basic block to control the effect of duplicating a basic block. Moreover, the optional suffix naming feature in the cloned basic blocks can cause a stack overflow problem when the basic blocks in a function are obfuscated by a large number of times, leading to excessively long instruction names.
- *Addition of per-basic block control:* We add two pass options, both as vectors. The first vector specifies the indices of the basic blocks to be cloned and the other the number of times each basic block should be cloned (i.e., vector M in Eq. (3)).

Consistent basic blocks. The premise of our algorithm is that cloning a basic block by k times should lead to linear changes on the counts of opcode affected. This, however, is not always true for all basic blocks, as explained in Table 3. The number in each entry of the table gives the increased count of `mov` instructions in the current sample over that derived from the previous column. For example, for basic block 86, after it is cloned by k times, where $k \in \{2, 3, 4, 5\}$, the increase of the `mov` instructions in the sample is always 5 compared to the one after it is cloned by $k - 1$ times; however, after basic block 86 is cloned by 100 times, the count of the `mov` instructions in the sample has increased by 478 over that after it is cloned by five times. Therefore, it is *not* a consistent basic block because if the increase is linear, the last sample should have 475 more `mov` instructions than that with the basic block cloned by five times. By contrast, basic block 3 is consistent because except

Table 3: Illustration of consistent basic blocks

Basic block index	Opcode	Iteration of basic block cloning					
		1	**2**	**3**	**4**	**5**	**100**
3	*mov*	9	10	10	10	10	950
86	*mov*	6	5	5	5	5	478
		$\underbrace{\qquad}_{\Delta_1}$	$\underbrace{\qquad}_{\Delta_2}$				

the first transformation cloning the basic block by one more time increases the count of mov instructions by exactly 10.

It is time consuming to check consistency of each basic block by cloning it many times. We thus use the following method to obtain consistent basic blocks. Define an ordered list $L = [1, 2, 3, 4, 5, 100]$, where the index starts from 1. Let B include all the basic blocks in the program. For each basic block $b \in B$, we create $|L|$ samples by cloning it by l times where $l \in L$. We calculate the difference in the count of each opcode in the basic block between the $(l-1)$-th sample and the l-th sample for any $l \in [2, ..., 5]$. If any of these differences is different from the others, it is not deemed as a consistent basic block. Let this difference be δ. For the last sample ($l = 100$) we expect its difference from the fifth sample to be exactly $95 \cdot \delta$. If it is not, b is ruled out as a consistent basic block.

We do not consider the first difference between the original sample and the one with the basic block cloned once, because the first cloning is performed on the original basic block while the others on a cloned basic block and the first difference may not be consistent with the others. This has implications on our implementation: as shown in Table 3, we keep two different Δ-vectors, one accounting for the difference due to the first transformation (i.e., cloning the basic block only once) and the other for the remaining ones. Function MODIFY in Algorithm 2 updates the feature vectors based on these two Δ-vectors and the number of transformation needed.

The CFG representation at the LLVM IR level compiled from the original Rbot source code contains 4,974 basic blocks, among which we use 194 consistent ones. In the genetic algorithm, each chromosome thus contains 194 elements (i.e., $|M| = 194$).

5.3 Experiments

We have implemented the genetic algorithm in Algorithm 1 with a single thread in Python and use multiple workstations to perform the experiments, each having an Intel i7-4790 8-core 3.60GHz CPU, 32G RAM, and a 2T hard disk and running 64-bit Ubuntu version 16.04. One of these workstations has a Linux-based license to run IDA Pro disassembler version 6.9. To validate the functionality of a malware sample, we execute it in a VM running Windows 7.

Successful rate. Our results for deceiving the opcode malware classifier $C_{mal}^{(1)}$ into targeted misclassification are summarized in Table 4. In total, we have mutated the original Rbot sample to successfully deceive the classifier for *nine* target families, leading to a successful rate of 75%. Even using all the available malware samples as the anchor samples, we could not have find a practical sample to deceive the classifier into classifying it into one of the three families, *Bagle*, *Ldpinch*, and *Lmir*. Our hypothesis is that because the samples in the three malware families are underrepresented (see Figure 1), the task of mutating the original Rbot sample through the genetic algorithm into the feature spaces of these families becomes more difficult than the others. To further validate the functionality of adversarial malware samples created, we monitor the list of API calls made by each sample recorded by the Cuckoo sandbox [2] and find that the number of distinct API calls made by each adversarial malware sample is exactly the same as that of the original Rbot sample, which is 72.

Table 4: Results of targeted misclassification by the opcode classifier. In the "time per anchor" column, if multiple anchor samples are used, x/y means that the amount of time spent on a failed anchor sample and a successful anchor sample is x and y time units, respectively.

Target Family	Successful?	Anchors	Time per anchor	File size (KB)
Benign	✔	1	6s	1,247
Bagle	✗	82	24h	-
Bifrose	✔	1	8s	1,315
Hupigon	✔	1	2m57s	2,044
Koobface	✔	1	1m40s	4,433
Ldpinch	✗	63	24h	-
Lmir	✗	47	24h	-
Sdbot	✔	9	24h/1h44m	489
Swizzor	✔	8	24h/5h8m	320
Vundo	✓	26	24h/1h0m	1,063
Zbot	✔	3	24h/1h20m	20,406
Zlob	✔	1	15m28s	2,298

Execution performance. Table 4 also depicts the number of anchor samples used for each target family as well as the average execution time spent on each anchor sample. The execution time excludes validation time through dynamic analysis, because in our experiments whenever an adversarial sample was found to deceive the classifier successfully, it always passed validation. Generally speaking, if the genetic algorithm fails to find any adversarial sample that can deceive the opcode classifier into targeted misclassification, its execution time is approximately 24 hours. For each of the five target families with only one anchor sample needed to find a successful adversarial sample, the execution time of the genetic algorithm is short, varying from a few seconds to 16 minutes. For those target families using multiple anchors to achieve targeted misclassification, it may take the genetic algorithm several hours to find a successful variant.

File sizes. One side effect of cloning basic blocks to achieve targeted misclassification is that it increases the malware size. Table 4 depicts the file size of each malware sample successfully created for targeted misclassification. Compared with the sample compiled with clang++, on average the size of each adversarial sample has increased by 13.1 times, while the largest one, which is created to target the Zbot family, is 77 times as large as the original one.

6 DECEIVING API CLASSIFIER

In this section, we show how to deceive classifier $C_{mal}(A)$ into targeted misclassification.

6.1 Methodology

As API function names provide hints about malware behavior, various API obfuscation techniques have been used to confuse malware analysts [60]. The robustness of these techniques varies, but for the purpose of deceiving malware classifier $C_{mal}(A)$ into targeted misclassification, we only need an API obfuscation technique that can prevent the API function feature extraction tool from finding a selected list of API functions actually used by the malware.

Let V_{api} be an ordered set that includes all the API functions that can be called by a PE program. For malware classifier $C_{mal}(A)$, the feature vector of the original malware is given as X_p which is a vector of size $|V_{api}|$ including binary values indicating whether the API functions in V_{api} have been used by the original PE sample.

By abusing notation M first used in Section 5.1, we let it denote an obfuscation vector of size $|V_{api}|$. Each element in the obfuscation vector M takes only binary values, indicating whether the corresponding API function in the ordered set V_{api} should be obfuscated or not. Hence, the feature vector of the obfuscated sample becomes $X_p \odot M$ where operator $\odot$ means element-wise multiplication.

To find an adversarial sample that can deceive malware classifier $C_{mal}(A)$ into targeted misclassification, we use the same genetic algorithm as shown in Algorithm 1 along with different support functions defined in Algorithm 3. Similar to our method for deceiving the opcode classifier, each chromosome h is an instantiation of vector M. The fitness score of a feature vector $X_{p'}$ is defined to be its L2 distance (i.e., Euclidean distance) from that of the anchor sample X_q.

Algorithm 3 Support functions for deceiving API classifier

```
function MODIFY($X_p$, $h$)
    return $X_p \odot h$                      ▷ Element-wise product
end function
function FITNESS($X_{p'}$, $X_q$)
    return $D_{L2}(X_{p'}, X_q)$              ▷ $L_2$ distance
end function
```

We define a subset of API function features among V_{api} that are allowed to be mutated in the genetic algorithm and use set I to include their indices in V_{api}. We call I the *operable set.* As the chromosome h takes only binary values, the three genetic operations are slightly different from those in deceiving the opcode classifier. Each generation contains 600 individuals. In each generation, the best 200 chromosomes are selected onto list L. We have the following three types of genetic operators. (1) *Full mutation:* For each element in vector M that belongs to the operable set, it is randomly assigned to be 0 or 1. We perform full mutation for 200 times. (2) *Partial mutation:* For each element in vector M that belongs to the operable set, we choose to mutate it with probability 0.5. If it is chosen to be mutated, its value is randomly chosen between 0 and 1. We perform partial mutation on each of the 200 chromosomes on list L. (3) *Crossover:* We randomly choose two chromosomes h_0 and h_1 from list L to create a new variant. For each element in vector M that belongs to the operable set, we randomly choose its value from h_0 and h_1. For those elements not in the operable set, their values must be the same for h_0 and h_1, which are copied into the new variant. The crossover operation is performed for 200 times.

An alternative greedy algorithm. It is noted that the genetic algorithm augmented with support functions in Algorithm 3 tries to find an optimal feature vector that has the shortest L2 distance from that of the anchor sample. As we only obfuscate API functions that appear in the operable set, there exists a *greedy algorithm* that directly obtains the unique optimal solution: for each API function in the operable set, we obfuscate it in the adversarial sample *if and only if* its value in the anchor sample's feature vector (e.g., X_q) is 0, meaning that this API function is either obfuscated or not used in the anchor sample. Later we shall demonstrate through experiments that the genetic algorithm outperforms the greedy algorithm in finding adversarial samples that can successfully deceive the API classifier into targeted misclassification.

6.2 Implementation

As we use IDA Pro [4] to extract imported API functions, we use a simple API obfuscation technique discussed in [60]. For each Windows API function used by the Rbot source code, if it is chosen for obfuscation, we apply the `GetProcAddress` method from *kernel32.dll* to obtain its function address, which is further used to replace every direct call to the API function in the source code. Interestingly, the original Rbot source code has used this technique to obfuscate a subset of the Windows API functions used by it. We remove these obfuscations in the original source code and use this as the *baseline sample*, whose size is 211KB after compilation with MSVC++ in a Windows 7 VM. Using the baseline sample, we obfuscate the API functions based on the solution found by the genetic algorithm to create adversarial samples. In addition to indirect API function calls, we also obfuscate the DLL name strings and API function name strings with the base-64 encoding scheme. Although these obfuscation techniques are rudimentary, they are sufficient to fool the IDA Pro tool [4] used for feature extraction.

From the original Rbot sample we can extract 103 API functions directly, but its source code has obfuscated 110 others. We remove these obfuscations and obtain a total amount of 213 imported API functions. We observe that 50 of these API functions cannot be obfuscated through the aforementioned technique. Hence, the operable set I used by the genetic algorithm contains 163 indices.

6.3 Experiments

Successful rate. We use the same workstations as mentioned in Section 5.3 to perform the experiments aimed at deceiving the API classifier $C_{mal}(A)$. The experimental results are summarized in Table 5. Using the genetic algorithm, we have deceived malware classifier $C_{mal}(A)$ into targeted misclassification for 10 families, leading to a successful rate of 83.3%. The two exceptions are the Bagle and Vundo families: even using all the samples from these two families in the malware dataset as anchor samples, the genetic algorithm still could not find the adversarial samples achieving targeted misclassification.

Execution performance. Table 5 shows that for each family with successful targeted misclassification, only a single anchor sample is needed from the target family, and the execution time spent by the genetic algorithm is short, varying from a few seconds to three minutes. For the Bagle and Vundo families without successful targeted misclassification, the execution time spent by the genetic algorithm on each anchor sample can be as long as over three hours.

File sizes. We also compare the file sizes of the adversarial samples against that of the baseline sample in which no API function obfuscations are used. Table 5 tells us that due to API function

Table 5: Performance comparison of the greedy algorithm and the genetic algorithm in searching adversarial samples

Target	Genetic Algorithm				Greedy Algorithm			
Family	Successful?	Anchors	Time per Anchor	File size	Successful?	Anchors	Time per Anchor	File size
Benign	✓	1	5s	217 KB	✓	2	0.05s	220 KB
Bagle	✗	all	~3h12m	-	✗	all	0.05s	-
Bifrose	✓	1	1m24s	217 KB	✗	all	0.05s	-
Hupigon	✓	1	16s	216 KB	✓	226	0.05s	217 KB
Koobface	✓	1	5s	217 KB	✓	1	0.05s	221 KB
Ldpinch	✓	1	2m31s	217 KB	✗	all	0.05s	-
Lmir	✓	1	6s	217 KB	✓	32	0.05s	220 KB
Sdbot	✓	1	5s	216 KB	✓	3	0.05s	219 KB
Swizzor	✓	1	9s	217 KB	✓	279	0.05s	217 KB
Vundo	✗	all	~3h14m	-	✗	all	0.05s	-
Zbot	✓	1	15s	217 KB	✓	12	0.05s	218 KB
Zlob	✓	1	8s	216 KB	✗	all	0.05s	-

obfuscation each adversarial sample successfully created is slightly larger than the baseline one; on average, the increase in file size is 2.9% over the 10 adversarial samples successfully created.

Comparison with the greedy algorithm. As mentioned earlier, given the feature vector X_q of the anchor sample, a greedy algorithm can directly find an optimal feature vector for an adversarial sample whose L2-distance from X_q is minimized. Table 5 compares the performances of the greedy algorithm and the genetic algorithm in searching successful adversarial samples. Although, unsurprisingly, the greedy algorithm spends much shorter time (less than one second) on finding the optimal API call feature vector directly for a given anchor sample, it finds adversarial samples that achieve targeted misclassification successfully for only seven families, leading to a successful rate of 58.3%. Moreover, even for those families that creation of adversarial samples is successful, more anchor samples are needed; on average 79.3 anchor samples are used to find a successful adversarial sample for each target family. In contrast, for the genetic algorithm its successful rate is much higher and for those successful families only a single anchor sample was used. Our results suggest that *the evolution process in the genetic algorithm is instrumental in finding successful adversarial examples.*

7 DECEIVING SYSTEM CALL CLASSIFIER

This section describes how to deceive classifier $C_{mal}(S)$.

7.1 Methodology

Different from the previous two malware classifiers, $C_{mal}(S)$ is trained from features extracted from dynamic execution of malware samples in a sandbox. Our work assumes that the sandbox environment can be inferred by the attacker through techniques such as wear-and-tear artifacts [39]. Once the sandbox environment is known, the attacker can predict the execution path of a malware sample executed inside it. When deceiving classifier $C_{mal}(S)$ into targeted misclassification, a major challenge is that system calls *cannot* be added directly to the source code of the malware. To circumvent this problem, we consider adding a selected set of API calls A on the predicted execution path of the malware program. We provide dummy inputs to these API calls so their executions would not affect the functionality of the original malware program. For each of these API calls, we monitor the difference in the system calls invoked by the malware. More specifically, for each API call $a \in A$, we define $\Delta(a)$ to denote the changes in the count of system calls invoked by the malware if a is added to the execution path once. Accordingly we define the Δ-vector to be $\{\Delta(a) \mid \forall a \in A\}$.

We use the same genetic algorithm described in Algorithm 1 to search adversarial samples that can deceive $C_{mal}(S)$ into targeted misclassification. The *count vector* of malware program p, denoted by X_p, includes the number of times each system call has been invoked by the malware in the sandbox. In the genetic algorithm, each chromosome h indicates the number of times each API should be added to the execution path of the malware.

Algorithm 4 Support functions for deceiving system call classifier

```
function MODIFY($X_p$, $h$)
    return $X_p + h \cdot \Delta$-vector
end function
function FITNESS($X_{p'}$, $X_q$)
    return $D_{JS}(H(X_{p'}), H(X_q))$        ▷ JS divergence
end function
```

The support functions for deceiving system call classifier are given in Algorithm 4. When the MODIFY function is called with the original count vector X_p and chromosome h, it returns a count vector that includes the number of times each system call is expected to be invoked by the sample mutated from malware program p with chromosome h. Also, given the feature X_p, let $H(X_p)$ denote its corresponding frequency histogram of all the system calls. From the two count vectors $X_{p'}$ and X_q, we first obtain their corresponding frequency histograms $P' = H(X_{p'})$ and $Q = H(X_q)$. The fitness score is given by the Jensen-Shannon divergence between P' and Q, which is defined as follows:

$$D_{JS}(P', Q) = \sqrt{\frac{D_{KL}(P', \frac{P'+Q}{2}) + D_{KL}(Q, \frac{P'+Q}{2})}{2}} \tag{5}$$

In the genetic algorithm, the population size (i.e., $|H|$) is 1000. In the initial population, each element in the chromosome is randomly chosen between 0 and 1000. The maximum number of iterations (i.e., n) is set to be 100. Parameter t is set to be 200. The same

Table 6: Results of targeted misclassification by the system call classifier. They use the same notations as in Table 4.

Target Family	Success-ful?	Anchors	Time per anchor	File size (KB)
Benign	✓	1	4s	216
Bagle	✓	1	57s	217
Bifrose	✓	1	1s	216
Hupigon	✓	1	4s	216
Koobface	✓	1	56s	216
Ldpinch	✓	2	1h58m/47s	217
Lmir	✓	1	8s	217
Sdbot	✓	1	1s	216
Swizzor	✓	27	1h58m/43m33s	217
Vundo	✗	all	~2h	-
Zbot	✓	3	1m43s	217
Zlob	✓	1	1m38s	216

three genetic operators (i.e., full mutation, partial mutation, and crossover) are used as in deceiving the opcode classifier.

When the genetic algorithm finds an appropriate feature vector $X_{p'}$ that can deceive classifier $C_{mal}(s)$ successfully, we use the corresponding chromosome h to generate adversarial malware sample p'. Recall that each element in h indicates how many times each API call should be added to the execution path.

7.2 Implementation

We use the implementation of JS divergence in SciPy [1] for Eq. (5). To choose the API calls for influencing the system calls invoked, we start from more than 100 candidate API calls. We then eliminate those that do not trigger any system call or do not result in consistent changes of system calls when invoked by multiple times. We also exclude those API calls whose executions depend on the presence of others. For example, in order to use the `socket` API from the `ws2_32.lib` library to create a socket, the Winsock library must be initialized by calling the `WSAStartup` function. The eventual set of API calls used in our experiments includes 43 ones.

7.3 Experiments

Our experimental results are summarized in Table 6. Our method has deceived $C_{mal}(A)$ into targeted misclassification for all 12 families except Vundo, leading to a successful rate of 91.7%. It is also noted that for the majority of these families, only a few anchor samples are needed to find a successful adversarial malware sample, suggesting that the genetic algorithm has been effective in most cases. Compared with the results seen in Table 4, the adversarial samples created to deceive the system call classifier does not have the side effect of bloating the malware file sizes.

8 DISCUSSIONS

Although our work successfully creates practical samples for deceiving malware classifiers, its limited scope is discussed as follows.

First, this work has considered only three malware classifiers, each using a random forest, but many other malware classifiers can be trained based on different malware features. In our future work we will investigate the robustness of other types of malware classifiers, such as those based on deep learning methods, against adversarial machine learning attacks. This study shows that different techniques are needed to achieve targeted malware misclassification, suggesting that using an ensemble of malware classifiers can raise the bar for adversarial machine learning attacks against them.

Second, the genetic algorithm has been shown successful in creating adversarial samples to achieve targeted misclassification for a majority of malware families. Although our study has not found as much success with other methods such as simulated annealing and mixed integer programming solvers, it does not mean that the genetic algorithm should be the optimal solution. We plan to explore other solution searching methods in our future work.

Third, existing research on adversarial machine learning focuses on image classification, for which similarity between original and adversarial examples is crucial and thus used as a constraint for finding adversarial examples. As this constraint is unnecessary for creation of adversarial malware samples, without it the optimization problem becomes easier to solve. In contrast, the constraint for adversarial malware creation is keeping its functionality and practicality, as we cannot directly modify the features upon which malware classifiers are trained. Because such a constraint cannot be formulated into the optimization problem directly, we have chosen to use dynamic analysis to verify whether it is satisfied.

Fourth, underground malware authors are usually more interested in malware evasion than the generic problem of malware deception. For malware evasion, malware packers are commonly used to hide details of malware programs through encryption, compression, or code obfuscation. We are however witnessing increasing use of malware as cyber weapons among groups, organizations, or even nations. In such scenarios, deceptive malware samples may be used as tactical tools to mislead the target's malware mitigation efforts. This study has demonstrated the possibility of creating not only evasive but also deceptive malware samples.

9 CONCLUSIONS

In this work, we have studied a new type of threats aimed at deceiving a multi-class PE malware classifiers into targeted misclassification. This study uses a malware dataset with tens of thousands of PE samples belonging to 13 families. From this malware dataset we extracted three types of malware features, the first one based on frequencies of opcodes in the disassembled malware code, the second one the list of API functions imported by a PE sample, and the third one the list of system calls observed in dynamic execution. We construct three random forest classifiers accordingly. Our study has shown that using a genetic algorithm augmented with different support functions, we can create adversarial PE samples that can deceive the three classifiers into targeted misclassification with a successful rate of 75%, 83.3%, and 91.7% respectively.

ACKNOWLEDGMENT

We acknowledge NSF Award CNS-1618631 for supporting this work.

REFERENCES

[1] https://scipy.github.io/devdocs/generated/scipy.spatial.distance.jensenshannon.html.
[2] Cuckoo Sandbox. https://cuckoosandbox.org.
[3] FakeNet. https://github.com/fireeye/flare-fakenet-ng.
[4] IDA Pro. https://www.hex-rays.com/.
[5] Microsoft Malware Classification Challenge (BIG 2015). https://www.kaggle.com/c/malware-classification.
[6] Pin - A Dynamic Binary Instrumentation Tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[7] Rbot source code 0.0.3. https://github.com/ytisf/theZoo/tree/master/malwares/Source/Original/rBot0.3.3_May2004.

[8] Stuxnet. https://en.wikipedia.org/wiki/Stuxnet.

[9] VirusShare.com - Because Sharing is Caring. https://virusshare.com/.

[10] VirusTotal. https://www.virustotal.com/.

[11] Y. Aafer, W. Du, and H. Yin. Droidapiminer: Mining api-level features for robust malware detection in android. In *International Conference on Security and Privacy in Communication Systems*, pages 86–103. Springer, 2013.

[12] A. Abusnaina, A. Khormali, H. Alasmary, J. Park, A. Anwar, U. Meteriz, and A. Mohaisen. Examining adversarial learning against graph-based iot malware detection systems. *arXiv preprint arXiv:1902.04416*, 2019.

[13] F. Ahmed, H. Hameed, M. Z. Shafiq, and M. Farooq. Using spatio-temporal information in api calls with machine learning algorithms for malware detection. In *Proceedings of the ACM workshop on Security and artificial intelligence*, 2009.

[14] A. Al-Dujaili, A. Huang, E. Hemberg, and U.-M. O'Reilly. Adversarial deep learning for robust detection of binary encoded malware. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 76–82. IEEE, 2018.

[15] H. Anderson, A. Kharkar, B. Filar, D. Evans, and P. Roth. Learning to evade static PE machine learning malware models via reinforcement learning. *arXiv preprint arXiv:1801.08917*, 2018.

[16] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck. Drebin: Effective and explainable detection of android malware in your pocket. In *Proceedings of the Network and Distributed System Security Symposium*, 2014.

[17] AV-TEST. Malware statistics & trends report. https://www.av-test.org/en/statistics/malware/, Accessed in March 2018.

[18] B. Biggio, I. Corona, D. Maiorca, B. Nelson, N. Šrndić, P. Laskov, G. Giacinto, and F. Roli. Evasion attacks against machine learning at test time. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, 2013.

[19] D. Bilar. Opcodes as predictor for malware. *International Journal of Electronic Security and Digital Forensics*, 1(2):156–168, 2007.

[20] D. Canali, A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda. A quantitative study of accuracy in system call-based malware detection. In *International Symposium on Software Testing and Analysis*. ACM, 2012.

[21] R. J. Canzanese Jr. *Detection and Classification of Malicious Processes Using System Call Analysis*. PhD thesis, Drexel University, 2015.

[22] N. Carlini and D. Wagner. Towards evaluating the robustness of neural networks. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 39–57. IEEE, 2017.

[23] S. Cesare and Y. Xiang. Classification of malware using structured control flow. In *Proceedings of the Eighth Australasian Symposium on Parallel and Distributed Computing-Volume 107*, pages 61–70. Australian Computer Society, Inc., 2010.

[24] P.-Y. Chen, Y. Sharma, H. Zhang, J. Yi, and C.-J. Hsieh. Ead: elastic-net attacks to deep neural networks via adversarial examples. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[25] A. Demontis, M. Melis, B. Biggio, D. Maiorca, D. Arp, K. Rieck, I. Corona, G. Giacinto, and F. Roli. Yes, machine learning can be more secure! a case study on android malware detection. *IEEE Transactions on Dependable and Secure Computing*, 2019.

[26] Y. Dong, F. Liao, T. Pang, H. Su, J. Zhu, X. Hu, and J. Li. Boosting adversarial attacks with momentum. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 9185–9193, 2018.

[27] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[28] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel. Adversarial examples for malware detection. In *European Symposium on Research in Computer Security*, pages 62–79. Springer, 2017.

[29] S. Hou, Y. Ye, Y. Song, and M. Abdulhayoglu. Hindroid: An intelligent android malware detection system based on structured heterogeneous information network. In *Proceedings of ACM Conference on Knowledge Discovery and Data Mining*, pages 1507–1515, 2017.

[30] W. Hu and Y. Tan. Generating adversarial malware examples for black-box attacks based on gan. *arXiv preprint arXiv:1702.05983*, 2017.

[31] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin. Obfuscator-LLVM – software protection for the masses. In B. Wyseur, editor, *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection*, 2015.

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

[33] J. Z. Kolter and M. A. Maloof. Learning to detect malicious executables in the wild. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 470–478. ACM, 2004.

[34] D. Kong and G. Yan. Discriminant malware distance learning on structural information for automated malware classification. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2013.

[35] F. Kreuk, A. Barak, S. Aviv-Reuven, M. Baruch, B. Pinkas, and J. Keshet. Deceiving end-to-end deep learning malware detectors using adversarial examples. *arXiv preprint arXiv:1802.04528*, 2018.

[36] J. Lin. Divergence measures based on the shannon entropy. *IEEE Transactions on Information theory*, 37(1):145–151, 1991.

[37] X. Liu, Y. Lin, H. Li, and J. Zhang. Adversarial examples: Attacks on machine learning-based malware visualization detection methods. *arXiv preprint arXiv:1808.01546*, 2018.

[38] S. B. Mehdi, A. K. Tanwani, and M. Farooq. Imad: in-execution malware analysis and detection. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1553–1560. ACM, 2009.

[39] N. Miramirkhani, M. P. Appini, N. Nikiforakis, and M. Polychronakis. Spotless sandboxes: Evading malware analysis systems using wear-and-tear artifacts. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 1009–1024. IEEE, 2017.

[40] A. Mohaisen, O. Alrawi, and M. Mohaisen. Amal: High-fidelity, behavior-based automated malware analysis and classification. *Computers & Security*, 52, 2015.

[41] S.-M. Moosavi-Dezfooli, A. Fawzi, and P. Frossard. Deepfool: a simple and accurate method to fool deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[42] R. Moskovitch, C. Feher, N. Tzachar, E. Berger, M. Gitelman, S. Dolev, and Y. Elovici. Unknown malcode detection using opcode representation. In *European conference on intelligence and security informatics*. Springer, 2008.

[43] N. Nissim, A. Cohen, C. Glezer, and Y. Elovici. Detection of malicious pdf files and directions for enhancements: a state-of-the art survey. *Computers & Security*, 48:246–266, 2015.

[44] 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, 2016.

[45] R. Perdisci, A. Lanzi, and W. Lee. Mcboost: Boosting scalability in malware collection and analysis using statistical classification of executables. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2008.

[46] K. Raman. Selecting features to classify malware. In *Proceedings of InfoSec Southwest*, 2012.

[47] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov. Learning and classification of malware behavior. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2008.

[48] 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 *International Symposium on Research in Attacks, Intrusions, and Defenses*.

[49] C. Rossow, C. J. Dietrich, C. Grier, C. Kreibich, V. Paxson, N. Pohlmann, H. Bos, and M. Van Steen. Prudent practices for designing malware experiments: Status quo and outlook. In *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012.

[50] J. Sahs and L. Khan. A machine learning approach to android malware detection. In *2012 European Intelligence and Security Informatics Conference*. IEEE, 2012.

[51] A. Sami, B. Yadegari, H. Rahimi, N. Peiravian, S. Hashemi, and A. Hamze. Malware detection based on mining api calls. In *Proceedings of the ACM Symposium on Applied Computing*. ACM, 2010.

[52] M. G. Schultz, E. Eskin, E. Zadok, and S. J. Stolfo. Data mining methods for detection of new malicious executables. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2001.

[53] A. Shabtai, R. Moskovitch, C. Feher, S. Dolev, and Y. Elovici. Detecting unknown malicious code by applying classification techniques on opcode patterns. *Security Informatics*, 1(1):1, 2012.

[54] M. Z. Shafiq, S. M. Tabish, F. Mirza, and M. Farooq. PE-Miner: Mining structural information to detect malicious executables in realtime. In *International Symposium on Recent Advances in Intrusion Detection*. Springer-Verlag, 2009.

[55] F. Shahzad and M. Farooq. Elf-miner: Using structural knowledge and data mining methods to detect new (linux) malicious executables. *Knowledge and information systems*, 30(3):589–612, 2012.

[56] C. Smutz and A. Stavrou. Malicious pdf detection using metadata and structural features. In *Annual Computer Security Applications Conference*. ACM, 2012.

[57] C. Smutz and A. Stavrou. When a tree falls: Using diversity in ensemble classifiers to identify evasion in malware detectors. In *Proceedings of the Network and Distributed System Security Symposium*, 2016.

[58] N. Šrndić and P. Laskov. Practical evasion of a learning-based classifier: A case study. In *Proceedings of the IEEE symposium on security and privacy*, 2014.

[59] N. Šrndić and P. Laskov. Hidost: a static machine-learning-based detector of malicious files. *EURASIP Journal on Information Security*, 2016(1):22, 2016.

[60] M. Suenaga. A museum of API obfuscation on win32. *Symantec Security Response*, 2009.

[61] Y. Vorobeychik and M. Kantarcioglu. Adversarial machine learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 12(3):1–169, 2018.

[62] W. Xu, Y. Qi, and D. Evans. Automatically evading classifiers. In *Proceedings of the 2016 Network and Distributed Systems Symposium*, 2016.

[63] G. Yan, N. Brown, and D. Kong. Exploring discriminatory features for automated malware classification. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 41–61. Springer, 2013.

[64] W. Yang, D. Kong, T. Xie, and C. A. Gunter. Malware detection in adversarial settings: Exploiting feature evolutions and confusions in android apps. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, 2017.

[65] Y. Ye, D. Wang, T. Li, and D. Ye. IMDS: Intelligent malware detection system. In *International conference on Knowledge Discovery and Data Mining*, 2007.