Evaluating Feature Robustness for Windows Malware Family Classification

Adam Duby
United States Military Academy
West Point, NY
adam.duby@westpoint.edu

Teryl Taylor
IBM Research
Yorktown Heights, NY
terylt@ibm.com

Gedare Bloom, Yanyan Zhuang
University of Colorado Colorado Springs
Colorado Springs, CO
{gbloom, yzhuang}@uccs.edu

2022 International Conference on Computer Communications and Networks (ICCCN) | 978-1-6654-9726-8/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICCCN54977.2022.9868914

***Abstract*—Machine learning approaches to classify malware by family save analysts valuable time during incident response. A key challenge for these approaches is selecting features that are robust against concept drift, which describes the change in malware over time. In this paper, we evaluate a dynamic feature set based on Windows handles (e.g., files, registry keys) for malware family classification. Specifically, we examine the features' vulnerabilities and evaluate their robustness against concept drift. We curated a novel dataset that simulates the manipulations that attackers may invoke on malware samples. We demonstrate improved robustness to concept drift over traditional API call-based features by training machine learning classifiers on malware collected in the wild, and testing the classifiers against samples that underwent manipulations. Further, we investigate time decay due to concept drift using temporally consistent evaluations that do not assume access to newer information. The evaluation shows that our features are robust against malware obfuscation. Furthermore, we empirically demonstrate how malware labeling conventions (malware type or family) can affect results, and make recommendations for dataset construction.**

***Keywords*—Malware analysis, forensics, malware classification**

I. Introduction

Malware classification research has shown promising results, with some approaches demonstrating near perfect accuracy [14], [21]. Such results may suggest that the problem is largely solved. However, temporal experimental bias [31] can inflate the performance metrics. Temporal experimental bias is introduced when experiments assume the distribution of features at test-time and train-time are the same; this assumption does not hold when the features change over time due to concept drift [20]. Concept drift occurs when the statistical relationship between input features and output labels change, decaying classifier performance. In software, concept drift can occur naturally because of version changes in code or operating systems that in turn affect input features for classification algorithms. For example, the shift from 32-bit to 64-bit executables changed program features despite the programs being semantically equivalent. Concept drift can also be intentionally induced by an adversary to thwart a classifier [16]. We refer to such drift as adversarial drift [22].

Many studies neglect the influence of adversarial drift, which often comes in the form of program/machine code obfuscation [1]. Heavy obfuscation, such as packing, is highly transformative, making it a challenging adversarial case study. Packing is also a realistic and prevalent problem, as it is estimated that about 80% of the malware in the wild is packed [29]. Packing also provides reusability, affording attackers a means to create new malware from existing samples. In this paper, we explore the robustness of dynamic features for Windows malware classification on a large real-world malware dataset of 27 malware families. To evaluate robustness against concept drift, we compare the performance of a temporally biased dataset with one that removes temporal bias by ensuring temporal consistency in the training and testing data. By leveraging compile timestamps, we train our classifier on older malware, and test on more recent malware to capture the time decay. To further evaluate robustness against packing, we augment the dataset with packed malware to simulate adversarial drift.

We present and analyze two sets of dynamic features with an emphasis on robustness: process handle counts by type, and named memory-mapped files. Handles are references to resources, such as files, events, and registry keys that are made by a process; memory-mapped files are known as section objects, most of which are dynamically loaded libraries (DLLs). We present a dynamic feature extraction process to provide additional robustness for capturing DLLs that could otherwise be hidden through stealthy techniques, such as import obfuscation. Results show that our proposed features offer improved robustness against concept drift over state-of-the-art dynamic feature sets. Further, unlike past studies that used adversarial training and retraining to boost classifier performance, we found that such training had a negligible impact on our classifier. This suggests that robust feature sets reduce the need for retraining.

In addition to temporal bias, the ease of distinguishability of malware labels in popular datasets may further inflate reported performance metrics [43]. For example, the label 'botnet' loosely classifies the malware based on its *type* labels and provides little valuable intelligence for a defender. By contrast, a *family* label, such as 'Zeus', informs the defender about the malware's specific tactics, techniques, and procedures. Although family labeling increases classification difficulty because families are more fine-grained and harder to distinguish, families improve classifier utility. We demonstrate the impact labeling can have on malware classification. Indeed,

This work is supported in part by NSF grants OAC-2115134, OAC-1920462, OAC-2001789, CNS-2046705, and Colorado State Bill 18-086. The views expressed in this paper are those of the authors and do not reflect the official policy or position of the United States Military Academy, the United States Army, the Department of Defense, or the United States Government.

978-1-6654-9726-8/22/$31.00 ©2022 IEEE

whether malware is labeled based on family or type can have a huge effect on performance. In response, we provide a set of recommendations around labeling practices for future work, and discuss techniques for building datasets for analysis.

In this paper, we make the following contributions:

- We present an in-depth feature analysis of dynamic features based on handle references.
- We show that handle-based features are more robust to concept drift in general over state-of-the-art API-based feature sets.
- We highlight how the choice of labeling malware datasets can drastically affect classification performance, and make recommendations around labeling conventions for future studies.

II. Background and Related Work

This paper focuses on feature analysis and labeling for a machine learning-based malware classifier trained with a set of malware samples. The machine learning input is in the form of feature vectors, extracted from malware samples, and the labels are known ground truth class associations. We denote this classifier as $f : X \rightarrow Y$, where X is the malware feature space and Y is the label space. Once trained and deployed, given a feature vector $x \in X$, the classifier returns a predicted label, $f(x) = y$, where $y \in Y$. Let M represent the sample space of raw malware files (i.e, PE32 files in Windows). For each $m \in M$, a feature extraction and selection function yields the feature vectors, $\phi(m) = x$. While feature extraction pulls features from m, feature selection is the process of choosing which extracted features to include in x.

A. Malware Labeling

Machine learning has been applied to the malware domain to tackle two major problems: malware detection and malware classification. *Malware detection* is a binary classification to detect malicious software [15]. In this case, the label space is simply Y ={benign,malicious}. By contrast, the *multiclass malware classification* problem assumes the input software is malware and predicts the type or family membership of m. Related work has not clearly defined the labeling terminology malware *type* and malware *family*; however, the distinction between labeling schemes is extremely important, and can have an impact on classification performance.

A malware *type* is a high-level description of the general functionality of the malware. Example labels of malware types include worm, trojan, ransomware, and spyware. By contrast, a malware *family* describes a grouping of malware that utilizes similar low-level primitives or a similar code base to achieve similar effects. While a type describes the overall effect, a family categorization represents how the effects are achieved. A family can describe widely available commodity malware (e.g., Papras, Delf), or proprietary malware deployed in an intrusion set campaign (e.g., BlackEnergy).

The disadvantage of type labels is that they do not expose the information that analysts can use to reason about the attack for follow-on defensive operations, such as threat attribution, hunting for additional indicators, and performing damage assessments. For example, type classification as a backdoor still requires that an analyst manually reverse engineer the malware to find related network traffic and follow-on adversary activity. If the malware was classified into a more granular family name, such as NewsReels (an instance of a backdoor), then the analysts can immediately start hunting for malicious HTTP traffic indicative of this malware family.

Once a label space Y that meets the problem's objective is defined, one must determine the appropriate features X and algorithm. Possible algorithms include random forests, k-nearest-neighbors (kNN), support vector machines, and artificial neural networks. Feature extraction and selection are more important to a model's success than the choice of algorithms [9], [25], so our emphasis is on feature selection.

B. Malware Features

Feature extraction and selection is the process of pulling observable information from m to produce a feature vector x as input to a malware classifier. In software, there are two main approaches for feature extraction: static and dynamic.

Static features are extracted from the program file and do not require program execution. This can include structural file information [42], [51], statistical information [3], [6], information from the file header [50], or information obtained directly from opcodes [47], [49]. Other techniques leverage information from the program's import table and loaded libraries [6], [28]. Approaches use more rigorous feature selection based on information gain to extract a combination of informative static features [55]. Static features are easy to extract, but they tend to reason about surface signatures [43]. For example, static features extracted from packed malware train classifiers to learn about packing instead of the characteristics of the malware [1], [26]. As such, these features are not resilient to adversarial attacks.

Dynamic features are extracted from the runtime analysis of a program. Related work has proposed API and system call information as dynamic features [2], [4], [5], [17], [34], [40]. Despite promising results, attacks against system call and API based approaches have been shown effective at defeating classifiers by perturbing the order of calls, invoking additional calls, and obfuscating the API calls [38], [44]. API calls can also change over time due to variations in the execution environment, obfuscation, and version changes in software and operating systems. For example, between Windows 7 SP0 and Windows 10 v1809, there have been 69 new native (NT) APIs introduced, and three removed completely; a cursory examination of the Win32 documentation reveals that many Win32 API calls were gradually deprecated.

Behavioral approaches have also been proposed to approximate the behavior of malware. Research demonstrated the feasibility of behavioral approaches in malware detection by using file systems interactions, mutexes, and network resources as features [19], [24], [33], [45]. Most closely related to our work, Mosli et al. proposed using handle statistics for malware detection [27], i.e., binary classification. Our approach expands on these observations to utilize handles and memory-mapped file names as features for malware family classification. Further, we filter out handles and names that may change over time or can be easily manipulated by an attacker and we evaluate these features' robustness to concept drift.

C. Adversarial Considerations

We are interested in identifying features that are robust to concept drift. Consider an adversary who wants to re-purpose an existing malware m. The adversary's goal is to perturb the feature space in a manner that forces a misclassification. Such an attack is realized via some transformation function t applied against the original malware sample, $t : M \rightarrow M'$. The challenge for the attacker is developing t such that $t(m) = m'$ and $\phi(m) \neq \phi(m')$ while preserving the original semantics and desired effects of m.

It may be trivial to conceive transformations that alter the feature space; however, not all transformations are easily *realizable*. Due to the complex nature of compiled software, it is difficult to randomly perturb features while preserving desired semantics [32]. Due to such difficulty, attackers commonly use packing as an adversarial technique. Packing approximates many adversarial transformation techniques because it changes the code layout and obfuscates the import table. Further, packing changes the PE32 header, which has been shown to be one of the most discriminative and important features used in malware classifiers [35]. Packing also alters the API call sequence by invoking additional API calls to unwrap the packed executable at process start-up.

Attacks can also easily perturb features based on loaded libraries [6], [7]. If extraction is performed statically, the attacker can deceive malware classification through delay loading libraries, import address table (IAT) obfuscation, and stealthy loading [23]. These stealthy techniques unlink loaded libraries in a binary sample, causing related features to be omitted from the classifier's input.

The names of operating system resources, such as mutexes and accessed files, are also popular features for malware classification [19], [24], [33], [36], [45]. For example, Backoff malware variants created predictable and stable named mutexes as part of its infection routines. However, attacks on these features can be as simple as changing the names of these resources – e.g., TreasureHunter malware used dynamically generated mutex names that were not trivial to predict.

In addition to changes induced by the adversary, concept drift introduces time decay in models whose chosen features change over time [20]. Recent work proposed time-aware splits based on malware compile timestamps to create temporally consistent training and testing windows for malware detection [31]. They observed a decrease in F1-scores by approximately 0.20 in malware detection models when retrained and tested using time-aware splits. As such, they recommend using robust features that are less likely to deviate over time due to concept drift and adversarial drift. We use these observations to incorporate time-aware evaluations against our chosen features for family classification.

D. Feature Robustness

It is important to select features that are less likely to change over time and are costly for an attacker to manipulate [54]. Several techniques have been proposed to enhance feature robustness. Xu et. al. suggested *feature squeezing* to reduce the attack surface by selecting only features imperative to the classification task [53]. Identifying features with a strict monotonic increase constraint also reduces the adversary's options to perturb features [18]. A monotonically increasing feature is one that if removed, the malware will lose its desired functionality. For example, imports are a monotonically increasing feature set [18]. A removed library reference removes the malware's ability to interact with said library; however, these features are susceptible to import obfuscation, packing and delay loading.

We desire features that are both monotonically increasing and unaffected by import obfuscation and packing. Further, static techniques for transforming malware are relatively low cost for attackers and they can easily fool classifiers dependent on a static feature set. By contrast, dynamic features are riskier for an attacker to manipulate because behavioral changes may violate semantic preservation [10], [11], [32]. This suggests that we need an approach based on dynamic features that are costly for adversaries to remove to attain feature robustness.

E. Dataset Considerations

Existing malware datasets are difficult to reuse because they are either distributed in non-executable format [37], or only provide certain features [5], making it challenging to extract new features from the same malware samples for comparison. We also desire a *hard* dataset where malware classes are not easily distinguishable. Smith et al. [43] discuss the *ease* of label distinguishability in malware datasets. We expand upon their observations and propose the following criteria for a *hard* dataset: (i) adopt meaningful labeling conventions that lead to actionable intelligence; (ii) testing sets should contain some adversarial transformations to simulate malware changing over time; (iii) ensure temporal consistency in the training and testing sets to avoid inflated results.

Some existing datasets favor high classification accuracy rather than useful malware labels [43]. We suggest this (unintentional) bias comes from *dirty labels*, which are class labels that are either incorrect or too vague to derive proper conclusions for a given problem [13]. Extending this concept to malware analysis, we present several examples of dirty labeling that may inadvertently lead to reporting of unrealistic results.

First, we consider the Microsoft Malware Classification Challenge [37] which provides a dataset of nine malware families and is used heavily in the literature [43]. One class in the dataset is `OBFUSCATOR.ACY` and serves as a bucket for all obfuscated or heavily packed malware. Obfuscated malware is not a family, and, as such, provides little information for defenders to pursue follow-on defensive activities. By encapsulating obfuscated malware into one family, the remaining families become easily distinguishable. This over-simplification of the obfuscation (e.g., packing) problem degrades reliability in deployed environments, because malware of the same family can appear as obfuscated variants.

Another example of dirty labeling is the use of antivirus detection names as ground truth family labels. Antivirus detection names are vendor-specific, and often based on specific hand-crafted signatures [12], [43]. This approach can create easily distinguishable classes not representative of the true complexity of malware family classification. Further, it has been shown that detection names are unreliable for proper malware family identification [12], [41]. To demonstrate this, we took one sample of the Masad Stealer malware and packed it with UPX[1] then ran both samples through 27 anti-virus engines. We observed different detection names for the two semantically identical samples. For example, Kaspersky named the original as `Trojan-Spy.Win32.AutoIt.arg`, and the packed version as `Trojan-PSW.Win32.Masqulab.b`. If we curated a dataset by pulling all samples that matched the `Trojan-Spy.Win32.AutoIt.arg` detection name and assigned them the same label, we risk missing packed and obfuscated variants. Therefore, this type of labeling in dataset construction may inadvertently create class labels that are more easily distinguishable than what is representative in the real world. Further, because anti-virus detection names are crafted by manual signatures, training a classifier with such labels teaches the classifier to reason about the signatures instead of reasoning about the malware [43].

III. Approach

The goal of our work is to evaluate feature robustness against concept drift for the malware classification problem. To do so, we must extract a set of features from malware that are robust against the attacker's ability to modify the malware through manipulation techniques. Specifically, we desire a small feature set to reduce the attack surface (i.e., feature squeezing [53]); and our features should meet the monotonic increase constraint to reduce adversarial options [18]. In this section, we present our features (Section III-A), dataset (Section III-B), and our feature analysis and selection process (Section III-C).

A. Features

We do not use any static features because of their known limitations and vulnerabilities to packing [1], [43]. Similarly, we avoid using most API calls as features because they do not meet the monotonic increase constraint [18], are vulnerable to diversification tactics [38], and are subject to concept drift. Instead, our approach to dynamic feature selection utilizes handles, which are references to resources that the Windows kernel exposes to applications. Handle information has been shown to be effective for malware detection [27] (i.e., binary classification); however, we focus on malware family classification; furthermore, to improve robustness we remove malleable features that can change over time or be easily modified by the adversary, as described below.

1) Memory-Mapped Files: Given that many Windows resource names (e.g., mutexes) can be easily modified by the attacker, we only identify memory-mapped file names (e.g., DLL names) from the pool of named resources as features due to their stability over time. Although the import of a DLL can be hidden through import obfuscation, the name of the file is less prone to attacker manipulation since DLLs are an integral part of the Windows OS. However, malware can still hide the import through delay loading. Therefore, we need a feature extraction method that can detect these hidden DLLs.

We detect hidden DLLs by monitoring a special Windows data structure called `OBJECT_ATTRIBUTES`. Stealth loaders [23] and reflective loaders hide loaded libraries by avoiding file mapping and loader APIs. Instead, they use memory functions like the Windows API call `CreateFile` to load their DLLs. `CreateFile` eventually calls `NtCreateFile`, which populates `OBJECT_ATTRIBUTES` with the name of the memory-mapped file during execution. By monitoring this structure, we can capture all loaded libraries that reference a file name, even if they are otherwise hidden from traditional Windows loader data structures. Furthermore, while packing, IAT obfuscation, and delay loading are all straightforward methods to hide an import from the static IAT, these objects can still be detected dynamically using our approach, as shown via the experiment results in Section IV.

2) Handle Counts: The other feature set we consider are handle counts, which we represent as a histogram of the unique handles a malware process accesses categorized by type (e.g., mutex, memory-mapped file, etc.). In contrast to related work [27], our categorizations of handles completely ignore named objects (except memory-mapped file names) because they are easy to manipulate; however, keeping the counts of these objects is important because they approximate the semantics of the program. Beyond naming, our categorizations also ignore the order of handle references in the program, which can be easily manipulated and may vary between systems.

B. Dataset

In this section, we present our malware dataset and feature extraction process.

1) Dataset Details: In the pursuit of realistic performance metrics, our dataset was designed with no ambiguity in the family labels. We use labeling that associates the malware with a specific intrusion set malware campaign, or a commodity malware that has a shared code base with similar properties. This avoids excessive generalization of the malware family and exposes the required malware intelligence for defenders to tailor follow-on defensive activities. Specifically, our malware corpus contains a total of 1,804 samples from 27 different malware families. We used a combination of various sources, such as Variant [48] and Contagio[2], to build the corpus and we verified the ground truth labeling manually.

The malware had many diverse samples within their families. For example, most of the original Bifrose samples were packed with either `aspack` or `scpack`. Several families, including CookieBag, utilized delay loading, while some Bredolab samples used a code obfuscation technique which made some of the code appear as unstructured data; furthermore, many malware samples had varied compiler options and optimizations including some variants of Masad Stealer, which were compiled with Nullsoft Scriptable Install System (NSIS) along with Microsoft Visual C compiler. Finally, malware such as Tbot and Masad are trojans, with their benign functionality widely different across samples.

To evaluate features against a realistic adversary, we created two sub-datasets: M and M'. M contains the original

[1] Ultimate Packer for eXecutables: https://github.com/upx/upx

[2] http://contagiodump.blogspot.com/

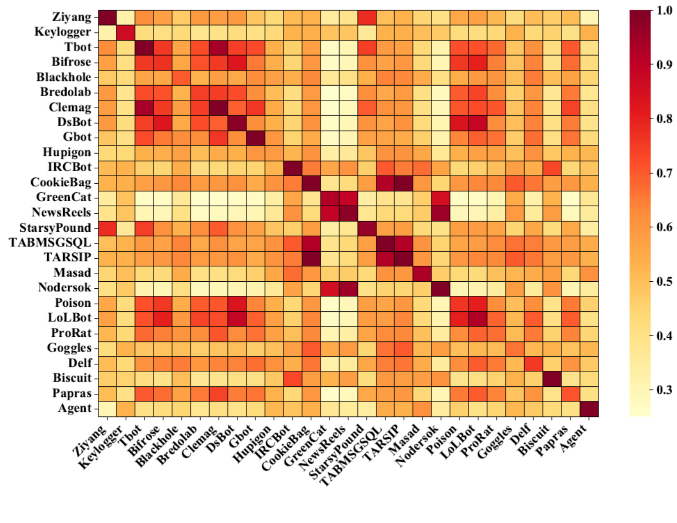

Fig. 1. Pairwise Jaccard similarity of memory-mapped files.

1,044 malware samples collected from the wild; M' was generated by packing each $m \in M$ with UPX to simulate adversarial transformations that may occur at test time. Because packing sometimes breaks the functionality of the program, some generated variants did not execute in our test environment, leaving 1,804 samples of usable malware.

2) Feature Collection: Each malware sample was executed on a Windows 10 virtual machine for two minutes, or until process termination, whichever came first. Samples executed their default code execution paths. Although it is challenging to guarantee complete code coverage in a dynamic environment, our results in Section IV-D suggest that the default code execution paths approximate the malware enough to extract features sufficient for classification.

We used `strace` from DynamoRIO to monitor for system calls that either query or create section objects to capture all memory-mapped files (DLLs). Furthermore, we incorporated the `handles` utility from the SysInternals tool suite [39] to capture the handle count information. Raw feature vectors X are represented as integers for the handle counts, and as a set of strings for memory-mapped files. The integer values were normalized with a min-max scaler, where $x_{scaled} = \frac{x - \min(x)}{\max(x) - \min(x)}$. Strings were vectorized using binary encoding.

C. Feature Analysis and Selection

In this section, we provide insight into the feature analysis and selection process. We analyze and select features based on 80% of the original malware from M. The remaining are omitted to avoid overfitting in the evaluation. Feature analysis and selection also omits any features extracted from M' in order to fairly evaluate the approach against unseen samples.

1) Memory-Mapped Files: As mentioned in Section III, we avoid using most object names as features due to their susceptibility to manipulation (e.g., renaming mutex and file names). The only names that can provide stability over time are names of memory-mapped files, which contain DLLs. When selecting which of the remaining files we use, we first remove any names that appear prominently across all malware families. This reduced the number of unique DLLs from 162 to 99. Removed DLLs included `ntdll.dll`, `kernel32.dll`, and `kernelbase.dll`. All Windows processes map these DLLs since they are required to interface with the native API. We manually removed other DLLs that may be subject to differences between the execution environments. For example, Windows-on-Windows (WOW) subsystem DLLs were removed, which include `wow64.dll`, `wow64win.dll`, and `wow64cpu.dll`.

Feature	**Utility**
`TotalHandles`	High Information Gain
`EtwRegistrationHandles`	
`NamedMappedFiles`	
`ALPCPorts`	Moderate Information Gain
`EventHandles`	
`FileHandles`	
`KeyHandles`	
`MutexHandles`	
`ThreadHandles`	
`UnnamedSectionHandles`	
`SemaphoreHandles`	Low Information Gain
`DirectoryHandles`	
`ChildProcesses`	Low Variance
`DesktopHandles`	
`WindowStationHandles`	
`KeyedEventHandles`	High Correlation
`TimerHandles`	
`TpWorkerFactoryHandles`	
`IoCompletionHandles`	

TABLE I: Features from the handle count class. All features are integer counts of the number of handles. We discarded the features with low information gain, low variance, and high correlation.

We use the pairwise Jaccard Similarity (JS) score between each sample's set of remaining memory-mapped files to examine uniqueness between families and similarities within families. A high JS score (e.g., close to 1) indicates that there is a significant feature overlap between two families. Intuitively, a high JS score within a family but a low one across families is desired for effective classification. We visualize the mean pairwise JS scores and the resulting confusion matrix as a heatmap in Figure 1. Dark spots in the heatmap indicate high feature overlap.

From the diagonal in Figure 1, we gather that we have a good feature set. Due to some intra-family variance in the features, the diagonal is not perfect (i.e., 1.0), with the lowest intra-family similarity score being 0.55 for family Hupigon. Hupigon is configurable through an exploit kit software that can add functionality to meet the attacker's objectives. This variation in functionality accounts for the higher variance in the loaded libraries, thus the lower similarity score. Figure 1 reveals that families GreenCat and NewsReels follow similar intra-family scores. They are both backdoors with different communication mechanisms. Interestingly, they are also both attributed to the same advanced persistent threat (APT1). Other noticeable patterns include inter-family similarity between CookieBag, TABMSGSQL, and TARSIP (all are backdoors), and similarity between Tbot and Clemag (both are trojans that install backdoors). Despite these feature similarities between families, the general trend shows high intra-family similarity scores and low inter-family scores.

2) Handle Counts: The handle counts feature set represents a histogram of unique resource handles of a particular type. Table I shows a complete list of the features we examined. Three types of handles, `ChildProcesses`, `DesktopHandles`, and `Window-Station-Handles`, were removed from the original handles feature set due to low variance. The most surprising removal was the child process count. Indeed, few malware samples in our dataset actually created child processes during analysis. We also calculated the pairwise Spearman rank correlation coefficient to identify correlated features. We found that not only did the number of desktop handles and windows station handles have low variance, they were also linearly correlated. As a result, these features were not considered.

We also observe a correlation between the number of timers and keyed events, which are both synchronization objects. Timers are often used when a thread is waiting to be signaled at a specific time. Keyed events allow threads to set an event or wait on some event for synchronization. Upon examining the system's behavior, we observe that a timer is associated with each keyed event to serve as a timeout in case the event is not released.

There is also a relationship between the number of worker threads in the system's thread pool (`TpWorkerFactory` threads) and the number of I/O ports. Worker threads execute asynchronous callbacks on behalf of the running process, while I/O ports serve as thread-safe queues to transfer work between threads. There are twice as many worker threads as I/O ports because two threads are associated with each I/O port. The thread pool is managed by the operating system, and can vary between OS versions; therefore, these features are subject to change between targets and malware variants. For these reasons, we remove them from the feature set.

3) Feature Importance: To assess feature importance, we converted the strings in the memory-mapped files into binary vectors and normalized the object count features using a simple min-max scaler. Next, we investigate feature importance using information gain and mean decrease impurity (MDI). Specifically, we use permutation importance [46] to find the statistical mean of scores across ten random permutations of the feature space. Let $H(A)$ represent the entropy of some node A in a decision tree, $H(A) = -\sum_{j=1}^{N} p_j \log p_j$, where N is the number of classes (malware families) and p_j is the fraction of samples labeled as malware family j. The information gain, $I(A, B) = H(A) - H(A|B)$, is the reduction in entropy when the data is split from node A to a child node B. We also consider the mean decrease impurity (MDI) using the Gini impurity formula, $H(A) = 1 - \sum_{j=1}^{N} {p_j}^2$.

We use a random forest classifier with 100 trees and ten random feature permutations. The random permutations of features allow us to monitor how scores decrease when the relationship between a feature and the target malware family is broken by removing a feature. This type of permutation importance removes bias from features that are correlated. Figure 2 shows the mean information gain and MDI from the permutations. The feature set of memory-mapped files had 99 unique strings, and hence 99 unique features. Principal component analysis (PCA) [52] was applied to reduce the dimensionality into a single feature set for feature importance comparison. Results showed that the memory-mapped file names, the number of memory-mapped handles, the total number of handles, and the number of event registration handles are among the features that are most important for classification, while the number of semaphores and directories are less important for analysis. Therefore, in Table I both `SemaphoreHandles` and `DirectoryHandles` are removed.

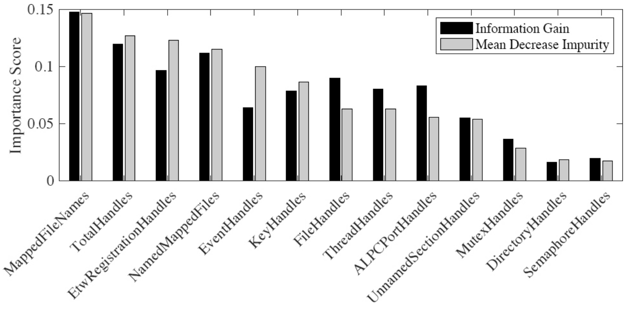

Fig. 2. Mean feature importance scores. `MappedFileNames` is the PCA generated vector representing the names of memory-mapped files. The remaining features are a subset of the handle counts feature set.

IV. Evaluation

In this section, we evaluate the feature set to understand its robustness against concept drift. All experiments were conducted on the dataset described in Section III-B, which contains 1,044 malware samples (M) collected from the wild across 27 malware families, and augmented with 760 samples (M') packed with UPX to simulate adversarial drift.

A. Classifier

The focus of our study is on feature analysis for malware classification, rather than optimizing the actual classifier algorithm. Therefore, we chose a simple ensemble classifier that takes a majority vote of individually trained multiclass classifiers: k-nearest-neighbor (kNN), support vector classifier (SVC), and extra trees classifiers (ETC). The classifiers were implemented with Scikit-learn [30] in Python. For the kNN classifier, we use the Euclidean distance metric for the object counts feature set since the normalized feature space contains continuous data. The names of the memory-mapped files are represented as dichotomous (binary) data after text vectorization, so a Jaccard distance is used for this feature set's distance metric.

B. Experiment Design

We conducted five experiments using various configurations to capture the influence of concept drift that occurs naturally within the malware, and our adversarial drift that was simulated using packing. As discussed in Section III-B, the set M represents the original malware as collected in the wild, while M' are the modified variants. The experiments are visualized in Figure 3, and are designed as follows:

Experiment 1 - Random Splits on M. This experiment evaluates the features without simulated adversarial drift by training and testing the classifier using the non-overlapping subsets in the original malware M. Training and testing

Features	Family Classification					Type Classification				
	Exp. 1	Exp. 2	Exp. 3	Exp. 4	Exp. 5	Exp. 1	Exp. 2	Exp. 3	Exp. 4	Exp. 5
Object Counts	0.88	0.85	0.88	0.88	0.84	0.94	0.95	0.95	0.94	0.93
Memory-Mapped File Names	0.87	0.86	0.86	0.87	0.80	0.93	0.93	0.92	0.93	0.92
Fused	0.88	0.83	0.89	0.88	0.83	0.95	0.94	0.95	0.95	0.94
API Calls (Binary Encoded)	0.74	0.72	0.73	0.68	0.64	0.95	0.95	0.95	0.93	0.93
API Calls (Histogram)	0.90	0.82	0.89	0.84	0.81	0.95	0.95	0.94	0.95	0.93

TABLE II: F1-Score results. The API histogram features performed the best when the training and testing split is not time-aware. The introduction of concept drift (Exp. 2), simulated adversarial drift (Exp. 4), or both (Exp. 5) shows a greater decline in performance for the API histogram features over that of the object counts and memory-mapped file names.

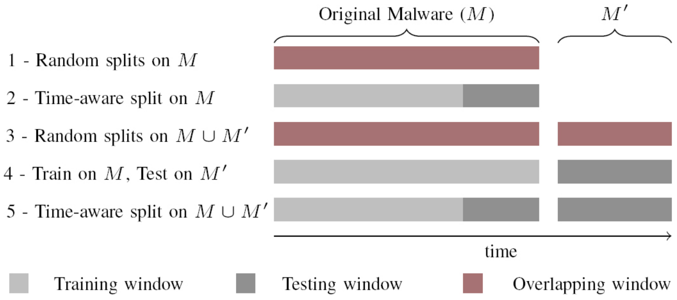
Original Malware (M)
M'
1 - Random splits on M
2 - Time-aware split on M
3 - Random splits on $M \cup M'$
4 - Train on M, Test on M'
5 - Time-aware split on $M \cup M'$
time
Training window
Testing window
Overlapping window

Fig. 3. Experimental design. Experiment 1 uses cross validation (CV) across M, which is not temporally consistent. Experiment 2 is trained and tested using temporally consistent splits on M. Experiment 3 introduces additional artificial drift in M' and uses CV across $M \cup M'$. Experiment 4 trains on M and tests on M'. Experiment 5 has temporally consistent windows and does not train on M'.

were done via a 3-fold cross validation and ignores temporal consistency. The analysis is formally defined as: ${}_{train}, X_{test} = \{x | x \subset X \text{ and } (X_{train} \cap X_{test}) = \emptyset\}$, where X is the feature representation of M.

Experiment 2 - Time-aware Splits on M. This experiment evaluates the features against M using time-aware splits for training and testing to ensure a temporally consistent evaluation. Using compile timestamps, training is conducted on older malware (80%), while testing is conducted on newer malware (20%). More formally: $X_{train} = \{x | x \in X_{old}\}$, and $X_{test} = \{x | x \in X' \text{ and } x \in X_{new}\}$.

Experiment 3 - Random Splits on $M \cup M'$. This experiment teaches the classifiers to learn about the packed samples by including them in training [16]. In this experiment, the original malware and the transformed malware are combined for training and testing using 3-fold cross validation. Formally: $X_{train}, X_{test} = \{x | x \subset (X \cup X') \text{ and } (X_{train} \cap X_{test}) = \emptyset\}$.

Experiment 4 - Train on M, Test on M'. To evaluate the robustness of the feature sets against our simulated adversarial drift, we train the models on M, and test using M'. In this configuration, the models are not aware of the packing transformations. i.e., $X_{train} = \{x | x \in X\}$, and $X_{test} = \{x | x \in X'\}$.

Experiment 5 - Time-aware Split on $M \cup M'$. This experiment considers both concept drift that naturally occurs in the dataset, and our adversarial drift that was simulated with packing. As such, this is the most rigorous and realistic evaluation configuration. Formally: $X_{train} = \{x | x \in X_{old}\}$, and $X_{test} = \{x | x \in X' \text{ and } x \in X_{new}\}$.

To compare our approach to a popular set of dynamic features [5], [40], we extracted the API calls from each sample using the Cuckoo sandbox[3]. We observed 206 unique API calls across the dataset and treated each unique call as a feature. We used two API feature representations: a binary encoded representation and an API histogram. The binary encoded representation encodes each call with a 1 if the call was observed, and 0 otherwise. The histogram represents a bag-of-words to capture the count corresponding to each call. The API count histogram underwent the same preprocessing pipeline (i.e., min-max scaler) as the object counts.

C. Labeling

To measure the impact of labeling on results, the five experiments were performed using two different labeling conventions. First, we used malware *family* labels, shown as **Family Classification** in Table II. Second, we labeled each malware based on its *type* descriptor via manual inspection of the malware's primary capabilities. For type descriptions, we follow the Structured Threat Information Expression (STIX) v2.1 grammar for describing malware[4]. STIX is an open standard language for communicating cyber threat information. By using STIX vocabulary, we ensure a type labeling consistent with the cyber threat intelligence community. The eight type labels generated from our dataset are backdoor, bot, downloader, exploit kit, keylogger, resource exploitation, spyware, and trojan. These results are shown as **Type Classification** in Table II.

D. Results

The results from our experiments are shown in Table II. We present our results as F1-Scores, or the harmonic means of the precision and recall.

1) Summary of Results: Experiment 1 evaluated the chosen features using a traditional cross validation on the original malware. These results indicate that handle counts are slightly more effective features for malware classification than memory-mapped file names, while the fusion of both offers no performance benefit over the handle counts alone, while the fusion of both performs slightly better in type classification. *Experiment 2* shows a performance hit when the training and testing windows are temporally consistent, demonstrating evidence of concept drift. While the API histogram features suffered a 0.08 drop in F1-Score, our fused features only dropped by 0.05, suggesting improved robustness to the drift over API calls.

[3] https://github.com/cuckoosandbox

[4] https://oasis-open.github.io/cti-documentation/

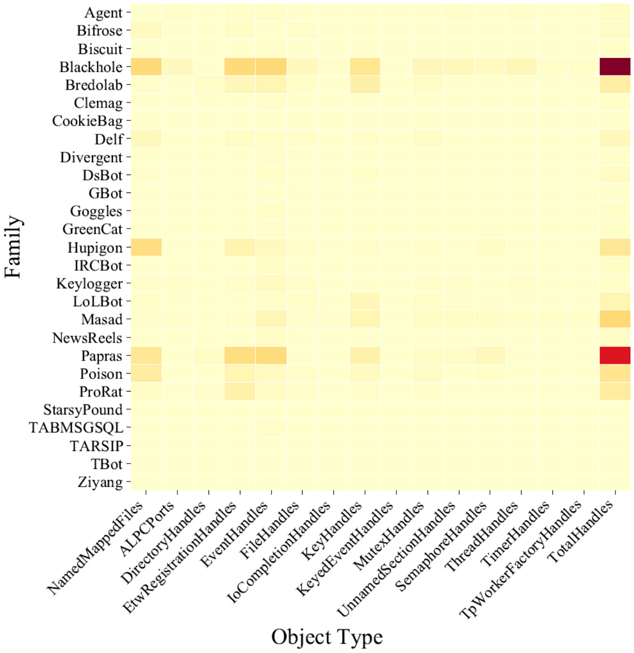

Fig. 4. Mean difference of object counts by family. Darker boxes indicate a larger difference.

Training classifiers on anticipated changes has been shown to improve classifier robustness [16]. Although this type adversarial training can be a useful technique to improve classifier robustness, it assumes the feature perturbations are known during training. This assumption abuses some temporal bias, where the training data assumes future knowledge of the testing data [31]. *Experiment 3* validated this claim, demonstrating negligible performance hits when our simulated drift (via packing) was included in testing and training. However, *Experiment 4* suggests that our features are more robust to packing than API calls. The F1-scores for the classifiers trained on API calls dropped by 0.05 when the packed samples were neglected from training, while classifiers trained on our features only dropped by approximately 0.01.

Experiment 5 contains the most rigorous and realistic setting, where the classifiers were only trained on older instances of the original malware. Testing included the newer instances and the packed versions to simulate concept drift and an adversarial drift. The F1-scores from API-based features dropped by approximately 0.09, while our features only dropped by 0.05, suggesting improved robustness against concept drift and packing over API calls. Next, we will show that this is because the packed samples do not significantly change the variance of the more robust features.

2) Feature Robustness: We analyzed the variance of the object counts between feature vectors X and X', and calculated the perturbation vector as $\delta(x) = x - x'$. Figure 4 shows the delta of the mean object counts between the packed and unpacked samples in the entire dataset. As can be seen, packing generally had little influence on the object counts, demonstrating their stability and robustness to packing. However, we do see larger differences in object count values for commodity malware in Figure 4, specifically Hupigon, Blackhole, Masad, and Papras. Deeper inspection revealed that these malware use a variety of packers already, and our adversarial samples created a double-packed effect, causing unpacking issues at runtime.

The API histogram suffered a performance hit in the temporally consistent evaluations (Experiments 2 and 5), especially for family classification. Packing adds a layer of noise to the API calls. Specifically, the unpacking process makes additional calls to the OS loader API to resolve the import table at process start-up. For example, we observe a large increase in calls to `LdrLoadDll` and `LdrGetProcAddress` at the beginning of the call sequence, altering the histogram's distribution of API call counts. In this case, adversarial training indeed offers a performance benefit. The binary encoded API representation performed worse than all the other techniques evaluated; however, the approach's performance was not affected by packing because the feature representation ignores the number of times each API was invoked.

Inherently robust features reduce the need for additional training and should be a focus of future research in malware classification. Our results demonstrate improved feature robustness by comparing performance degradation when faced with temporally consistent training and testing windows.

3) Labeling Conventions: Our labeling experiments demonstrate the performance implications of choosing a particular labeling convention for malware: type classification or family classification. Due to the decreased specificity of the clusters, type classification experiments yield better results. Most notably, type classification improves the recall scores by ≈ 0.10 due to fewer false negatives. Although type classification performs better, the decreased distinctiveness in type labels reduces the amount of usable information one can infer about the malware. By contrast, specific family labels aid defenders in reasoning about the attack, performing attribution, and assessing damage. For example, we grouped the malware that primarily acted as trojans into a single label for type classification. Indeed, they were accurately classified at test time; however, discriminating between which variant of the trojan (i.e., Clemag vs. DsBot) can expose the necessary intelligence for analysts to pursue follow-on defensive actions. Therefore, we recommend defenders perform a cost-benefit analysis to determine if the increased probability of predicting the correct type is worth the decrease in knowledge about the malware. Next, we take a closer look at the malware samples that were misclassified in our experiments.

E. On Misclassifications

Across all family classification experiments, the CookieBag malware was misclassified into the TARSIP family. Both are backdoors that use HTTPS headers for command-and-control; they are also both attributed to APT1. Despite this misclassification, the families are similar in semantics and origin, making the incorrect result understandable. When these two families were combined into a single family labeled *APT1 HTTPS Backdoors*, we observed the accuracy and F1-Scores of family classifiers improved by $\approx 4\%$ in all family classification experiments. This is another observation on how labeling can influence performance reporting.

We also observe some misclassifications within the commodity malware families (e.g., Delf, Papras, Hupigon, Poison).

These malware samples are built from modular exploit kits and are a part of a malware distribution platform with an infection payload, so the capabilities can vary widely between samples. For example, Delf malware is known to vary in capabilities and intent based on which modules the attacker utilizes. This phenomena is observed again in our Type Classification experiments. All of the type misclassifications came from exploit kits misclassified as backdoors. However, many exploit kits can include a subset of backdoor functionality. In order to improve classifier performance for modular exploit kit malware, we would need to separate the payloads from the broader malware platform. This further illustrates how a "hard" dataset that incorporates classes that are not easily distinguishable can capture performance that is more representative of what may be encountered in a deployed environment.

V. Discussion

Our results indicate that handle counts and memory-mapped file names are effective features for Windows malware classification. The primary benefits include a high degree of robustness against packing and a reduced performance hit due to concept drift.

Labeling Conventions. Our results also expose the effects of labeling on performance. Although malware type labeling is popular and they yield better performance in our study, we suggest their usability in malware analysis is limited. Further, we recommend using standardized type label approaches such as STIX, to create labeling consistent with the cyber threat intelligence community. We caution against the use of anti-virus labels in malware classification as these labels are based on signatures that can be easily evaded through adversarial techniques. We recommend expert malware analyst domain knowledge as the optimal family labeling method.

Dataset Curation. We use a novel dataset that is *hard* due to the variance of commodity malware, informative family labels, an additional testing set of packed malware, and the availability of time-aware training/testing splits. Packing is highly transformative, making it a realizable and effective method to generate testing samples. Therefore, we make three recommendations for curating malware datasets: (i) the labeling methodology should provide analysts with actionable information that addresses the underlying problem of malware classification; (ii) provide testing datasets that are representative of the manipulations that may be observed in the wild; (iii) ensure temporal consistency in training and testing.

Feature Selection and Domain Knowledge. Finally, we utilized domain knowledge derived from systems analysis when choosing features rather than deep learning based feature selection approaches. Deep learning feature selection is a black-box approach to selecting and weighting features based on the training data. Unfortunately, deep learning cannot predict how an attacker will modify malware features to force a misclassification, nor can it provide insight as to why particular features were chosen, making them vulnerable to adversarial samples [8]. As a result, malware analysis expert knowledge is critical for building more robust classifiers.

VI. Conclusion

In this paper, we present handle counts and memory-mapped file names as dynamic features for malware classification. Our selected features are generally agnostic of implementation specifics, making them robust against concept drift and common adversarial manipulations, like obfuscation. Unlike past studies that used adversarial training to boost classifier performance, we found that such training had a negligible impact on our classifier, because the features were already inherently more robust. The results indicate that researchers should continue pushing to uncover robust feature sets rather than relying on adversarial training, which tries to predict how an attacker will manipulate malware. We also show how malware labeling techniques can influence performance reporting, and make recommendations on labeling and dataset practices to create more realistic studies. Future work should adopt rigorous evaluations and feature justification for use in malware classification.

References

[1] H. Aghakhani, F. Gritti, F. Mecca, M. Lindorfer, S. Ortolani, D. Balzarotti, G. Vigna, and C. Kruegel, "When malware is packin'heat; limits of machine learning classifiers based on static analysis features," in *Network and Distributed Systems Security Symposium (NDSS)*, 2020.

[2] M. Ahmadi, A. Sami, H. Rahimi, and B. Yadegari, "Malware detection by behavioural sequential patterns," *Computer Fraud & Security*, 2013.

[3] M. Ahmadi, D. Ulyanov, S. Semenov, M. Trofimov, and G. Giacinto, "Novel feature extraction, selection and fusion for effective malware family classification," in *6th ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2016.

[4] R. Canzanese, S. Mancoridis, and M. Kam, "Run-time classification of malicious processes using system call analysis," in *IEEE Conference on Malicious and Unwanted Software (MALWARE)*, 2015.

[5] F. O. Catak, A. F. Yazı, O. Elezaj, and J. Ahmed, "Deep learning based sequential model for malware analysis using windows exe api calls," *PeerJ Computer Science*, vol. 6, 2020.

[6] J. Choi, H. Kim, J. Choi, and J. Song, "A malware classification method based on generic malware information," in *International Conference on Neural Information Processing (ICONIP)*, 2015.

[7] J. Choi, Y. Han, S.-j. Cho, H. Yoo, J. Woo, M. Park, Y. Song, and L. Chung, "A Static Birthmark for MS Windows Applications Using Import Address Table," in *7th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS)*, 2013.

[8] L. Demetrio, B. Biggio, G. Lagorio, F. Roli, and A. Armando, "Explaining vulnerabilities of deep learning to adversarial malware binaries," in *3rd Italian Conference on Cyber Security (ITASEC)*, 2019.

[9] P. Domingos, "A few useful things to know about machine learning," *Communications of the ACM*, vol. 55, no. 10, pp. 78–87, 2012.

[10] A. Duby, T. Taylor, G. Bloom, and Y. Zhuang, "Detecting and Classifying Self-Deleting Windows Malware Using Prefetch Files," in *2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC)*, Jan. 2022, pp. 0745–0751.

[11] A. Duby, T. Taylor, and Y. Zhuang, "Malware family classification via residual prefetch artifacts," in *2022 IEEE 19th Annual Consumer Communications & Networking Conference (CCNC)*. IEEE, 2022, pp. 256–259.

[12] F. N. Ducau, E. M. Rudd, T. M. Heppner, A. Long, and K. Berlin, "Automatic malware description via attribute tagging and similarity embedding," *arXiv preprint arXiv:1905.06262*, 2019.

[13] A. Ghiassi, T. Younesian, Z. Zhao, R. Birke, V. Schiavoni, and L. Y. Chen, "Robust (deep) learning framework against dirty labels and beyond," in *1st IEEE Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*, 2019.

[14] L. Ghouti and M. Imam, "Malware classification using compact image features and multiclass support vector machines," *IET Information Security*, 2020.

[15] D. Gibert, C. Mateu, and J. Planes, "The rise of machine learning for detection and classification of malware: Research developments, trends and challenges," *Journal of Network and Computer Applications*, vol. 153, 2020.

[16] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel, “Adversarial examples for malware detection,” in *22nd European Symposium on Research in Computer Security (ESORICS)*, 2017.

[17] M. Hassen and P. K. Chan, “Scalable function call graph-based malware classification,” in *7th ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2017.

[18] Í. Íncer Romeo, M. Theodorides, S. Afroz, and D. Wagner, “Adversarially robust malware detection using monotonic classification,” in *4th ACM International Workshop on Security and Privacy Analytics (IWSPA)*, 2018.

[19] C. Jindal, C. Salls, H. Aghakhani, K. Long, C. Kruegel, and G. Vigna, “Neurlux: dynamic malware analysis without feature engineering,” in *ACM Annual Computer Security Applications Conference (ACSAC)*, 2019.

[20] R. Jordaney, K. Sharad, S. K. Dash, Z. Wang, D. Papini, I. Nouretdinov, and L. Cavallaro, “Transcend: Detecting concept drift in malware classification models,” in *26th USENIX Security Symposium*, 2017.

[21] M. Kalash, M. Rochan, N. Mohammed, N. D. Bruce, Y. Wang, and F. Iqbal, “Malware classification with deep convolutional neural networks,” in *9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, 2018.

[22] A. Kantchelian, S. Afroz, L. Huang, A. C. Islam, B. Miller, M. C. Tschantz, R. Greenstadt, A. D. Joseph, and J. Tygar, “Approaches to adversarial drift,” in *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, 2013.

[23] Y. Kawakoya, E. Shioji, Y. Otsuki, M. Iwamura, and T. Yada, “Stealth loader: Trace-free program loading for api obfuscation,” in *20th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, 2017.

[24] A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda, “Accessminer: using system-centric models for malware protection,” in *17th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2010.

[25] H. Lawrence, U. Ezeobi, G. Bloom, and Y. Zhuang, “Shining New Light on Useful Features for Network Intrusion Detection Algorithms,” in *2022 IEEE 19th Annual Consumer Communications Networking Conference (CCNC)*, Jan. 2022, pp. 369–377, iSSN: 2331-9860.

[26] H. D. Menéndez, S. Bhattacharya, D. Clark, and E. T. Barr, “The arms race: Adversarial search defeats entropy used to detect malware,” *Expert Systems with Applications*, 2019.

[27] R. Mosli, R. Li, B. Yuan, and Y. Pan, “A behavior-based approach for malware detection,” in *IFIP International Conference on Digital Forensics*. Springer, 2017.

[28] M. Narouei, M. Ahmadi, G. Giacinto, H. Takabi, and A. Sami, “DLLMiner: structural mining for malware detection,” *Security and Communication Networks*, vol. 8, no. 18, 2015.

[29] L. H. Park, J. Yu, H.-K. Kang, T. Lee, and T. Kwon, “Birds of a feature: Intrafamily clustering for version identification of packed malware,” *IEEE Systems Journal*, vol. 14, no. 3, 2020.

[30] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss *et al.*, “Scikit-learn: Machine learning in python,” *The Journal of Machine Learning Research*, vol. 12, 2011.

[31] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro, “TESSERACT: Eliminating experimental bias in malware classification across space and time,” in *28th USENIX Security Symposium*, 2019.

[32] F. Pierazzi, F. Pendlebury, J. Cortellazzi, and L. Cavallaro, “Intriguing properties of adversarial ml attacks in the problem space,” in *41st IEEE Symposium on Security and Privacy*, 2020.

[33] R. S. Pirscoveanu, S. S. Hansen, T. M. T. Larsen, M. Stevanovic, J. M. Pedersen, and A. Czech, “Analysis of malware behavior: Type classification using machine learning,” in *International Conference on Cyber Situational Awareness, Data Analytics and Assessment (CyberSA)*, 2015.

[34] D. Rabadi and S. G. Teo, “Advanced windows methods on malware detection and classification,” in *ACM Annual Computer Security Applications Conference (ACSAC)*, 2020.

[35] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. K. Nicholas, “Malware detection by eating a whole exe,” in *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[36] 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 (DIMVA)*. Springer, 2008.

[37] R. Ronen, M. Radu, C. Feuerstein, E. Yom-Tov, and M. Ahmadi, “Microsoft Malware Classification Challenge,” *arXiv:1802.10135 [cs]*, 2018.

[38] 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 (RAID)*, 2018.

[39] M. Russinovich, “Sysinternals suite,” *Microsoft TechNet*, 2009.

[40] M. Schofield, G. Alicioglu, R. Binaco, P. Turner, C. Thatcher, A. Lam, and B. Sun, “Convolutional neural network for malware classification based on api call sequence,” in *International Conference on Artificial Intelligence and Applications (AIAP 2021)*, 2021.

[41] S. Sebastián and J. Caballero, “Avclass2: Massive malware tag extraction from av labels,” in *Annual Computer Security Applications Conference (ACSAC)*, 2020.

[42] M. Z. Shafiq, S. M. Tabish, F. Mirza, and M. Farooq, “PE-Miner: Mining structural information to detect malicious executables in realtime,” in *Recent Advances in Intrusion Detection (RAID)*, 2009.

[43] M. R. Smith, N. T. Johnson, J. B. Ingram, A. J. Carbajal, B. I. Haus, E. Domschot, R. Ramyaa, C. C. Lamb, S. J. Verzi, and W. P. Kegelmeyer, “Mind the gap: On bridging the semantic gap between machine learning and malware analysis,” in *13th ACM Workshop on Artificial Intelligence and Security (AISec)*, 2020.

[44] A. Srivastava, A. Lanzi, and J. Giffin, “System call api obfuscation,” in *Recent Advances in Intrusion Detection (RAID)*, 2008.

[45] J. Stiborek, T. Pevnỳ, and M. Rehák, “Multiple instance learning for malware classification,” *Expert Systems with Applications*, vol. 93, pp. 346–357, 2018.

[46] C. Strobl, A. Boulesteix, T. Kneib, T. Augustin, and A. Zeileis, “Conditional variable importance for random forests,” *BMC Bioinformatics*, vol. 9, no. 1, 2008.

[47] Z. Sun, Z. Rao, J. Chen, R. Xu, D. He, H. Yang, and J. Liu, “An opcode sequences analysis method for unknown malware detection,” in *2nd International Conference on Geoinformatics and Data Analysis*, 2019.

[48] J. Upchurch and X. Zhou, “Variant: a malware similarity testing framework,” in *IEEE Conference on Malicious and Unwanted Software (MALWARE)*, 2015.

[49] ——, “Malware provenance: Code reuse detection in malicious software at scale,” in *IEEE Conference on Malicious and Unwanted Software (MALWARE)*, 2016.

[50] G. D. Webster, B. Kolosnjaji, C. v. Pentz, J. Kirsch, Z. D. Hanif, A. Zarras, and C. Eckert, “Finding the needle: A study of the pe32 rich header and respective malware triage,” in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2017.

[51] G. Wicherski, “peHash: A novel approach to fast malware clustering.” *2nd USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2009.

[52] S. Wold, K. Esbensen, and P. Geladi, “Principal component analysis,” *Chemometrics and intelligent laboratory systems*, vol. 2, no. 1-3, 1987.

[53] W. Xu, D. Evans, and Y. Qi, “Feature squeezing: Detecting adversarial examples in deep neural networks,” in *25th Network and Distributed Systems Security Symposium (NDSS)*, 2018.

[54] F. Zhang, P. P. Chan, B. Biggio, D. S. Yeung, and F. Roli, “Adversarial feature selection against evasion attacks,” *IEEE Transactions on Cybernetics*, 2015.

[55] J. Zhao, S. Zhang, B. Liu, and B. Cui, “Malware detection using machine learning based on the combination of dynamic and static features,” in *International Conference on Computer Communication and Networks (ICCCN)*, 2018.