Automatically Evading Classifiers
A Case Study on PDF Malware Classifiers

Weilin Xu, Yanjun Qi, and David Evans
University of Virginia
http://www.EvadeML.org

Abstract—**Machine learning is widely used to develop classifiers for security tasks. However, the robustness of these methods against motivated adversaries is uncertain. In this work, we propose a generic method to evaluate the robustness of classifiers under attack. The key idea is to stochastically manipulate a malicious sample to find a variant that preserves the malicious behavior but is classified as benign by the classifier. We present a general approach to search for evasive variants and report on results from experiments using our techniques against two PDF malware classifiers, PDFrate and Hidost. Our method is able to automatically find evasive variants for both classifiers for *all* of the 500 malicious seeds in our study. Our results suggest a general method for evaluating classifiers used in security applications, and raise serious doubts about the effectiveness of classifiers based on superficial features in the presence of adversaries.**

I. Introduction

Machine learning models are popular in security tasks such as malware detection, network intrusion detection and spam detection. From the data scientists' perspective, these models are effective since they achieve extremely high accuracy on test datasets. For example, Dahl et al. reported achieving 99.58% accuracy in classifying Win32 malware using an ensemble deep neural network with dynamic features [9]. Šrndic et al. achieved over 99.9% accuracy in a PDF malware classification task using an SVM-RBF model with structural path features [28].

However, it is important to realize that these results are for particular test datasets. Unlike when machine learning is used in other fields, security tasks involve adversaries responding to the classifier. For example, attackers may try to generate new malware deliberately designed to evade existing classifiers. This breaks the assumption of machine learning models that the training data and the operational data share the same data distribution. As a result, it is important to be skeptical of machine learning results in security contexts that do not consider attackers' efforts to evade the generated models.

The risk of evasion attacks against machine learning models under adversarial settings has been discussed in the machine learning community, mainly focused on simple models for spam detection (e.g., [10, 18]). However, evasion attacks against malware classification can be much more complex in terms of the classification algorithm and the feature extraction as well as the mutability of highly-structured samples. Consequently, though evading malware classifiers has been partially explored by classifier authors as well as security researchers, previous studies significantly under-estimate the attackers' ability to manipulate samples. For example, previous studies may mistakenly assume the attackers can only insert new contents because removing existing contents would easily corrupt maliciousness [4, 20, 28]. In addition, previous works are ad hoc and limited to particular target classifiers or specific types of samples [20, 29]. Other than suggesting point solutions, they do not provide methods to automatically evaluate the effectiveness of a classifier against adaptive adversaries.

We present a generic method to assess the robustness of a classifier by simulating attackers' efforts to evade the classifier. We do not assume the adversary has any detailed knowledge of the classifier or the features it uses, or can use targeted expert knowledge to manually direct the search for an evasive sample. Instead, drawing ideas from genetic programming (GP) [11, 15], we perform stochastic manipulations and then evaluate the generated variants to select promising ones. By repeating this procedure iteratively, we aim to generate evasive variants. A sophisticated attacker, of course, can do manipulations that would not be found by a stochastic search, so we cannot claim that a classifier that resists such an attack is necessarily robust. On the other hand, if the automated approach finds evasive samples for a given classifier, it is a clear sign that the classifier is not robust against a motivated adversary.

We evaluated the proposed method on two PDF malware classifiers, and found that it could automatically find evasive variants for all the 500 sample seeds selected from the Contagio PDF malware archive [5]. The evasive variants exhibit the same malicious behaviors as the original samples, but are sufficiently different in the classifier's feature space to be classified as benign by the machine learning-based models.

Our analysis of the discovered evasive variants reveals that both classifiers are vulnerable because they employ non-robust features, which can be manipulated without disrupting the desired malicious behavior. Superficial features may work well on test datasets, but if the features used to classify malware are shallow artifacts of the training data rather than intrinsic properties of malicious content, it is possible to find ways to preserve the malicious behavior while disrupting the features.

Contributions. Our primary contributions involve developing and evaluating a general method for automatically finding variants that evade classifiers. In particular:

- We propose a general method to automatically find evasive variants for target classifiers. The method does not rely on any specific classification algorithms or assume detailed knowledge of feature extraction, but only needs the classification score feedback on

NDSS '16, 21-24 February 2016, San Diego, CA, USA
Copyright 2016 Internet Society, ISBN 1-891562-41-X
http://dx.doi.org/10.14722/ndss.2016.23115

generated variants and rough knowledge of the likely features used by the classifier (Section II).

- We implement a prototype system that automatically finds variants that can evade structural feature-based PDF malware classifiers. This involves designing operators that perform stochastic manipulations on PDF files, an oracle that determines if a generated variant preserves maliciousness, a selection mechanism that promotes promising variants during the evolutionary process, and a fitness function for each target classifier (Section IV).
- We evaluate the effectiveness of our system in evading two recent PDF malware classifiers: PDFrate [25] and Hidost [28], a classifier designed with the explicit goal of resisting evasion attempts. Our system achieves 100% success rates in finding evasive variants against both classifiers in an experiment with 500 malware sample seeds. An analysis of the discovered evasive variants in the feature space of each classifier shows that many non-robust features employed in the classification facilitate evasion attacks (Sections V and VI).

We provide background on machine learning classifiers in Section II and on PDF malware in Section III. Section VIII discusses related work on evasion attacks.

II. Overview

We propose an automated method to simulate an attacker attempting to find an evasive variant for a desired malware sample which is detected by a target classifier. The attacker's goal is to find a malware variant that preserves the malicious behavior of the original sample, but that is misclassified as benign by the target classifier. In addition to improving our understanding of how classifiers work in the presence of adaptive adversaries, we hope our results will lead to strategies for constructing classifiers that are more robust to adversaries, but in this work we focus on assessing evadability.

A. Machine Learning Classifiers

Machine learning learns from and makes predictions on data. A machine learning-based classifier attempts to find a hypothesis function f that maps data points into different classes. For example, a malware classification system would find a hypothesis function f that maps a data point (a piece of malware sample) into either *benign* or *malicious*.

The effort to train a machine learning system starts with feature extraction. As most machine learning algorithms cannot operate on highly-structured data, the data samples are usually represented in a specially-designed feature space. For example, a malware classifier may extract the file size and the function call traces as features. Each feature is a dimension in the feature space; consequently, every sample is represented as a vector. An extra step of feature selection may be performed to reduce the number of features when the number of features is too large for the classification algorithm.

The most widely used machine learning algorithms in security tasks use *supervised learning*, in which the training dataset comes with labels identifying the class of every training sample. The hypothesis function f is trained to minimize the prediction error on the training set. This function usually results in a low error rate on the operational data under the *stationarity* assumption that the distribution over data points encountered in the future will be the same as the distribution over the training set.

Machine learning has produced impressive results and is widely deployed for specific security tasks including malware classification. Without examining the behavior of suspicious malware in a real system, malware classifiers often employ static properties to predict maliciousness such as the file structure, file size, metadata, grams of tokens or system calls. Although this approach often achieves high accuracy in validation tests, the classifier may learn properties that are superficial artifacts of the training data, rather than properties that are inherently associated with malware. This is because malware samples in the training data are likely to differ from the benign samples in many ways that are not essential to their malicious behavior.

B. Threat Model

We assume an attacker starts with a desired malicious sample that is (correctly) classified by a target classifier as malicious, and wants to create a sample with the same malicious behavior, but that is misclassified as benign. The attacker is capable of manipulating the malicious sample in many ways, and is likely to have knowledge of samples that are (correctly) classified as benign.

We assume the attacker has black-box access to the target classifier, and can submit many variants to that classifier. For each submitted variant, the attacker learns its classification score. The classification score is a number (typically a real number between 0 and 1) that indicates the classifier's prediction of maliciousness, where values above some threshold (say 0.5) are considered malicious and samples with lower classification scores are considered benign. We do not assume the attacker has any internal information about the classifier, only that it can use it as a black-box that outputs the classification score for an input sample. We assume the classifier operator does not adapt the classifier to submitted variants (which must be the case if the attacker has offline access to the classifier).

C. Finding Evasive Samples

Our method uses genetic programming techniques to perform a directed search of the space of possible samples to find ones that evade the classifier while retaining the desired malicious behavior.

Genetic programming (GP) is a type of evolutionary algorithm, originally developed for automatically generating computer programs tailored to a particular task [11, 15]. It is essentially a stochastic search method using computational analogs of biological mutation and crossover to generate variants, and modeling Darwinian selection using a user-defined *fitness function*. Variants with higher fitness are selected for continued evolution, and the process continues over multiple generations until a variant with desired properties is found (or the search is terminated after exceeding a resource bound). Genetic programming has been shown to be effective in many

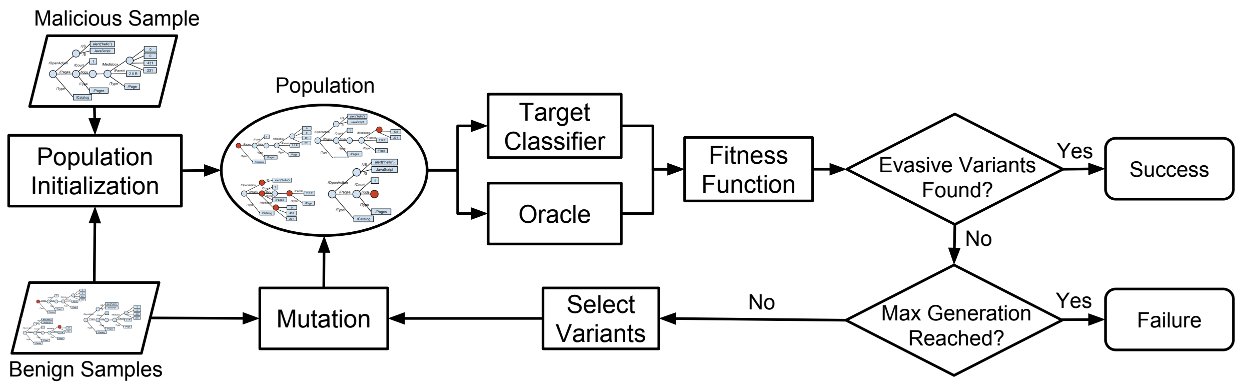


Fig. 1. Generic classifier evasion method.

tasks including fixing legacy software bugs [17], software reverse engineering [13], and software re-engineering [23].

Method. Our procedure is illustrated in Figure 1. It starts with a seed sample that exhibits malicious behavior, and is classified as malicious by the target classifier. Our method aims to find an evasive sample that preserves the malicious behavior but is misclassified as benign by the target classifier.

First, we initialize a population of variants by performing random manipulations on the malicious seed. Then, each variant is evaluated by a target classifier as well as an oracle. The *target classifier* is a black box that outputs a number that is a measure of predicted maliciousness of an input sample. There is a prescribed threshold used to decide if it is malicious or benign. The *oracle* is used to determine if a given sample exhibits particular malicious behavior. In most instantiations, the oracle will involve expensive dynamic tests.

A variant that is classified as benign by the target classifier, but found to be malicious by the oracle, is a successful evasive sample. If no evasive samples are found in the population, a subset of the generated variants are selected for the next generation based on a fitness measure designed to reflect progress towards finding an evasive sample. Since it is unlikely that the transformations will re-introduce malicious behaviors into a variant, corrupted variants that have lost the malicious behavior are replaced with other variants or the original seed.

Next, the selected variants are randomly manipulated by mutation operators to produce next generation of the population. The process continues until an evasive sample is found or a threshold number of generations is reached.

To improve the efficiency of the search, we collect traces of the mutation operations used and reuse effective traces. If a search ends up finding any evasive variants, the mutation traces on the evasive variants will be stored as successful traces. Otherwise, the mutation trace of a variant with the highest fitness score is stored. These traces are then applied to other malware seeds to generate variants for their population initialization. Because of the structure of PDFs and the nature of the mutation operators, the same sequence of mutations can often be applied effectively to many initial seeds.

III. PDF Malware and Classifiers

This section provides background on PDF malware and the two target PDF malware classifiers.

A. PDF Malware

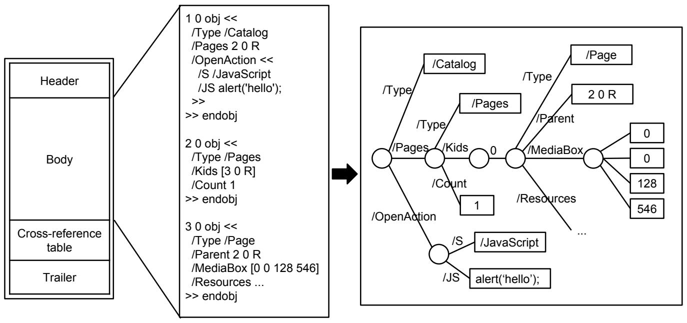
The *Portable Document Format* (PDF) is a popular document format designed to enable consistent content and layout in rendering and printing on different platforms. Although it was not openly standardized until 2008 [1], and there are various non-standard extensions supported by different PDF reader products, all PDF files roughly share the same basic structure depicted in Figure 2.

A PDF file consists of four parts: *header*, *body*, *cross-reference table* (CRT) and *trailer*. The *header* contains the PDF magic number and a format version indicator. The *body* is a set of PDF objects that comprise the content of the file, while the CRT indexes the objects in *body*. The *trailer* specifies how to find the CRT and other special objects such as the root object. Thus, PDF readers typically start reading a PDF from the end of the file for efficiency.

The *body* is the most important to a PDF since it holds almost all the visible document data. It contains eight basic types of objects, namely Booleans, numbers, strings, names, arrays, dictionaries, streams and the null objects. The objects can be labeled with a pair of integer identifiers as indirect objects so that they can be referenced by other objects. The inter-referencing objects form a tree-like logical structure, as is shown in the right of Figure 2. This tree-like structure is ideally suited to genetic programming techniques since it is easy to alter and move sub-trees to generate new variants.

PDF malware is becoming increasing prevalent because PDF is a widely accepted document format and victims are more willing to open PDFs than other files. According to a recent Internet security threat report [30], PDF is in top 7 attachment types in spear-phishing emails in 2014. We expect there will be continuing opportunities for PDF malware attacks because 128 new vulnerabilities in Acrobat readers have been reported in CVE so far in 2015 (through 8 December), which is almost three times the total number in 2014 [8].

PDF malware typically contains exploits in JavaScript objects or other objects that take advantage of vulnerabilities of particular PDF readers (most commonly, Adobe Acrobat).



```
1 0 obj <<
 /Type /Catalog
 /Pages 2 0 R
 /OpenAction <<
  /S /JavaScript
  /JS alert('hello');
 >>
>> endobj

2 0 obj <<
 /Type /Pages
 /Kids [3 0 R]
 /Count 1
>> endobj

3 0 obj <<
 /Type /Page
 /Parent 2 0 R
 /MediaBox [0 0 128 546]
 /Resources ...
>> endobj
```

Fig. 2. The physical and logical structure of a PDF file.

PDF malware may also carry other encoded payloads in stream objects which will be triggered after exploits [25].

B. *Target Classifiers*

Several projects have built PDF malware classifiers using machine learning techniques. Earlier works, such as Wepawet [7] and PJScan [16], focused on the embedded malicious JavaScript in PDF malware. These tools consist of a JavaScript code extractor and a dynamic or static malicious JavaScript classifier.

Since not all PDF malware involves embedded JavaScript, and PDF malware authors have found many tricks for hiding JavaScript code [24], recent PDF malware classifiers have focused on structural features of PDF files. In this work, we target state-of-the-art structural feature-based classifiers.

Structural feature-based classifiers assume that the malicious PDFs have different patterns in their internal object structures than those found in benign PDFs. For example, the PDF Malware Slayer tool uses the object keywords as features, where each feature corresponds to the occurrences of a given keyword [19]. For our experiments, we selected PDFrate [25, 26] and Hidost [28] as the target classifiers. They are representatives of recent PDF malware classifiers, and Hidost was developed with a particular goal of being resilient to evasion attacks. Both classifiers achieve extremely high accuracy in malware detection on their testing datasets. The other reason for choosing these classifiers as our targets is the availability of the open source implementations. Although our method only requires black-box access to the classifier, having access to the internal feature space is beneficial for understanding our results (Section VI).

PDFrate. PDFrate is a random forest classifier that uses an ensemble learning model consisting of a large number of decision trees designed to reduce variance in predictions. With a random subset of training data and a random subset of features, each decision tree is trained to minimize the prediction error on its training subset. After training, the output score of PDFrate is the fraction of trees that output "malicious", ranging from 0 to 1. The threshold value is typically 0.5, although the PDFrate authors claim that adjusting the threshold from 0.2 to 0.8 has little impact on accuracy because most samples have scores very close to either 0 or 1.

Besides object keywords, PDFrate also employs the PDF metadata and several properties of objects as the classification features. The PDF metadata includes the author, title, and creation date. The object properties includes positions, counts, and lengths.

PDFrate was trained with a random subset of the Contagio dataset [5] with 5,000 benign and 5,000 malicious PDFs. The two parameters are respectively the number of trees ($ntree = 1,000$) and the number of features in each tree ($mtry = 43$). The feature set is a total of 202 integer, floating point, and Boolean features, but only 135 of the features are described in the PDFrate documentation.

What we use in this work is an open-source re-implementation of PDFrate named Mimicus [27], implemented by Nedim Šrndic and Pavel Laskov to mimic PDFrate for malware evasion experiments [29]. Mimicus was trained with the 135 documented PDFrate features and the same training set as PDFrate.[1] Mimicus has been shown to have classification performance nearly equivalent to PDFrate [29].

Hidost. Hidost is a support vector machine (SVM) classification model. SVM is an optimal margin classifier that tries to find a small number of support vectors (data points) that separate all data points of two classes with a hyperplane of a high-dimensional space. With kernel tricks, it can be extended as a nonlinear classifier to fit more complex classification problems. Hidost uses a radial basis function (RBF) kernel to map data points into an infinite dimensional space. At testing time, the (positive or negative) distance of a data point to the hyper-plane is output as the prediction result. A positive distance is interpreted as malicious, and negative as benign.

Hidost uses the structural paths of objects as classification features. For example, the structural path of a typical Pages object is */Root/Pages*. If that object appears in the PDF file, its feature value is 1; if not, its feature value is 0. Since the number of possible structural paths of PDF objects is infinite, Hidost uses 6,087 selected paths as features. The selected paths are those which appeared in at least 1,000 of the files in a pool of 658,763 benign and malicious PDFs collected from VirusTotal [31] and a Google search. The resulting model provided by the authors of Hidost was trained using the randomly-sampled 5,000 malicious and 5,000 benign files. It is reported to be robust against adversaries, where the number of false negatives on another 5,000 random malicious files only increased from 28 to 30 under what the authors claim is the "strongest conceivable mimicry attack" [28].

IV. EVADING PDF MALWARE CLASSIFIERS

The proposed method could be applied to any security classifier, although its effectiveness depends on being able to find good genetic programming operators to search the feature space efficiently and an appropriate fitness function to direct the search. In this section, we show how to instantiate our design to find evasive PDF malware.

A. *PDF Parser and Repacker*

The first step is to parse the PDF file as a tree-like representation. We will also need to regenerate a PDF file from

[1]The Mimicus authors were unable to locate one malicious file with the MD5 hash 35b621f1065b7c6ebebacb9a785b6d69 in Contagio.

the tree representation, after it has been manipulated to produce a new variant. For this, we use pdfrw [21], a python-based open source library for parsing PDF files into the tree-like structure and serializing that structure into an output PDF file.

It is important to note that pdfrw is not a perfect PDF parser and repacker, and a number of PDF malware samples have been malformed intentionally to bypass or confuse PDF parsers used in malware detectors (while still being processed by target PDF readers due to parser quirks). This means we cannot test our method on PDF seed samples that cannot be parsed by pdfrw, or that no longer exhibit malicious behavior when they are unpacked and packed using pdfrw.

To avoid losing too many samples because of PDF parsing issues, we modified pdfrw to loosen its grammar checking. This significantly increased the success rate of repacking PDF malware samples. The modified version of pdfrw is available at https://github.com/mzweilin/pdfrw.

In our modified pdfrw, we ignore several potentially corrupted, malformed, or misleading auxiliary elements. The *EOF* marks in PDF raw bytes are ignored; instead, the parser reads in all bytes of a file. The *cross-reference tables* are ignored; instead, it parses objects in the *body* directly without any index. Stream length indicators are ignored; instead, the parser detects the stream length with the *endstream* token. The unpaired keys or values are also ignored in parsing a dictionary. Ignoring these auxiliary elements significantly decreases parsing efficiency, thus, is only suitable for repacking seed malware samples. All seeds are repacked with correct auxiliary elements for efficient parsing later. In addition, we added support for parsing empty objects, which do exist in the malware samples. The dictionary data structure was modified to enable deep-copy in duplicating variants from seeds.

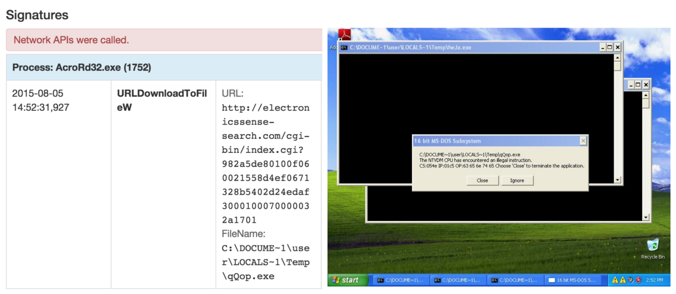
B. Genetic Operators

Since both of the classifiers we target employ the object structure of the PDF file as features, we need to generate variants by manipulating the PDF files at that level. (If we were targeting JavaScript-based classifiers instead, we would instead need to generate variants by manipulating the embedded JavaScript code.) Due to the limited number of possible static features, we believe it is reasonable to assume the attackers have the knowledge of the manipulation level.

We use computational analogs of *mutation* in biological evolution to generate evasive PDF malware variants. The mutation operator changes any object in a PDF file's tree-like structure with low probability. An object is mutated with probability given by the *mutation rate*, typically a number smaller than 0.5. The mutation is either a *deletion* (the object is removed), an *insertion* (another object is inserted after it), or a *replacement* (this object is replaced with some other object).

We choose among these options with uniform random probability. In the case of an insertion or replacement, a second object is also chosen uniformly at random from a large pool of objects segmented from benign PDFs. The external genome helps to generate a more diverse population.

The other well-known operator, *crossover*, commonly used in genetic algorithms, is not used in this work. We found it



Signatures

Network APIs were called.

Process: AcroRd32.exe (1752)

2015-08-05 14:52:31,927	**URLDownloadToFileW**	URL: `http://electronicssense-search.com/cgi-bin/index.cgi?982a5de80100f060021558d4ef0671328b5402d24edaf3000100070000032a1701` FileName: `C:\DOCUME~1\user\LOCALS~1\Temp\qQop.exe`

C:\DOCUME~1\user\LOCALS~1\Temp\fwJx.exe

16 bit MS-DOS Subsystem

C:\DOCUME~1\user\LOCALS~1\Temp\qQop.exe
The NTVDM CPU has encountered an illegal instruction.
CS:054e IP:01c5 OP:63 65 6e 74 65 Choose 'Close' to terminate the application.

Fig. 3. A PDF malware detection result given by the Cuckoo sandbox. The left side shows the key API execution trace, the right is a screenshot captured from the virtual machine.

was possible to achieve an 100% evasion rate only using the simple mutation operations.

C. Oracle

We need an oracle to determine if a variant preserves the seed's malicious behavior. There is no perfectly accurate malware detection technique that works universally (indeed, if such a technique existed our work would not be necessary). In this case we have one advantage that enables a highly-accurate oracle for testing variants: we do not need an oracle that can test for arbitrary malicious behavior, but instead only need to verify that a particular known malicious action is performed by the variant.

To do this, we use the Cuckoo sandbox [12]. Cuckoo runs a submitted sample in a virtual machine installed with a PDF reader and reports the behavior of the sample including network APIs called and their parameters. Figure 3 shows an example of malware detection results from Cuckoo. The malware sample opened in a virtual machine exploited a disclosed buffer overflow vulnerability in Acrobat Readers (CVE-2007-5659). The injected shellcode downloads four additional pieces of malware from Internet and executes them. Since the execution of Cuckoo was isolated from the Internet to avoid spreading malware, the shellcode just received malformed executable files provided by INetSim, a network service simulator [14]. However, the downloading and execution behaviors detected by Cuckoo are enough to show that the shellcode has been executed. By comparing the behavioral signature of the original PDF malware and the manipulated variant, we determine if the original malicious behavior is preserved. The details on how we select and compare behavioral signatures are deferred to Section V-A.

We only focus on the network behaviors of malware samples in this work. Although this setting prevents our method from working on malware samples without network activity, we believe it is not a real constraint in practice since malware authors could always develop a way to verify the desired malicious behaviors.

Cuckoo sandbox works well as an oracle, but is computationally expensive. We experimented with other possible oracles, including using Wepawet. Wepawet and similar detection techniques only detect the malicious payloads, but do not verify that the payload is actually executed in a real PDF reader. Because many of the genetic mutations will disrupt that execution, oracles that do not actually dynamically observe the variant exhibiting the malicious behavior result in many false positives (apparently evasive variants that would not actually work as malware). Hence, it is important to use an oracle that confirms the malicious behavior is preserved through actual execution. This limits the samples we can use in our experiments to ones for which we can produce the malicious behavior in our oracle's test environment (Section V-A).

D. Fitness Function

A fitness function gives the fitness score of each generated variant. Higher scores are better. Given 0 as a threshold value, a variant with a positive fitness score is evasive: it is classified as benign and retains the malicious behavior.

In our case, the fitness function captures both the output of the oracle and the predicted result of the target classifier. The oracle is modeled as a binary function: $oracle(x) = 1$ if x exhibits the malicious signature; otherwise, $oracle(x) = 0$. In order to eliminate corrupted variants, we always assign the lowest possible fitness score to variants with $oracle(x) = 0$.

Based on the different scoring methods used by the target classifiers, the fitness functions are defined separately. PDFrate, as a random forest classifier, outputs a confidence value of maliciousness from 0 to 1, typically with a threshold of 0.5. Thus, we define its fitness function as

$$fitness_{pdfrate}(x) = \begin{cases} 0.5 - pdfrate(x) & oracle(x) = 1 \\ LOW\_SCORE & oracle(x) = 0 \end{cases}$$

with evasive range of $(0, 0.5]$.

The SVM model of Hidost outputs negative (positive) distance of a benign (malicious) sample to hyperplane. Therefore, for Hidost the fitness function is defined as

$$fitness_{hidost}(x) = \begin{cases} hidost(x) \times (-1) & oracle(x) = 1 \\ LOW\_SCORE & oracle(x) = 0 \end{cases}$$

with evasive range of $(0, +\infty)$.

E. Selection

A selection process in GP can be as simple as always selecting variants with higher fitness scores in a generation. However, it might happen that very few or even none of the variants in a generation preserve the malicious behavior during the evolutionary process. If the malicious behavior is lost from the population, it is very unlikely the GP will ever find an evasive sample that exhibits the original malicious behavior.

In order to avoid degeneration in the population, we designed a replacement mechanism in addition to the naïve selection process. The corrupted variants, which are judged by the oracle as non-malicious, are assigned the lowest fitness score (*LOW_SCORE*) and are replaced by either the original malicious PDF, the best variant found so far, or the best variant found in the previous generation. We choose among these options with uniform random probability when corrupted variants occur, which ensures that a fixed number of variants are retained in each generation.

F. Trace Collection and Replay

The most common way to initialize a population is duplicating the original seed and performing a random mutation operation on each copy. Considering the potentially common properties across evasive variants, we accelerate the search by reusing mutation traces that successfully led to evasive or promising variants.

A mutation trace consists of a series of mutations defined by 3-tuple (*mutation operator*, *target object path*, *file id: source object path*). For example,

(insert, */Root/Pages/Kids/1*, *1: /Root/Pages/Kids/4*)

inserts an external Page object from a benign file *1* to the targeted PDF file. The three possible mutation operators are defined in Section IV-B. Though the *target object path* has the same format as the *source object path*, they are paths in different PDF files. The *target object path* refers to an object in the variant, while the *source object path* points to an object in an external benign file with the specified file id.

Mutation traces are added to two pools at the end of each GP search. If a GP search successfully generates evasive variants, all of the corresponding mutation traces are added to the *success trace pool*. Otherwise, a mutation trace that generates the variant with the highest fitness score is added to the *promising trace pool*.

The traces in the two pools are replayed in the population initialization to produce some variants for the first generation. If the number of usable traces is smaller than the population size, additional variants are generated in the conventional way. If the number is larger than the population size, the selection process described in Section IV-E shrinks the population to the specified size.

V. Experiment

We evaluate the effectiveness of the proposed method by conducting experiments on the two target PDF malware classifiers.

A. Dataset and Experiment Setup

We started with the 10,980 PDF malware samples in the Contagio archive [5], from which we selected 500 suitable samples for evaluation. These samples are verified by the oracle as exhibiting malicious behavior, are classified by both target classifiers as malicious, and can be correctly repacked by pdfrw.

Malicious PDF Dataset. Table I summarizes the sample selection procedure.

TABLE I. Seed selection.

Description	Number
PDF Malware samples in Contagio	10,980
Samples with network API calls detected by Wepawet	9,688
Samples with network activities observed by Cuckoo	1,414
Unique samples correctly repacked by pdfrw	1,384
True positives of PDFrate	1,378
True positives of Hidost	502
Intersection of TPs in PDFrate and Hidost	500

First, we filtered out the samples that don't have any network API calls by the shell code analysis of Wepawet, leaving 9,688 out of 10,980 samples. This is not necessary for our method, but useful since we use Wepawet to obtain additional information about the samples.

Second, the remaining samples were tested in the Cuckoo sandbox. According to the vulnerability information of each sample provided by Wepawet, Adobe Acrobat Reader 8.1.1 is the most common target PDF reader, except for CVE-2009-9837 which targets Foxit readers. Thus, these samples were loaded with Acrobat Reader 8.1.1. However, not all network behaviors indicated by the static analysis on shell code can be observed in Cuckoo even though we have selected a targeted PDF reader due to the imperfect network simulation in virtual machines as well as the potential sandbox detection features in malware. As a result, only 1,414 out of the 9,688 samples were observed to have malicious network activities running on Acrobat Reader 8.1.1 inside the Cuckoo sandbox.

Next, the 1,414 samples were repacked by the modified pdfrw with less strict grammar checking, then re-tested by Wepawet and Cuckoo. This resulted in 1,384 unique samples. Eleven of the samples were corrupted during repacking and no longer behaved maliciously in Wepawet or Cuckoo. The other 19 samples were found to be duplicates after being repacked. This is a clear sign that malware authors have attempted to evade detection through parsing obfuscation.

Since our goal is to evaluate the effectiveness of an evasion attack, we need to filter out the false negative samples of the target classifiers. PDFrate correctly classified 1,378 out of the 1,384 samples as malicious, while Hidost only correctly classified 502 of them. The intersection of the true positives from both classifiers left a suitable evaluation set of 500 unique PDF malware samples.

According to results from Wepawet, these 500 malware samples exploit two different vulnerabilities in Acrobat Readers: 333 of them exploit multiple buffer overflows reported in CVE-2007-5659, the other 167 exploit a stack-based buffer overflow reported by CVE-2009-0927. Both vulnerabilities can be exploited to execute arbitrary code. In summary, the payloads in the 500 samples access 255 different hosts to download additional malware from the Internet.

The selection process leaves us with 500 samples from the original 10,980 malware samples in the Contagio archive. Although this selects less than 5% of the original samples, it does not have implications for the success rate of a malware author attempting to find an evasive sample so long as the selection criteria have no biases which would impact our results. Many of the down-selects are due to artifacts of the experiment, not reflective of what an actual malware producer would observe. For example, the most significant reduction is because of the particular dynamic environment we selected to verify the malicious behaviors. Malware authors can easily design an oracle that verifies the presence of the particular malicious behaviors they intend to inflict.

Reliable Malware Signatures. Since the dynamic behavior of malware samples may vary across executions, we need to select a reliable malware signature from a group of candidates. Even though the malware is executing in the same virtual environment, its behavior may be effected by the timing of events, service failures, and other sources of non-determinism.

Focusing on the network behaviors of malware samples, we may extract various network behaviors reported by Cuckoo as signatures, such as DNS queries, HTTP URL requests, and network destinations. Cuckoo generates these reports from the network-related API execution traces and the captured network traffic. Table II compares the effectiveness of six different types of signatures extracted from Cuckoo reports.

We tested the 500 malware seeds in Cuckoo virtual machines, running each seed ten times. Our goal is to determine which type of signature will have the best precision in capturing observed malicious behavior, while being consistent across multiple executions of the same sample.

If a signature extracts any relevant behavior for a seed in any of the ten tests, we count the signature effective on the seed. Obviously, an ideal signature would be effective on all 500 seeds. We also measure the consistency of a signature over the 10 repeated tests. We designate the extracted behavior observed most frequently over the ten tests as the reference signature for a seed. The consistency on a seed is calculated as $\frac{mode}{10}$ (that is, the fraction of times the reference signature occurred across the 10 trials).

The average and the minimum consistency of each type of signature over the ten executions for each of the 500 seeds are listed in Table II. In general, the signatures extracted from API traces are more consistent than those extracted from network traffic. We choose the union of the HTTP URL requests and host queries extracted from API traces as the signature for our experiments. By combining those two behavioral signatures, we obtain a signature that is effective on all 500 malware seeds and has the highest average and minimum consistency.

Benign PDF Dataset. We collected a set of 179 benign PDF documents using a Google search with filetype:pdf and no keywords. All files were confirmed to be benign by both VirusTotal [31] and Wepawet [7]. We only included files smaller than 1 MB to avoid introducing unnecessary computation costs manipulating extremely large PDF files. We picked the 3 benign samples with the lowest scores (that is, most benign) to the target classifiers as the source of external objects in the experiment. Our results show that just a few benign samples is sufficient for generating successful evasion attacks.

GP Parameters. Several GP parameters are arbitrarily chosen without any parameter fine-tuning other than one obvious constraint: we want the experiment to finish in a reasonable time. The population size is 48 and the maximum generation is 20. The mutation rate is 0.1. The fitness stop threshold is 0.0, which indicates that an evasive variant has been found.

Target Classifiers. Since we don't want to abuse the online deployed malware classification systems by submitting too many automatically generated malware variants, we always prefer locally executable code. We used the Mimicus re-implementation of PDFrate and the Hidost classifier, configured and trained as described in Section III-B.

Machine. We used one typical desktop PC in the experiment (Intel Core i7-2600 CPU @ 3.40GHz and 32GB of physical

TABLE II. COMPARISON OF NETWORK-BASED MALWARE SIGNATURES.

Source	Description	Example	Effective	Consistency	
				Average	Minimum
API traces	Combination of HTTP URL requests and host queries	[http://stortfordaircadets.org.uk/flash/exe.php?x=pdf, stortfordaircadets.org.uk]	500	0.95	0.50
API traces	Hosts queried through getaddrinfo()	[stortfordaircadets.org.uk]	497	0.95	0.50
Network traffic	Transport layer destination IP addresses	(udp: [192.168.57.2:53], tcp: [192.168.57.2:80])	476	0.85	0.10
API traces	URLs requested through raw socket, URLDownloadToFileW(), InternetOpenUrlA()	[http://stortfordaircadets.org.uk/flash/exe.php?x=pdf]	473	0.95	0.50
Network traffic	DNS queries	[stortfordaircadets.org.uk]	462	0.93	0.10
Network traffic	HTTP URL requests	[http://stortfordaircadets.org.uk/flash/exe.php?x=pdf]	460	0.93	0.10

memory running 64-bit Ubuntu 14.04 Server). The Cuckoo sandbox consists of 16 virtual machine instances running Windows XP SP3 32 bit and Adobe Acrobat Reader 8.1.1. The resources required to find evasive samples using our approach are readily available.

VI. RESULTS

The GP-based method achieves surprisingly good results in evading the two target classifiers. For both of the classifiers, it is able to generate a variant that preserves the malicious behavior but is classified as benign for all 500 seeds in our test set. Our code and data are available under an open source license from http://www.evadeML.org

A. PDFrate

After approximately one week of execution, the algorithm found 72 effective mutation traces that generated 16,985 total evasive variants for the 500 malware seeds (34.0 evasive variants per seed in average), achieving 100% evasion rate in attacking PDFrate.

Trace Analysis. All the mutation traces that generated evasive variants were re-executed on all of the 500 seeds afterwards to investigate the efficacy of each trace. Efficacy here measures for how many of the malware seeds applying the given trace produces an evasive variant.

The length of each mutation trace and its efficacy are illustrated in Figure 4. The traces are sorted by trace ID, which reflects the order in which traces are found. From the figure we observe that the method generally finds longer mutation traces as the evolution proceeds. Part of the reason for this is the initial population for later seeds is generated using the collected traces. If those initial variants are not evasive, subsequent mutations will be added to the original traces.

The efficacy of each seed is not strongly correlated with its length. One mutation trace consisting of a single operation that inserts a page object generated evasive variants for 155 malware seeds. There was also mutation trace with 189 operations that was effective for only two seeds.

The accumulated evasions sorted by the length of mutation traces are given by Figure 5 (for comparison, the figure show results for Hidost as well, which we discuss later). The difficulty of generating variants to evade PDFrate varies substantially over the seeds. It only took 15 short mutation traces (none longer than 45 operations) to generate evasive variants for 400 of the 500 seeds. Finding evasive variants for the other 100 seeds took 57 long mutation traces with lengths ranging from 48 to 354.

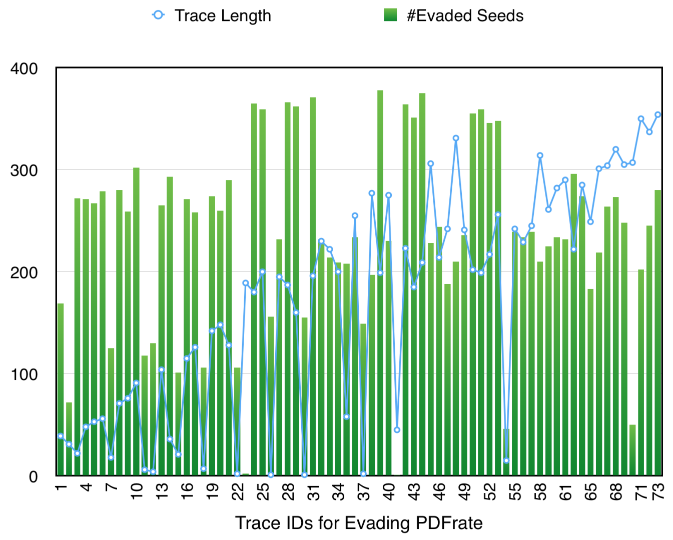


Fig. 4. The length and efficacy of mutation traces for evading PDFrate.

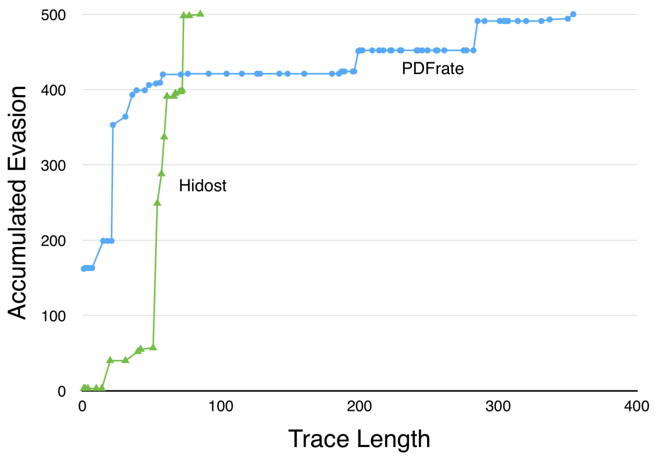


Fig. 5. Accumulated evasions against PDFrate and Hidost, sorted by trace length.

In order to understand why it takes much longer traces to generate evasive variants for those 100 seeds, we examined the original classification scores of each seed. Figure 6 groups the seeds by the minimum trace length required for generating evasive variants. The left side shows the original classification score distribution in PDFrate. We found that the original seeds with lower classification scores (<0.95) are mostly evadable by short traces. Thus, we believe some seeds require more mutations to evade because they are originally more clearly malicious to the classifier. (This is more obvious in Hidost as we discuss later.)

Feature Analysis. To understand the evasion attacks, we examine the impact of the changes on the feature space used by PDFrate.

We first look at the two simplest mutation traces in length of 1 that are effective for 162 seeds:

(insert, */Root/Pages/Kids*,
3:/Root/Pages/Kids/4/Kids/5/)
(replace, */Root/Type*, *3:/Root/Pages/Kids/1/Kids/3*)

Even though they are different operations, the common effect of the two mutations is that they both introduce new Page objects from external benign PDFs, resulting in significant changes in the feature space of PDFrate.

Table III lists one example of feature changes by simply inserting several Page objects. The classification score of the original seed is 0.998, approaching the maximum malicious score of 1.0. After inserting the new Page objects, the classification score decreases to 0.43, which is below the normal malware threshold of 0.5. The simple insert resulted in a large number of changes in the feature space. The counters of some objects like pages, fonts and streams as well as the file size directly increase due to the newly introduced objects. The object length statistics are decreased or increased due to the change of the object population. Some other features on object positions are also changed due to the relocation of objects at the raw byte level. All feature values are in the raw formats because feature normalization is not required with random forests. Even though the feature changes are so significant that PDFrate classifies the new variant as benign, the malicious behavior of the original seed does not change at all. The change just added some pages to the PDF file.

One simple manipulation introduces many feature changes, but the impact of changing each feature is not equivalent due to the varying importance of features in the classification. Though random forest is a complex non-linear model that is difficult to interpret, we estimate the impact of altering each feature independently. Intuitively, changing a high impact feature should significantly affect the classification scores.

For an original malware sample, if the feature is changed to a different value in evasive variants, the classification score should decrease (indicating it appears more like benign samples to the classifier). On the other hand, for an evasive variant, if the feature value is reset to the original value, the classification score should increase (appears more like malicious samples to the classifier). Therefore, we model the impacts with two factors. The decrease of the classification score of a malware seed when a feature is changed to the evasive value is reported as $\Delta score1$. The increase of the classification score of an evasive variant when the feature is reset to the original value is reported as $\Delta score2$. The impact is the sum of the two scores. Table III lists the impact ranking of the affected features, which roughly matches with the feature importance ranking in PDFrate [26].

The most critical feature change for this example is *count_font*. The original malware sample does not have any font objects as fonts are not needed for the exploit. The classifier learns that this feature is important because most of the malware samples in the training set do not contain any font objects as the malware authors are too lazy to insert any text, but it is unlikely that any benign PDF file has no font objects. However, this is an artifact of the malware samples in the training set, not an inherent property for malicious PDFs. It is trivial to add font objects to an existing PDF malware sample to alter the value of this feature.

TABLE III. IMPACT OF PDFRATE FEATURES.

Feature	Original	Evasive	Δscore1	Δscore2	Impact
count_font	0.0	70.0	0.114	0.392	0.506
count_obj	11.0	230.0	0.067	0.110	0.177
count_endobj	11.0	230.0	0.056	0.069	0.125
count_box_other	3.0	140.0	0.038	0.043	0.081
count_endstream	4.0	74.0	0.011	0.054	0.065
pos_box_max	0.0	0.8	0.052	0.013	0.065
count_stream	4.0	74.0	0.021	0.041	0.062
pos_box_avg	0.0	0.5	0.022	0.022	0.044
pos_eof_avg	1.0	1.0	0.000	0.032	0.032
pos_eof_min	1.0	1.0	-0.002	0.029	0.027
pos_page_max	0.0	0.8	0.003	0.018	0.021
pos_eof_max	1.0	1.0	0.002	0.016	0.018
pos_page_avg	0.0	0.5	0.002	0.010	0.012
size	36,028.0	503,739.0	-0.001	0.005	0.004
ratio_size_page	36,028.0	7,407.9	0.001	0.002	0.003
ratio_size_obj	3,275.3	2,190.2	0.000	0.002	0.002
ratio_size_stream	9,007.0	6,807.3	0.002	0.000	0.002
len_obj_avg	3,234.9	2,157.2	0.001	0.000	0.001
count_page	1.0	68.0	-0.004	0.004	0.000
len_obj_max	27,455.0	34,314.0	0.000	0.000	0.000
len_obj_min	44.0	49.0	0.000	0.000	0.000
len_stream_avg	8,700.3	6,390.2	0.000	0.000	0.000
len_stream_max	27,392.0	34,246.0	0.000	0.000	0.000
pos_page_min	0.0	0.0	-0.002	0.000	-0.002
pos_box_min	0.0	0.0	-0.003	0.000	-0.003

The difference in the feature space of PDFrate for a selected seed sample (with score 0.998) and corresponding evasive variant (with score 0.43). Original is the feature value of the original seed malware sample; Evasive is the feature value of the evasive variant. Features with the same value for both samples are not included. $\Delta score1$ is the original score subtracting the score after that feature is changed to Evasive value; $\Delta score2$ is the evasive score subtracted by the score after that feature is reset to the Original value. Impact is $\Delta score1 + \Delta score2$.

There are longer traces which contain at most 354 mutations and influence more features in PDFrate. Table IV lists the features that were most frequently increased and decreased across all 16,985 evasive variants found. (The full list of all 68 mutable features of PDFrate found in evasion attacks is found in Appendix A.) The count is how many times the value of the feature is different for the evasive variant found compared to the original seed. High counts imply these features are not robust and should not be used in malware classification because they are easy to change without corrupting the malicious properties for many malware seeds.

Most non-robust features are unsurprising, because a PDF malware author can always change the visible contents (such as pages, text, images and metadata) in PDF malware samples without corrupting the malicious payloads. The only surprising feature is *count_javascript*. Since PDF malware heavily relies on JavaScript to carry exploits and shell code, it seems surprising that it is possible to decrease *count_javascript* without disrupting the malicious behavior. However, the *count_javascript* feature is not an accurate count of the number of embedded JavaScript code pieces in a PDF. It just extracts the number of JavaScript keywords, but these keywords are optional in script execution. The targeted PDF reader will execute the JavaScript even without the */Javascript* keyword.

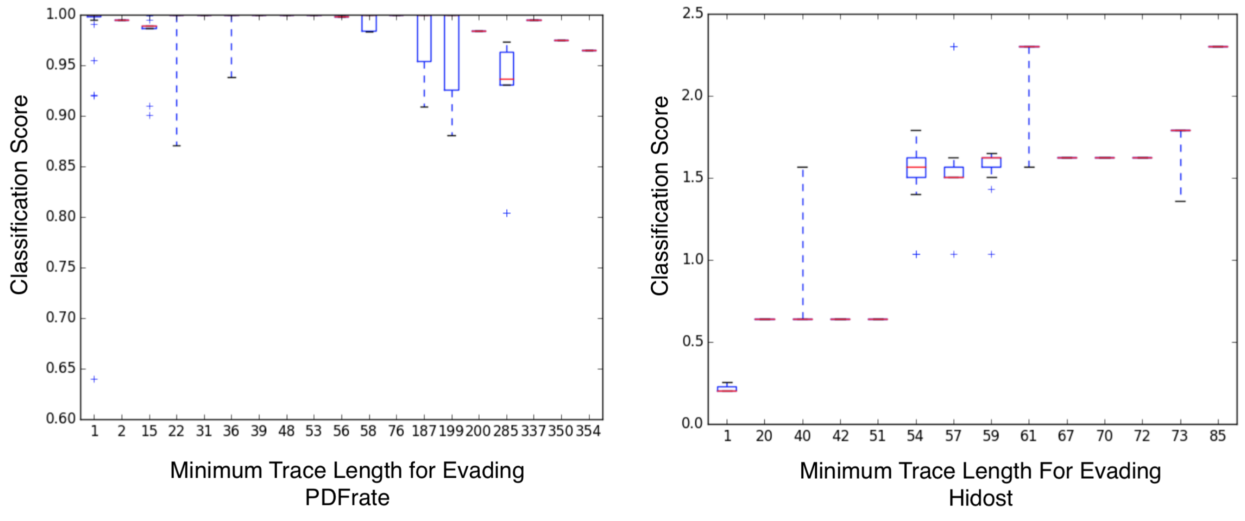


Fig. 6. The distribution of the original classification score of seeds.

TABLE IV. MOST ALTERED FEATURES EVADING PDFRATE

Counts	Increased features	Counts	Decreased features
16,985	pos_eof_avg	14,234	pos_page_min
16,985	pos_eof_max	10,806	len_obj_min
16,985	pos_eof_min	10,728	count_javascript
16,985	size	8,834	len_stream_min
16,975	count_endstream	7,637	ratio_size_stream
16,975	count_stream	4,742	createdate_tz
16,941	count_endobj	4,742	delta_tz
16,941	count_obj	4,250	ratio_size_page
16,862	len_stream_max	3,448	len_stream_avg
16,812	pos_box_max	3,137	pos_page_avg

B. Hidost

The experiment of evading Hidost took around two days to execute. Although Hidost was designed specifically to resist evasion attempts,[2] our method achieves a 100% evasion rate, generating 2,859 evasive samples in total for 500 seeds (5.7 evasive samples per seed in average).

Trace Analysis. We analyze the efficacy of each mutation trace which is examined in the same way as for PDFrate. The length and efficacy of each mutation trace are shown in Figure 7. In general, it required shorter mutation traces to achieve 100% evasion rate in attacking Hidost than it did for PDFrate.

We observed two major differences compared to PDFrate. First, there is no increasing trace length trend for newly found mutation traces, unlike for PDFrate where the trace length increases with the trace ID. Second, the trace length is more correlated with the efficacy: longer traces tend to be more effective in generating evasive variants. Several short mutation traces with fewer than 5 mutations are only effective on 1 or 2 malware seeds. In contrast, a long mutation trace containing 61 mutations is effective on 334 malware seeds.

[2]Specifically, the Hidost authors claim, "The most aggressive evasion strategy we could conceive was successful for only 0.025% of malicious examples tested against an off-the-shelf nonlinear SVM classifier with the RBF kernel using the binary embedding. Currently, we do not have a rigorous mathematical explanation for such a surprising robustness. Our intuition suggests that the main difficulty on attacker's part lies in the fact that the input features under his control, i.e., the structural elements of a PDF document, are only loosely related to the true features used by a classifier. The space of true features is hidden behind a complex nonlinear transformation which is mathematically hard to invert." [28]

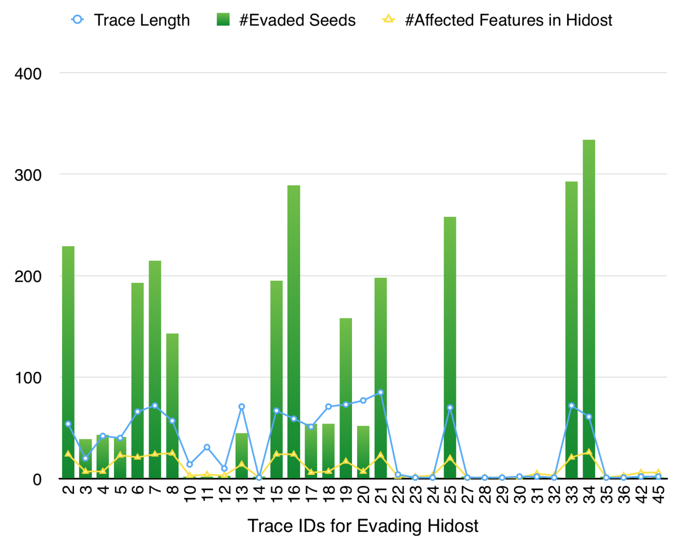


Fig. 7. The length and efficacy of mutation traces for evading Hidost.

The accumulated number of evasions found sorted by the length of mutation traces is given in Figure 5. The plot is closer to linear, suggesting that, in contrast to PDFrate, there is little variation in the difficulty of finding evasive variants for different seeds. We believe the differences from PDFrate stem from the different feature set in Hidost. The mutation operations have more direct influence on the structural path features in Hidost. For example, an object deletion operation just deletes the corresponding path of a feature (along with those of its descendants). In contrast, feature changes in PDFrate resulting from the same operation are less tangible. Besides decreasing the counts of specific objects that we can expect, the other positional features may also change due to the relocation of objects in repacking the modified variant. As a result, there are more inter-influences among the mutation operations in evading PDFrate, and a larger number of mutations may be required to reach the evasion threshold. The box plot of the original classification score in Hidost of each seed shown in the right side of Figure 6 suggests that it usually requires more mutations to find an evasive variant for seeds that appear to be more clearly malicious to the classifier.

Feature Analysis. The binary features used in Hidost are much easier to interpret than the variety of features used by PDFrate.

We first look at the simplest mutation traces. There are 5 mutation traces in length 1, which are only effective on 1 or 2 malware seeds. They are:

(delete, */Root/OpenAction/JS/Length*)
(delete, */Root/Names*)
(delete, */Root/AcroForm/DR*)
(replace, */Root/AcroForm/DR*,
    3: /Root/OpenAction/D/0/.../FontBBox/3)
(replace, */Root/AcroForm/DR*,
    3: /Root/Pages/Kids/3/.../DescendantFonts/0/DW)

The first three mutations each delete a node from the original malware seeds, changing the value of the corresponding Hidost feature from 1 to 0. The first deleted object similar to the *count_javascript* feature in PDFrate. Both capture properties that frequently exist in malware samples but not in benign files. However, they are optional in malicious code execution. The other deleted objects are artifacts in the training dataset that are not closely tied to malicious behavior. Although the last two traces use replace operations, the important effects of the replacements are to remove the features extracted from the children objects of the original */Root/AcroForm/DR* node.

Simply deleting some objects is not sufficient to evade Hidost (it is only effective on 1 or 2 malware seeds in our experiment), but additional mutations are enough to find evasive variants for all of the seeds. The longest mutation trace contains 85 operations, which is effective on 198 malware seeds for generating evasive variants to bypass Hidost. Table V lists the all of feature changes observed over the 198 malware seeds when executing that mutation trace. Unsurprisingly, several auxiliary objects are added or deleted to fool Hidost. For example, several metadata objects are inserted. Metadata widely exists in benign PDFs when users generate PDF documents with popular PDF writers. On the other hand, it is rare in PDF malware because malware authors did not add metadata in hand-crafting PDF exploits. However, this is just an artifact in the training dataset and not an essential difference between PDF documents and PDF malware. Inserting metadata into a PDF malware sample increases the likelihood of the sample being considered benign by Hidost.

As seen from this example, trace length itself is not a good measure of evasion complexity. Although the stochastic search process found an 85-operation trace to create these evasive variants, the trace only impacts the 23 features (each corresponding to a node in the PDF file) showing in Table V. That is to say, there is a 23-operation trace that would be just as effective (and probably shorter traces since one mutation can impact many features), and the trace found by the search includes many useless or redundant mutations. For the purposes of creating evasive malware, it is not important to find the shortest effective trace, although it would be possible to develop techniques to automatically pare down a trace to its essential operations if desired. The yellow triangle plot in Figure 7 shows the number of affected features for each trace.

TABLE V. FEATURE CHANGES PRODUCED BY LONGEST HIDOST MUTATION TRACE.

Added Features	Deleted Features
Threads	AcroForm
ViewerPreferences/Direction	Names/JavaScript/Names/S
Metadata	AcroForm/DR/Encoding/PDFDocEncoding
Metadata/Length	AcroForm/.../PDFDocEncoding/Differences
Metadata/Subtype	AcroForm/.../PDFDocEncoding/Type
Metadata/Type	Pages/Rotate
OpenAction/Contents	AcroForm/Fields
OpenAction/Contents/Filter	AcroForm/DA
OpenAction/Contents/Length	Outlines/Type
Pages/MediaBox	Outlines
	Outlines/Count
	Pages/Resources/ProcSet
	Pages/Resources

Although its authors claimed that Hidost was robust against evasion attacks involving just feature addition, we found many evasive variants that only added features. Among the 2,859 evasive variants, 761 are pure feature addition attacks, 21 of them are pure feature deletion attacks, and the other 2,077 involved both feature addition and deletion. It is already unrealistic to assume attackers can only insert features, and, as shown in the claims about non-evadability of Hidost, dangerous to assume a technique cannot be evaded because particular manual techniques fail.

A complete list of mutated features in evading Hidost is given in Appendix B. These non-robust features should not be used in a malware classifier, as they can be easily changed while preserving the original malicious properties.

C. Cross-Evasion Effects

Even though the classifiers are designed very differently and trained with different training datasets, we suspected they must share some properties in the same classification task. Therefore, we conducted a cross-evasion experiment by feeding one classifier with the evasive variants found in evading the other classifier.

For 388 of the malware seeds, the evasive variants found by evading Hidost are also effective in evading PDFrate. That is to say, without any access to PDFrate, a malware author with access to Hidost could find evasive variants for 77.6% of the seeds. In contrast, the evasive variants found by evading PDFrate are only effective against Hidost for two of the malware seeds.

The significant difference in the cross evasion effects is due to the different feature sets in the two classifiers. Indeed, the primary design goal for Hidost was to be less easily evaded than other classifiers by using features based on structural properties. The evasive variants generated by the algorithm in evading PDFrate do change the measured features significantly, however, they have little effect on the structural features used in Hidost. In the reverse direction, the evasive variants targeting Hidost by directly altering structural features (necessary to evade Hidost), incidentally impact the features used by PDFrate.

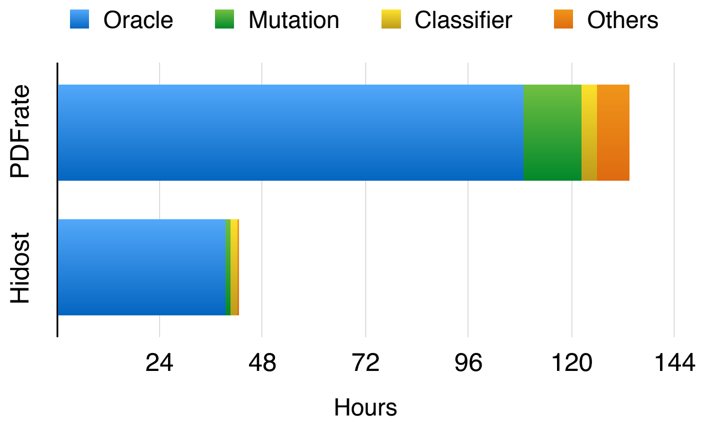


Fig. 8. Time required to find evasive variants for 500 malware samples.

D. *Execution Cost*

One drawback of evolutionary algorithms is that they provide no guarantees about generating good results within a specific duration. For many problems, these methods can require a huge amount of computing resources before a desired result is found. Further, failing to find the desired result may be a sign that it doesn't exist, or just that more computing resources are required. Our experiments show that the resources required for this instantiation are very reasonable.

For each classifier target, the experiment was run in several rounds. The first round started with empty trace pools, so the search for evasive variants relies solely on the stochastic search directed by the fitness function. In later rounds, the successful and promising mutation traces found previously were used to accelerate the search. All the failed jobs in a round were rerun in the next round with the all mutation traces stored in previous rounds.

For PDFrate, it took four rounds to reach 100% evasion. The evasion rate on Hidost reached 100% in the second round. Figure 8 shows the total time needed to find evasive variants for all 500 malware seeds in evading the two classifiers. The duration for each job is not meaningful because it mainly depends on the job sequence. Later jobs are usually much faster because they may benefit from mutation traces found earlier, but the order of trying the seeds is arbitrary.

It took less than 6 days to achieve a 100% evasion rate on PDFrate. In other words, our method found an evasive variant for each seed in 16 minutes on average running on a commodity desktop. Evasive variants were found against Hidost three times faster, taking 5 minutes per seed in average.

The main computation time is running the generated variants in the Cuckoo sandbox, which we use as the oracle in our process. The machine with 16 virtual machines running in parallel is able to test 1,000 samples per hour. This could easily be accelerated by using more machines, since there are no dependencies between the executions.

We also observed that the time spent on other tasks (including mutation) in attacking PDFrate takes a larger proportion of the total duration than for Hidost (8.3% vs. 4.1%). This is because the benign files used as external object genome are larger than those in attacking Hidost. Hence, it produced larger variants, increasing the computational burden for parsing, manipulating, and repacking.

VII. Discussion

In this section we discuss the potential defenses and future directions suggested by our results.

A. *Defense*

Beyond understanding the vulnerabilities of current classifiers, our ultimate goal is to improve the robustness of classifiers under attack. Based on the evasive samples we generated, and the non-robust features we found in Section V, we consider several possible approaches.

Information Hiding and Randomization. One of the most direct solutions to protect classifiers is hiding the classification scores from the users or adding random noise to the scores [2]. Another proposed method is the multiple classifier system, in which the classification scores are somewhat randomly picked from different models trained with disjoint features [3]. As our method heavily relies on the classification scores of variants to calculate fitness scores that direct the evolution, the lack of accurate score feedback makes the search for evasive variants much harder and may make our approach infeasible.

However, the intrinsic non-robustness of superficial features should not be simply ignored. Considering the potential cross-evasion effects (Section VI-C), hiding or randomizing the information may not help much against an adversary who can infer something about the types of features used by the target classifier. Moreover, previous work has shown that accurately re-implementing a similar classifier with a surrogate training set is possible (indeed, this is what the authors of Mimicus did to experiment with evadability of PDFrate [26, 29]).

Adapting to Evasive Variants. Our experiments assume that adversary can test samples without exposing them to the classifier operator. In an on-line scenario, the classifier may be able to adapt to attempted variants. Note, however, that retraining is expensive and opens up the classifier to alternate evasion strategies such as poisoning attacks.

Chinavle et al. proposed a method that would automatically retrain the classifier with pseudo labels once evasive variants were detected by a mutual agreement measure on the ensemble model, which had been shown effective on a spam detection task [6]. However, adapting to users' input without true labels introduces a new risk of poisoning attacks.

Defeating Overfitting. The evadability of classifiers we demonstrate could be just an issue of overfitting, in which case, well known machine learning practices should work to defeat overfitting. For example, collecting a much larger dataset for training the model, or using model averaging to lower the variance.

We don't expect these conventional methods will help, however. It is impossible to collect a complete dataset of future malware, and none of these techniques anticipate an adversary who is actively attempting to evade the classifier.

Selecting Robust Features. We found many non-robust features from the two classifiers in the evasion experiments.

Obviously, they should be removed from the feature set as they can be easily manipulated by the attacker without corrupting the malicious properties. The problem with the features used by both Hidost and PDFrate, however, is that *all* of the features are likely non-robust. The superficial features used by these classifiers do not have any intrinsic distinguishability between benign and malicious PDFs, and it would be very surprising if superficial features were found that could be used for robust classification. Instead, it seems necessary to use deeper features to build classifiers that can resist evasion attempts by sophisticated adversaries. Such features will depend on higher-level semantic analysis of the input file, in ways that are difficult to change without disrupting the malicious behavior.

B. Improving Automatic Evasion

Our automatic evasion method provides a general method to evaluate the robustness of classifiers for security tasks. Its ability to find evasive variants against a target classifier demonstrates clear weaknesses, but if our method fails to find evasive variants against a particular classifier this is certainly not enough to be confident that other techniques (including manual effort) would not be able to find evasive variants. Hence, it is valuable to improve the method to enable more efficient searching to target more challenging classifiers.

Parameter Tuning. In this work, we just arbitrarily choose the search parameters. Tuning the parameters, or even trying dynamic mechanisms like parameter decay, could make the search algorithm more efficient.

Learnable GP. The current method we use to generate evasive variants is essentially a random search algorithm. Hence, it often generates corrupted variants that lose the malicious behavior. A probabilistic model would learn which mutations are more effective for generating evasive variants to direct the search more efficiently.

Other Applications. Our case study focused on PDF malware, but we believe similar approaches could be effective against other machine-learning based malware classifiers. The main challenges in applying our approach to a new domain are to develop suitable genetic mutation operations and find an appropriate oracle.

VIII. Related Work

There have been several papers on evasion attacks against classifiers in the machine learning community, mostly focused on spam detection with simple models (e.g., [6, 10, 18]. Chinavle et al. argued that the adversarial problem is essentially *concept drift*, which is a well studied field in machine learning that considers data distributions which change over time [6]. However, the concept drift solutions assume the data distribution changes are not due to the classifier itself, not resulting from an adversary intentionally adapting to it.

Evasion attacks against malware classifiers have been studied previously by Biggio et al. from the angle of classification models [4] and by Šrndic et al. [28]. However, these studies assumed that attackers can only insert new features and they conducted evasion experiments in the feature space without generating actual evasive PDF malware. In fact, the experiments in our work show attackers can also delete features while preserving maliciousness, and our experiments verified that the resulting evasive variants preserved maliciousness through dynamic execution in a test environment.

Šrndic et al. demonstrated how PDFrate could be evaded by exploiting an implementation flaw in the feature extraction [29]. Our method does not rely on any particular implementation flaw in a target classifier. Instead, it exploits the weak spots in a classifier model's feature space and employs a stochastic method to manipulate samples in diverse ways.

Maiorca et al. proposed reverse-mimicry attacks against PDF malware classifiers [20]. In reverse-mimicry, a benign sample is manipulated into a malicious one by inserting malicious payloads into the structure. The attack is generic to a class of classifiers based on structural features. However, the hand-crafted attack only works on malware with simple payloads. In contrast, our GP-based method is automatic and does not have this limitation.

Evolutionary algorithms have also recently been used to fool deep learning-based computer vision models [22]. In contrast, this work uses genetic programming, an important branch of evolutionary algorithms for generating highly-structured data like computer programs.

IX. Conclusions

Our experiments show how the traditional approach of building machine learning classifiers can fail against determined adversaries. We argue that it is essential for designers of classifiers used in security applications to consider how adversaries will adapt to those classifiers, and important for the research community to develop better ways of predicting the actual effectiveness of a classifier in deployment.

Availability

The source code of our automatic evasion tool, along with the data from all of our experiments, is available at http://www.EvadeML.org.

Acknowledgments

This work was partially supported by grants from the Air Force Office of Scientific Research and the National Science Foundation. The authors gratefully thank Giovanni Vigna for helpful assistance, Westley Weimer for fruitful discussions about this work and help with the genetic programming experiments, and Hongning Wang for insightful comments and assistance. We thank Nedim Šrndic and Pavel Laskov for making an open source version of PDFrate and providing Hidost, which were essential for our experiments.

References

[1] Adobe, Inc. PDF Reference and Adobe Extensions to the PDF Specification. http://www.adobe.com/devnet/pdf/pdf_reference.html.

[2] Marco Barreno, Blaine Nelson, Russell Sears, Anthony D Joseph, and J Doug Tygar. Can Machine Learning Be Secure? In *First ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2006.

[3] Battista Biggio, Giorgio Fumera, and Fabio Roli. Multiple Classifier Systems for Adversarial Classification Tasks. In *Multiple Classifier Systems*. Springer, 2009.

[4] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Šrndić, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. Evasion Attacks against Machine Learning at Test Time. In *6th European Machine Learning and Data Mining Conference (ECML/PKDD)*. 2013.

[5] Stephan Chenette. Malicious Documents Archive for Signature Testing and Research - Contagio Malware Dump. http://contagiodump.blogspot.de/2010/08/malicious-documents-archive-for.html.

[6] Deepak Chinavle, Pranam Kolari, Tim Oates, and Tim Finin. Ensembles in Adversarial Classification for Spam. In *18th ACM Conference on Information and Knowledge Management (CIKM)*, 2009.

[7] Marco Cova, Christopher Kruegel, and Giovanni Vigna. Detection and Analysis of Drive-By-Download Attacks and Malicious JavaScript Code. In *19th International World Wide Web Conference (WWW)*, 2010.

[8] CVE Details. Adobe Acrobat Reader — CVE Security Vulnerabilities, Versions and Detailed Reports. http://www.cvedetails.com/product/497.

[9] George E Dahl, Jack W Stokes, Li Deng, and Dong Yu. Large-Scale Malware Classification Using Random Projections and Neural Networks. In *38th IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2013.

[10] Nilesh Dalvi, Pedro Domingos, Sumit Sanghai, and Deepak Verma. Adversarial Classification. In *10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2004.

[11] Stephanie Forrest. Genetic Algorithms: Principles of Natural Selection Applied to Computation. *Science*, 261 (5123), 1993.

[12] Claudio Guarnieri, Alessandro Tanasi, Jurriaan Bremer, and Mark Schloesser. Cuckoo Sandbox: A Malware Analysis System. http://www.cuckoosandbox.org/.

[13] Mark Harman, William B Langdon, and Westley Weimer. Genetic Programming for Reverse Engineering. In *20th IEEE Working Conference on Reverse Engineering (WCRE)*, 2013.

[14] Thomas Hungenberg and Matthias Eckert. INetSim: Internet Services Simulation Suite. http://www.inetsim.org/.

[15] John R Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, volume 1. MIT press, 1992.

[16] Pavel Laskov and Nedim Šrndić. Static Detection of Malicious JavaScript-Bearing PDF Documents. In *27th ACM Annual Computer Security Applications Conference (ACSAC)*, 2011.

[17] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering*, 2012.

[18] Daniel Lowd and Christopher Meek. Adversarial learning. In *11th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2005.

[19] Davide Maiorca, Giorgio Giacinto, and Igino Corona. A Pattern Recognition System for Malicious PDF Files Detection. In *8th International Conference on Machine Learning and Data Mining in Pattern Recognition*. 2012.

[20] Davide Maiorca, Igino Corona, and Giorgio Giacinto. Looking at the Bag Is Not Enough to Find the Bomb: An Evasion of Structural Methods for Malicious PDF Files Detection. In *8th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2013.

[21] Patrick Maupin. PDFRW: A Pure Python Library That Reads and Writes PDFs. https://github.com/pmaupin/pdfrw.

[22] Anh Nguyen, Jason Yosinski, and Jeff Clune. Deep Neural Networks are Easily Fooled: High Confidence Predictions for Unrecognizable Images. In *28th IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[23] Conor Ryan. *Automatic Re-Engineering of Software Using Genetic Programming*, volume 2. Springer Science & Business Media, 2012.

[24] Karthik Selvaraj and Nino Fred Gutierrez. The Rise of PDF Malware. https://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/the_rise_of_pdf_malware.pdf, March 2010.

[25] Charles Smutz and Angelos Stavrou. Malicious PDF Detection Using Metadata and Structural Features. Technical report, 2012.

[26] Charles Smutz and Angelos Stavrou. Malicious PDF Detection Using Metadata and Structural Features. In *28th ACM Annual Computer Security Applications Conference (ACSAC)*, 2012.

[27] Nedim Šrndic and Pavel Laskov. Mimicus: A Library for Adversarial Classifier Evasion. https://github.com/srndic/mimicus.

[28] Nedim Šrndic and Pavel Laskov. Detection of Malicious Pdf Files Based on Hierarchical Document Structure. In *20th Network and Distributed System Security Symposium (NDSS)*, 2013.

[29] Nedim Šrndic and Pavel Laskov. Practical Evasion of a Learning-Based Classifier: A Case Study. In *35th IEEE Symposium on Security and Privacy (Oakland)*, 2014.

[30] Symantec Corporation. Symantec Internet Security Threat Report, 2015.

[31] VirusTotal. Free Online Virus, Malware and URL Scanner. https://www.virustotal.com/.

Appendix

A. *Mutated Features in PDFrate*

The 68 features mutated in our experiments evading PDFrate are listed in Table VI. It is important to note, however, that just because a feature does not appear here does not mean it is robust to evasion. The features listed are those that were sufficient for achieving 100% evasion rate in our experiment. Similarly, the unidirectional mutations are how observed in the evasion attack experiment. It doesn't necessarily mean that these features cannot also be mutated in the reverse direction without corrupting malware samples in practice.

TABLE VI. 68 Features Modified Evading PDFrate

Feature Name	Mutability	Feature Name	Mutability
box_nonother_types	↓	image_totalpx	↑
box_other_only	↑	len_obj_avg	↑↓
count_acroform	↑↓	len_obj_max	↑↓
count_action	↑↓	len_obj_min	↑↓
count_box_letter	↓	len_stream_avg	↑↓
count_box_other	↑↓	len_stream_max	↑
count_endobj	↑↓	len_stream_min	↑↓
count_endstream	↑↓	pos_acroform_avg	↑↓
count_font	↑↓	pos_acroform_max	↑↓
count_image_med	↑	pos_acroform_min	↑↓
count_image_small	↑	pos_box_avg	↑↓
count_image_total	↑	pos_box_max	↑↓
count_image_xsmall	↑	pos_box_min	↑↓
count_javascript	↓	pos_eof_avg	↑
count_js	↓	pos_eof_max	↑
count_obj	↑↓	pos_eof_min	↑
count_objstm	↓	pos_image_avg	↑
count_page	↑↓	pos_image_max	↑
count_page_obs	↓	pos_image_min	↑
count_stream	↑↓	pos_page_avg	↑↓
createdate_mismatch	↑	pos_page_max	↑↓
createdate_ts	↑	pos_page_min	↑↓
createdate_tz	↓	producer_dot	↑↓
createdate_version_ratio	↑	producer_lc	↑↓
creator_dot	↑↓	producer_len	↑↓
creator_lc	↑↓	producer_mismatch	↑
creator_len	↑↓	producer_num	↑↓
creator_mismatch	↑	producer_oth	↑↓
creator_num	↑↓	producer_uc	↑↓
creator_oth	↑↓	ratio_imagepx_size	↑↓
creator_uc	↑↓	ratio_size_obj	↑↓
delta_ts	↑	ratio_size_page	↑↓
delta_tz	↓	ratio_size_stream	↑↓
image_mismatch	↑	size	↑

B. *Mutated Features in Hidost*

The 24 inserted features and the 19 deleted features in finding the 2,859 evasive variants against Hidost are listed in Table VII. As with PDFrate, the features that are not listed are not necessarily robust features.

The "counts" are the number of evasive variants mutated that feature. Note that some features are hierarchically dependent in the PDF object structure, so one insertion or deletion may impact many features. For example, inserting a complete *Metadata* object (as is done in 2,507 of the variants) also introduces several child objects: *Metadata/Length*, *Metadata/-Subtype* and *Metadata/Type*.

TABLE VII. Features Altered Evading Hidost

Counts	Inserted Feature
2,507	Metadata
2,507	Metadata/Length
2,507	Metadata/Subtype
2,507	Metadata/Type
2,454	PageLabels
2,363	ViewerPreferences/Direction
1,991	Pages/Resources/ProcSet
1,968	Pages/Resources
1,702	Pages/Rotate
1,382	Pages/MediaBox
825	Threads
718	OpenAction/MediaBox
385	OpenAction/Contents/Filter
385	OpenAction/Contents/Length
369	OpenAction/Contents
319	OpenAction/Resources
319	OpenAction/Resources/ProcSet
158	OpenAction/Rotate
158	OpenAction/CropBox
51	OpenAction/Type
51	OpenAction
41	PageLabels/Nums
41	PageLabels/Nums/S
40	PageLayout
Counts	**Deleted Feature**
1,345	Names/JavaScript/Names/S
865	PageLayout
615	Outlines/Type
615	Outlines
615	Outlines/Count
502	AcroForm/Fields
500	AcroForm
330	OpenAction/JS/Length
54	Pages/Rotate
14	Pages/Resources/ProcSet
12	AcroForm/DR/Encoding/PDFDocEncoding
12	AcroForm/DR/Encoding/PDFDocEncoding/Differences
12	AcroForm/DR/Encoding/PDFDocEncoding/Type
11	Pages/Resources
9	AcroForm/DA
8	Pages/MediaBox
4	OpenAction/S
3	Names/EmbeddedFiles
2	Names