UNIVERSITY OF CALIFORNIA

Los Angeles

Detecting Mimicry Attacks in Windows Malware

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in Computer Science

by

Haikuo Yin

2023

ABSTRACT OF THE DISSERTATION

Detecting Mimicry Attacks in Windows Malware

by

Haikuo Yin
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2023
Professor Leonard Kleinrock, Chair

Ever since the earliest days of the Internet, malware has been a problem for computers. Since then, this problem's severity has only increased, with important organizations like universities and hospitals suffering major security breaches due to malware. As detection techniques get more advanced, so do attackers' evasion attempts. One such method, called the mimicry attack, introduces benign behavior to malware to produce a benign classification in detectors even while retaining its malicious behaviors. In this document, I will describe the work we did on developing malware detection methods that remain effective in the presence of such evasion attacks. Using Windows APIs, our detection pipeline generates a summary of program behavior, vectorizes it in a way that's robust to modifications, and constrains features to reduce the impact of added benign behaviors. We use two methods of constraining features, enforcing monotonicity on them and removing them from the feature vector. To evaluate this detection pipeline and other methods, we use hooking and injection techniques to generate mimicry attacks that can insert benign behavior in more locations than prior work, and are thus produce stronger attacks than prior work. Our results show that our methods can effectively and consistently detect malware that use both mimicry attacks and adversarial attacks with minimal accuracy loss in vanilla data.

The dissertation of Haikuo Yin is approved.

Peter L. Reiher

George Varghese

Glenn D. Reinman

Cho-Jui Hsieh

Leonard Kleinrock, Committee Chair

University of California, Los Angeles

2023

DEDICATION

To my parents, who came from small rural villages and brought me
to this country to give me this opportunity, and I will always be grateful.
To my friends and family, who have been there for me and
supported me throughout this journey.

Contents

List of Figures

List of Tables

Acronyms

API Application Programming Interface.

DLL Dynamic Link Library.

EAT Export Address Table.

FPR False Positive Rate.

IAT Import Address Table.

LGBM Light Gradient-Boosting Machines.

PE Portable Executable.

RVA Relative Virtual Address.

TPR True Positive Rate.

Acknowledgements

Some parts of Chapters 3, 4, 5, 6, 8, and 9 contain content similar to portions of:

- Haikuo Yin, Brandon Lou, and Peter Reiher. 2023. A Method for Summarizing and Classifying Evasive Malware. In Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '23). Association for Computing Machinery, New York, NY, USA, 455–470. https://doi.org/10.1145/3607199.3607207

Brandon Lou wrote and ran scripts to download benign software samples from online repositories, which resulted in a significant addition to the dataset used in the evaluation. He has also helped look for other sources of data, although none were used for this work. Brandon has also worked a lot on implementing and testing some related works, although those results were not included in this work. Still, his work helped decide on which related works to compare with. More generally, he helped discuss ideas on how to respond to reviews and improve the work for resubmission. Peter Reiher, as my advisor, has helped me develop the idea, has read and edited manuscripts for submission, and has pointed out areas to improve or explain more to conform with high standards of review.

I would like to thank Dr. Leonard Kleinrock for his continued guidance and support of our work through the UCLA Connection Lab. Also, this work was done with an Academic API from VirusTotal, which allowed us to look up information of malware samples.

Vita/Biographical Sketch

I attended University of California, San Diego for my undergraduate and master's studies, where I received my Bachelor of Science in Computer Science in 2016 and my Master of Science in Computer Science in 2018. I've published the paper "To Catch a Ratter: Monitoring the Behavior of Amateur DarkComet RAT Operators in the Wild" at the 2017 IEEE Symposium on Security and Privacy, and more recently, I've published the paper "A Method for Summarizing and Classifying Evasive Malware" at the 26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2023), where I attended the conference and presented the work. I have also done internships at Qualcomm (3 separate times), Naval Postgraduate School, Sandia National Labs, and Meta (formerly known as Facebook). In 2021, I won the bonus prize for the Best Attack using Counterfit (a Microsoft attack generation tool) at MLSEC, a Microsoft-sponsored adversarial malware competition.

Chapter 1

Introduction

As early as the 1970s, the first known computer malware, the Creeper Worm and Wabbit Virus, demonstrated the ability to spread over a network and cause damage to host machines [1]. Over the years, countermeasures against malware have grown more sophisticated, but this problem is far from solved: malware is currently still causing massive problems for businesses and organizations [2].

Because this problem is so widespread, there is no shortage of malware detection systems in both industry and academia. While some detection systems rely on signatures generated by a human expert through manual analysis, most newer systems use some form of machine learning to determine whether a sample is malicious or benign. This, in turn, exposes these newer systems to adversarial machine learning techniques that defeat them by modifying the input in ways the system developer had not anticipated.

One class of attacks is the mimicry attack, which was first proposed to defeat intrusion detection systems (IDS) that analyzed system call traces [3]. While the authors proposed many specific methods of evasion, the main idea behind all of them is to simulate the execution of benign software so as to appear harmless to a detector. As the authors state in the paper: "In essence, we are mimicking the behavior of the [benign] application, but with a malicious twist" [3].

More recently, research in this area [4, 5, 6] have shown that malware detectors using Windows APIs (functions used to make system calls in Windows) with machine learning are susceptible to similar manipulations made to malware (such as inserting additional APIs) that can cause malware to be classified as benign even though all the malicious actions are performed. This type of attack can potentially defeat many proposed API-based malware detectors, and real-world attackers can be even more aggressive in their manipulations. However, most recent malware detection papers do not even consider the possibility of such attacks, let alone provide any results that demonstrate their effectiveness against them. It is unknown exactly how often these types of attacks occur in the wild, since by their nature they are difficult analyze, but evasive malware in general (malware that attempt to evade detection) are increasingly more common and make up most of current malware threats [7].

The rest of this document will describe our work in building a malware detection method that can effectively classify software as benign or malicious even in the presence of these evasion methods. Since Windows is the most popular desktop operating system, this work focuses mainly on Windows malware, but I will also discuss how our methods can apply to other operating systems as well. The major contributions of this work include:

1. We develop a method of summarizing API traces as a hierarchical behavior model by grouping together APIs that access related resources into API sequences.

2. We vectorize this summary using relative orderings of APIs in the sequences which can extract important sequential information without being disrupted by inserted API calls.

3. We identify and constrain the most benign features in our machine learning models, so manipulations such as inserting API calls do not cause malicious samples to be misclassified. We constrain features both by enforcing monotonicity and removing them from the feature vector.

4. We improve existing mimicry attack techniques to generate stronger mimicry attacks

in real malware to more thoroughly evaluate the proposed classification methods.

Our results show that our proposed methods most consistently and effectively classifies mimicry attacks in malware across different scenarios as compared to other common methods. We also do additional evaluations and demonstrate that our methods are robust against adversarial attacks as well.

In Chapters 2 and 3, I will provide background information and discuss related works on malware detection and mimicry attacks respectively. In Chapter 4, I will describe the threat model and assumptions about the attacker that we are operating under to set boundaries on the attacks considered in this work. In Chapter 5, I will describe our design and implementation of the first part of a malware detection system, which is a method to summarize API traces to group together and highlight any malicious behaviors. In Chapter 6, I will describe the second part of this system, which is a machine learning model that takes as input the API summary and output whether it describes a malicious or benign sample. That chapter will include information on how we vectorize the behavior summary to be used as input for the model as well as our methods of constraining benign features to limit their impact on a producing a misclassification. In Chapter 7, I will describe how we evaluate our approach, using both popular methods from the literature and methods developed as part of this research, which are based on generating existing forms of mimicry and adversarial attacks, and improvements upon existing attack methods. Chapter 8 will describe the evaluations we did, including the dataset used, the evaluations results on mimicry and adversarial attacks, as well as a case study to help the reader gain a more intuitive understanding of why the proposed methods are effective. Chapter 9 will discuss the implications of these results, the feasibility of using these methods in real-world settings, and possible directions for future research. Finally Chapter 10 will be the summary and conclusion of this document.

Chapter 2

Malware Detection Terms and Definitions

The field of malware detection is large and expansive, and in this chapter, I will describe some basic concepts of malware detection that are pertinent to the rest of this work. These concepts will also be described when they appear in the rest of this document, but this section is included to provide the reader as a resource where these terms are all defined in one place. Note than many of these concepts can apply more broadly to other areas of machine learning or security, but in this work they will be used in the malware detection context.

2.1 Definitions and Background Concepts

First, some common terms and concepts will be defined here as reference for the rest of the document. They can be grouped into machine-learning related terms, statistical measures used to evaluate detection accuracy, and Windows terms.

2.1.1 Machine Learning Terms

- *Decision Tree*: A model design based on a binary tree structure. A prediction is made by traversing the tree from the root to a leaf node, which contains the final prediction. Every non-leaf node contains a feature and a threshold, and the path taken for a sample depends on whether its specified feature value is above or below the threshold.

- *Loss Function*: A function that evaluates the error of a model's prediction as compared the ground truth. This is what the training process attempts minimize. For binary classification, binary cross-entropy is usually used as the loss function.

- *Log Loss/Cross-entropy Loss*: A loss function based on the cross-entropy between the predicted values and true values from information theory [8]. When output as a prediction for a single sample, it can be converted to the probability that the sample belongs to certain class through the logistic function, $f(x) = \frac{1}{1+e^{-x}}$. In this work, this corresponds to the probability that the sample is malicious, which we will call "maliciousness score" in later parts of this document to emphasize this interpretation.

- *Gradient Boosting*: A machine learning technique that builds an ensemble of weaker models (decision trees in this work) where each additional model is built to correct the error of the previous models. The final prediction is the weighted sum of the predictions of all the weak models [9]. This term is usually abbreviated as GBM for Gradient Boosting Machines.

- *LightGBM*: A gradient-boosting model framework that uses additional techniques (gradient-based one-side sampling and exclusive feature bundling) to improve speed and memory usage while maintaining accuracy [10]. This term will be abbreviated as LGBM in later parts of this document.

2.1.2 Evaluation Terms

The following are some common terms when measuring detection accuracy [11].

- *Recall/Sensitivity/True Positive Rate (TPR)*: The proportion of actual positives that are correctly identified as positives by the classifier.
- *False Positive Rate (FPR)*: The proportion of actual negatives that are incorrectly identified as positives by the classifier.
- *Precision*: The proportion of samples identified as positives that are actual positives.
- *Accuracy*: The proportion of all samples that are identified correctly, positive and negative.
- *F1 Score*: A combined score of precision and recall, calculated by taking their harmonic mean.
- *N-fold Cross-validation*: This is a method of evaluating machine learning models where the dataset is divided into N non-overlapping subsets, and the model is evaluated N times, where at each time it is tested on one of the N subsets while being trained on the remaining N-1 subsets. The final result is usually the average of the N runs. For example, in 10-fold cross-validation, the dataset is divided into 10 subsets, and the model is tested on each of the 10 subsets, while being trained on the remaining 9 each time. This is done so that the accuracy is not biased towards any particular split of training data to testing data.

2.1.3 Windows Terms

- *API calls*: Functions used to make system calls in Windows. Can be from the Windows API [12] or Native API [13].

- *Portable Executable (PE)*: The file format for Windows binaries, for both executables and libraries.
- *Dynamic Link Library (DLL)*: A shared library file in the PE format. DLLs export functions for other executable to call. Windows comes installed with several DLLs that export important system calls (also known as API calls) for use by Windows software.
- *Import Address Table (IAT)*: A table in the PE that maps imported functions to their addresses. When the PE is loaded, the Windows loader fills the table with function addresses from loaded DLLs.
- *Export Address Table (EAT)*: A table in the DLL that maps exported functions to their addresses. This is used by the Windows loader or other software to find relevant function addresses.
- *Relative Virtual Address (RVA)*: Address offset relative to the image base address (the address of the first byte of a loaded file). Thus, the actual virtual address is sum of the image base address and the RVA.
- *Registry*: A method of storing system data in Windows, such as system or hardware settings. Registries keys are directory-like containers that hold name-data pairs where the data is stored. Like directories, a key can contain subkeys so the Windows Registry is heirarchical in nature.
- *Mutant*: Another name for mutex in some Windows APIs.
- *Service*: A background process in Windows, similar to a daemon in Linux.
- *Hook*: A method of intercepting events in Windows that allows the user to modify an event, discard an event, or take additional actions.

2.2 Signature-Based vs. Anomaly-Based Detection

Malware detection is generally seen as having two different approaches: signature-based detection and anomaly-based detection [14]. Signature-based detection works by matching the target with a pre-defined list of known malicious patterns (also known as Indicators of Compromise or IOC) and triggering an alert if there is a match. Therefore, signature-based detection focuses mostly on the malicious class and requires a lot of malicious samples to be effective. In contrast, anomaly-based detection compares the target to a known benign baseline and triggers an alert if the target deviates too much from that baseline. Thus, anomaly-based detection focuses mostly on the benign class and requires an accurate description of benign behavior to be effective.

Both methods have their drawbacks. The effectiveness of signature-based detection is entirely dependent on the database of known malware signatures, so it is usually ineffective against new malware or unknown malware, resulting in lower recall than anomaly-based detection in those cases. On the other hand, since anomaly-based detection compares the target to the normal baseline, there is a higher false-positive rate and lower precision (e.g., if the target deviates from the baseline but is not malicious). If machine learning is used, then both detection methods are susceptible to mimicry attacks. For signature-based detection, machine learning would be used to determine if the target is similar to known malware signatures, so the mimicry attack would mimic benign samples with the goal of decreasing its similarity to the signatures. For anomaly-based detection, machine learning would be used to determine if the target is sufficiently dissimilar from the benign baseline, so the mimicry attack would mimic benign samples with the goal of increasing its similarity to the baseline.

With the advent of machine learning, there is a third detection approach that does not fit well in either category. It is sometimes called supervised learning (because it uses labeled data with machine learning) or simply classification. In this approach, both benign and malicious samples are used to train a classifier (hence the need for labeled data), so there is

not a heavier focus on either one. In essence, the classifier compares the target to both the benign and malicious data, outputting the class that it is more similar too. This approach is also susceptible to mimicry attacks, as mimicking benign samples will increase its similarity to the benign data and decrease its similarity to the malicious data. This type of detection will be the focus of the rest of this document, i.e., the detector will be trained on both benign and malicious data to be able to determine whether a given sample is malicious or benign.

2.3 Static Analysis vs. Dynamic Analysis

Orthogonal to signature-based versus anomaly-based detection, malware detection can also be divided into static analysis and dynamic analysis. Static analysis is done by analyzing static artifacts of the malware sample (e.g., the source code or compiled executable) to obtain features used in detection. Static analysis can be done without the execution of a potentially malicious sample, so it is much safer since there is little chance of the malicious sample affecting the host machine or spreading to other devices. However, techniques like packing and encrypting can make static features very difficult if not impossible to access. Furthermore, with the rise of stealthy malware like fileless attacks [15], even being able to obtain static artifacts for analysis becomes less certain.

Dynamic analysis, on the other hand, monitors the behavior of malware either in a controlled virtual environment or in real machines to decide if malicious actions are taking place. This circumvents some of the problems of static analysis, but the analysis is more resource intensive since resources are needed for running the sample and collecting its runtime behavior, and there is some danger of a malicious sample causing harm. Moreover, if executed in a virtual environment, there is also a chance that the malware notices this and stops any malicious behaviors, although this problem is not present if the dynamic analysis is performed on real machines (e.g., as part of a real-time detection system). Nevertheless, dynamic analysis is popular due to the richness of available information as it captures the

actual behavior of the malware, and as such, it will be the focus of this document.

Within dynamic detection, the use of system calls is popular because it provides a good overview of program behavior, since all programs need to use them to access system resources (both for malicious and benign purposes). Furthermore, it is straightforward to capture system calls, due to the well-defined boundary between the program and the operating system, and can be done entirely in software. As discussed in the introduction, because Windows is so popular, this document will focus entirely on the Windows operating system (noting, however, that the same techniques could be applied to other operating systems, with suitable adjustments for their different APIs). In Windows, system calls are invoked through the Native API [13] or the Windows API [12], so dynamic detection in Windows usually monitors these API calls. Additionally, since API calls are inherently sequential, the order in which the calls are made encode important context about the calls, so it is important to capture that sequential information, rather than discarding it, e.g., only considering the frequency of each API call. Therefore, the main focus of this document will be on dynamic detection based on Windows API sequences. It is important to note that while this document focuses on malware detection, any malicious process will need to invoke API calls, whether it is a custom piece of malware or a piece of benign software that has been exploited. Thus, theoretically, this work has contributions beyond just traditional malware detection, such as intrusion detection or detection of fileless or stealthy malware [15].

Chapter 3

Related Works

3.1 API-based Dynamic Analysis for Malware

There are numerous works related to API-based malware detection, and many use n-grams (consecutive API sequences of length n) or tuples to generate signatures (see survey [16]). In this section, we will discuss some recent works that most closely relate to ours (i.e., use more sophisticated models like graphs) to inform the reader how our work is positioned relative to them.

Some works [17, 18, 19] built API graphs from APIs that access the same resource, and represented samples as either binary vectors [17, 18] or n-grams [19] before using machine learning to classify them. However, these works do not extract information from resource names or buffers (such as file contents), which can be very important for classification, as noted in prior work [16].

Other works [20, 21] used resources to assign risk scores to APIs [21] or high-level behaviors [20], which are then used together with behaviors or APIs as features. However, both works used only 5 levels of risk, which only provides very coarse-grained information. Additionally, resource risk may change depending on context, so static risk scores may not always be appropriate, e.g., writing to a data file may be low risk normally, but is high risk

if the file is renamed (indicating ransomware). Finally, neither work considers buffer information. Chen et al. [21] does demonstrate some robustness against adversarial attacks, but they train their surrogate and target models on data sets with no overlap, do not use target model predictions as labels for surrogate model training, and perturbed only APIs but not parameters; all of which we do, so our models are evaluated on more effective adversarial attacks.

Li et al. [22] built API graphs, extracted information from both resource strings and buffers, used graph neural networks to generate feature vectors, and used a multilevel perceptron to classify samples. While extracting more information than previous works, there are still some drawbacks to this work. First, resource strings are analyzed for similarity with known strings, so dissimilar strings that are semantically similar (e.g., "McAfee" and "Kaspersky") are treated as not similar, while our work takes semantic information into account. Additionally, they only check whether buffers contain executables and ignore other important information like how buffers change (like in ransomware), which we do consider. Finally, in their API graphs, every API with the same name is mapped to the same node, with edges linking consecutive APIs, so inserting APIs (see Chapter 4) can have a large effect on the graph, as every inserted API removes one edge and adds two. Thus, this design is very vulnerable to evasion, while our model is designed to be robust to this type of manipulation.

Overall, previous API-based detection methods use various models to extract behavioral information, but important information such as resource names and buffers are either not considered or are inadequately considered. More importantly, these previous works in API-based detection methods do not adequately consider potential manipulations by an attacker to evade detection. Our work introduces a behavior model that encodes sequential API information in a manner that is robust to manipulations, extracts fine-grained, semantic information from resources and buffers, and has demonstrated robustness against effective evasion attacks.

3.2 Evasive Attacks and Countermeasures

Naturally, adversaries want to avoid detection, so we need to consider various evasion techniques. Note that throughout this document, we use the term "evasion" to refer to methods that cause a benign classification when malicious actions are performed, rather than anti-analysis techniques that avoid malicious actions when being analyzed, like in [23, 24]. Those methods are important to study, but they are orthogonal to the work described in this document and can be used in conjunction with our methods.

For malware/intrusion detection, the most popular method involves adding benign elements to malicious samples, either to mimic benign samples [25, 3, 26, 27] (i.e., "mimicry" attacks) or through adversarial machine learning [28, 5, 29, 6, 4]. For API-based attacks, this means adding benign API calls [25, 3, 5, 29, 6, 4], although other methods include replacing API call parameters or generating equivalent attacks [3, 25, 30]. Of these, only adding benign APIs is scalable (see Chapter 4), and this type of evasion was shown to be effective against API sequence malware classifiers that use n-grams or binary features as input [29, 5, 6, 4], which describes most of the related works discussed in Section 3.1, so this type of attack can potentially effect many proposed API-based malware classifiers. Thus, this is the focus of this document, and we will review potential countermeasures against it.

3.2.1 Adversarial Training

Adversarial training is a method that trains the model on generated adversarial samples to increase its robustness to them. Tong et al. [31] evaluated iterative retraining (iteratively performing attacks and training on successful attack samples) against evasive PDF malware. They found that any attacks not used in adversarial training remain effective, similar to an observation made by Šrndić and Laskov [26] that adversarial training only works well if the defenders can anticipate the type of attack performed.

To augment iterative retraining, Tong et al. [31] also introduced the concept of conserved

features, which are features that cannot be modified or the attack will lose its effectiveness. When retraining, the authors added the constraint that conserved features cannot be modified during evasion, which leads to robust detectors against mimicry attacks. While useful for PDF malware (where conserved features are largely features related to embedded JavaScript), conserved features are less straightforward to apply to sequential data like API traces, since malicious sequences can be of varying lengths and may even be broken up by other API calls. Nevertheless, the idea of constraining certain features can be useful and is present in our proposed method.

Grosse et al. [28] has also evaluated adversarial training against evasive Android malware, but they found that it is largely ineffective. Even with adversarial training, misclassification rates range from 67% to 79%, which is still very high. Chen et al. [32] also evaluated adversarial training in their attack paper on Android malware detectors and found them to be moderately successful, increasing the F-score of detecting malware from 0.64 to 0.87. Like Šrndić and Laskov [26], they observed that this relies on a priori knowledge of the attacks, which can be difficult in practice.

3.2.2 Monotonic Models

One method that seems well-suited against evasion by adding benign features is monotonic models. Monotonicity is the property enforced on certain features such that increasing these feature values cannot decrease the final score, so malware cannot induce a misclassification by increasing benign features.

Íncer et al. [33] noticed that certain features of malware are easy to increase but hard to decrease (or vice versa) for the attacker, so they identified these features and made them monotonic. Their monotonic model achieves a detection rate of 62%, down from around 75% for the original model, which the authors attribute as the cost of achieving monotonicity. However, such numbers clearly indicate that there is more work to be done to increase detection rates. Notably, the authors omitted API sequences as features because they did

not fit into the monotonic model.

Chen et al. [34] used verifiably robust training to enforce certain robustness properties in PDF malware classifiers, one of which is the bounded monotonic property (bounded because changes can only be made within a certain distance). One attack they evaluated is the *reverse mimicry* attack, which is when the attacker injects malicious payloads into a benign sample. For these attacks, their best model only achieves about 51% detection accuracy and with a higher false positive rate (FPR) than their other models on vanilla data. The authors also evaluated a number of other methods, including adversarial training, ensemble training, and monotonic classifiers. For reverse mimicry attacks, adversarial and ensemble training both achieved 0% detection accuracy, and monotonic classifiers achieved about 49% detection accuracy. These results show that it is very difficult to defend against reverse mimicry attacks.

Chen et al. [35] has also built a framework that can train classifiers to satisfy 5 robustness properties, one of which is monotonicity. They evaluated their models on cryptojacking websites, Twitter spam accounts, and Twitter spam URLs. They showed that they were able to train classifiers satisfying those properties with minimal to moderate drop in accuracy as compared to baseline models. While their results were impressive, they did not evaluate their models against any evasive attacks, so it is not clear how their models would perform against such scenarios. Additionally, their datasets only require a small number of features to be monotonic (< 10), so their performance on perhaps hundreds of features, as API-based malware detection may require, is not clear.

Other works have used monotonic models on hardware performance counters [36] or static analysis features [37], but to the best of our knowledge, we are the first to use monotonic models for API sequence-based detection for evasive malware.

3.2.3 Metamorphic Detection

Singh et al. [38] used metamorphic testing (analyzing the outputs from additional generated inputs) to detect Android malware that have been repackaged in benign software. They used a standard neural network classifier, but during inference, they took the samples that are classified as benign, removed the N most benign features (using interpretable models), and inputted them into the classifier again. The main idea is that removing benign features would not change the classification of truly benign samples but would expose the malicious features in repackaged malware and allow them to be detected. Thus, a major difference between this work and our proposed "Super Ensembles" (see Section 8.2.2) is that they train a single model on vanilla data, while we train one model on vanilla data and another model on data with benign features removed. In their evaluations, they achieved 94.56% accuracy, 0.98 precision, 0.95 recall, and 0.96 F1-score on repackaged malware. These results are quite good, so we adapt (and improve) their work for API-based malware detection and evaluate it on mimicry attacks, but as our results will show, it performs worse than our methods in this task (see Section 8.3.2).

Chapter 4

Threat Model

For this work, we use the API trace as input and assume that it accurately captures any malicious behaviors in malware. If malicious actions are not reflected in the API trace, then malicious and benign traces are indistinguishable and classification is impossible for methods that rely on API calls. Fundamentally, this assumption means that the same sequence of APIs (including API argument and buffer information) are always considered the same level of malicious, so if the same API sequence is considered either malicious or not depending on external factors not captured in the trace (e.g., some real-world event), then API-based detection will not be effective. For example, a driving application giving the wrong driving directions will not be detectable by API-based detection methods. This is not an issue in most cases because for the most malicious behaviors, we usually consider an API sequence to be malicious as long as it has the potential to be malicious, even if not necessarily so. For example, creating a process and writing to its memory may not necessarily be malicious, but because it is used in many process injection attacks, it is always considered malicious. Nevertheless, it is important to clearly define this assumption for this work. Note that this discussion only applies to malware themselves and not malware that corrupt the data of another piece of software (e.g., malware that corrupts the map data from an otherwise benign driving application), because corrupting the data or another piece of software likely

will require API calls. It is also important to note that this limitation applies to all API-based detection methods, so it is not unique to this work.

In this work, to generate API traces from samples, we use Cuckoo [39] to execute samples and capture their API traces, i.e., perform dynamic analysis, so there is also the implicit assumption that malicious behaviors can be captured in API traces produced by Cuckoo. However, both the attack and defense methods described in this document can work with any API trace, so any limitations of Cuckoo can be overcome by using another dynamic analysis platform.

As discussed in Section 3.2, evasion by adding benign calls to malicious samples can be effective against many proposed API-based malware classifiers, so we use this threat model for all the experiments in this document. Like prior work [28, 4, 5, 6], we allow the attacker to make additional calls to attempt evasion, but not remove, reorder, or modify existing calls. As noted by Rosenberg et al. [5], even though it is sometimes possible to produce malware with equivalent functionality using these methods that we do not consider, doing so requires complex analysis and may not be possible for every sample, and is thus not scalable. Thus, an attacker who wants to create a large number of reasonably evasive malware at scale may find meerly adding API calls attractive. Even with this restriction, we can produce very effective attacks (see Section 8.3). One caveat to these assumptions is we do take some steps to defend against simple, one-to-one replacements of API calls by treating very similar APIs as the same (e.g., `DeleteFileA` and `DeleteFileW` are treated as the same API), so in some cases, the proposed malware detection method is robust to attacks that modify existing APIs, but it is not effective against more sophisticated modifications of APIs.

However, within these constraints, we make generous assumptions about attacker capabilities. We allow any number of additional calls to be made, do not limit their parameters, and allow malicious APIs to be split among many processes or separated by benign APIs. To be feasible, the attacker will need to make the benign calls fully independent from the malicious calls (e.g., modifying different file/registries) and be willing to accept some side

effects (e.g., new files being created), but we believe a motivated attacker is able and willing to do so. As this is a strong estimation of attacker capabilities, we generate attacks to evaluate both this worst-case scenario by using simulated evasive malware and a more realistic scenario by creating real evasive malware (see Chapter 7). These assumptions are a notable departure from prior attacks on sequential API malware classifiers, where the parameters are limited to *no-ops* [5, 6, 29], and the number of additional calls are either explicitly limited [5] or in practice ends up being a small fraction of the original calls [6, 29]. In our evaluations, we mostly insert more APIs than the original (and many more for PE Injection, see Section 8.2), resulting in much more difficult attacks for us to defend against.

We evaluate two types of evasion in this work: *mimicry attacks*, which generates malicious sample without access to the target classifier by mimicking benign samples, and *adversarial attacks*, which do make repeated queries to the target model. Mimicry attacks can be attractive for attackers without access to the target model or who wish to target a large variety of models, while adversarial attacks are suitable for attackers with access to, and a desire to attack, a particular model. For adversarial attacks, our malware detection pipeline must be at least as hard to attack as the underlying machine learning model, and our models take a binary vector as input and output a binary label (see Chapter 6). To the best of our knowledge, there are no adversarial algorithms that can efficiently generate black-box attacks (the most realistic scenario) directly against this type of model, other than brute-force combinatorial enumeration. Instead, we train a surrogate model, attack it with white-box attacks, and use the generated adversarial samples on the target model.

Chapter 5

Hierarchical Behavior Model (Defense)

In this chapter, we will discuss the first part of our detection method. Our detection method consists of 3 stages: 1) the preprocessing stage, where the raw API trace is converted into a behavior summary, 2) the vectorization stage, where the behavior summary is converted into a feature vector, and 3) the classification stage¸ where the behavior summary vector is used as input to a machine learning model for training and classification. See Figure 5.1 for an illustration of this pipeline. The behavior summary not only reduces the size and complexity of the data, but it can also group together related API calls separated by other calls or in different processes, extracting important information for classification and can even provide analysts with a summary of program behavior.

This chapter will discuss mostly the behavior summary (i.e., the preprocessing stage), while the next chapter discusses how this summary is converted into a feature vector and used to train and classify samples (i.e., the vectorization and classification stages).

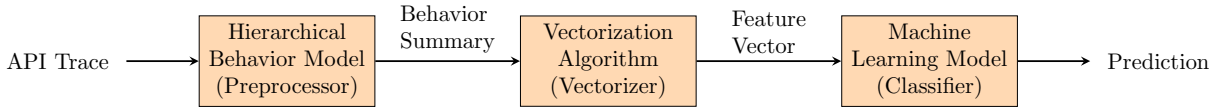


Figure 5.1: The proposed detection pipeline. First the API trace is summarized by the hierarchical behavior model as a behavior summary. Next, this behavior summary is converted to a vector by the vectorization algorithm. Finally, the vector is used as input to a machine learning model for training and classification.

5.1 Behavior Summary Design

The first stage in the classification pipeline is to summarize an API trace into a much shorter summary that highlights its main behaviors. At a high level, the preprocessing stage separates the API trace into groups of calls that correspond to distinct behaviors but are still robust to minor variations in APIs. We use the term "behavior" loosely here to refer to any group of related calls, e.g., reading, writing to, and renaming a file can be one behavior.

The most important aspect of the behavior summary is to group together APIs that access the same resource, because they have a direct impact on each other, but we can also group additional calls together to further reduce the size of the summary. Through reviewing our dataset, we make the following observations about API call sequences:

1. Calls that access the same resource or related resources can often be grouped as one behavior.

2. Calls that access different resources are often separate behaviors, even when they overlap in the API call trace.

3. The sequences of API calls applied to similar or related resources are often the same.

4. Resources (files, registries, etc.) are often hierarchical.

Thus, we designed the preprocessor to group API calls based on resources accessed. For each call in the trace, the preprocessor attempts to find previous calls that have accessed "related" resources (see below for how we define "related"), and if it succeeds, groups the

current call with those previous calls. Calls to the same resource are saved as a sequence in that group. This allows calls affecting the same resource to be grouped together even when separated by other calls. If no related calls are found, a new group is created so that future calls can be added to it. See Algorithm 1 for the pseudocode of this algorithm. The **extract_information** subroutine returns the API function name and the resource accessed by the API call. The **is_related** subroutine returns whether two resources are related (see below). Finally, *calls_seq_map* is a map structure that maps every resource in a group to a list of API calls that accessed it.

Here we define two resources as related if 1) they share the same parent; 2) one resource is the parent of the other; 3) there are other resources already seen that connect them through parent-child relationships; or 4) they are the same resource. By parent, we mean the resource that is one level higher in the resource hierarchy, e.g. the parent of a file is the directory the file is in. The resources we consider in this work are files, registries, mutants, processes, threads, services, hostnames/IPs, hooks, and windows. Calls on resources with no inherent hierarchy (e.g., mutants) are only grouped together if they access the same resource. See Appendix A for all the APIs considered in this work.

Once all the calls have been grouped, the groups need to be summarized. Within each group, calls to the same resource are formed into an ordered API sequence, and all the API sequences in the group (one for each resource) are aggregated and counted (the same API sequence may appear for several resources). Therefore, a group is summarized as a set of API sequences.

For space efficiency reasons and to improve aggregation for similar API sequences, we performed two types of repetition removal on all API sequences before aggregation. First, we replaced consecutive, repeated calls to the same API with one call to that API and the number of repetitions of that call. For example, the API sequence $A \rightarrow B \rightarrow B \rightarrow B \rightarrow C$ would become $A \rightarrow B\{3\} \rightarrow C$. Then, we replaced consecutive, repeated patterns of up to 4 calls long with just one instance of that pattern. For example, the sequence $A \rightarrow B\{3\} \rightarrow$

Algorithm 1 Grouping API Calls

```
Input: list of API calls - api_calls

all_groups ← Empty List
calls_seq_map ← Empty Map
for all call ∈ api_calls do
    func_name, resource ← extract_info(call)
    group_found ← False
    for all group ∈ all_groups do
        for all resource_g ∈ group do
            if is_related(resource, resource_g) then
                group.add(resource)
                calls_seq_map[group, resource].append(call)
                group_found ← True
            end if
        end for
    end for
    if group_found ≠ True then
        new_group ← create new group
        new_group.add(resource)
        calls_seq_map[new_group, resource].append(call)
        all_groups.add(new_group)
    end if
end for
return all_groups, calls_seq_map
```

$C \to B\{2\} \to C$ would become $A \to (B\{3\} \to C)\{2\}$. As seen in this example, if individual calls in the patterns have different numbers of repetitions, the maximum is kept.

To include information about the resources accessed by the APIs, every resource is tagged with information based on its resource name or path. This is done by searching the resource name or path for a predefined list of keywords, each of which maps to a tag. Some API calls (e.g. `CreateProcessInternalW`) include a resource string followed by commands (e.g. "vssadmin.exe Delete Shadows"). In this case, all the command strings are analyzed as well. We develop 36 tags in total, and some important tags include Internet Settings, Security Center, etc. Both tags and their associated keywords are developed manually to demonstrate the effectiveness of our method, but more tags can be added later on. Each resource can be matched to multiple tags, so there is a set of tags associated with each resource.

For these tags, we avoid tagging information specific to certain malware and only tag information general to all Windows machines. For example, we will tag "Internet Explorer" as *browser* or "System32" as *system*, but not a mutant name used by a specific family of malware. This is so we can generalize better against malware that may use different strings. This also means that the keywords only need to be defined once until a new version of Windows is released, and even then many keywords will likely remain the same.

We also decided to tag APIs with buffer information, specifically the buffers of file and memory API calls. This way, we can include information on the data moved between different resources. The information we tag include whether the buffer is or contains a PE (portable executable), whether the buffer is text (all ASCII), and whether there is a change to these tags (e.g., reads a file with ASCII data but writes to it with non-ASCII data, which may indicate ransomware). Like resources, each buffer may be matched to multiple tags. See Appendix B for a list of all resource and buffer tags used in this work.

See Algorithm 2 for the pseudocode of the summarization function. The subroutines **eliminate_API_reps** and **eliminate_pattern_reps** eliminate repetitions as described previously and returns the new, shorter API sequence along with the repetitions for each API. The **tag_information** subroutine generates tags based on the resource name and buffer, and *calls_seq_map* is a map structure that maps every resource in a group to a list of API calls that accessed it. See Appendix C for a short example of a behavior summary group.

5.2 Grouping Implementation

Now that we have an understanding of the high-level design of behavior summary from the previous section, I will describe the implementation details of the grouping algorithm in this section. Note that all code described in this document is written entirely in Python.

Since the core idea of the behavior summary is grouping API calls by resources hierarchically, the main data structure of the grouping algorithm is a list of trees, where each tree

Algorithm 2 Summarizing Groups

Input: list of API groups - all_groups
Input: map of groups and resources to API seq - $calls\_seq\_map$

$all\_summaries \leftarrow$ Empty List
for all $group \in all\_groups$ **do**
    $seq\_reps \leftarrow$ Empty Map
    $seq\_info \leftarrow$ Empty Map
    for all $resource_g \in group$ **do**
        $api\_seq \leftarrow calls\_seq\_map[group, resource]$
        $apis, reps \leftarrow$ **eliminate_API_reps**(api_seq)
        $tags \leftarrow$ **tag_information**$(resource)$
        if $api\_seq \in seq\_count$ **then**
            $max\_reps = seq\_reps[api\_seq]$
            $seq\_reps[api\_seq] \leftarrow max(reps, max\_reps)$
            $seq\_info[api\_seq].add(tags)$
        else
            $seq\_reps[api\_seq] \leftarrow reps$
            $seq\_info[api\_seq] \leftarrow tags$
        end if
    end for
    $seq\_reps \leftarrow$ **eliminate_pattern_reps**(seq_reps)
    $summary \leftarrow (seq\_reps, seq\_info)$
    $all\_summaries.append(summary)$
end for
return $all\_summaries$

represents a group of related resources and the API calls that act on it. Trees are chosen because they are well suited to representing hierarchical resources, where parent nodes can represent parent resources (e.g., parent directories for files or parent registries for registry keys). This allows related resources to be easily grouped in such a structure. For example, suppose the algorithm sees the following four file/directory resources: `C:\`, `C:\Users`, `C:\Users\file1`, and `C:\Users\file2`. The tree that contains these four resources have one node for `C:`, which has a child node for `Users`, which itself has 2 child nodes for `file1` and `file2`. See Figure 5.2 for an illustration of the tree containing these resources.

While the tree structure is used because it organizes resources well, resources are not the only information stored in these trees. Since our overall goal is to summarize an API trace,

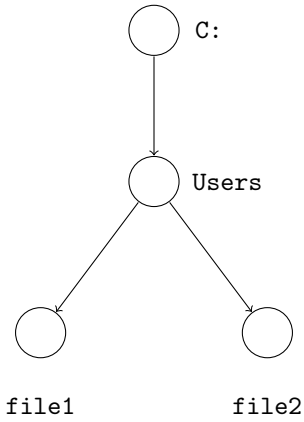


Figure 5.2: A tree that represents the following four resources grouped together: `C:`, `C:\Users`, `C:\Users\file1`, and `C:\Users\file2`.

the APIs that access a resource are also stored in the same node. As described in Chapter 5.1 (and more concretely in Algorithm 1), the APIs that access the same resource need to be stored as a sequence, so in our tree structure, we can store these APIs as a sequence in the node corresponding to their accessed resource. For example, suppose we have the following API trace:

1. `NtOpenFile("C:\")`
2. `NtOpenFile("C:\Users")`
3. `NtOpenFile("C:\Users\file1")`
4. `NtReadFile("C:\Users\file1")`
5. `NtOpenFile("C:\Users\file2")`
6. `NtReadFile("C:\Users\file2")`

Then, the nodes for each resource would contain the APIs that access it (either `NtOpenFile` or `NtOpenFile` and `NtReadFile`). See Figure 5.3 for an illustration of the tree generated for this API trace.

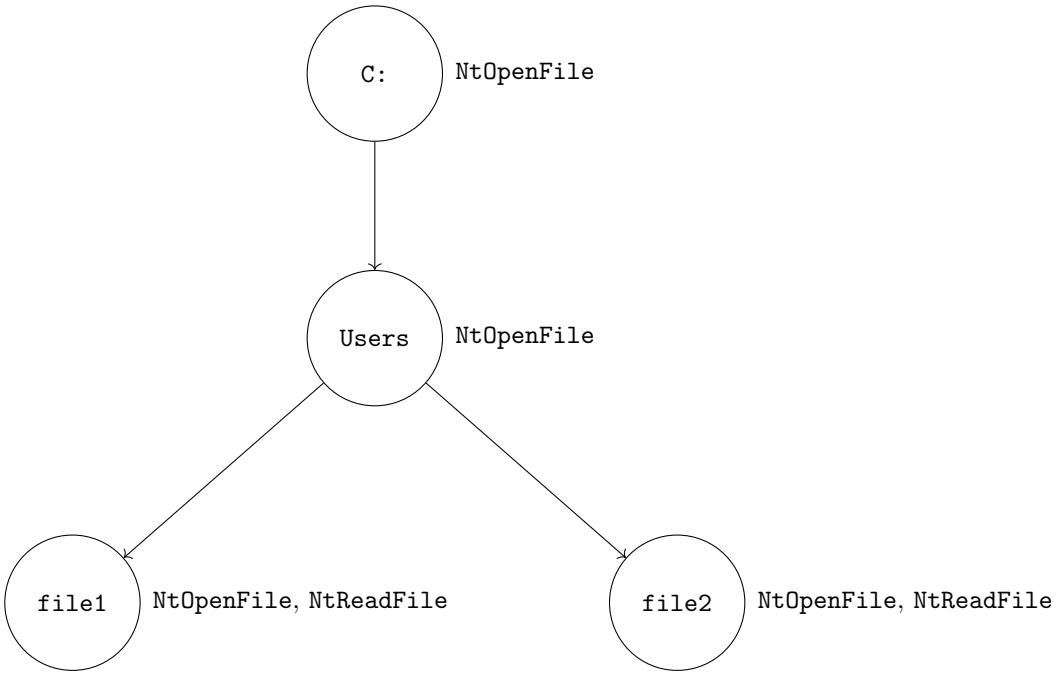


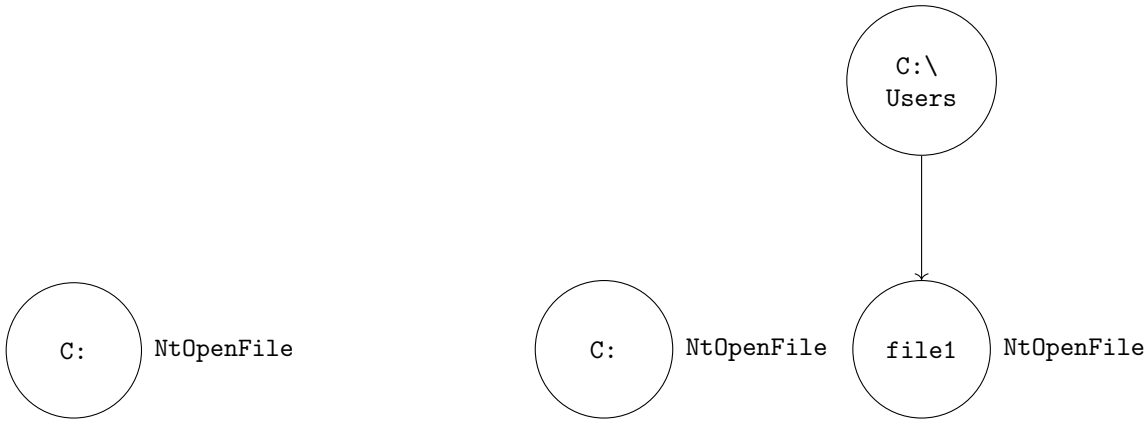
Figure 5.3: A tree that represents the following API trace: `NtOpenFile("C:\")`, `NtOpenFile("C:\Users")`, `NtOpenFile("C:\Users\file1")`, `NtReadFile("C:\Users\file1")`, `NtOpenFile("C:\Users\file2")`, and `NtReadFile("C:\Users\file2")`.

Recall from Section 5.1 that we defined two resources are related if 1) they share the same parent; 2) one resource is the parent of the other; 3) there are other resources already seen that connect them through parent-child relationships; or 4) they are the same resource. Thus, in all 4 situations, as long as we ensure that for every resource we also add a node for its parent resource (if it doesn't already exist), any new resource can be added to a tree if it or its parent can be found in a previous tree. If the algorithm does not find the resource or its parent in an existing tree, then a new tree is created (with a node for its parent resource).

See Figure 5.4 for an example of the resource trees for the following API trace:

1. `NtOpenFile("C:\")`
2. `NtOpenFile("C:\Users\file1")`.

After the first API, since no trees exist, a new tree is created with its resource and API (Figure 5.4a). Since the resource is a root resource (`C:`), no parent node is added because it



(a) One tree is created. **(b)** Second tree is added for a total of 2 trees.

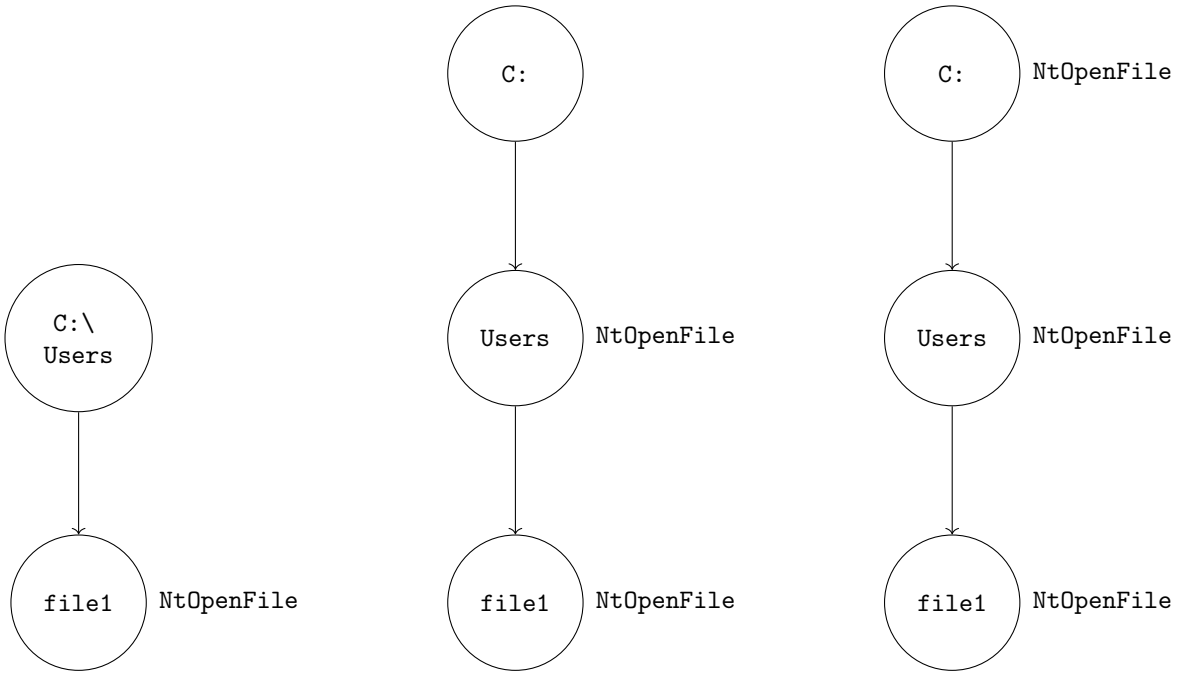
Figure 5.4: Trees that represents the following API trace: `NtOpenFile("C:\")`, `NtOpenFile("C:\Users\file1")`. After the first API, one tree is created (a). After the second API, since the parent resource (`C:\Users`) is not found, a second tree is added with the parent node as the root, for a total of two trees (b).

does not have a parent. After the second API, since the parent of the resource (`C:\Users`) is not found in a previous tree, a new tree is created with the parent resource as the root (Figure 5.4b). This is the desired outcome because `C:\` and `C:\Users\file1` do not have a parent-child or sibling relationship, so they should be in separate trees.

If a resource is found in an existing tree, the API that accesses it is simply added to the existing node, but a parent node may also need to be created for the resource (if a parent node does not already exist). This is so that if a sibling or parent of the parent is encountered later on, then a new tree does not need to be created. Consider the following example API trace:

1. `NtOpenFile("C:\Users\file1")`
2. `NtOpenFile("C:\Users")`
3. `NtOpenFile("C:")`

See Figure 5.5 for the trees created for this example. After the first API, a tree is created for `C:\Users\file1` with `C:\Users` as the root (Figure 5.5a). After the second API, since a node for `C:\Users` already exists, the API is simply added to the existing node, and a



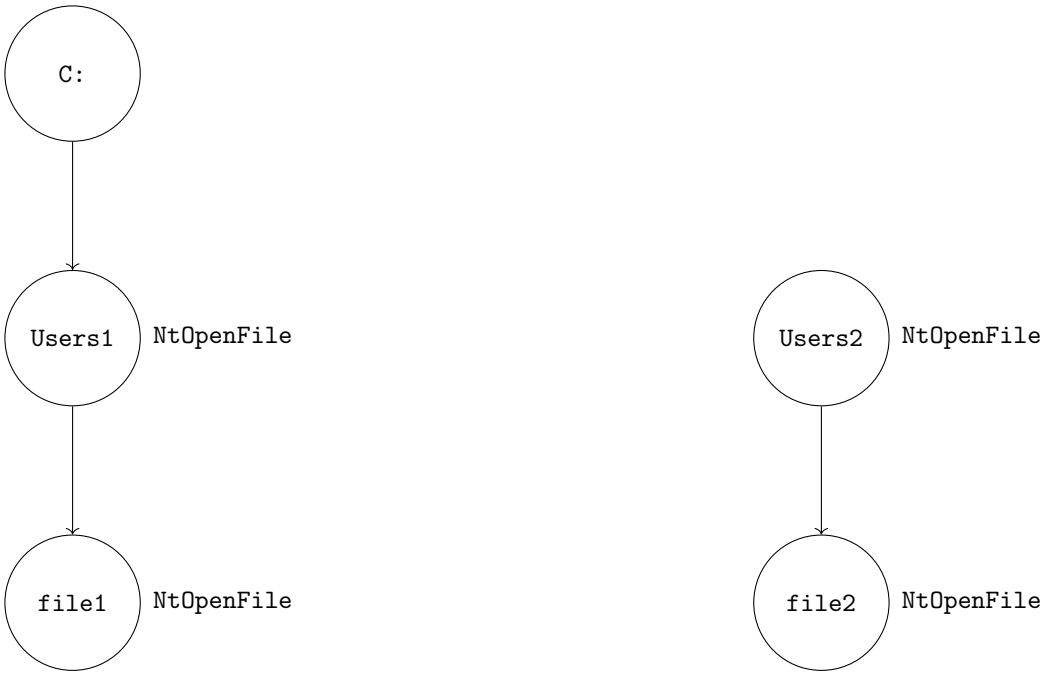
(a) Tree after first API **(b)** Tree after second API **(c)** Trees after third API

Figure 5.5: Trees that represents the following API trace: `NtOpenFile("C:\Users\file1")`, `NtOpenFile("C:\Users")`, `NtOpenFile("C:")`. After the first API, one tree is created with its parent as the root. After the second API, since the resource (`C:\Users`) is found, its API is added to that node, and a node is created for its parent. After the third API (`C:`), because its child (e.g., `C:\Users`) has been encountered, its node has been already created, and it can be added to the same tree (c).

parent node is created (Figure 5.5b). Thus, if the parent of `C:\Users` is encountered (i.e., `C:`), a new tree does not to be created for it (Figure 5.5c). The parent node is not created when a node for the parent resource already exists, to prevent duplicate nodes being created for the same resource in the same tree.

When these parent nodes are created, it is possible that the newly created parent node is the parent or child of some other nodes that already exist. In these cases, we do not merge these potential children nodes into the newly created parent node because we believed doing so would incur too high of a performance cost for a small benefit; for every newly created parent node, we will need to check all existing nodes for any potential children or parent nodes to merge with. Even though we can use sophisticated methods to store and search for possible children nodes (such as creating a map that, for every new resource encountered, maps its parent resource to the newly created node of the current resource for faster search in the future), we find that this situation is so rare that we did not believe the benefits to be worth the additional overhead. Additionally, as will be described below, some APIs access multiple resources, which can lead to the same resource being stored in multiple nodes, which will need to be merged into one node, or the parent node will contain multiple children nodes with the same resource, increasing complexity and overhead in both cases. Thus, for these reasons, we choose to not merge existing children nodes into newly created parent nodes, which does result in some resources that can be grouped together to not be in these rare situations, but since we found that it does not affect detection rates, we do not believe it is worth the additional complexity and inefficiency. For an example of this situation, consider the following API trace:

1. `NtOpenFile("C:\Users1\file1")`
2. `NtOpenFile("C:\Users1")`
3. `NtOpenFile("C:\Users2\file2")`
4. `NtOpenFile("C:\Users2")`



(a) First Tree **(b)** Second Tree

Figure 5.6: Trees created that represents the following API trace: `NtOpenFile("C:\Users1\file1")`, `NtOpenFile("C:\Users1")`, `NtOpenFile("C:\Users2\file2")`, `NtOpenFile("C:\Users2")`. After accessing `"C:\Users1"`, a parent node for `"C:"` is created. Then, when `"C:\Users2"` is accessed, a node for `"C:"` is not created because it already exists, and any future calls to `"C:"` will be added to the existing node. Since we do not migrate existing children nodes when parent nodes are created, these two trees remain separate.

See Figure 5.6 for an illustration of the trees after all the APIs have been added for this trace. After `NtOpenFile("C:\Users1")`, a parent node is created for `"C:"` because it is the parent of `"C:\Users1"`. However, after `NtOpenFile("C:\Users2")`, no parent node is created because since a node for `"C:"` already exists. Since we do not merge existing nodes, these two trees stay separate and will be summarized as two separate groups in the behavior summary.

One may notice that in many of the previous figures, some nodes have no APIs associated with them, because no APIs accessed them (e.g., if they were created as the parent node of an API resource). This is not an issue because if they do not contain an API sequence, then no API sequence is added to the behavior summary, so these nodes do not affect the final

behavior summary.

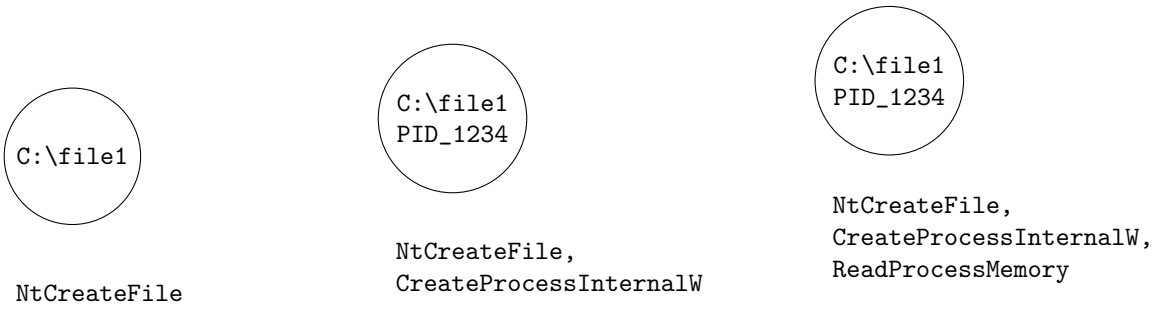
Additionally, to reduce unnecessary variation as well as to limit avenues of attack, some APIs with very similar or identical functionalities are treated as the same API. Most commonly, this involves certain APIs that have two versions with identical functionalities but use either ANSI or Unicode strings and end in either 'A' or 'W' respectively. For example, DeleteFileA and DeleteFileW are identical in functionality (i.e., they both delete a file), but DeleteFileA takes the input filepath as a ANSI string, while DeleteFileW takes the input filepath as an Unicode string. In these cases, the ending 'A' or 'W' is removed.

Certain API calls can also contain multiple relevant resources. For example, CopyFileA can have both a source and a destination filepath for copying files. For some APIs, these resources may not be of the same type. For example, CreateProcessInternalW can have both a filepath and a PID as resources. The filepath is the file used to create the process, and the PID is the PID of the created process. For these cases, the node adds both resources to it and can add other APIs or children/parent nodes for both resources. For example, consider the following API trace:

1. `NtCreateFile("C:\file1")`
2. `CreateProcessInternalW("C:\file1", "PID_1234")`
3. `ReadProcessMemory("PID_1234")`

See Figure 5.7 for the illustration of the trees created for this example trace. After the first API, a node is created for `C:\file1` (Figure 5.7a) with `NtCreateFile` added to it. After the second API, because `C:\file1` has been found, `CreateProcessInternalW` is added to the same node, and since `PID_1234` is also accessed, it is also added to the same node as well (Figure 5.7b). Finally, after the third API, because `PID_1234` has been found, `ReadProcessMemory` is added to the same node as well (Figure 5.7c).

These nodes with multiple resources act just like nodes with a single resource. Future APIs that match any of the resources in the node can be added to it, and children nodes for



(a) Tree after first API. **(b)** Tree after second API. **(c)** Tree after third API.

Figure 5.7: Trees that represents the following API trace: `NtCreateFile("C:\file1")`, `CreateProcessInternalW("C:\file1", "PID_1234")`, and `ReadProcessMemory("PID_1234")`. After the first API, one tree is created (a). After the second API, since the resource is found, CreateProcessInternalW is added to that node. Because CreateProcessInternalW contains 2 resources, both are added to the node (b). After the third API, because the resource is found, it is added to the same node (c).

children resources of any of the resources in the node can be added as well. Parent nodes can be added as well, but since only one parent is added per node, for APIs with multiple potential parents (e.g., `CopyFileA` has both a source filepath and a destination filepath, and both filepaths may have parents), for simplicity, only the parent of the first resource is added as a parent node.

For APIs that access multiple resources, another consideration is the priority of these resources when used to match with existing resources. For example, `CreateProcessInternalW` contains both a filepath and a PID as resources, so if both resources have existing nodes, then a priority is needed to determine which node should this API be added to. For APIs with a source and a destination filepath (e.g., `CopyFileA`), then only the source filepath is used because the destination filepath either does not exist yet and is created with the API call or does is exist and is replaced by the API call. All other APIs that access multiple resources contain a filepath and a PID, and the filepath takes priority over the PID because that is the more important resource (e.g., the file from which to create the process). Note that this priority only applies to looking for existing resources for the current API, because any subsequent APIs can be matched with any resource in the existing nodes.

Some APIs also access or modify a buffer, which may contain important information. In particular, the APIs that may have useful buffer information are `NtReadFile`, `NtWriteFile`, and `WriteProcessMemory`. For `NtReadFile` and `NtWriteFile` buffers can indicate ransomware if the content read from the file is text (all ASCII characters), but the content written to the file is not. `NtWriteFile` buffers can also indicate if a particular type of file is written; e.g., an executable. Similarly, `WriteProcessMemory` can also indicate when an executable is written to memory (e.g, for PE injection). We did not include buffer information for `ReadProcessMemory` because it does not modify memory, so any tags (e.g., whether is it an executable), are not as useful, and since ransomware does not encrypt the contents of memory, the difference between read and written memory is also not as useful.

For the buffers of file APIs (`NtReadFile` and `NtWriteFile`), the algorithm keeps track of the buffer state for each file opened and stores a "buffer snapshot" (the current state of the buffer) after each `NtReadFile` and `NtWriteFile`. This is necessary because these calls do not necessarily access the entire file; a subsection of the file can be accessed, depending on the *offset* parameter and length of buffer. Thus, the algorithm modifies only the sections read or written and takes a snapshot after each file read or write. For the buffers of `WriteProcessMemory`, because we do not keep track of `ReadProcessMemory` buffers and because process memory can be modified so frequently, we simply store what is written at each call.

Finally, if a program consists of multiple processes, then this entire procedure is repeated for every process in the program, with the APIs from every process being added to the same list of trees. This allows API calls that access the same resource but are split across different processes to be placed in the same trees so sequential information can be extracted. Cuckoo already groups API calls by process, so this is straightforward to determine.

The following is a high level summary of the grouping algorithm: As the algorithm reads through the trace of API calls, for every call it attempts to find a previous tree that has a node with its resource or a node with the parent of its resource. If it is successful, then it

will add the current API call to either the node with the resource (if there is already a node with that resource) or create a new node as a child of the parent resource node. If the found resource node is currently a root of a tree, then a node of the parent resource (if it exists) is created and added as the parent node. Each node will contain a list of API calls that represent the API subsequence that has operated on that resource.

If it is unsuccessful in finding a suitable node, then a new tree will be created with the parent of the current resource as the root. This is so future resources that are siblings of the current resource (i.e., resources that share the same parent) can be added to the same tree, but resources that are distant descendants or ancestors will not be added to the same tree. This ensures that only related resources are added to the same tree. Recall that we defined two resources as related (not including same resource) if 1) they share the same parent; 2) one resource is the parent of the other; or 3) there are other resources already seen that connect them through parent-child relationships.

One additional detail to note is that the algorithms described in this section assumes resources are readily available in API parameters and abstracts away from the details of obtaining these resources. In actuality, many APIs operate not on resources directly, but on *handles*, which are unique identifiers for resources in Windows. For example, when a resource is created or opened (e.g., NtCreateFile or NtOpenFile) a handle is returned that identifies the newly created resource. Then, when other APIs access the resource (e.g., NtReadFile), the handle is passed in as a parameter to the function rather than the resource name. Thus, this handle is what's recorded in the API trace. The code implementation keeps track of the mapping between handles and resources when resources are created or opened and converts handles to resource names when they are used by subsequent APIs, so this detail can be abstracted away from in the algorithms described here.

5.3 Summarizing Trees Implementation

Once all the API calls have added to trees, all the trees are traversed and summarized, which results in a sequence of summarized trees (where each tree represents a group of APIs) in the same order that they were created. To summarize a tree, the algorithm performs a pre-order depth-first traversal where each node is processed and aggregated with the summarized information from its child nodes.

For each node, the algorithm first extracts the sequence of APIs associated with it. Recall that most of the time, each node contains one resource, so the sequence of APIs associated with it is the sequence of APIs that accessed that resource. In cases where API calls access multiple resources, these resources are stored in the same node, so the sequence of APIs associated with it is the sequence of APIs that accessed any of the resources. The API sequence is then processed to remove repetitions of consecutive APIs as described in the `eliminate_API_reps` subroutine from Algorithm 2. This results in a shorter API sequence (with no consecutive, repeated calls) along with the number of repetitions for each call.

The pseudocode for `eliminate_API_reps` is described in Algorithm 3; it iterates through all the APIs in a sequence, and if the current API is the same as the previous API, the API is not added to the new sequence and the number of repetitions of the previous API is increased by 1. If the current API is different from the previous API, the API is added to the new sequence with a repetition of 1. The algorithm then outputs this new, shorter sequence along with the number of repetitions for each API in the new sequence.

Next, the information tags for resources and buffers are applied to the current node. For resource tags, the resource string is matched against a pre-defined list of keywords, and for any keyword matches, the tag is applied. See Appendix B, for the list of keywords and their associated tags. To store the tags, a binary vector is used where each element corresponds to a tag, with a 1 indicating the presence of the tag while a 0 indicates its absence. This way, the tags can be stored compactly and can be combined with tags of other nodes using the binary OR operation. Similarly, for buffers, the sequence of buffer snapshots are processed

Algorithm 3 Eliminating API Repetitions

```
1: Input: API sequence - api_sequence
2:
3: prev_api ← null
4: new_sequence ← Empty List
5: repetitions ← 0
6: for all api ∈ api_sequence do
7:     if prev_api = null then
8:         repetitions ← 1
9:     else if prev_api = api then
10:        repetitions ← repetitions + 1
11:    else
12:        new_sequence.append((prev_api, repetitions))
13:        repetitions ← 1
14:    end if
15:    prev_api ← api
16: end for
17: if prev_api ≠ null then
18:    new_sequence.append((prev_api, repetitions))
19: end if
20: return new_sequence
```

for buffer tags (e.g., if the buffer starts with "MZ", it is an executable). A set of tags is added for each snapshot, forming a sequence of tag sets corresponding to the sequence of buffer snapshots. These will be used later to determine the presence or absence of certain buffer tags, as well as any changes in these tags.

After all the information in the current node is collected, the same information from all its child nodes are collected through a recursive call of the current function and aggregated with the information of the current node. Resource tag vectors of the current node and all its child nodes are OR'ed together, and buffer tag sets are unioned together. Aggregating API sequences is more complicated. The API sequences of the current node and its child nodes are also collected in sets. If an API sequences has not been seen before, then it and its API repetitions are added to the set. If an API sequence has been seen (i.e., the same APIs in the same order has been seen in a previous sequence, although not necessarily with the same number of repetitions), then the sequence is not added to the set, but the repetitions

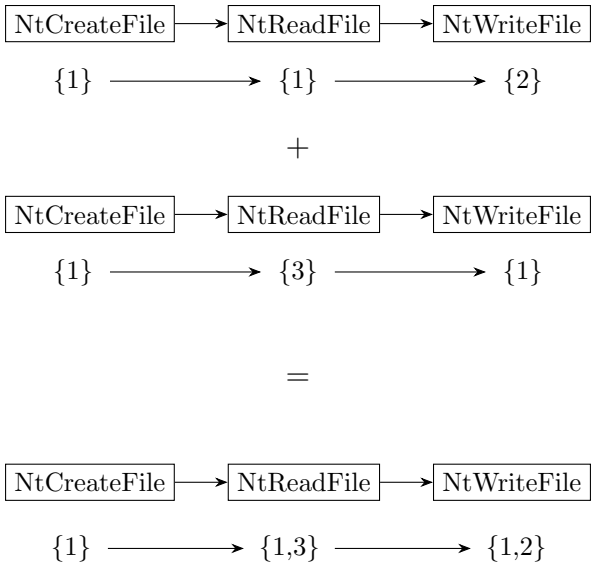


Figure 5.8: Illustration of aggregation of API sequences. After removal of consecutive APIs, each sequence is transformed into an API sequence and a sequence of numbers representing the number of repetitions for each API. When aggregating sequences that are the same, all the repetitions for each API are placed into a set, to calculated minimum and/or maximum repetitions.

of each API are added to a set of repetitions for each API in the previously seen sequence. See Figure 5.8 for an illustration of the aggregation of API sequences with repetitions. We do this so the maximum and minimum repetitions for each API in every sequence can be extracted later. Starting at the root node, the information from its children is aggregated recursively until all the information in the tree is aggregated together.

Once all the information for each tree is aggregated, the **eliminate_pattern_reps** subroutine from Algorithm 2 is applied to the aggregated API sequences to further reduce their length. See Algorithm 5 for the pseudocode of this subroutine. This algorithm makes use of two additional subroutinges, **get_all_partitions** and **eliminate_consecutive_repetitions**. The **get_all_partitions** subroutine partitions a sequence into subsequences of a specified window size w, starting with the first k elements of the sequence, where k ranges from 1 to w. For example, consider the following sequence:

$$[a, b, c, d, e, f, g]$$

For a window size of 3, the partitions returned by the subroutine would be:

- $[(a), (b, c, d), (e, f, g)]$
- $[(a, b), (c, d, e), (f, g)]$
- $[(a, b, c), (d, e, f), (g)]$

Notice that for a window size of 3, 3 partitions are returned. See Algorithm 4 for the pseudocode of the implementation of this subroutine. It adds the first n elements of the sequence to each partition as the first subsquence (for $n = 1$ to $window\_size$) and then continuously adds subequences of length $window\_size$ to each of the partitions until the entire sequence is added for every partition. In Algorithm 4, $sequence[i : j]$ refers to the subsequence from index i (inclusive) to index j (exclusive). The **eliminate_consecutive_repetitions** subroutine works very similarly to **eliminate_API_repetitions** from Algorithm 3, but can work on a sequence of any elements (e.g., a sequence of subsequences like in a partition) rather than just for APIs.

For window sizes of 2, 3, and 4, Algorithm 5 uses **get_all_partitions** to get partitions with subsequences of length $window\_size$ and then uses **eliminate_consecutive_repetitions** to eliminate repetitions in these subsequences, thereby eliminating pattern repetitions. The partition that reduces the sequence length the most is kept for each window size. For example, consider the sequence:

$$[a, b, c, a, b, c, a, b, a, b]$$

For window size 2, the partition $[(a, b), (c, a), (b, c), (a, b), (a, b)]$ can be reduced to $[(a, b), (c, a), (b, c), (a, b)\{2\}]$, which is shorter than the partition $[(a), (b, c), (a, b), (c, a), (b, a), (b)]$, which cannot be reduced at all. Then, for window size 3, the most effective partition for the new sequence, $[a, b, c, a, b, c, (a, b)\{2\}]$, is $[(a, b, c), (a, b, c), ((a, b)\{2\})]$, because the repeating (a, b, c) can be eliminated, resulting in $[(a, b, c)\{2\}, (a, b)\{2\}]$ as the final sequence.

Algorithm 4 Getting All Partitions

```
Input: Sequence of Elements - sequence
Input: Window Size - window_size

all_partitions ← Empty Array of Size window_size
for index_w ∈ [0, ..., window_size − 1] do
    all_partitions[index_w] ← Empty List
end for
for index_s ∈ [0, ..., sequence.length − 1] do
    if index_s < window_size & index_s ≠ 0 then
        all_partitions[index_s].append(sequence[0 : index_s])
    end if
    all_partitions[index_s mod window_size].append(sequence[index_s : index_s + window_size])
end for
return all_partitions
```

This algorithm is done for each of the API sequences in every tree.

Next, the buffer tags are vectorized as feature vectors for each of the sequences. Some tag features include whether there is a change in tags between buffer snapshots (e.g. all ASCII to not all ASCII, see Appendix B). Thus, for each sequence, all considered tags and changes in tags are searched for across the stored buffer snapshots. This is then converted into a binary vector where each element represents the presence of absence of each tag feature (including both tags and changes in tags).

Finally, after the information in the trees is aggregated and repetitions have been removed from API sequences, the information is used to summarize a tree. For each API sequence in the tree, the sequence and its associated resource and buffer vectors are printed, and a tree is summarized as a set of API sequences and tag vectors, which we call a "group" in the behavior summary because it represents a group of APIs. The list of trees of is thus summarized as a list of groups, which summarizes the entire API trace of a sample.

Algorithm 5 Eliminating Pattern Repetitions

Input: API sequence - *api_sequence*

for *window_size* $\in [2, 3, 4]$ **do**
    shortest_sequence $\leftarrow$ *null*
    partitions $\leftarrow$ **get_all_partitions**(*api_sequence*, *window_size*)
    for all *partition* $\in$ *partitions* **do**
        temp_sequence $\leftarrow$ **eliminate_consecutive_repetitions**(*partition*)
        if *temp_sequence.length* $<$ *shortest_sequence.length* **then**
            shortest_sequence $\leftarrow$ *temp_sequence*
        end if
    end for
    api_sequence $\leftarrow$ *shortest_sequence*
end for
return *api_sequence*

5.4 Behavior Summary Example

To illustrate the full behavior summary pipeline, we will provide an example and illustrate every step of the behavior summary process. Consider the following API trace:

1. `NtOpenFile("C:\Users1\file1.txt")`
2. `NtReadFile("C:\Users1\file1.txt")` Buffer: "Hi"
3. `NtOpenFile("C:\Users1\file1.txt")`
4. `NtReadFile("C:\Users1\file1.txt")` Buffer: "There"
5. `NtOpenFile("C:\Users1\")`
6. `NtOpenFile("C:\Users2\file2.txt")`
7. `NtReadFile("C:\Users2\file2.txt")` Buffer: "MZ\u001f"
8. `NtReadFile("C:\Users2\file2.txt")` Buffer: "\u00e4\u0001"
9. `NtOpenFile("C:\Users2\")`
10. `NtOpenFile("C:\")`

11. `NtCreateFile("C:\virus.exe")`

12. `NtWriteFile("C:\virus.exe")` Buffer: "MZ\u0002\u000e"

13. `CreateProcessInternalW("C:\virus.exe", "PID_1234")`

14. `ReadProcessMemory("PID_1234")` Buffer: "\u0000\u0000"

15. `WriteProcessMemory("PID_1234")` Buffer: "MZ\u002f\u0020"

16. `NtResumeThread("PID_1234")`

Notice that some of these APIs contain buffer information as well as API and resource information, and these buffers are strings of bytes read from or written to a resource. ASCII characters are written as characters, while non-ASCII characters are written as unicode characters in the form of "\uXXXX".

The resulting trees created are illustrated in Figure 5.9. After APIs 1 - 10, the resulting trees look very similar to Figure 5.6 because these APIs follow the same bottom up pattern (with two separate trees for `"C:\Users1\file1.txt"` and `"C:\Users2\file2.txt"` because we do not merge trees), although there are some additional APIs in some of the nodes. APIs 11 and 12 create a new node for `"C:\virus.exe"` under the `"C:\"` node, and API 13 adds the resource `PID_1234` to the same node. Finally, APIs 14 - 16 are also added to that node because they also access the resource `PID_1234`.

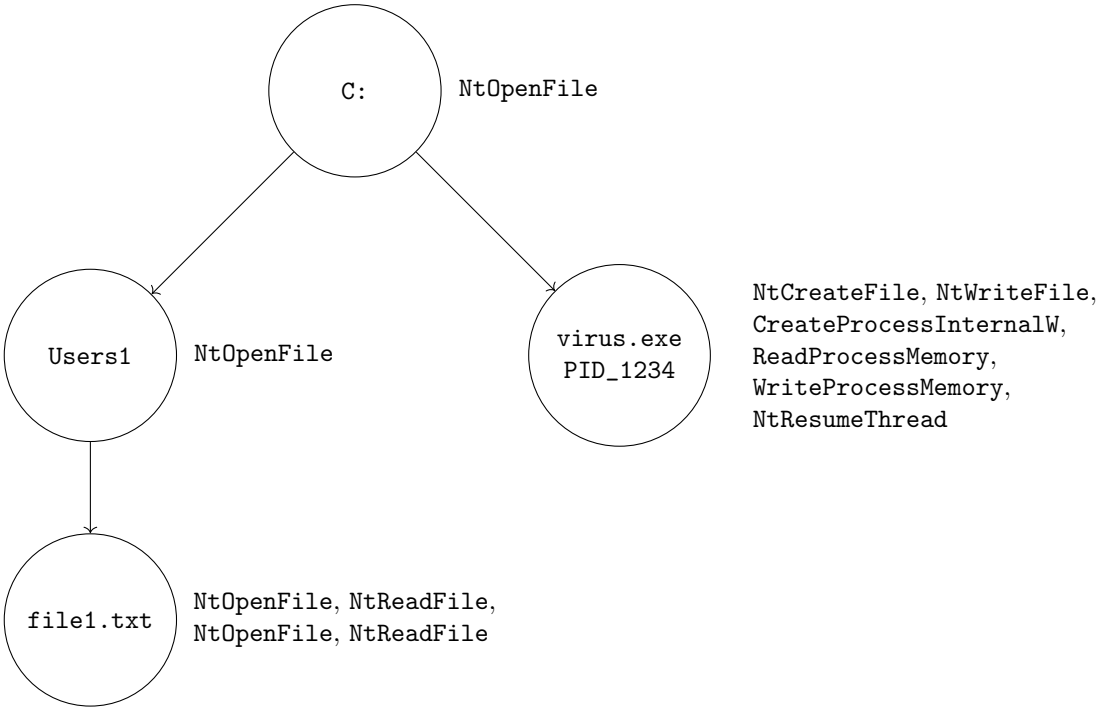
Next to summarize these trees, the API sequences in each tree are aggregated and have consecutive, repeated APIs removed. For the first tree, the unique API sequences are:

1. `NtOpenFile`

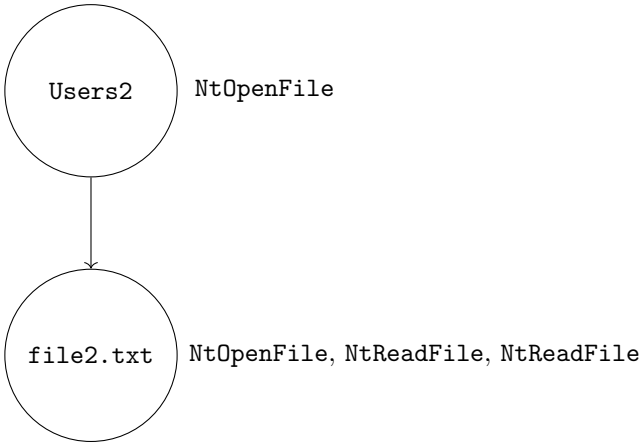
2. `NtOpenFile`, `NtReadFile`, `NtOpenFile`, `NtReadFile`

3. `NtCreateFile`, `NtWriteFile`, `CreateProcessInternalW`, `ReadProcessMemory`, `WriteProcessMemory`, `NtResumeThread`

For the second tree, the API sequences are:



(a) First Tree



(b) Second Tree

Figure 5.9: Trees created after the full example.

1. `NtOpenFile`

2. `NtOpenFile`, `NtReadFile`{2}

Assume for simplicity that there are only two resource tags and three buffer tags. The resource tags are whether the resource string contains the word "user" and the word "virus". These will be represented by a binary vector of length 2, where the first element is whether "user" exists in the resource name and the second is whether "virus" exists in the resource name. The buffer tags used are whether any of the buffers is an executable (begins with "MZ"), whether the buffer changes from all ASCII characters to not all ASCII characters, and vice versa. These will be represented by a binary vector of length 3 where the 3 elements each represent one of these features in that order. Applying these tags, we get for the first tree:

1. `NtOpenFile` - *Resource Tags*: [1,0]

2. `NtOpenFile`, `NtReadFile`, `NtOpenFile`, `NtReadFile` - *Resource Tags*: [1,0] - *Buffer Tags*: [0,0,0]

3. `NtCreateFile`, `NtWriteFile`, `CreateProcessInternalW`, `ReadProcessMemory`, `WriteProcessMemory`, `NtResumeThread` - *Resource Tags*: [0,1] - *Buffer Tags*: [1,0,0]

For the second tree, we get:

1. `NtOpenFile` - *Resource Tags*: [1,0]

2. `NtOpenFile`, `NtReadFile`{2} - *Resource Tags*: [1, 0] - *Buffer Tags*: [1,0,0]

Finally, after removing repeated patterns, we get the following groups which together form our behavior summary:

Group 1

- `NtOpenFile` - *Resource Tags*: [1,0]
- (`NtOpenFile`, `NtReadFile`){2} - *Resource Tags*: [1,0] - *Buffer Tags*: [0,0,0]
- `NtCreateFile`, `NtWriteFile`, `CreateProcessInternalW`, `ReadProcessMemory`, `WriteProcessMemory`, `NtResumeThread` - *Resource Tags*: [0,1] - *Buffer Tags*: [1,0,0]

Group 2

- `NtOpenFile` - *Resource Tags*: [1,0]
- `NtOpenFile`, `NtReadFile`{2} - *Resource Tags*: [1,0] - *Buffer Tags*: [1,0,0]

Chapter 6

Classification and Constraining Features (Defense)

Once the behavior summary has been generated, it is then used as input to a classifier that outputs whether the behavior summary represents a benign or malicious program. The machine learning model we used as the classifier is LightGBM (Light Gradient Boosting Machines or LGBM) [40], a modern boosting framework with improved memory efficiency and speed over other designs while maintaining similar accuracy [10]. The behavior summary is vectorized, processed, and then used to train and evaluate the classifier. There are two important steps in this stage: 1) transforming the behavior summary into a feature vector, and 2) constraining some number of benign features to eliminate the possibility of attackers increasing them to produce a misclassification.

6.1 Behavior Summaries to Feature Vectors

To utilize the behavior summaries as input to the classifier, they need to be first converted into a vector of numerical features. There are several design decisions we made with regards to generating feature vectors:

1. **Unlimited API Sequences** - The first requirement we imposed is to ensure that the feature vector can accommodate potentially unbounded API sequences. API traces can be arbitrarily long, and in our threat model, we allow the attacker to make as many additional APIs as needed to attempt evasion (e.g., add many benign file read API calls), so the feature vector would need to capture the malicious features from among many benign API calls. As an example, a binary vector where each element indicates the existence or absence of an API can work because even if the attackers add many calls to an API, they are unable to remove any existing features. This also has the benefit that no matter how long the API trace it, the feature vector is of constant size.

2. **Binary Vectors** - Another design decision we made is to ensure that all features are binary features. This limits the impact an attacker has when adding benign features. Using the previous example where each feature indicates the presence or absence of an API, if the attacker intends to add a certain benign API call, adding one API call or adding thousands of calls both have the same effect of changing a feature from 0 to 1. However, if the feature vector contains the counts of APIs, adding thousands of API calls can increase the value of a feature by thousands, moving it much further in the feature space. Also, to the best of our knowledge, no adversarial attack algorithms exist for models with binary vector inputs and a binary output, so this design choice also increases robustness to adversarial attacks.

3. **API Sequential Information** - One additional design decision we made is to capture sequential information from APIs because it provides much richer information than information from individual APIs, but we need to do so with our threat model in mind. Recall that the attacker can insert as many additional APIs as desired, which may cause certain malicious features to be removed if not careful.

   For example, assume each element of the feature vector represents the existence of a

particular API 2-gram (e.g., feature (A, B) indicates that APIs A and B are called consecutively), and suppose the 2-gram (A, B) indicates some malicious action in the training data. If the attacker inserts API call C between every instance of (A, B) , then the attacker has removed feature (A, B) from the program and replaced it with (A, C) and (C, B), thus decreasing its probability of being classified as malicious without changing its functionality. The feature vector will need to preserve malicious features in the presence of such changes.

With the aforementioned design choices in mind, we now describe our feature vector design. We start from the individual API sequences in the summary. For each API sequence we generate a vector in the following format:

$$[APIs][API\ Relative\ Ordering][Tags]$$

APIs is a binary vector with each element representing the presence or absence of each API. Similarly, *Tags* is a binary vector where each element represents the presence or absence of each tag. Finally, *API Relative Ordering* is a binary vector designed to capture sequential API information while being robust to the API insertions described above. Each element represents the existence of a certain relative ordering between API calls in the sequence. For example, (A, B) means that in this API sequence, A has appeared before B (or B has appeared after A). This way, no matter what APIs are inserted, feature (A, B) will always remain for that sample. This is only useful because of the short sequences in the behavior summary. For an entire API trace of tens or hundreds of thousands of calls, it is likely that most calls have appeared before and after most other calls.

As described in Section 5.1, the behavior summary removes repetitions, so *API Relative Ordering* would need to take that into account. For repeated consecutive APIs, we set the feature for a certain API appearing after itself to be positive, e.g., the sequence $A \rightarrow B\{3\} \rightarrow C$ would have (B, B) as a feature. For repeated patterns, we set the feature for every pair

of APIs in the pattern to be positive because if a pattern of APIs repeats, then every API appears after every other API in the pattern, including itself. For example, the sequence $(A \rightarrow B \rightarrow C)\{2\}$ would have (A, B), (B, A), (B, C), (C, B), (A, C), (C, A), (A, A), (B, B), and (C, C) as features.

Because the summarization algorithm groups APIs based on resources, APIs without overlapping resources will never appear in the same API sequence. Thus, the APIs naturally partition into sets of APIs that can appear in the same sequence, which we call *categories*. These categories largely correspond to the categories assigned to each API by Cuckoo, although with some modifications (e.g., we separated the "misc" category into multiple categories like Console and Windows). They are: File/Thread/Process/Service, Registry, Mutant, Network, Console, System, Hooks, Windows, and Exceptions. We considered File, Thread, Process, and Service APIs to be in the same category because Process and Service APIs can have files as arguments, and Process and Thread APIs can have process IDs as arguments, so APIs from all 4 types can be in the same API sequence and thus need to be in the same category.

Because *API Relative Ordering* encodes pairs of APIs in API sequences, it only needs to accommodate APIs in the same category, so its maximum length is the number of APIs in a category squared. For example, our largest category, File/Thread/Process/Service, contains 31 APIs (after considering certain APIs with identical functions as the same, e.g., DeleteFileA and DeleteFileW), so the length of *API Relative Ordering* for that category is at most 961, although it can be shorter because we remove empty features, (see Section 6.3).

Therefore, a separate vector (i.e., *APIs*, *API Relative Ordering*, and *Tags*) is generated for each category. Like *API Relative Ordering*, *APIs* only contains the APIs in a category, but *Tags* is the same for all categories. We use a separate *Tags* vector for each category, so different types of resources can get tagged with (sometimes the same) resource tags. For example, a file resource and registry resource may both be related to the browser, so they can both be tagged as such. For each group, the vectors for all the API sequences in that

Algorithm 6 Vectorizing a Behavior Summary

Input: Behavior Summary (Sequence of Groups) - *behavior_summary*
Input: List of API Categories - *categories*
Input: Map containing number of APIS for each category - *num_apis_cat*

all_vectors ← Empty Map
for all *category* ∈ *categories* **do**
    $n\_apis \leftarrow num\_apis\_cat[category]$
    $all\_vectors[category] \leftarrow [0] \times (n\_apis + n\_apis^2 + n\_tags)$ Array of 0's
end for

for all *group* ∈ *behavior_summary* **do**
    for all *sequence* ∈ *group* **do**
        category ← **get_category**(*sequence*)
        $n\_apis \leftarrow num\_apis\_cat[category]$
        tags_vector ← **extract_tags**(*sequence*)
        api_vector ← **vectorize_api_sequence**(*sequence*, *category*, n_apis)

        sequence_vector ← *tags_vector* || *api_vector* //concatenate these vectors
        $all\_vectors[category] \leftarrow$ **bitwise_or**($all\_vectors[category]$, *sequence_vector*)
    end for
end for
return *all_vectors*

group are OR'ed together to create a group vector, and the vectors for all the groups in each category are OR'ed together to create a category vector. The resulting vectors for the different categories are concatenated to create the final feature vector.

6.2 Vectorization Implementation

See Algorithm 6 for the implementation of the vectorization algorithm. First, for each of the categories considered, a zero vector of length $(n\_apis + n\_apis^2 + n\_tags)$ is created and stored, where n_apis is the number of APIs considered in each category and n_tags is the number of tags considered, both resource and buffer tags. Then, for every API sequence in a group, for every group in the behavior summary, the sequence is vectorized, concatenated with its tags vector (as indicated by the || operator in the algorithm), and bitwise OR'ed

with every other vector in the same category. Then, all these vectors (one for each category) are returned.

Vectorizing the tags is straightforward; recall from Section 5.4 that resource tags and binary tags have both already been converted into vectors in the behavior summary, so extracting these vectors (i.e., the **extract_tags** subroutine from Algorithm 6) is simply reading them from the behavior summary. Vectorizing the API sequences is more complicated; as discussed in the previous section, we need to generate a vector that captures both individual API information and API relative ordering information as binary vectors.

See Algorithm 7 for the pseudocode to vectorize API sequences. This algorithm encodes both individual API information and sequential API information present in relative orderings between APIs (i.e., both the *APIs* and *API Relative Ordering* parts of the feature vector from the Section 6.1). As a result, if n_apis is the number of APIs considered, then the returned vector is of length $n\_apis + n\_apis^2$, where the first n_apis elements represent individual API information and the last n_apis^2 elements represent API relative ordering information. Note that because only APIs in the same category can be in the same sequence (as discussed in Section 6.1), n_apis is the number of APIs considered for a specific category, and a different sized vector is returned depending on the category. However, this algorithm is generalizable for any value of n_apis.

Therefore, the algorithm treats the two halves of the feature vector as independent. Because the first n_apis elements of the feature vector are API features, setting an API feature only requires indexing into the feature vector using the index of the API, e.g., line 11 from Algorithm 7 that sets the API feature to 1. The index of the API is a number assigned from 0 to $n\_apis - 1$ for every API in each category and is represented in the pseudocode by the **api_to_index** function.

The second half of the feature vector represents API relative ordering information, so it needs to encode pairs of APIs, which it does with a flattened 2-D array (hence why it has length n_apis^2). Accessing elements is also similar to indexing into a flattened 2-D array, e.g.,

Algorithm 7 Vectorizing an API Sequence

Input: Sequence of APIs - *api_sequence*
Input: API category - *category*
Input: Number of possible APIs in category - *n_apis*

$feature\_vector \leftarrow [0] \times (n\_apis + n\_apis^2)$ Array of 0's of length $n\_apis + n\_apis^2$
$prev\_apis \leftarrow$ Empty List
$pattern\_rep\_apis \leftarrow$ Empty List
for all $api \in api\_sequence$ **do**
    $idx \leftarrow$ **api_to_index**(api)

    $feature\_vector[idx] \leftarrow 1$
    for all $p\_api \in prev\_apis$ **do**
        $p\_idx \leftarrow$ **api_to_index**(p_api)
        $feature\_vector[n\_apis + n\_apis \times p\_idx + idx] \leftarrow 1$ //Set feature (p_api, api)
    end for

    if is_API_rep(api) **then**
        $feature\_vector[n\_apis + n\_apis \times idx + idx] \leftarrow 1$ //Set feature (api, api)
    end if

    if is_pattern_rep(api) **then**
        $feature\_vector[n\_apis + n\_apis \times idx + idx] \leftarrow 1$ //Set feature (api, api)
        for all $pr\_api \in pattern\_rep\_apis$ **do**
            $pr\_idx \leftarrow$ **api_to_index**(pr_api)
            $feature\_vector[n\_apis+n\_apis\times idx+pr\_idx] \leftarrow 1$ //Set feature (api, pr_api)

            if last_pattern_rep(api) **then**
                $pattern\_rep\_apis \leftarrow$ Empty List
            else
                $pattern\_rep\_apis.append(api)$
            end if
        end for
    end if
$prev\_apis.append(api)$
end for
return $feature\_vector$

accessing $array[idx1][idx2]$ is equivalent to accessing $flattened\_array[idx1 \times n\_apis + idx2]$. Because the first n_apis elements of the feature vector are API features, we need to skip over those elements by adding n_apis to the index, e.g., $feature\_vector[n\_apis + idx1 \times n\_apis + idx2]$. Whenever the pseudocode uses any form of $feature\_vector[n\_apis + idx1 \times n\_apis +$

$idx2]$, it can be interpreted as simply indexing into API relative ordering information at $(idx1, idx2)$. Recall from Section 6.1) that for API relative ordering, feature (A,B) indicates that API A has appeared before API B in the sequence.

As an example, supposed there are 5 APIs considered in total: A, B, C, D, and E with indices 0, 1, 2, 3, and 4 respectively. Thus, $n\_apis = 5$, so the length of the API feature vector will have length $5 + 25 = 30$, with the first 5 elements representing individual API features and the last 25 representing API relative ordering features. Suppose we want to vectorize the simple API sequence: $(A \rightarrow B \rightarrow C)$. First, the individual API features for the three APIs (i.e., indices 0, 1, and 2) are set to 1. Then, the API relative ordering features need to be encoded. Because there are only three APIs in the sequence, there are only three API relative ordering features: (A, B), (B, C), and (A, C). For each of these features, if we replace the APIs with their indices, we get the following features: (0, 1), (1, 2), and (0, 2). Converting to feature vector indices using $n\_apis + idx1 \times n\_apis + idx2$, we get $5 + 1 = 6$, $5 + 7 = 12$, and $5 + 2 = 7$ as indices in the feature vector, which we set to 1.

See Figure 6.1 for a visual depiction of the feature vector. The vector is arranged as a matrix in rows of length n_apis (i.e., 5) to more clearly demonstrate the intuition behind the vector design. The numbers to the left of the matrix describe the indices of of the elements for each row. The first row (i.e., the first n_apis features with indices 0 - 4) are the individual API features, so the elements at indices corresponding to the APIs A, B, and C (indices 0, 1, and 2) are set to 1. The rest of the vector, a $n\_apis \times n\_apis$ matrix, represent the pairs of APIs as API relative ordering features. In our example, features (A, B), (B, C), and (A, C) become (0, 1), (1, 2), and (0, 2) if we replace the APIs with their indices, which correspond to indices 6, 12, and 7 in the feature vector. Thus, the corresponding elements in Figure 6.1 are set to 1. From Figure 6.1, we can also see that (0, 1), (1, 2), and (0, 2) can be interpreted as (row number, column number) if the first row is skipped. Note that n_apis being 5 means there are 5 possible APIs, but since not all of them appear in this sequence (i.e., APIs D and E do not appear), some single API features and API relative ordering features are 0

0 - 4	1	1	1	0	0
5 - 9	0	1	1	0	0
10 - 14	0	0	1	0	0
15 - 19	0	0	0	0	0
20 - 24	0	0	0	0	0
25 - 29	0	0	0	0	0

Figure 6.1: Illustration of the given feature vector example. The vector is organized in rows of *n_apis* (5), and the vector indices for each row are listed on the left. The first row (indices 1-4) are the individual API features, while the remaining 5 rows are API relative ordering features.

(i.e., columns 3 and 4 and rows 3, 4, and 5).

This simple example illustrates lines 11 - 15 and line 34 in Algorithm 7. For every API in the sequence, it first sets the individual feature for that API and then the API relative ordering feature of the current API relative to all the previous APIs in the sequence. The remaining sections of the algorithm deal with sequences with API repetitions or pattern repetitions. Note that the specifics of determining whether APIs have repetitions is abstracted away in Algorithm 7 using subroutines **is_API_reps** and **is_pattern_reps**, but in the code, this is determined by whether there is a repetition number after the API or pattern.

For any APIs with repetitions, the feature for that API appearing after itself is set to 1 (line 18 in Algorithm 7) because if an API is repeated, then by definition it has appeared after itself in the sequence. Similarly, for any APIs that appear in a repeated pattern, the feature for that API appearing after itself is also set to 1 (line 22 in Algorithm 7). Additionally, for all previous APIs in the repeated pattern, the reversed API ordering feature relative to the current API is also set to 1. For example, if the repeated pattern is (A, B){2}, then both (A, B) and (B, A) features are set to 1 because (A, B){2} represents the sequence A, B, A, B, so A appears before B and vice versa (note, this is not shown in Figure 6.1). In Algorithm

7, the API relative ordering features in the forward direction has already been set in lines 11 - 15, so lines 23 - 32 is used to set the features in the reverse direction.

Vectorization Example

To illustrate the vectorization algorithm, we will continue using the example from Section 5.4 but removing the last API sequence in Group 1 to keep the number of APIs (and thus vector length) small. Thus, the behavior summary is:

Group 1

- `NtOpenFile` - *Resource Tags*: [1,0]
- (`NtOpenFile`, `NtReadFile`){2} - *Resource Tags*: [1,0] - *Buffer Tags*: [0,0,0]

Group 2

- `NtOpenFile` - *Resource Tags*: [1,0]
- `NtOpenFile`, `NtReadFile`{2} - *Resource Tags*: [1,0] - *Buffer Tags*: [1,0,0]

Assume that there are only two APIs considered, `NtOpenFile` and `NtReadFile` with indices 0 and 1 respectively, so $n\_apis = 2$ and the API sequence vectors will be of length 6. Based on Algorithm 7), the first API sequence in Group 1 will have the API vector $[1, 0, 0, 0, 0, 0]$ because the API feature for `NtOpenFile` is set to 1 but since the sequence length is 1, there are no API relative ordering features to add. The second API sequence contains both `NtOpenFile` and `NtReadFile` as individual APIs, and because the pattern `NtOpenFile`, `NtReadFile` is repeated, both APIs appear after each other and both APIs also appear after themselves, so all API relative ordering features are set to 1. Thus, the vector will be $[1, 1, 1, 1, 1, 1]$.

For Group 2, the first API sequence is identical to the first sequence in Group 1, so the API vector will also be $[1, 0, 0, 0, 0, 0]$. The second API sequence contains both `NtOpenFile` and `NtReadFile` as individual APIs as well as the relative ordering feature (`NtOpenFile`,

`NtReadFile`), and since `NtReadFile` is repeated, it also contains the relative ordering feature (`NtReadFile`, `NtReadFile`). Based on Algorithm 7), (`NtOpenFile`, `NtReadFile`) would be index 3 in the feature vector and (`NtReadFile`, `NtReadFile`) would be index 5 in the feature vector, so the final feature vector will be $[1, 0, 0, 1, 0, 1]$.

Next, when concatenated with their respective resources and buffer tag vectors (where API sequences without a buffer vector will get a zero vector), the four feature vectors, in order, are:

- $[1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0]$
- $[1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0]$
- $[1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0]$
- $[1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0]$

Finally, all the vectors will be OR'ed together to produce the final vector for the category, and since there is only one category in this example, the category vector is the feature vector for the whole summary: $[1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0]$. This vector represents a single sample, and one is generated for each sample in the dataset, so we have converted behavior summaries into vectors that we can use for training and classification with machine learning models.

6.3 Constraining Benign Features

To decrease the impact of adding benign features, our key idea is to constrain the "most benign" features in the feature vector (see below for how we calculate this). We propose and evaluate two methods of constraining the attacker in this way: removing these features or enforcing monotonicity on them. If these features are removed, then it completely eliminates the possibility of the attacker using them to make a sample seem more benign. Unlike metamorphic detection [38] which uses a model trained on vanilla data, our feature removal is applied to all training and evaluation data. If monotonicity is enforced on these features,

then increasing them won't decrease the final maliciousness score, also limiting the attacker. However, in both cases, if too many features are removed or made monotonic, then the accuracy on vanilla (non-evasive) data may suffer. One major portion of our evaluation is determining the optimal number of features to remove or make monotonic. LGBM has a built-in option for monotonicity, so we only need specify which features we want to make monotonic, and LGBM will do so.

To calculate the most benign features, we split our training data into a malicious and a benign set. For each set, we create a vector where each element is the proportion of that set where a certain feature is positive. We then divide the benign vector by the malicious vector element-wise. The result is a vector where the highest numbers appear most often in the benign set and least often in the malicious set (i.e., "most benign"). We then either eliminate the top N most benign features or enforce monotonicity on them.

Note that for efficiency reasons, "empty" features are also removed before benign feature removal. "Empty" features are features that do not exist in any of the samples in our data. The *API Relative Ordering* portion of the feature vector (for all 10 categories) can have many empty features since it is not expected that every API has appeared after every other API. Therefore, these features are removed.

See Algorithm 8 for the pseudocode of calculating the N most benign features. The subroutines that start with "`elementwise_`" perform the specified operation (add or divide) element by element on corresponding elements of the two vectors (that must have the same lengths). They are used to determine the proportion of each set (benign or malicious) where each feature is set to 1 (*ben_proportion* and *mal_proportion*), as described in the previous paragraph. `elementwise_divide` is also used to divide *ben_proportion* by *mal_proportion* to get a ratio vector where the elements with the highest values are the "most benign". Finally, the `largest_elements` subroutine returns the N largest elements of the input vector, which is returned as the N most benign vectors.

Since we are dividing, some situations to be aware of are any divide by zero edge cases.

Algorithm 8 Calculating the Most Benign Features

1: **Input:** Number of Most Benign Features to Calculate - N
2: **Input:** List of Benign Feature Vectors - $ben\_vectors$
3: **Input:** List of Malicious Feature Vectors - $mal\_vectors$
4: **Input:** Number of Features in Each Feature Vector - $n\_features$
5:
6: $all\_ben\_vectors \leftarrow [0] \times n\_features$
7: $n\_ben \leftarrow 0$
8: **for all** $ben\_vector \in ben\_vectors$ **do**
9: $\quad all\_ben\_vectors \leftarrow$ **elementwise_add**$(all\_ben\_vectors, ben\_vector)$
10: $\quad n\_ben \leftarrow n\_ben + 1$
11: **end for**
12: $ben\_proportion \leftarrow$ **elementwise_divide**$(ben\_apis, [n\_ben] \times n\_features)$
13:
14: $all\_mal\_vectors \leftarrow [0] \times n\_features$
15: $n\_mal \leftarrow 0$
16: **for all** $mal\_vector \in mal\_vectors$ **do**
17: $\quad all\_mal\_vectors \leftarrow$ **elementwise_add**$(all\_mal\_vectors, mal\_vector)$
18: $\quad n\_mal \leftarrow n\_mal + 1$
19: **end for**
20: $mal\_proportion \leftarrow$ **elementwise_divide**$(mal\_apis, [n\_mal] \times n\_features)$
21:
22: $ben\_over\_mal\_ratio \leftarrow$ **elementwise_divide**$(ben\_proportion, mal\_proportion)$
23: $most\_benign\_features \leftarrow$ **largest_elements**$(ben\_over\_mal\_ratio, N)$
24: **return** $most\_benign\_features$

Because both the benign and malicious sets are nonempty, there is no divide by zero error in lines 12 and 20. However, this can occur in line 23 if any elements of *mal_proportion* are zero. For these elements, if the corresponding element of *ben_proportion* is nonzero, the result of the division in the code is inf, a value that is considered higher than any other number, which works with the algorithm because features that appear in the benign set but not in the malicious set can be considered the "most benign" features. Because empty features are removed, there should not be any elements that are zero in both *mal_proportion* and *ben_proportion* (except for some very rare cases, discussed below).

An astute reader may notice that this algorithm is equivalent to treating the benign and malicious data as two documents, calculating the TF-IDF of all features for each document as a vector, and dividing the benign vector by the malicious one. For each feature, TF

(term frequency) is the proportion of each set that's positive (which we calculated), and IDF (inverse document frequency) is the same for both documents so they cancel out. With two documents, each feature can only appear in 0, 1 or 2 documents, and features that appear in 0 documents are removed. Features that appear in 1 document can only have an TF quotient of 0 or undefined (because either the numerator or denominator is 0), so they are not affected by the IDF term. This only leaves features that appear in both documents, and by definition the IDF term must be the same for both documents. Thus, we can interpret this score as the importance of a feature for benign samples over the importance of the feature for malicious samples, measuring "how benign" each feature is.

Finally, as mentioned above, there are rare cases where some elements are zero in both *mal_proportion* and *ben_proportion* even when empty features are removed. This is because of n-fold cross validation, where some subset of the training data is removed and used as testing data (see Section 8), so while the feature is not empty for the training data as a whole, if some data is removed for cross validation, then that feature might become empty for the remaining training data. The code treats 0 divided by 0 as the value NaN (not a number) which is also considered larger than any other number including inf, so they are the first features to be constrained (either removed outright or made monotonic). To keep the number of features consistent across different folds, and because these features are so rare ($\sim$ 10 out of around $1,000$ features only for cross validation) and are the first to be constrained, we did not do additional processing to remove these features.

As a simple example, suppose we have 5 vectors in each of the benign and malicious sets as follows:

- Benign: $[1,0,0,0,1]$, $[0,0,0,0,1]$, $[1,1,0,0,0]$, $[1,0,0,0,0]$, $[1,0,1,0,1]$
- Malicious: $[1,1,0,0,0]$, $[0,1,1,0,1]$, $[0,1,1,0,0]$, $[0,0,1,1,0]$, $[0,1,1,0,0]$

Summing up these vectors elementwise and dividing by 5, we get:

- Benign: $[0.8, 0.2, 0.2, 0, 0.6]$
- Malicious: $[0.2, 0.8, 0.8, 0.2, 0.2]$

Finally, dividing these vectors elementwise, we get:

- Benign Over Malicious Ratio: $[4, 0.25, 0.25, 0, 3]$

Thus, in this example, feature 0 and feature 4 are the "most benign", because they appear most often in the benign set and least often in the malicious set. Conversely, features 1, 2, and 3 are the "most malicious" because they appear more often in the malicious set. If $N = 2$, then features 0 and 4 would be removed or made monotonic, and features 1, 2, and 3 would remain for classification. Thus, classifiers trained on the remaining features would mostly use features that appear more often in the malicious set than in the benign set, limiting the possibility of attackers producing a misclassification the by adding benign features. The N values given here are kept simple for illustration purposes, but will be chosen experimentally based on data for the evaluations. More on choosing the appropriate N values will also be discussed in Chapter 9.

Chapter 7

Attack Generation Methods

To evaluate the proposed detection methods, we need to generate attacks and evaluate the effectiveness of the detection methods on them. For these attacks, we will generate both mimicry attacks and adversarial attacks. Additionally, we generate two types of mimicry attacks; first, we simulate them at the API sequence level by combining malicious and benign API sequences, to simulate the resulting API trace of a real mimicry attack; then, we create real mimicry attacks in malware by modifying existing malware executables to make additional benign API calls. Simulating mimicry attacks allows us to fully implement our assumptions in the threat model and evaluate our detection methods with the strongest attacks possible, while creating real mimicry attacks allows us to create a more realistic attack scenario for evaluations. Section 7.1 will describe the process of simulating mimicry attacks, while the following section (Section 7.2) will describe creating real mimicry attacks. Finally Section 7.3 will describe creating adversarial attacks.

7.1 Simulated Mimicry Attack Generation Method

To simulate evasive malware, we take a malicious API sequence and insert them at random locations into a benign API trace (while maintaining their order), which simulates adding all the API calls of some benign software to the malicious calls. If the benign trace has multiple

Algorithm 9 Generating Simulated Mimicry Attack Samples

```
Input: List of Benign Samples - ben_samples
Input: List of Malicious Samples - mal_samples

mal_i ← 0
mal_samples ← randomize_order(mal_samples)
for all ben_sample ∈ ben_samples do
    mal_sample ← mal_samples[mal_i]
    mal_apis ← get_all_apis(mal_sample)
    n_processes_to_split ← min(ben_sample.processes.length, 5)
    mal_apis_chunks ← split_evenly(mal_apis), n_processes_to_split)

    for all ben_process ∈ ben_sample.processes do
        ben_apis ← ben_process.apis
        insertion_indices ← random_indices([0, ben_apis.length], mal_apis.length)
        for all index ∈ insertion_indices do
            ben_apis.insert(index, mal_apis[index])
        end for
    end for
    mal_i ← mal_i + 1
end for
return ben_samples
```

processes, then the malicious calls are split evenly among the processes, up to a maximum of 5 (because our shortest malicious API sequence in our evaluation contains only 11 API calls and many API calls only make sense in pairs, e.g., opening and reading a file). The malicious calls can be API sequences that are manually constructed to represent attacks or API traces read from the malicious samples in our dataset. The constructed attacks will be used to evaluate the detection methods on specific malicious sequences, while using random malicious traces evaluates the detection methods on a larger variety of more realistic API sequences found in actual malware.

See Algorithm 9 for the pseudocode of the process of generating mimicry attacks. It takes as input a list of benign samples and a list of malicious samples. Each sample contains a list of processes, and each process contains a list of API calls. For the malicious samples, each sample can also be a list of APIs, and the code is compatible with both. The malicious samples can all be the same if we want to use the same constructed malicious sequence for

all the benign samples, or they can be different if they're read from the malicious dataset. On line 5, the order of the malicious samples is randomized (with the **randomize_order** subroutine) to randomize benign and malicious sample pairings. but if all the malicious samples are the same, then this randomization does not have any effect.

Then, for each benign sample, the corresponding malicious sample is chosen, and its APIs are extracted using the **get_all_apis** subroutine. This function either extracts and combines the APIs for all the processes of a sample if the sample contains processes, or it simply returns the list of APIs if the sample is a list of API calls. This sequence of malicious API calls is then split into *n_processes_to_split* chunks using the **split_evenly** subroutine, where *n_processes_to_split* is the number of processes in the benign sample up to a maximum of 5. Next, random indices are generated for insertion using the **random_indices** subroutine, which generates *mal_apis.length* integers in the range $[0, ben\_apis.length]$. If $mal\_apis.length \leq ben\_apis.length$, then no integers are repeated, but if $mal\_apis.length > ben\_apis.length$, then the indices are generated such that every malicious and benign API alternates and all the additional malicious APIs are inserted at the end. This is to ensure that every pair of consecutive benign APIs is separated by at least one malicious API. Finally, the malicious APIs are inserted at the generated indices in the list of benign APIs.

7.2 Real Mimicry Attack Generation Method

In this section, I will describe the method used to create the real mimicry attacks in malware. Since this is more complicated than combining sequences of APIs in the previous section, I have split this section into subsections of different concepts. I will first define terms and introduce necessary background information on Windows executables, and then I will describe the techniques used to modify malware executables to make additional API calls along with illustrative examples to show the properties of these techniques. Next I will describe how we implemented these techniques and use them to create mimicry attacks. Finally, I will

Table 7.1: Windows related terms and acronyms used to describe generating real mimicry attacks in malware.

Term	Full Name	Definition
PE	Portable Executable	File format for Windows binaries (executables and libraries)
DLL	Dynamic Link Library	A shared library file in Windows (is in PE format)
IAT	Import Address Table	A table that maps imported functions to their addresses
EAT	Export Address Table	A table that maps exported functions to their addresses
RVA	Relative Virtual Address	Address offset relative to the image base address

discuss some limitations of this method and our implementation.

7.2.1 Terms and Definitions

Table 7.1 contains the terms and definitions related to Windows executables that I will be using throughout this section. They will be explained in more detail later on when they are referenced, but they are included here in this table as a convenient resource for the reader to look up these terms in one location.

7.2.2 Windows Imports Background

Windows binaries, including both executables and libraries, are written in the Portable Executable (PE) file format [41]. There are two important structures in the PE format that relate to making API calls: the Import Address Table (IAT) and the Export Address Table (EAT).

The Import Address Table (IAT) is a structure in the PE file format that maps imported functions to their addresses when they are loaded in memory. When an executable is loaded for execution, the entire PE file is loaded into memory, which means the IAT is also loaded into memory [42]. The Windows loader then loads the imported functions and overwrites the corresponding IAT entries with their addresses [42, 43, 44]. Whenever an imported function is called, a subroutine accesses the IAT and jumps to the corresponding address to execute the function [42].

The complement to the IAT is the EAT, the Export Address Table. As the name suggests, the EAT maps all the functions exported by a PE file, usually a library file (i.e., a Dynamic Link Library or DLL), to its loaded address [41]. Thus, the EAT is consulted whenever addresses of exported functions are needed, either when an executable is first loaded or during execution.

Since the IAT is filled by the Windows loader, the imported functions and DLLs are linked at load time, which is called Load-Time Dynamic Linking [45]. In contrast, imported functions can also be linked by an application during execution, which is called Run-Time Dynamic Linking [46]. This is usually done by manually loading the library (usually through calling the `LoadLibrary` or `LoadLibraryEx` API functions) and looking up the exported function address (usually through calling the `GetProcAddress` API function). Both load-time and run-time linking can be used by applications to import functions.

7.2.3 IAT and EAT Hooking

Based on the functionalities of the IAT and the EAT, if we modify their contents, we can redirect Windows API calls to custom functions that we can use to create mimicry attacks. For example, if we redirect calls to `NtReadFile` to a custom function that makes additional API calls as well as a call to `NtReadFile`, we can insert additional calls in API trace. This process of redirecting calls is called *hooking*, and modifying the IAT or EAT is called IAT hooking or EAT hooking respectively. In this section, we will refer to this custom function as the *hooked* function, while we will refer to the function being hooked as the *original* function.

In IAT hooking, the function addresses in the IAT are overwritten with the address of a custom function, so whenever the original function is called, the program jumps to the custom function instead [42]. Because the program expects a call to the original function (and pushes the appropriate arguments on the call stack), the custom function needs the have the same function signature as the original, but otherwise, there are no restrictions to the custom function. However, since we are using these techniques to insert APIs, all of

the custom functions we write also call the original function to maintain the functionality of original program.

Similarly, in EAT hooking, the function addresses in a DLL's EAT are overwritten, so whenever functions from that library are looked up, the address of the custom function is returned instead [47]. However, unlike the IAT, which uses virtual addresses (VA) directly, the EAT uses relative virtual addresses (or RVA), which is the offset from the base address of the loaded library file [48]. For example, if the DLL "kernel32.dll" is loaded starting at address 0x00400000, and function `ReadFile` is located at address 0x00401000, then the RVA of `ReadFile` is 0x00001000 [48]. Thus, to perform EAT hooking, the RVA of the hooked function relative to the base address of the original function needs to be calculated and used to overwrite the EAT entry, with possible integer overflows. For example, supposed as want to replace `ReadFile` from "kernel32.dll" with `Hooked_ReadFile` from "malicious.dll", and the functions and libraries are at the following addresses:

- "kernel32.dll" is at address 0x00400000
- `ReadFile` is at address 0x00401000
- "malicious.dll" is at address 0x00300000
- `Hooked_ReadFile` is at address 0x00301000

Thus, we need to calculate the address of `Hooked_ReadFile` minus the address of "kernel32.dll" (0x00301000 - 0x00400000) to get the required RVA (0xfff01000). This way, when the address of `ReadFile` is looked up (e.g., with `GetProcAddress`), the base address of "kernel32.dll" is added to the RVA (0x00400000 + 0xfff01000) to get the final address, which is 0x00301000, the address of `Hooked_ReadFile`.

See Figure 7.1 for an example of the calculated RVA from EAT hooking in an actual malware sample. Note that in this figure as well as the rest of this document, I use `OutputDebugStringA` as a "print" function to display strings. Because Cuckoo captures

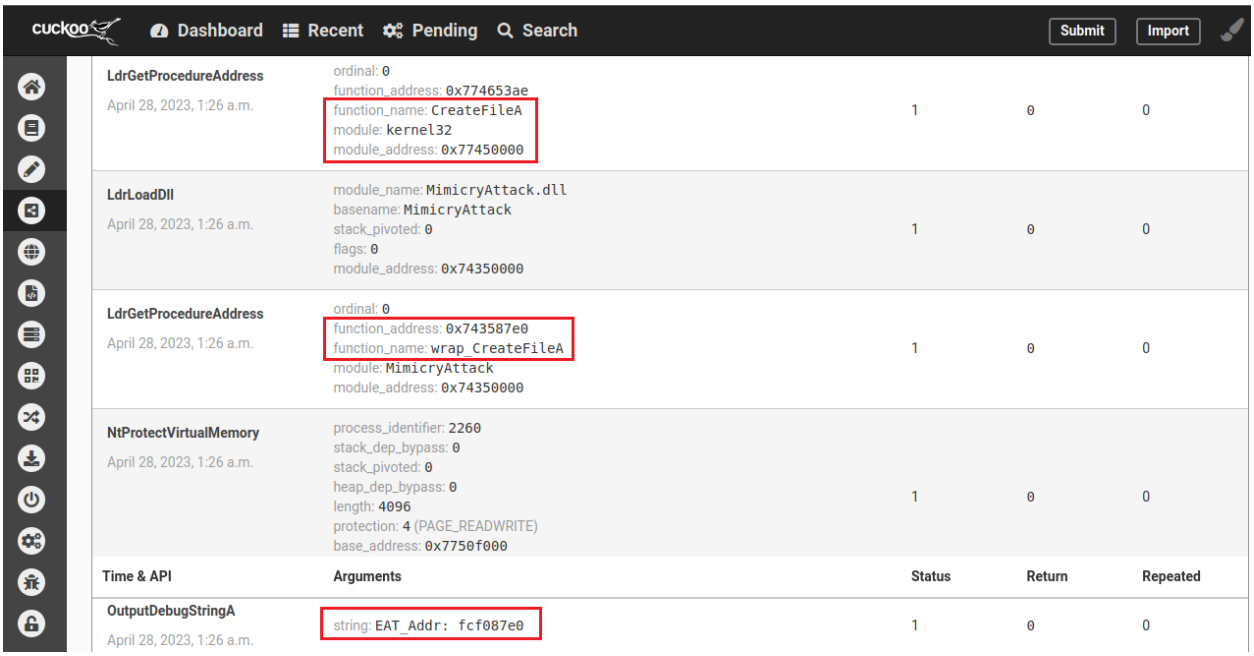


Figure 7.1: API trace that shows the address of "kernel32", `wrap_CreateFileA`, and the calculated RVA for EAT hooking.

APIs, this is a convenient method to print strings in an API trace that shows its position relative to other APIs. In Figure 7.1, `CreateFileA` is the original function from the library "kernel32", and `wrap_CreateFileA` is the hooked function from the custom library "Mimicry-Attack". From the first box, we can see that "kernel32" has an address of 0x77450000, and from the second box, we can see that `wrap_CreateFileA` has an address of 0x743587e0. Therefore, the calculated RVA should be 0x743587e0 - 0x77450000 = 0xfcf087e0 (after integer overflow), which is what we see in the third box (printed using `OutputDebugStringA`). This value when used in EAT hooking is able to successfully redirect calls to `CreateFileA` to `wrap_CreateFileA` instead.

7.2.4 IAT and EAT Hooking Characteristics

Because of their fundamental differences, IAT and EAT hooking have different characteristics and are effective under different conditions.

As expected, IAT hooking only works on functions found in the IAT, so only functions

that are load-time dynamically linked can be hooked. Thus, IAT hooking mostly does *not* work on functions that are run-time dynamically linked. Previous works [29, 6] have circumvented this limitation in some cases by IAT hooking `GetProcAddress` and modifying it to returned hooked functions whenever it is used to look up the original functions. However, this workaround is only effective when `GetProcAddress` itself is used and load-time dynamically linked (i.e., called from the IAT), but there are methods of calling `GetProcAddress` that is run-time dynamically linked, e.g., by searching through the loaded "kernel32" module to find its address [49], or even using the same method to find other function addresses, entirely without `GetProcAddress`. In particular, IAT hooking would not work on packers (malware that extracts a code payload in memory during execution and runs it [50]) if the payload's imports are reconstructed using run-time dynamically linked `GetProcAddress` calls.

In contrast, since EAT hooking modifies the loaded DLL image, all run-time dynamically linked functions can be hooked, whether `GetProcAddress` is run-time or load-time linked, and thus it does work on packers, unlike IAT hooking. However, since EAT hooking in effect changes the returned function address when it is looked up, it needs to occur before the application looks up the function addresses to be effective. This means it does not work on load-time dynamically linked functions, since the Windows loader loads the appropriate DLLs and fills the IAT before any application code (including the hooking code) runs. Thus, IAT and EAT hooking complement each other well, with one working for load-time dynamically linked functions and the other for run-time dynamically linked functions, so using both can hook more functions than either one individually.

Finally, both IAT and EAT hooking only work on the main process, so if the malware spawns any additional processes, those functions would not be hooked. For IAT hooking, this is because separate processes in Windows are separate images, so they each have their own IATs, which need to be hooked. For EAT hooking, this is because DLLs are loaded separately for each process, so the EATs of these DLLs need to be overwritten every time it is loaded. Thus, the hooking code needs to run inside newly spawned processes to be able to

hook those functions. Fortunately, both IAT and EAT hooking only need to be performed once per process, because the IAT of a process does not change after loading, and DLLs are only loaded once per process (even if the LoadLibrary function is called many times), so the modified EAT of that DLL also does not change. Methods to achieve this can include DLL injection (loading a DLL into a remote process) or code injection (writing code into the memory of the remote process) [51], but since hooked functions are usually written as part of a custom DLL, DLL injection is a simple and straightforward method of doing so that we will use.

Illustrative Malware Example

To illustrate these properties, I've hooked a piece of malware using IAT and EAT hooking and captured the resulting API trace with Cuckoo, a malware analysis sandbox [39]. This example malware both extracts a payload and spawns a new process to fully illustrate the properties of IAT hooking vs EAT hooking as well as the need to perform DLL injection.

Figure 7.2 shows the main function of the example malware decompiled using Ghidra [52], an open source reverse-engineering tool. The most important lines of code are in the red boxes. *VirtAllocAddr1* is the address of some memory allocated using a wrapper of the VirtualAlloc API call (`VirtualAlloc_Wrapper`). *VirtAllocAddr1_offset* is 512 bytes offset from *VirtAllocAddr1*. Then, some data (presumably the payload) is copied to *VirtAllocAddr1_offset* from some source address using the `memcpy` API call.

Next, `ExtractData` is a custom function that moves and modifies bytes in *VirtAllocAddr1*, which is presumably used to extract or modify the payload in memory, so I have given it the name `ExtractData`. Finally, *VirtAllocAddr3* is calculated as another offset from *VirtAllocAddr1* (presumably the beginning of the extracted code), and a custom function is used to jump to it (`JumpToVirtAddr3`). Thus, this decompiled code illustrates the main functions of this malware example; it extracts some code payload and executes it.

Figure 7.3 shows the outgoing function calls of the example malware executable, decom-

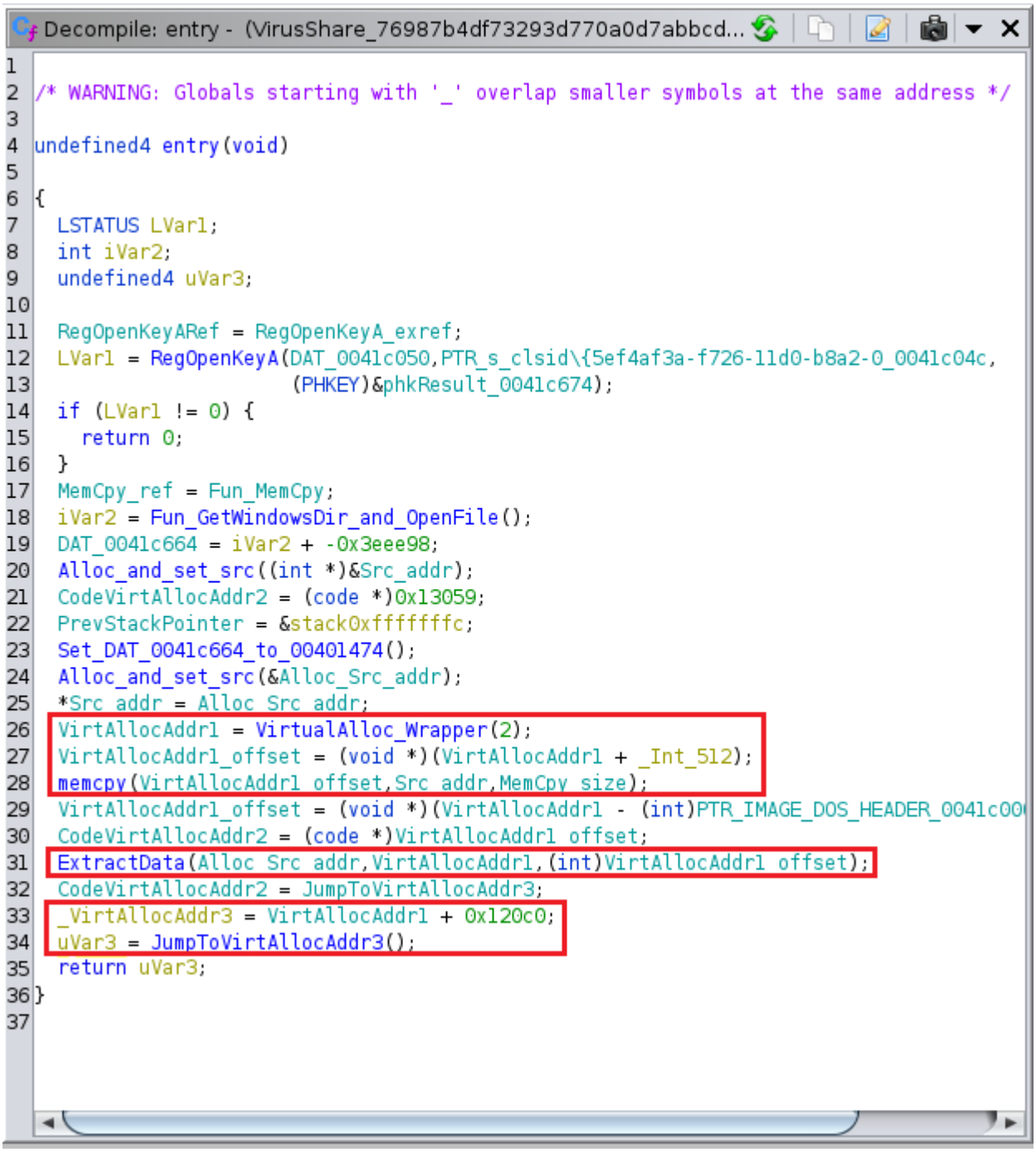


Figure 7.2: Decompiled code of example malware.

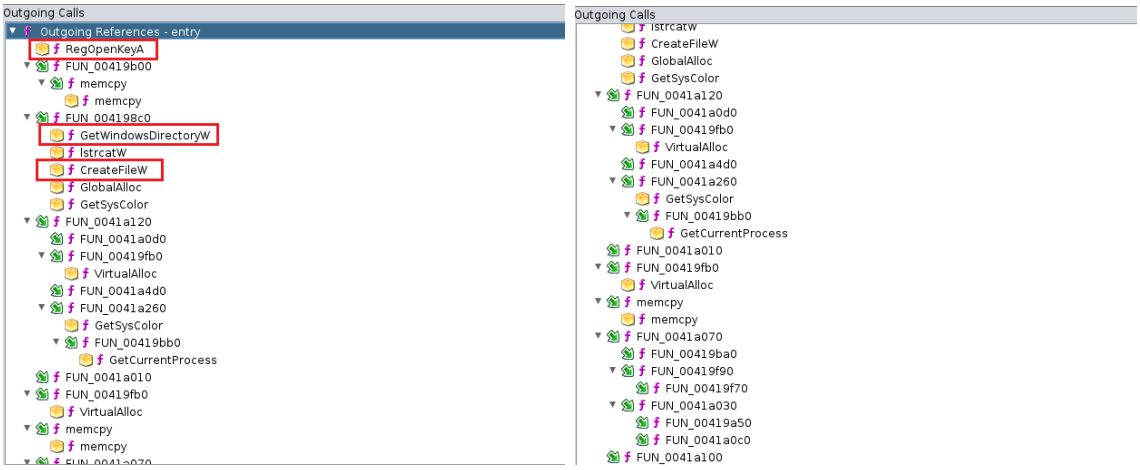


Figure 7.3: Example malware's outgoing function calls.

piled using Ghidra [52]. These outgoing functions are the imports called using the IAT, i.e., load-time dynamically linked functions. The Windows APIs we will focus on are in the red boxes: `RegOpenKeyA`, `GetWindowsDirectoryW`, and `CreateFileW`. Also note that certain APIs are *not* present, namely `CreateProcessA`, `CreateMutexA`, `ZwResumeThread`, and `GetProcAddress`.

In these examples, the hooked function uses `OutputDebugStringA` to print the name of the original function and then calls the original function, so the name of the function is printed, but its functionality otherwise remains the same. Thus, a successfully hooked function will print its name through `OutputDebugStringA`, while an unsuccessful hook will not.

IAT Hooking Only Example

Figure 7.4 shows the API trace when only IAT hooking is performed. In the red boxes, we can see that `RegOpenKeyA`, `GetWindowsDirectoryW`, and `CreateFileW` have been printed, so these functions are successfully hooked. The blue boxes shows the captured APIs from calling these functions, which are similar but not identical to the called functions. This is because Windows APIs internally may call other functions, usually Native APIs (which are functions that start with "Nt"), and Cuckoo mostly captures these Native APIs rather

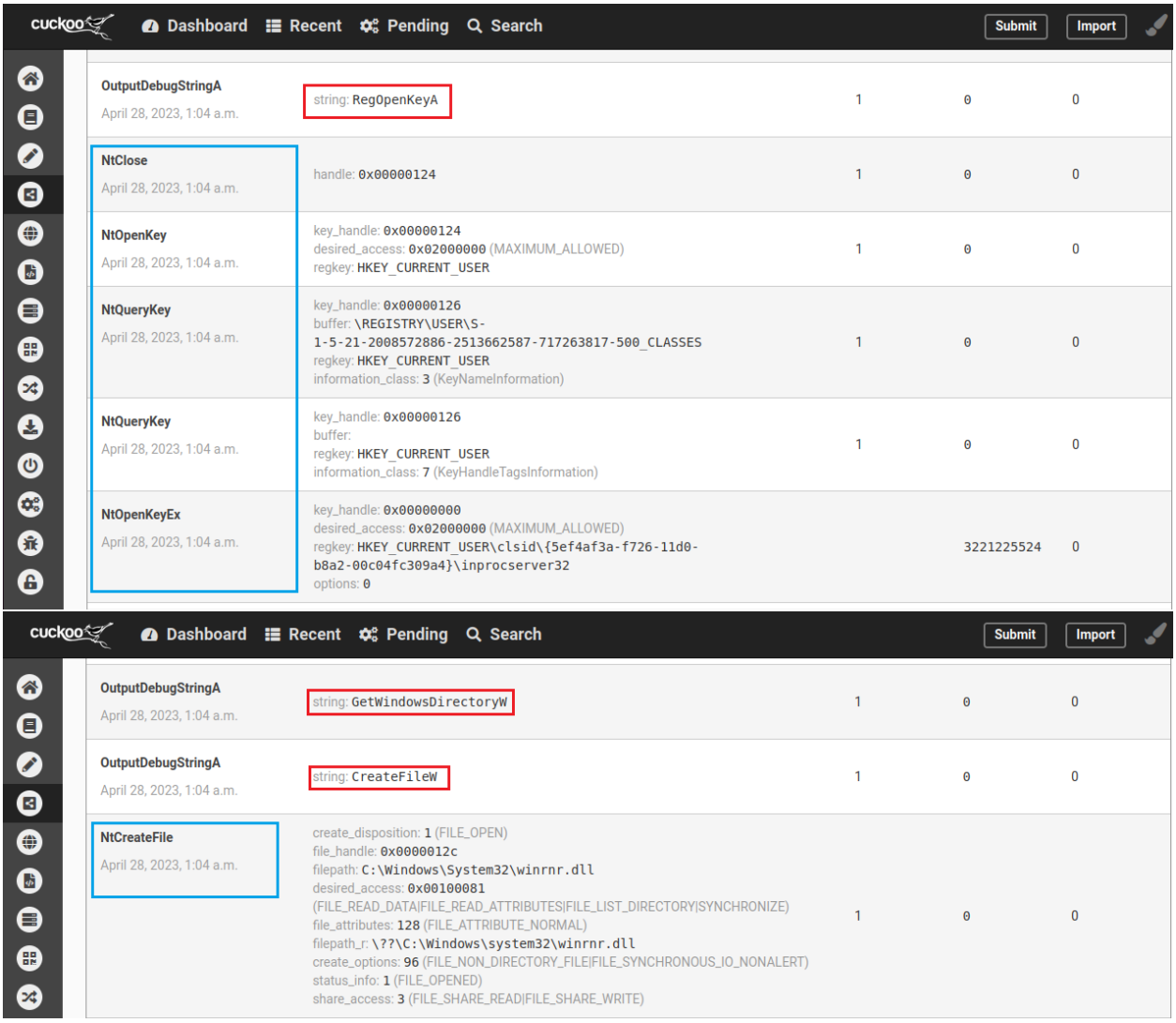


Figure 7.4: API trace with only IAT hooking that shows `RegOpenKeyA`, `GetWindowsDirectoryW`, and `CreateFileW` hooked properly.

than the Windows APIs. For example, `CreateFileW` internally calls `NtCreateFile`, which is captured by Cuckoo. `RegOpenKeyA` internally calls `NtOpenKey` and `NtQueryKey` several times, as can be seen from the trace. Cuckoo doesn't seem to capture `GetWindowsDirectoryW`, but the output shows that it has nevertheless been successfully hooked.

On the other hand, Figure 7.5 shows that `CreateProcessInternalW`, `NtCreateMutant`, and `NtResumeThread` do not have any function names printed with `OutputDebugStringA` before they are called. Thus, Windows APIs that call them have not been successfully hooked. In particular, if we look at the green boxes, we see that `CreateMutexA` and `ZwResumeThread` have both been looked up, so those are likely the functions that calls `NtCreateMutant` (mutant is another name for mutex in Windows) and `NtResumeThread` respectively, but these functions have not been hooked. Also, as we will later see, every call of `LdrGetProcedureAddress` results from a call to `GetProcAddress`, and since we do not see `GetProcAddress` printed, we can conclude that it has not been successfully hooked either and is also run-time dynamically linked.

The reason these functions have not been hooked is that they are not load-time dynamically linked, so IAT hooking does not work on them. Recall in Figure 7.3 that we do not see `CreateProcessA`, `CreateMutexA`, `ZwResumeThread`, and `GetProcAddress` in the outgoing functions list, so these functions are not linked through the IAT. Decompiling the executable shows that these calls are made after the IAT hooked functions, and after memory is allocated, a payload (presumably code) is copied to it, and a jump is performed to start executing the payload. Thus, these functions are likely run-time dynamically linked and called while executing the payload.

EAT Hooking Only Example

Next, we will shows the results of only performing EAT hooking. Figure 7.6 shows that `CreateProcessA`, `CreateMutexA`, and `ZwResumeThread` (shown in the red boxes) have been hooked successfully because their function names have been printed out by

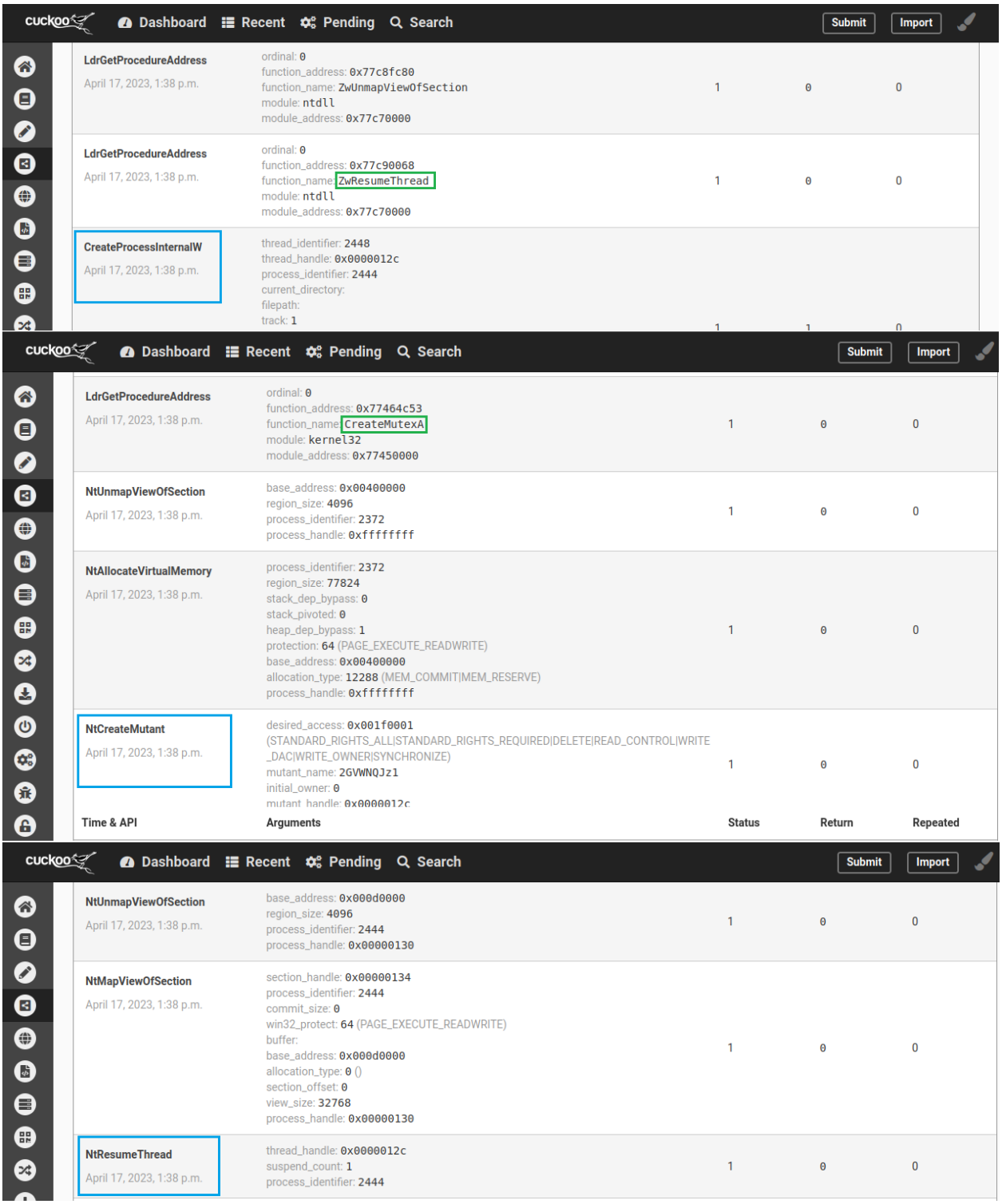


Figure 7.5: API trace with only IAT hooking that shows `CreateProcessInternalW`, `NtCreateMutant`, and `NtResumeThread` (blue boxes) are not hooked properly. We do see functions like `ZwResumeThread` and `CreateMutexA` (green boxes) being looked up, so they are likely the functions that call `NtCreateMutant` and `NtResumeThread` respectively, but we do not see these function names printed with `OutputDebugStringA`.

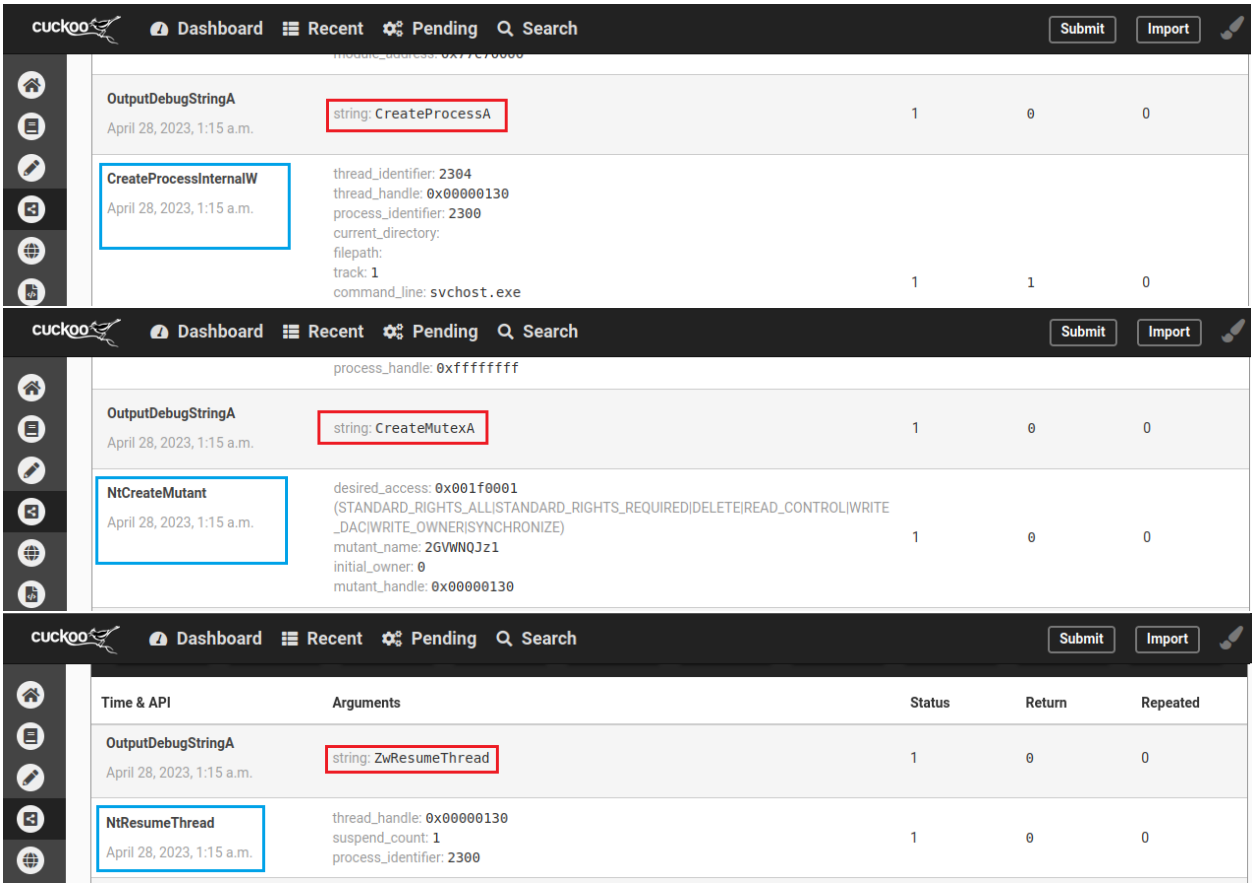


Figure 7.6: API trace with only EAT hooking that shows `CreateProcessA`, `CreateMutexA`, and `ZwResumeThread` (red boxes) are hooked properly as they are printed by `OutputDebugStringA`. They appear right before `CreateProcessInternaW`, `NtCreateMutant`, and `NtResumeThread` respectively (blue boxes), which they call internally.

`OutputDebugStringA`. Recall that these functions are run-time dynamically linked, so EAT hooking is effective on them.

Additionally, recall that we concluded `GetProcAddress` must also be run-time dynamically linked (since it did not appear in the out-going functions list), so EAT hooking should work on it as well. Figure 7.7 confirms this, since we see `GetProcAddress` printed out by `OutputDebugStringA` when EAT hooking is applied. For `GetProcAddress`, I have also added the function being looked up to the output string, and we can confirm that `CreateProcessA`, `CreateMutexA`, and `ZwResumeThread` are indeed run-time dynamically linked by calling `GetProcAddress`. This shows that run-time dynamically linked functions can be hooked through EAT hooking.

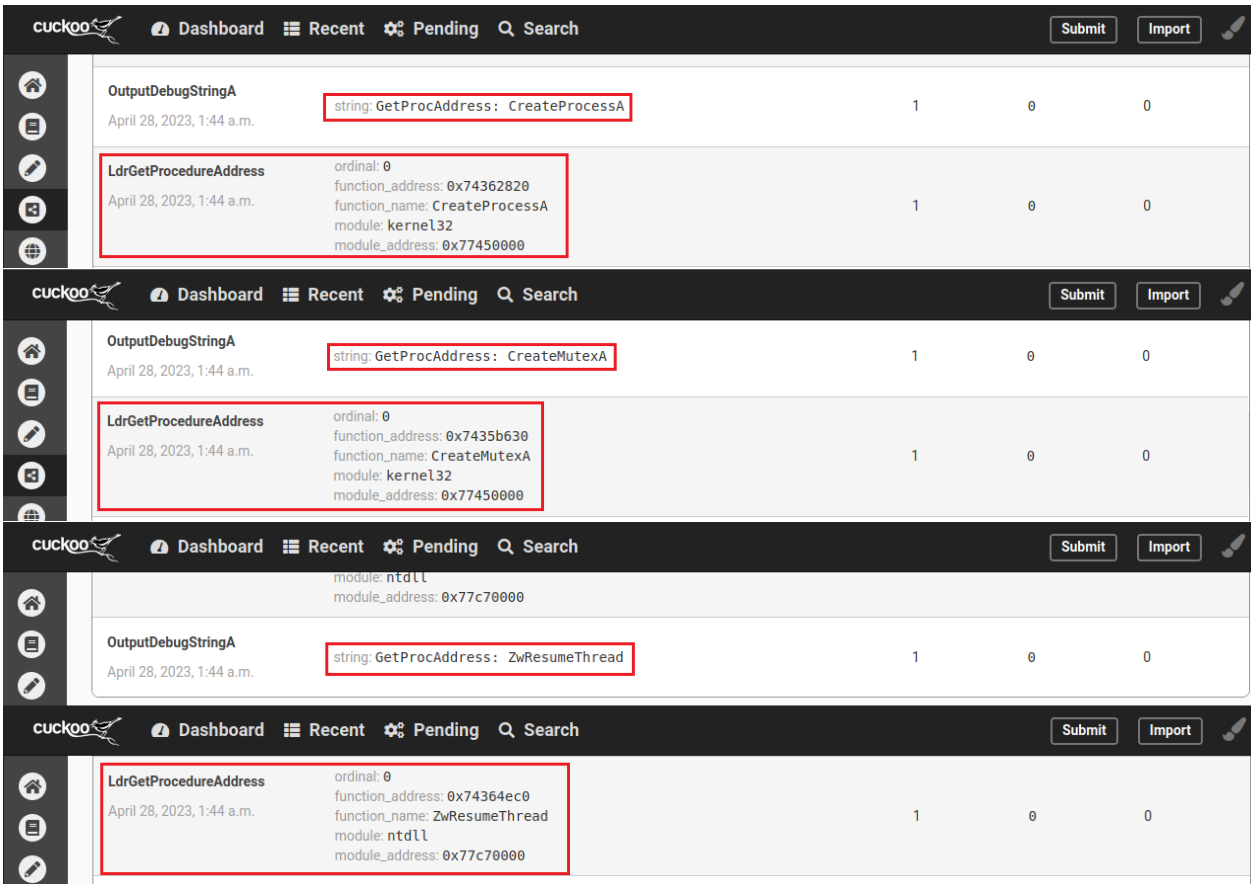


Figure 7.7: With only EAT hooking, `GetProcAddress` is hooked, which means it is likely that `GetProcAddress` is run-time dynamically linked. This also shows that `CreateProcessA`, `CreateMutexA`, and `ZwResumeThread` have been looked up using `GetProcAddress`, indicating run-time dynamic linking.

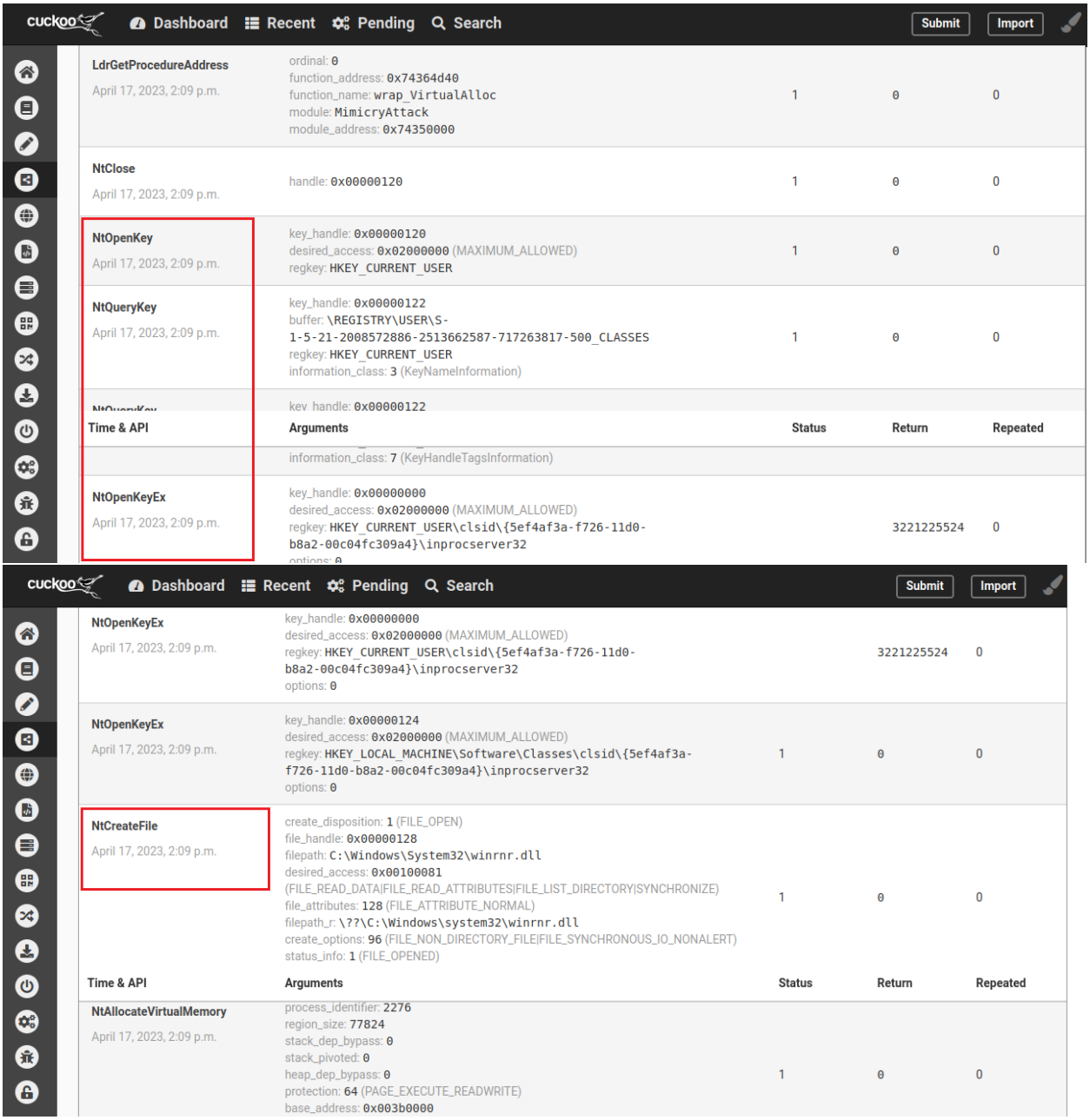


Figure 7.8: API trace with only EAT hooking that shows `RegOpenKeyA`, `GetWindowsDirectoryW`, and `CreateFileW` have not been hooked properly. Unlike in Figure 7.4, these function names have not been printed by `OutputDebugStringA` before their respective NT functions (red boxes).

However, as discussed in Section 7.2.4, since the IAT is filled with function addresses while the application is loaded, EAT hooking should not work on load-time dynamically linked functions (i.e., functions found in the IAT). Figure 7.8 confirms this, as `RegOpenKeyA`, `GetWindowsDirectoryW`, and `CreateFileW`, all load-time dynamically linked functions, have not been printed by `OutputDebugStringA`, as opposed to in Figure 7.4 when IAT hooking is used. In conclusion, these examples show scenarios where IAT hooking and EAT hooking complement each other, so using both techniques can hook more functions than just using one. This allows more verseatile and powerful mimicry attacks to be created (See Section 7.2.6).

Hooking Additional Processes Example

Next we will show an example hooking additional processes. Recall that IAT and EAT hooking only work for the process it is performed on and needs to be repeated for each additional process. Otherwise, the functions in the additional processes will not be hooked. See Figure 7.9 for an example where hooking is not performed on the second process. We can see that functions `GetFileInformationByHandle`, `NtReadFile`, `NtCreateFile` and `NtWriteFile` (red boxes), functions from the second process, have not been hooked since we do not see their function names printed by `OutputDebugStringA`.

See Figure 7.10 for the same functions when DLL injection [51] is applied to the second process. We can see that `GetFileInformationByHandle`, `NtReadFile`, `NtCreateFile` and `NtWriteFile` (blue boxes) have been successfully hooked since we do see their corresponding Windows API names being printed by `OutputDebugStringA` (red boxes). Thus, a technique like DLL injection is required to hook functions in additional processes.

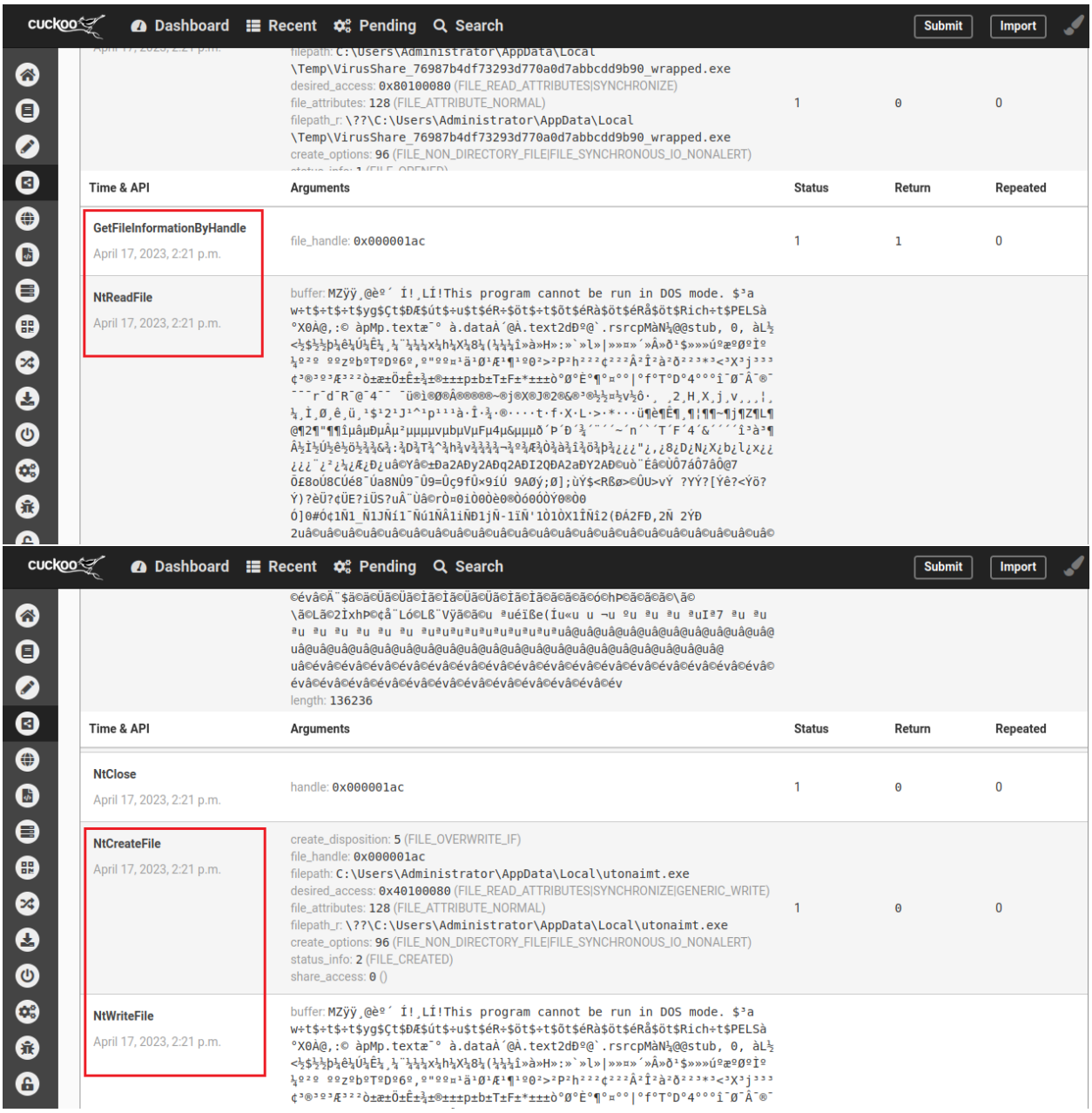


Figure 7.9: API trace with both EAT and IAT hooking but no DLL injection showing that `GetFileInformationByHandle`, `NtReadFile`, `NtCreateFile` and `NtWriteFile` (red boxes) have not been hooked.

Figure 7.10: API trace with both EAT and IAT hooking and with DLL injection showing that `GetFileInformationByHandle`, `NtReadFile`, `NtCreateFile` and `NtWriteFile` (blue boxes) have been successfully hooked since we see function names being printed out by `OutputDebugStringA` (red boxes).

7.2.5 Hooking Implementation

In this subsection, I will describe how these hooking techniques are implemented to hook actual malware, including the tools we used and the code we wrote.

IAT Patcher

Like previous works [29, 6], we modified an IAT hooking tool called IAT Patcher [53] to implement the hooking techniques described in this section. At a high level, IAT Patcher adds a stub to the target executable that acts as the new entry point. The stub contains code to load all the necessary libraries and functions and performs IAT hooking before jumping to the original entry point. An important characteristic of IAT Patcher is that it modifies executables directly, so these modified malware can be saved and run in Cuckoo, while most other tools can only hook an already running process.

To work, IAT Patcher requires a custom DLL that contains all the hooked functions to replace the original functions with, so the stub can load this DLL, get its function addresses, and replace IAT entries with them. For us, this custom DLL not only contains the hooked functions, but it also contains code to perform IAT and EAT hooking as well as DLL injection. We wrote our own IAT hooking code even though IAT Patcher already performs IAT hooking because we still need IAT hooking to run when the DLL is injected into additional processes.

IAT and EAT Hooking Implementation

Like executables, DLLs also have a main function, called DLLMain, that acts as the entry point for the DLL [54]. DLLMain is run when 1) it is loaded in a new process (called "process attach"), 2) it is unloaded from a process (called "process detach"), 3) a new thread is created in the process (called "thread attach"), and 4) a thread exits (called "thread detach"). There is a parameter in DLLMain that passes in which of these four conditions DLLMain is being run for. Thus, we can run code to perform IAT and EAT hooking whenever it is loaded in a new process (i.e., on process attach) to hook the necessary functions in the new process.

```
DWORD __declspec(dllexport)
WINAPI
wrap_GetFileType(
    _In_ HANDLE hFile
    ){
    makeCalls("GetFileType");
    typedef DWORD (WINAPI* PFunction)(HANDLE);
    PFunction pFunction = (PFunction)api_original_addresses["GetFileType"];
    return pFunction(hFile );
}
```

Figure 7.11: Example of a hooked function. First, `makeCalls` is called to insert additioanl APIs. Then, `PFunction` is defined as the function signature (same signature as the original function `GetFileType`), and `pFunction` is the address of the original function typecasted as `PFunction`. Finally, `pFunction` is called with the original arguments.

In our code, all the target original functions and their corresponding hooked functions are first loaded and have their addresses saved. For IAT hooking, the code loops through the IAT entries of the current process to search for the target functions and replace their addresses with the hooked function addresses. For EAT hooking, the code loops through the EAT entries of the target DLL, searches for the target functions, and replaces their addresses with the calculated RVAs relative to our custom DLL, as described in Section 7.2.3. The code for IAT and EAT hooking are written based on code snippets found in [55] and [56] respectively.

Hooked Functions

Recall from Section 7.2.3 that the hooked functions must have the same function signature (same parameters and return type) as the original function. This is convenient because it allows us to call the original function using the same arguments (recall that we also saved the original function addresses), so we can maintain the original functionality of that call. Thus, the hooked functions all call the original function and return its result. Before the call to the original function, the hooked functions also call a custom function (which we called `makeCalls`) that makes additional API calls, thereby inserting additional APIs into

```
//based on https://www.apriorit.com/dev-blog/679-windows-dll-injection-for-api-hooks
void dllInject(DWORD pid) {
    //path of custom DLL to load
    LPCSTR injectLibraryPath = "C:\\Users\\Administrator\\AppData\\Local\\Temp\\MimicryAttack.dll";

    //opens the remote process
    HANDLE processHandle = OpenProcess(
        PROCESS_VM_READ |
        PROCESS_QUERY_INFORMATION |
        PROCESS_CREATE_THREAD |
        PROCESS_VM_OPERATION |
        PROCESS_VM_WRITE,
        FALSE,
        pid);

    //allocates memory in remote process to write DLL path string
    int bytesToAlloc = strlen(injectLibraryPath) + 1;
    LPVOID remoteBufferForLibraryPath = VirtualAllocEx(processHandle, NULL, bytesToAlloc, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);

    //writes DLL path into buffer
    WriteProcessMemory(processHandle, remoteBufferForLibraryPath, injectLibraryPath, bytesToAlloc, NULL);

    //looks up function address of LoadLibraryA
    PTHREAD_START_ROUTINE loadLibraryFunction = (PTHREAD_START_ROUTINE)(GetProcAddress(GetModuleHandleW(L"kernel32.dll"), "LoadLibraryA"));

    //creates remote thread starting at function address with DLL path as argument
    HANDLE remoteThreadHandle = CreateRemoteThreadEx(processHandle, NULL, 0, loadLibraryFunction, remoteBufferForLibraryPath, 0, NULL, NULL);

    CloseHandle(remoteThreadHandle);
    CloseHandle(processHandle);
}
```

Figure 7.12: DLL Injection Code.

the trace. Exactly which APIs are called in this function are discussed in Section 7.2.6. See Figure 7.11 for an example of a hooked function for the API `GetFileType`.

DLL Injection Implementation

The goal of using DLL injection is to run the hooking code in the remote process. As discussed above, our custom DLL runs both IAT and EAT hooking when it is first loaded in a process, so we only need to inject code that loads the custom DLL. The main functionality of the DLL injection code relies on the `CreateRemoteThreadEx` API, which creates a thread in a remote process [57]. The thread begins execution at a specified address (passed in as a parameter), which must be a function in address space of the remote process. We can use `LoadLibraryA` as the function to load our DLL, and fortunately for us, `LoadLibraryA` is part of "kernel32.dll", which is loaded automatically at the same address in almost every process, so we can look up its address in the current process and pass it in to `CreateRemoteThreadEx` to run and load our custom DLL in the remote process.

Our DLL injection code is written based on an example found in [51]. First, a han-

dle to the remote process is opened using `OpenProcess` and memory is allocated in it to write the path of the custom DLL. Then, `GetProcAddress` is called to get the address of `LoadLibraryA`. Finally, `CreateRemoteThreadEx` is called with the address of `LoadLibraryA` and the address of the path string in the remote process passed in to create the process. See Figure 7.12 for the code.

This DLL injection code is run whenever a new process is created. This occurs most often in our data with `CreateProcessA` and `CreateProcessW` API functions, and since we have hooked these functions in the main process, we can run the DLL injection code whenever these functions are called. Sometimes processes are created in suspended mode, so if this is the case (which is passed in as a function argument), we run the DLL injection code in functions used to resume them, `NtResumeThread` and `ZwResumeThread`. Performing DLL injection on these 4 APIs results in over 50% of the malware samples using DLL injection, while around a third of the remaining samples have only 1 process, so DLL injection does not apply because no additional process is spawned.

For the remaining one third of the malware data, we decided not to use DLL injection on them for several reasons. First, we did not want most of the dataset to contain DLL injection because that may unfairly bias the data. For example, if a detection method can only detect DLL injection, it will still perform very well on a dataset where most samples contain DLL injection despite being a poor detector of general malware. Reducing the number of samples with DLL injection limits this effect. Next, there is a cost to inserting more APIs as well. Cuckoo limits the total execution time of each process to 2 minutes, so for each additional process hooked, there is an additional chance that the inserted APIs take up so much time that the original malicious functions are not executed. While this is not a major concern, it is still worth taking into consideration when deciding to use DLL injection on additional samples when a majority is already using it. Finally, some APIs that can create processes (e.g., `ShellExecuteA`/`ShellExecuteW` or `ShellExecuteExA`/`ShellExecuteExW`) have the additional issue that they don't *always*

create a new process. For this reason, `ShellExecuteA`/`ShellExecuteW` does not return the process handle. `ShellExecuteExA`/`ShellExecuteExW` does, but a flag needs to be set, which means the input to the original function needs to be modified, which may introduce problems with functionality. These challenges can certainly be overcome (e.g., by searching for the newly created process to find its handle), but they require additional complexity and additional API calls, which we have shown above to have a cost. For these reasons, we do not run the DLL injection code on any additional functions other than the 4 aforementioned ones (`CreateProcessA`, `CreateProcessW`. `NtResumeThread` and `ZwResumeThread`), but even with just these functions, we found that this is sufficient to create very effective mimicry attacks in malware (see Chapter 8).

7.2.6 Creating Mimicry Attacks

As discussed in Section 7.2.5, the end goal of hooking Windows API functions is to insert additional APIs into a trace and thereby create mimicry attacks in malware, which we can achieve by calling a custom function (called `makeCalls`) that makes additional API calls in the hooked functions. There are two main goals of inserting APIs: 1) inserting APIs that correspond to benign features will increase the likelihood of a malicious sample being classified as benign, and 2) for API 2-grams experiments, where features are pairs of consecutive APIs, inserting APIs may remove particularly malicious features and replace them with more benign ones.

To choose the most "benign" APIs to insert, we use a same method as the detection pipeline, which is the proportion of a feature in the malicious data over the proportion of the same feature in the benign data. This way, even if the attacker does not know the exact data used to train the models, using a similar set of malware and benign software can result in a similar ranking of maliciousness for the features. By using the same method as the detection system, this assumption also allows for the most effective attacks, even if it is somewhat generous for the attacker, so we can test our detection system on the most powerful attacks.

```
BOOL __declspec(dllexport)
WINAPI
wrap_WriteProcessMemory(
    _In_ HANDLE hProcess,
    _In_ LPVOID lpBaseAddress,
    _In_reads_bytes_(nSize) LPCVOID lpBuffer,
    _In_ SIZE_T nSize,
    _Out_opt_ SIZE_T* lpNumberOfBytesWritten
) {
    typedef BOOL(WINAPI* PFunction)(HANDLE, LPVOID, LPCVOID, SIZE_T, SIZE_T*);
    PFunction pFunction = (PFunction)api_original_addresses["WriteProcessMemory"];

    POINT point;
    GetCursorPos(&point);
    auto ret = pFunction(hProcess, lpBaseAddress, lpBuffer, nSize, lpNumberOfBytesWritten);
    GetCursorPos(&point);
    return ret;
}
```

Figure 7.13: An example of a particularly malicious API surrounded by specific benign APIs. `pFunction` here is `WriteProcessMemory`, and `GetCursorPos` is called both before and after it to eliminate the most malicious 2-gram features.

To be effective against methods that use the behavior summary, API subsequences with API orderings and resources that correspond to the most benign features are added. Whenever `makeCalls` is called, a random benign API sequence is chosen to be called. Some include opening and reading or writing benign files, creating mutexes with benign looking names, and calling standard system APIs like `GetUserName` or `GetSystemInfo`. To increase effectiveness against API 2-gram systems, for APIs that are particularly malicious (e.g., `WriteProcessMemory`, which is often used in process injection attacks), we surround the function with API calls that would result in the most benign 2-grams, so it would remove many of the very malicious API 2-gram features. See Figure 7.13 for an example.

In conclusion, as will be shown in the evaluation section, these result in very effective mimicry attacks. While prior work [29, 6] only used IAT hooking to create mimicry attacks, and only for one process, we used both IAT and EAT hooking as well as hooking additional processes. Thus, these APIs would be inserted in many more locations, resulting in more powerful attacks.

7.2.7 Limitations

The main limitation of the method is that we insert many additional benign APIs for every target API. Recall that whenever `makeCalls` is called, a random benign API sequence is chosen, and all the APIs in the sequence is called. This results in many inserted benign APIs for each hooked API, which has the benefit of adding many benign features to the sample, but can dramatically increase the length of the API trace as well as the length of time needed for execution. Cuckoo has a timeout for how long virtual machines are able to run a sample for. Thus, it is possible that because we inserted so many benign calls, some malicious calls may not have a chance to execute before the virtual machine times out. In our evaluations, we do filter out these cases, so they do not appear in the evaluation dataset, but it does mean that the dataset is smaller than if we added fewer benign calls. We believe that a smaller dataset is a worthwhile trade-off for being able to add more benign calls, which leads to stronger attacks and a more rigorous evaluation for our detection methods.

7.3 Adversarial Attack Method

In addition to mimicry attacks, another popular class of attacks is the adversarial attack, where a target classifier is repeatedly queried and a sample is repeatedly modified based on the output of the classifier until it is misclassified. Recall from our threat model (Section 4) that we perform attacks at the feature vector level because in our detection pipeline, API traces are first converted into feature vectors, so attacking at the feature vector level is at most as hard as attacking at the API sequence level because it is a more direct attack with one less intermediate step. Any successful adversarial API sequence is successful because it can be converted into a feature vector that produces a benign classification in the model.

Additionally, we assume a black-box scenario, so the attacker does not have access to the internal structure or parameters of the model because that's the most realistic scenario. Most black-box adversarial attacks in the literature depend on estimating a gradient for the

loss function, so the attacker knows what direction in vector space to perturb (i.e., which elements of the feature vector to modify and whether to increase or decrease these elements). This is can be done by perturbing the input by adding a random vector to it, subtracting the same vector (i.e., perturbing in the opposite direction), and taking the difference in model outputs. Repeating this allows for an estimation of gradient of the loss function (see [58]). This method requires continuous (non-discrete, especially non-binary) input and output, although the same paper has a modified method that works when the output is top-k most likely class labels. For classifiers with either a binary vector input or a binary output (but not both), attack methods like SCAR [59] and the boundary attack [60] have been developed to work in these scenarios respectively, but they require either a continuous output or continuous input to be effective. When applied to classifiers with both a binary vector input and a binary output (and with the restriction that features can only be added as in our scenario), these methods become simple combinatorial enumeration, which is scales very poorly for large vectors. To the best of our knowledge, there are no efficient black-box adversarial attack algorithms for classifiers with both a binary vector input and a binary output.

As a result, we default to transfer attacks, where we train a surrogate model, perform white-box attacks on the surrogate model, and then apply the successful adversarial samples to the target model. Because LGBM is a gradient-boosted decision tree model, we use the decision tree attack from Papernot et al. [61] as the basis for our adversarial attacks. This decision tree attack algorithm is described in Algorithm 10. Note that this is our interpretation of the algorithm, so for the original authors' description, refer to Algorithm 2 in their paper.

Decision trees are binary trees, so each internal node contains at most 2 children nodes. Each internal node (called "split node" in LGBM) contains a feature and a threshold for that feature. To classify a sample, the prediction algorithm traverses the tree, and at every internal node, if the specified feature is below the threshold, the left child is visited, and vice

Algorithm 10 Decision Tree Attack

Input: Malicious Sample Feature Vector - x
Input: Target Model Tree - *tree*

$current\_node \leftarrow$ **get_leaf_node**$(tree, x)$
$found \leftarrow$ `False`

while $found ==$ `False` **do**
    if is_left_child$(current\_node)$ **then**
        $found, adv\_leaf \leftarrow$ **search_subtree**$(current\_node.parent.right\_child)$
    else
        $found, adv\_leaf \leftarrow$ **search_subtree**$(current\_node.parent.left\_child)$
    end if

    if $found ==$ `False` **then**
        $current\_node \leftarrow current\_node.parent$
    end if
end while

$adv\_sample \leftarrow$ `null`
if $found ==$ `True` **then**
    $adv\_path \leftarrow$ **path_to_root**(adv_leaf)
    $adv\_sample \leftarrow$ **perturb_features**(x, adv_path)
end if
return adv_sample

versa if the feature is above the threshold, until a leaf node is reached. The leaf node then contains the final prediction of the sample. For classifiers with binary vector inputs, visiting the left or right child corresponds to whether the specified feature is a 0 or a 1 in the feature vector.

Algorithm 10 works as follows. For a given sample x, the corresponding leaf node is found by traversing the tree. Then, the subtree under the current node's other sibling is searched for a leaf node with the desired adversarial prediction. If none is found, this process is repeated for the current node's parent, and then the parent node's parent and so on, until an adversarial leaf node is found. Finally, the original sample has its features perturbed so that it follows the path to the adversarial node. See Figure 7.14 for an illustration of this process. The red node represents *current_node* from Algorithm 10, which can be the

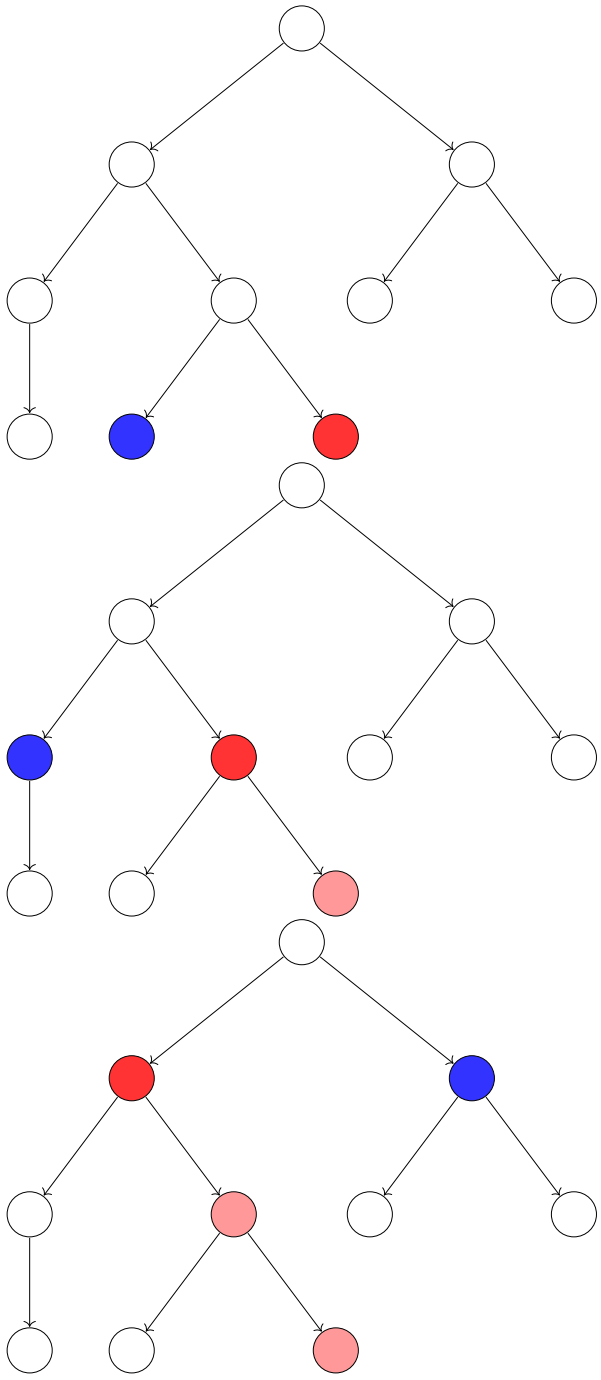


Figure 7.14: A illustration of the decision tree attack algorithm. Starting at the leaf node corresponding to a sample (red), it's sibling (blue) and the subtree under it are searched for a valid adversarial sample, recursively up the tree until an adversarial sample is found. Then, the sample is perturbed to match the path of the adversarial sample found. The lighter red nodes illustrate the path up the tree from the leaf node.

original sample's leaf node or its parent nodes, while the blue node represents the sibling node whose subtree is to be searched at each iteration. The light red nodes shows the progression up the tree from the original leaf node to the root. In the worst case, this method still exhaustively searches these trees for an adversarial sample, but because both the number of trees and nodes are limited in LGBM, this is much more efficient that searching through all permutations of the feature vector.

To use this algorithm to attack our models, however, we need to modify this algorithm because 1) we have the restriction that features can only be added but not removed, and 2) LGBM is not a single tree but consists of many trees (i.e., a "forest"), and our models can consist of multiple LGBM models. For the first point, we make two modifications to Algorithm 10. First, we remove the else condition of searching the subtree under the left sibling if the current node is a right child (removed lines 10-11). If a suitable leaf node is found in the subtree under the left sibling, then some feature will need to be changed from a 1 to a 0, which means removing that feature. Removing those lines of codes prevents one feature from decreasing in the adversarial sample. However, we do not modify the **search_subtree** subroutine, so there is the possibility that searching through the subtree results in some features needing to be changed from a 1 to a 0. Thus, we make the second modification to Algorithm 10 that when perturbing the original sample to match the found adversarial leaf node (**perturb_features** in line 22), we ignore changing any feature that needs to be decreased. Even though this results in an approximate solution, we've found that these changes along with some other techniques (see below) are sufficient to generate very effective attacks against LGBM models ($\sim$ 88% effective, see Section 8), so we did not restrict search paths in the **search_subtree** subroutine and increase its complexity for what's likely to be a small improvement.

For the second point, we simply use the attack algorithm iteratively on each tree in the LGBM model, starting from the first tree. For each iteration, we use the modified decision tree attack algorithm to generate a potential adversarial sample, test the sample on the model

to see if it evades detection, and if it does not, attack the next tree using the (unsuccessful) adversarial sample generated from the previous iteration as the original sample. This way, the sample accumulates changes as more trees are attacked until a successful sample is found. Since LGBM is a gradient-boosting machine, the first tree has a larger impact on the output than the subsequent trees, so applying the decision tree attack to the trees in order is an important aspect of the attack because potential adversarial sample candidates generated from first few trees are the most likely to evade the entire model. If the target is an ensemble of multiple LGBM models, then the attack is applied iteratively to the first tree of every LGBM model, then the second tree of every LGBM model, and so on, with the generated adversarial sample being tested on the target ensemble after every iteration.

Finally, after the decision tree attack algorithm has been run on all trees of all LGBM models, the differences between all the successful adversarial samples and their original samples are calculated and applied to all the remaining unsuccessful samples. This is done as a final attempt to generate successful adversarial samples by applying perturbations that have already been successfully used to generate adversarial samples.

See Algorithm 11 for the pseudocode of the full adversarial attack algorithm on LGBM models and ensembles. The code from lines 7-22 loops through all the malicious samples, iteratively performs the decision tree attack with our modifications (denoted as the **decision_tree_attack** subroutine in line 12) on the trees of LGBM models, and keeps track of the successful and unsuccessful adversarial samples after each iteration. Then, on lines 24-28, the differences between the successful adversarial samples and their original counterparts are calculated and stored as successful perturbations to be tried on unsuccessful samples. Note that since our modified decision tree attack algorithm only add features, the bitwise subtract function on line 26 does not result in any negative vector elements in the result. Finally, the code from lines 30-41 tries all the successful perturbations on all the remaining unsuccessful samples and keeps any changes that results in new successful samples.

As an additional note, for simplicity and clarity, Algorithm 11 depicts the attack as

Algorithm 11 LGBM Attack

Input: Malicious Samples - *samples*
Input: Target LGBM Ensemble Model - *model*

$adv\_samples \leftarrow samples$
$successful\_indices \leftarrow$ Empty List
$unsuccessful\_indices \leftarrow [0, ..., adv\_samples.length - 1]$
for $i \in [0, ..., adv\_samples.length - 1]$ **do**
    $x \leftarrow adv\_samples[i]$
    $tree\_idx \leftarrow 0$
    for all $lgbm\_model \in model$ **do**
        $curr\_tree \leftarrow lgbm\_model.trees[tree\_idx]$
        $adv\_x \leftarrow$ **decision_tree_attack**$(x, curr\_tree)$
        $label \leftarrow model.predict(adv\_x)$
        $adv\_samples[i] \leftarrow adv\_x$
        if $label ==$ `benign` **then**
            $successful\_indices.append(i)$
            $unsuccessful\_indices.remove(i)$
            break
        end if
    end for
    $tree\_idx \leftarrow tree\_idx + 1$
end for

$successful\_perturbations \leftarrow$ Empty List
for $i \in successful\_indices$ **do**
    $adv\_difference \leftarrow$ **bitwise_subtract**$(adv\_samples[i], samples[i])$
    $successful\_perturbations.append(adv\_difference)$
end for
for all $perturbation \in successful\_perturbations$ **do**
    for $j \in unsuccessful\_indices$ **do**
        $x \leftarrow samples[j]$
        $x\_adv \leftarrow$ **bitwise_add**$(x, perturbation)$
        $label \leftarrow model.predict(adv\_x)$
        if $label ==$ `benign` **then**
            $successful\_indices.append(j)$
            $unsuccessful\_indices.remove(j)$
            $adv\_samples[j] \leftarrow x\_adv$
        end if
    end for
end for
return $adv\_samples$

applying to one sample at a time (lines 8, 14, 32, 28, etc.), but in the actual code, the attack is done done on batches of samples because functions like *model.predict* (line 13) can be batched for higher performance. However, the decision tree attack algorithm (**decision_tree_attack**) cannot be batched because the attack needs to traverse the trees separately for each sample, so that portion of the code still needs to iterate through the samples in each batch and perform the attack separately.

Chapter 8

Evaluation

This chapter describes all the evaluations we did using the attacks described in the previous chapter. First, Section 8.1 describes how we collected our data, and the subsequent sections describe the mimicry and adversarial attack evaluations as well as a case study to help the reader gain an intuitive understanding of why our proposed detection system works.

8.1 Dataset

For this work, we created our own dataset by collecting samples and processing them with Cuckoo. The malicious samples are collected from VirusShare [62], and the benign samples are downloaded from the popular software repositories Software Informer, CNET, Softpedia, and Portable Freeware[63, 64, 65, 66]. We installed Cuckoo [39] and analyzed all these files with network connectivity disabled (to prevent malware spread). Before this data can be used however, they need to be sanitized to remove problematic samples. Note, that the samples are filtered in the order described, and any percentages given at each step is calculated by dividing the number of samples that pass all previous filters.

First, samples may not execute correctly, e.g., they may crash because required libraries are missing. To remove these samples, we use the durations of the VMs used to execute them. If the sample's duration is too short, then we assume it has not run correctly and

remove that sample. The VM durations of our data follows a bimodal distribution, with the vast majority of samples running for either around 20 seconds or above 120 seconds, with very few in between (∼10%). We assume that the shorter duration of 20 seconds indicate an incomplete execution, and since relatively few samples are between 20 and 120 seconds, we eliminate any samples with durations shorter than 120 seconds as an overestimation of incorrectly executed samples. Of the samples with durations less than 20 seconds, around 30% have zero APIs and at least 40% stop execution after failing to find "MSCOREE.DLL". Additionally, 85% of the removed samples are shorter than the latter, and samples failing to find other library files and samples failing for other reasons make up some proportion of the remaining 15% of the removed samples as well. Thus, the vast majority of these eliminated samples are very likely incompletely executed. This process removes around 41% of the samples, so it removes a significant portion of the data, but for the reasons described above, this process is necessary to removing incompletely executed samples.

For our malicious samples, we also need to ensure that malicious actions have been performed in the VM (see Section 4). Even if a sample is malicious, it may not perform any malicious actions, e.g. the sample has detected it is in a VM (a known issue with dynamic analysis), so those samples need to be filtered out. For this we use Cuckoo's threat scoring system. As discussed in [67], the scoring system can be unreliable in its threat classification, but very low scores can be helpful in determining when no malicious actions have taken place. We removed any malicious samples with a score of 3 or less (out of 10 theoretically, but some scores are higher). Cuckoo considers scores of up to 4 to be low risk, and 67% of the benign data have scores of 3 or more, so eliminating these samples removes malicious samples performing the least amount of malicious behaviors without making the dataset too imbalanced. This removes around 25% of the malicious samples (and thus around 13% of the benign and malicious data), so it also removes a significant portion of the malware data (although not a large percentage of the data overall, see the last paragraph in this section), but manual inspection of randomly selected samples shows that most of

the samples removed by this criteria indeed do not perform any malicious actions, so they need to be removed as well. One caveat for these duration and score statistics is that they are calculated on approximately 55% of the final dataset. Because of the large number of samples, we processed the data with Cuckoo in two batches, and unfortunately, for the second batch, we did not keep the samples that did not meet the duration or Cuckoo score criteria, so we cannot calculate statistics on them. However, since the data are gathered from the same sources (both benign and malicious), we believe these numbers are representative of the whole dataset, and we will extrapolate the same percentages to the whole dataset when estimating how many samples are removed at each step.

Next, we need to make sure that the samples we collected are correctly labeled. We scanned every sample on VirusTotal [68] and removed any benign sample with more than 2 malicious detections and any malicious sample with less than 15 malicious detections (out of $\sim$ 70 detectors). Zhu et al. [69] recommends a threshold of between 2 and 15 in their studies on VirusTotal aggregated labels, and we take the most conservative number for each set to filter out the largest number of incorrectly labeled samples. This removes around 1.2% of the malicious data and 8% of the benign data, so relatively small amounts of data are removed (especially compared to the filtering processes discussed above), but this process is still needed to ensure the samples are correctly labeled.

Finally, to ensure the diversity of malware in our dataset, we gathered VirusTotal [68] reports and processed them with AvClass2 [70] to get malware family statistics. For each sample, we used AvClass2 in compatibility mode to extract the family tag, and we capped any one family at 1200 samples ($\sim$ 11% of the malicious data) to prevent an imbalanced dataset. This filtering is used to only remove the most common family (Emotet), which was likely highly prevalent at the time when we collected the data. This removed around 13% of the malicious data (and thus around 7% of the malicious and benign data), so not a very large number removed, but it was needed to prevent too many Emotet samples from being in the dataset. Table 8.1 shows the 30 most common families in our data. Emotet, a banking

Table 8.1: Top 30 Most Common Malware Families in Our Malware Dataset

Family	Count	Family	Count	Family	Count
Emotet	1200	Virlock	249	Playtech	97
Reveton	1077	Bladabindi	246	Mepaow	89
Agenttesla	911	Xorist	235	Remcos	79
Domaiq	742	Nanocore	234	Shyape	71
Fareit	658	Wacatac	222	Appoffer	67
Poison	389	Lamer	189	Vebzenpak	66
Vilsel	333	Trickbot	186	Noon	59
Gamarue	282	Simda	168	Vbkryjetor	56
Locky	276	Prepscram	163	Gandcrab	52
Teslacrypt	252	Vobfus	152	Upatre	51

trojan/loader [71], is the most common, followed by Reveton (ransomware) and Agent Tesla (a remote access trojan or RAT). Other significant malware families include families of RATs (Poinson Ivy, Bladabindi, NanoCore, Remcos), adware (Domaiq), banking trojans (TrickBot), botnets (Gamarue, SIMDA), potentially unwanted programs (Prepscram, Playtech, Appoffer), ransomware (Locky, TeslaCrypt, Virlock, Xorist, Gandcrab), password stealers (Fareit, Wacatac), spyware (Noon) and other viruses/trojans (Lamer, Vobfus, Mepaow, Shyape, Vebzenpak, Vbkryjetor, Upatre). Malpedia [72], Trend Micro Threat Encyclopedia [73], and Microsoft Security Intelligence [74] were used to determine the categories for these malware families.

After sanitization, we are left with 10,711 benign and 10,563 malicious samples for a total of 21,274 samples. With the caveat of processing in two batches as discussed above, this means approximately 55% of the data is removed with all the steps. The vast majority is removed because of duration ($\sim 41\%$), with removal due to the Cuckoo score ($\sim 8\%$), incorrect labels ($\sim 2\%$), and malware families ($\sim 3\%$) making up a relatively small proportions of samples removed. As discussed above, we are reasonably certain that the samples removed because of duration are incorrectly executed, so we believe removing these data points is justified.

8.2 Mimicry Attack Evaluation

We first test the proposed behavior summary with feature constraining models on various mimicry attack scenarios to evaluate the robustness of these methods to mimicry attacks. We especially want to see how the effectiveness changes as the N value changes. Here we define the N value as the number of benign features removed or made monotonic, starting from the features that are the "most benign" (see Section 6.3).

For the attacks, we use both simulated mimicry attacks and real mimicry attacks to evaluate our detection methods both in the worst-case scenario and in a more realistic scenario respectively. Because simulated mimicry attacks simply combine a malicious trace with a benign trace, they can produce API traces that may not be feasible in real malware if the APIs from the two traces affect each other, e.g., a malicious API deletes a file, but a benign API is still able to read from the same file. For the simulated mimicry attacks, we both manually construct the API sequence of a real attack technique seen in malware (PE injection [75]) and use randomly selected API traces from our malware dataset. We chose PE injection because it is a short malicious API sequence (only 11 APIs long) that appears in a specific order and is well represented in the training data (see Appendix D for the exact APIs used). When used in a simulated mimicry attack, this allows us to evaluate detection methods in the worst-case scenario where malware traces consist mostly of benign APIs (since every API other than the 11 APIs of the PE injection attack came from a benign trace, which can be tens or hundreds of thousands of APIs long). In contrast, we also used randomly selected malware traces because they contain all the APIs present in a real piece of malware, which is a more complete and realistic representation of malware behavior than the PE injection sequence (i.e., a real piece of malware would likely never have only those APIs). However, even with randomly selected malware traces, using them in simulated mimicry attacks allows us to create API traces that can insert the malicious APIs in more locations than in malware with real mimicry attacks.

The process to generate these simulated mimicry attacks is discussed in Section 7.1.

Recall that if the benign trace has multiple processes, then the malicious calls are split evenly among at most 5 of the processes, since our shortest sequence, PE injection, contains only 11 API calls and many of them only make sense in pairs, e.g., opening and reading a file. To exclude benign traces that are too small (and thus may inadequately hide the malicious calls), we only used benign samples with at least 1000 API calls. Additionally, some benign traces are truly massive (many Gigabytes), so for efficiency reasons, we limited the size of the benign traces to 67 MB. All together, this leaves us with 8,676 suitable benign samples from which we randomly select 4,000 per experiment. Finally, as discussed in Section 7.1, for constructed attacks (i.e., PE injection), the same malicious sequence is inserted into all benign traces, while for randomly selected malware traces, a different malware trace is randomly selected for each benign trace.

For real mimicry attacks in malware, we used the techniques described in Section 7.2, which include IAT hooking, EAT hooking, and DLL injection. Since we are using these attacks to evaluate our detection methods, we only hooked APIs considered in our detection methods(see Appendix A), like in [29, 5]. The APIs we inserted are randomly chosen from considered APIs that we've seen appear most often in benign samples, such as reading/writing files/registries, getting system information, getting cursor/keyboard state, etc. (see Appendix E for the APIs we've inserted). Unlike prior work [29, 5, 6], our inserted APIs are not truly *no-op*; we do affect the system like writing files, but we only do so in a manner that's unlikely to affect malware functionality (e.g., creating files with names that are unlikely to exist on an average computer and only writing to or deleting files we've created).

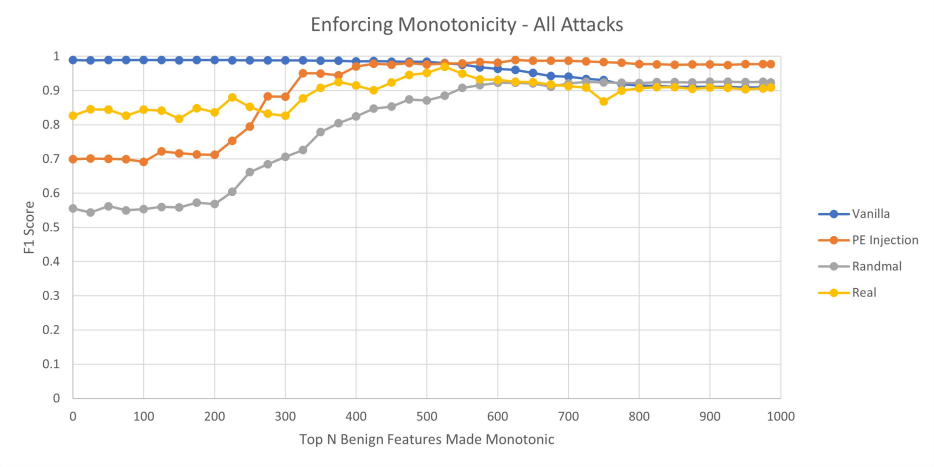
All evaluations are done with 10-fold cross-validation. For the experiments done on the vanilla dataset, 10% of the data is used as the test data while the remaining 90% is used as the training data, for each fold. For experiments with simulated mimicry attacks, we discard the 10% test data and instead use the mimicry attack malware samples and the benign files used to create them as the test data. In accordance with cross-validation, we remove any mimicry attack malware samples and their corresponding benign files from the test set if

the same benign files also appear in the training data. Since the training data is 90% of the entire dataset for each fold, this also removes roughly 90% of the mimicry attack malware samples and leaves 10% for testing, similar to cross validation. For real mimicry attacks, we keep the benign test data and use the real mimicry attack malware as the malicious test data, but we remove any malware where its original is in the training set. All results reported are averaged across the 10 folds. Note, we do not train on any of the mimicry attack malware because our methods are designed to work without having to accurately predict possible attacks, which can be very difficult in practice.

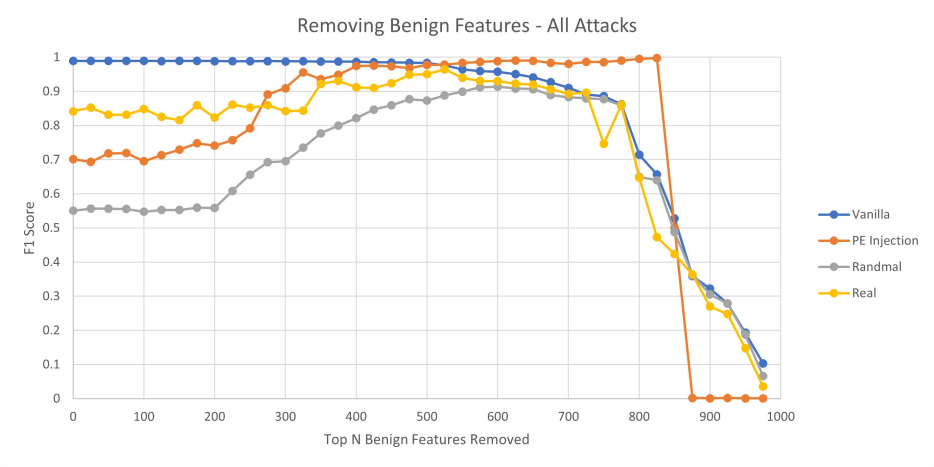
8.2.1 Results

See Figure 8.1 for the performance of both enforcing monotonicity on, and removing, the top N benign features in the behavior summary. F1 scores are graphed at different N values for the three evasive malware attacks (simulated mimicry attacks with PE injection, simulated mimicry attacks with random malware traces, and real mimicry attacks) as well as for vanilla data. Note that when $N = 0$, no features are removed or made monotonic, so $N = 0$ represents the effectiveness of the behavior summary without any feature constraining models.

When we look at Figure 8.1a, we see for the three mimicry attack experiments that as N increases, the F1 scores generally increase as well, peaking around 500 to 600 before mostly plateauing, although these is a slight decrease after the peak for real mimicry attacks. For the vanilla data experiment, the F1 scores start very high at low N values but decrease after the same peak, similar to real mimicry attacks. When we look at Figure 8.1b, we see similar results for removing the top N benign features up until approximately an N value of 700. We see a similar peak at around an N value of 500 to 600 for the three mimicry attacks, a similar decline for vanilla data at around the same N values, as well as a similar slight decline after the peak up until approximately $N = 700$. However, after $N = 700$, there is a large drop in performance. This is expected because if too many features are removed,



(a) Enforcing Monotonicity



(b) Removing Benign Features

Figure 8.1: An overview of the performance (F1 scores) of the proposed detection pipeline (behavior summary with feature constraining models) on all 3 attacks and vanilla data at different N values for both enforcing monotonicity (top) and removing benign features (bottom). The detection pipeline in these experiments use a single LGBM model rather than an ensemble like in subsequent experiments.

then there are fewer and fewer features left to differentiate between benign and malicious samples, causing the classifier performance to approach zero.

The different curves of the different attacks in both graphs of Figure 8.1 also show different properties of these attacks. When $N = 0$, simulated mimicry attacks with random malware traces has the lowest F1 score, followed by simulated mimicry attacks with PE injection, and ending with real mimicry attacks which has the highest F1 score. This indicates that without constraining features, the two simulated mimicry attacks are more difficult to detect than real mimicry attacks, which is expected because simulated mimicry attacks can interweave benign and malicious APIs more evenly than real mimicry attacks (as discussed above). However, as the value of N increases, the F1 scores for simulated mimicry attacks with PE injection quickly rises above the other two attacks, likely because the few but potent malicious features of the PE injection sequence are more exposed as more benign features get constrained, while simulated mimicry attacks with random malware traces and real mimicry attacks have more varied malicious features, so some samples will still be classified as benign even as more benign features are constrained (perhaps malware samples that lack the "most malicious" features).

Another interesting observation is that for both feature constraining techniques, vanilla data, random malware, and real evasive malware all start at varying F1 scores but converge as N increases. This indicates that a drop in performance is likely a fundamental trade-off from removing features or enforcing monotonicity when N is large, because it affects both vanilla data and mimicry attacks. Only PE injection is different, as its performance remains consistently higher than the other three, likely because, as mentioned above, it has only a few specific features that indicate maliciousness, so recall is not affected unless those features are constrained, while for the other three data sets, features that indicate maliciousness are more varied, so some samples will avoid detection as more of those features are removed or made monotonic.

Next, to further analyze the performance details these feature constraining techniques,

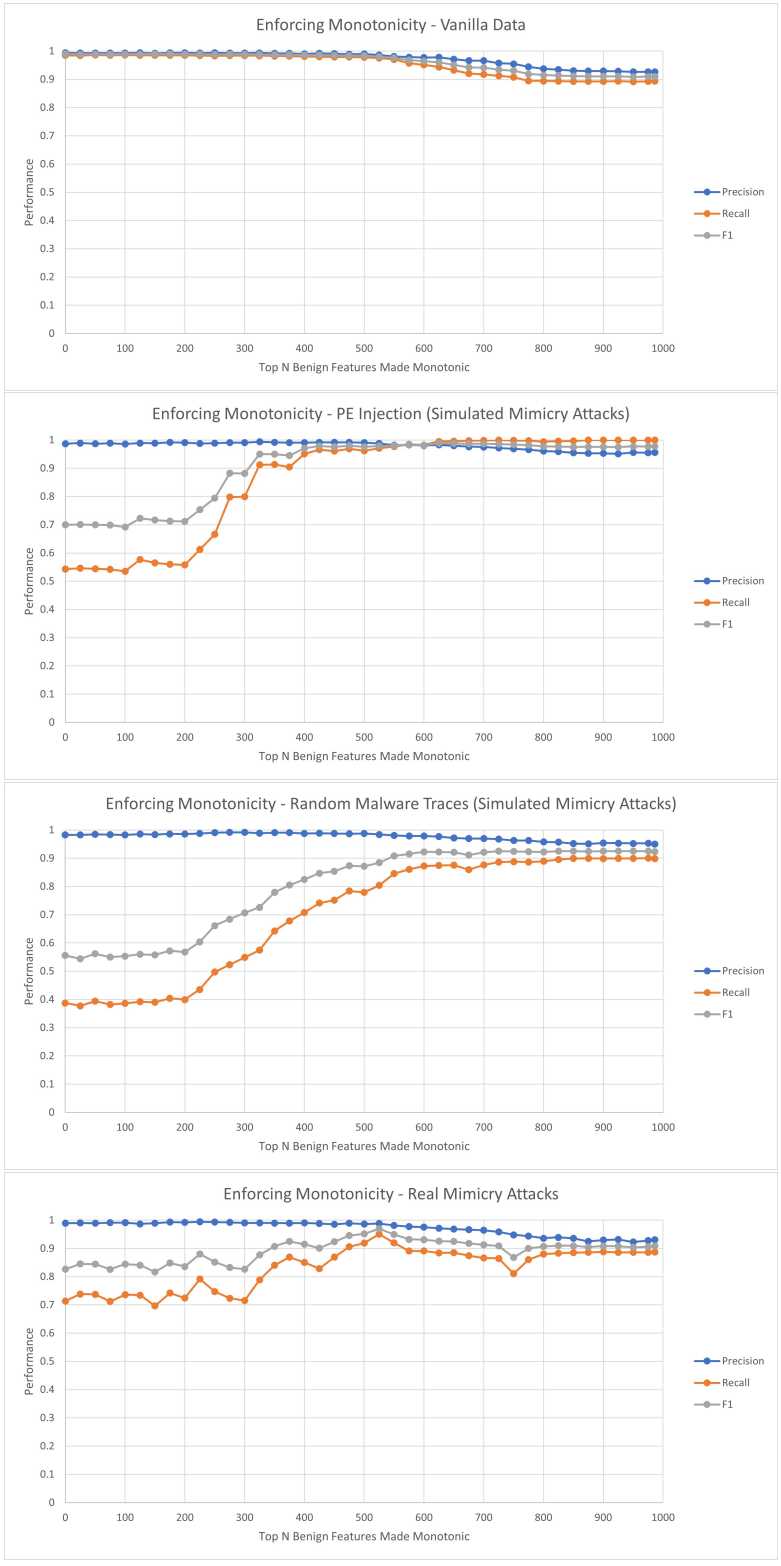


Figure 8.2: Precision, Recall, and F1 scores graphed versus N for the three attacks and vanilla data when enforcing monotonicity in a single LGBM model.

we graph the precision, recall, and F1 scores versus N for the three attacks and vanilla data. See Figure 8.2 for the graphs for enforcing monotonicity. Recall that we saw in Figure 8.1a a decrease to F1 scores after the peak when N is approximately between 500 and 600, in both vanilla data and real mimicry attacks when enforcing monotonicity. When we look at Figure 8.2, we see that for those two experiments, this is due to a decrease in both precision and recall. Similarly, both simulated mimicry attacks (PE injection and random malware traces) also suffer from a decrease in precision at around the same N values, but their recalls increase as well, which leads to overall more consistent F1 scores after the peak.

These performance numbers indicate that enforcing monotonicity on more features generally results in better detection (recall) of attacks, but only up to a certain point, and beyond that, there is a cost to precision and in some situations, a cost to recall as well. These results make sense, because as more features are made monotonic, the presence of those features no longer decreases the classifier output (i.e., the final maliciousness score), thus increasing the detection of attacks that evade detection by adding benign features, but past a certain N value, features that are already strong indicators of malicious behavior are made monotonic, so enforcing monotonicity on those features do not provide much additional benefit. Additionally, if too many features are made monotonic, some benign samples relying on the existence of some of those monotonic features for a benign classification may now be classified as malicious, thus decreasing precision as well.

The slight drop in recall for vanilla data and real mimicry attacks are likely related because Figure 8.1 shows that F1 scores for vanilla data, simulated mimicry attacks using random malware traces, and real mimicry attacks all converge after the aforementioned peak. If the drop in recall is present in vanilla data, then we expect to see the same drop in mimicry attacks with a similar distribution of malware samples. Since real mimicry attacks and simulated mimicry attacks with random malware traces both sample from the malicious dataset to generate their attacks, we would expect to see the same decrease in their recall as the vanilla data. However, we do not see this in simulated mimicry attacks with random

malware traces likely because its F1 score is much lower when $N = 0$, so it does not increase enough to drop after the peak, but it instead rises to converge with the other two experiments when N is large enough. We do not see this pattern PE injection because it uses a constructed API sequence as the malicious trace, which does not come from the same distribution as the malware dataset. As for why we see this slight decrease in recall for vanilla data, the reason is not immediately clear, but one possible explanation is that as more benign features are made monotonic, they cannot be used to lower the final maliciousness score, so to keep the number of false positives low, the training process decreases the impact of certain malicious features, resulting in some samples (both benign and malicious) less likely to be classified as malicious, thus also slightly reducing overall recall. As discussed above, this is likely the same reason why there is a corresponding drop in recall for real mimicry attacks and why the recall in simulated mimicry attacks with random malware traces does not rise higher than the recall in vanilla data.

See Figure 8.3 for the graphs of the precision, recall, and F1 scores versus N for the three attacks and vanilla data when removing features. Before approximately $N = 700$ when the large drop occurs, these graphs are very similar to the monotonic graphs. We see the same peak in F1 scores for the three attacks at an N value of between 500 and 600, while vanilla data starts high and begins to decrease at around the same N value. Also, increasing N generally increases recall before the peak, and precision starts to decrease after the peak. Recall for vanilla data and real mimicry attacks dip after the peak like in the monotonic experiments, but recall in simulated mimicry attacks using random malware traces also dip, likely because these N values are very close to the large drop, where recall will start to dramatically decrease across all four experiments.

As discussed above, unlike the monotonic experiments, these graphs show a large drop in recall after approximately $N = 700$. This is expected because if too many features are removed, features that indicate maliciousness will also start to be removed, including possibly the malicious features that are key characteristics of certain attacks. If those features

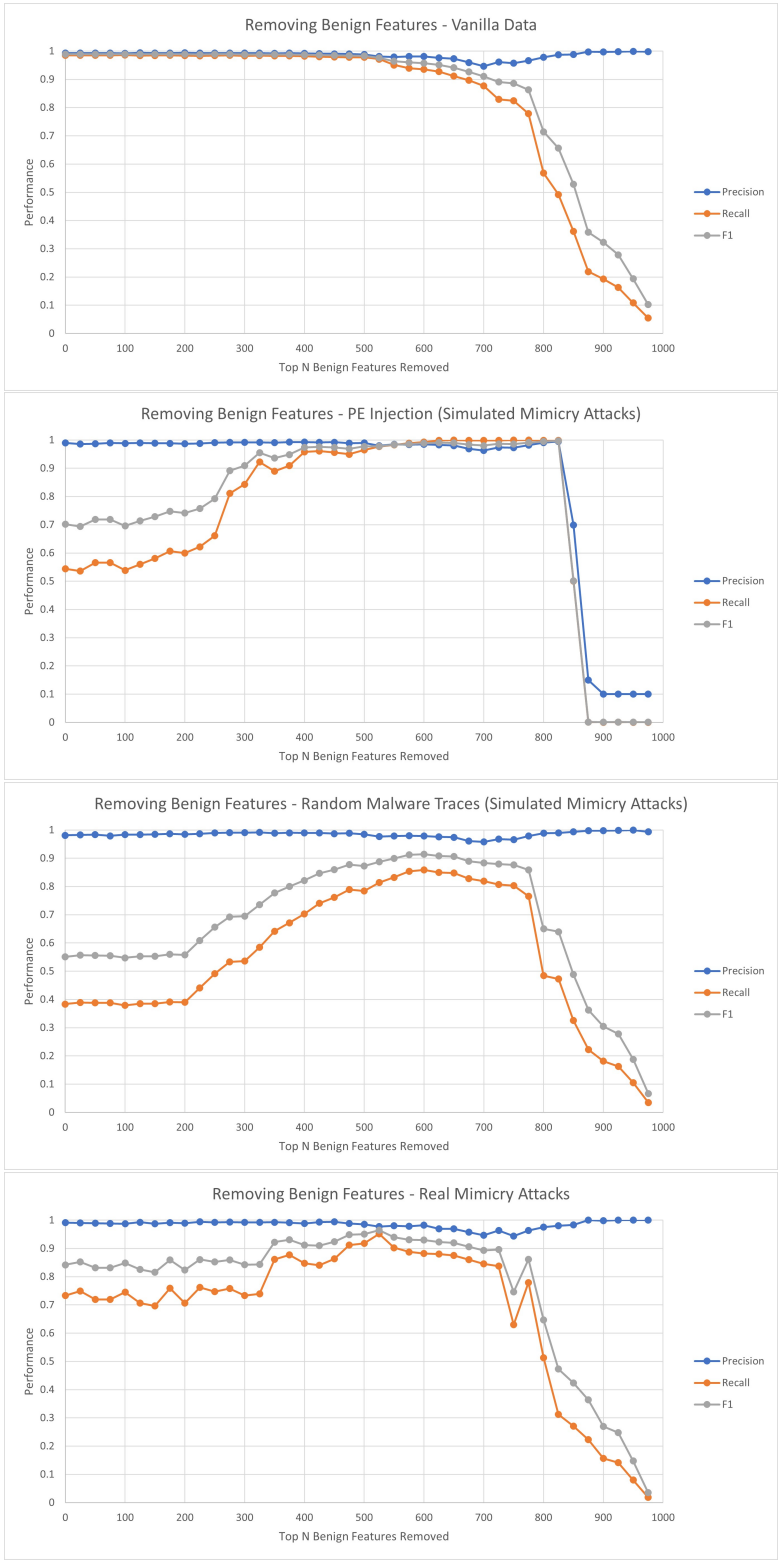


Figure 8.3: Precision, Recall, and F1 scores graphed versus N for the three attacks and vanilla data when removing benign features in a single LGBM model.

are removed, then malicious samples that contain those attacks will appear more benign, reducing recall overall. Recall for vanilla data, simulated mimicry attacks with random malware traces, and real mimicry attacks all drop quickly after approximately $N = 700$, while recall for simulated mimicry attacks with PE injection stays higher for longer, before dropping as well (and even more rapidly). This is also expected because PE injection contains a few very malicious features that won't be removed until very large values of N, which means recall also won't be affected until then, while the other three experiments use more varied malicious samples, so some samples will have important malicious features removed earlier.

Another interesting observation from these graphs is when the large drop in recall occurs, precision becomes even higher (> 0.998) in three of the experiments (vanilla data, simulated mimicry attacks using random malware traces, and real mimicry attacks), while it decreases drastically (to ~ 0.1) in simulated mimicry attacks using PE injection. This can be explained by the way precision is calculated, which is the proportion of samples classified as malicious that are indeed malicious (see Section 2.1.2). If very few samples are classified as malicious (even a single sample), but they are all actually malicious, then the precision becomes 1, but if *no* samples are classified as malicious, then a divide by zero error occurs and precision defaults to 0. Thus, a small change in how many samples are classified as malicious can result a very large difference in precision, which is what we see here. For vanilla data, simulated mimicry attacks using random malware traces, and real mimicry attacks, recall becomes much lower but does not reach 0, so precision can be close to 1 if the few samples classified as malicious are indeed malicious, which is likely to happen because only samples that contain the most malicious features (the only features that have not been removed) are likely to be classified as malicious. One the other hand, for simulated mimicry attacks with PE injection, once the features that correspond to the PE injection sequence are removed, malicious samples are almost indistinguishable from benign samples, so it is very likely that no samples are classified as malicious in many of the folds, leading to a precision close to 0. Because all these measurements are averaged across ten folds, a precision of 0.1 likely

indicates that nine folds have a precision of 0 (because no samples are classified as malicious), while one fold has a precision of 1 (because the small number of samples classified as malicious are all indeed malicious).

When removing features in a real detection system, these results create a dilemma for the defender; N will have to be large enough that mimicry attacks can be detected, but not so large that important malicious features are also removed, which can be a difficult task if future attacks cannot be accurately predicted. This is also somewhat true for enforcing monotonicity, but there is no large drop in recall, so the defender has more leeway when choosing the value of N. More on choosing the appropriate N value will be discussed in Chapter 9.

8.2.2 Super Ensembles

Even though recall can change drastically with different values of N, across all experiments for both enforcing monotonicity and removing benign features, precision remains consistently higher than recall (except for simulated mimicry attacks with PE injection, where recall is only slightly higher, but precision is still very high). Thus, we propose the "Super Ensemble" architecture, which consists of two models where one is trained on data with no features removed or made monotonic, while the other is trained on data with N of its most benign features removed or made monotonic, and a sample is considered malicious if either model classifies it as malicious. We call this a *Super Ensemble* because each of the two submodels can be an ensemble model (although not necessarily). Since recall in vanilla data decreases as N increases, while recall in mimicry attacks increases (generally) as N increases, the Super Ensemble design can ensure that recall on vanilla data remains high (close to 1) as the value of N is tuned to be effective against mimicry attacks. Thus, with the Super Ensemble, both vanilla data and mimicry attacks can be detected at high rates without decreasing precision too much ($\sim 1\%$ decrease) because of the relatively high precisions of these models.

See Figure 8.4 for the precision, recall, and F1 scores graphed versus N for the three

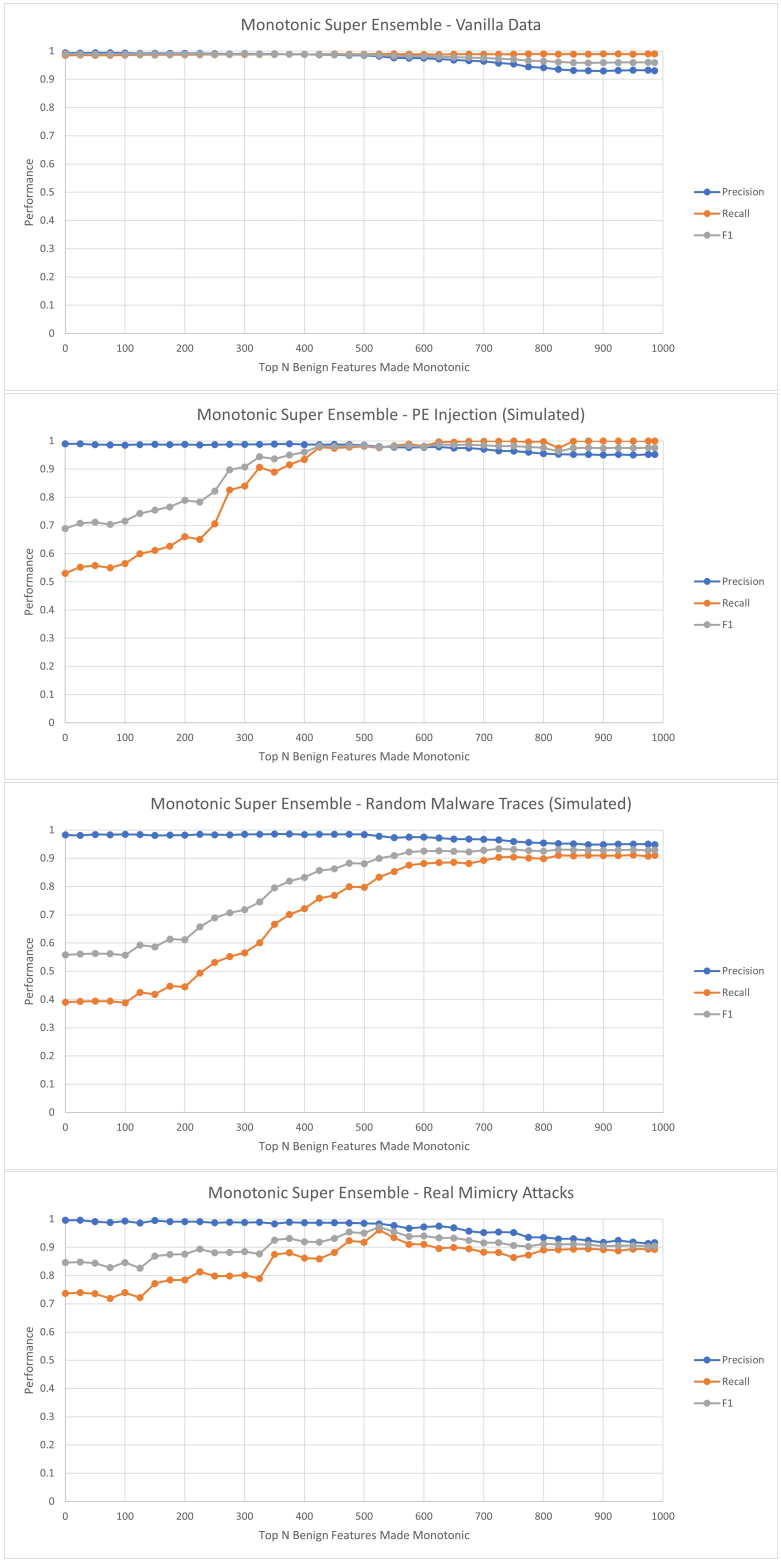


Figure 8.4: Precision, Recall, and F1 scores graphed versus N for the three attacks and vanilla data when enforcing monotonicity in a Super Ensemble.

attacks and vanilla data when enforcing monotonicity in a Super Ensemble. The biggest change from using a single model is that recall no longer decreases in vanilla data when N is large (as designed) because the submodel with no features made monotonic in the Super Ensemble is able detect the malicious samples even when the other submodel increases its N value. However, for the other three attacks, recall still improves but not by much (only by around one percentage point). For simulated mimicry attacks using PE injection, recall on a single model is already high when N is large, so an additional model does not provide much benefit. For simulated mimicry attacks using random malware traces and real mimicry attacks, the model with no monotonic features does not detect these attacks very effectively, and it is likely that most of the samples it can detect can already be detected by the model with the large N value.

See Figure 8.5 for the precision, recall, and F1 scores graphed versus N for the three attacks and vanilla data when removing benign features in a Super Ensemble. The benefits here are much more impactful than when enforcing monotonicity. Due to the design of the Super Ensemble, recall will never drop below the recall of any one of the submodels, and since one of the models does not have any features removed, the recall of the Super Ensemble will never drop below its recall at $N = 0$. This effect is mostly hidden when enforcing monotonicity because increasing N increases recall most of the time, and even when there is a decrease (i.e., in vanilla data and real mimicry attacks), the drop is relatively small. However, when removing features, all experiments experience a large drop in recall when N is large (see Figure 8.3), so this lower limit on recall is much more useful. This means that for vanilla data, recall is very good for all values of N. For the three attacks, there is still a large drop after the peak, but it is much smaller than when only a single model is used.

For both enforcing monotonicity and removing features, these results also show that precision does decrease but only by a relatively small amount (around one percentage point), especially for reasonable values of N (i.e., not at the very end of the graph). Thus, Super Ensembles offer significant benefits to recall, especially for vanilla data, with little cost to

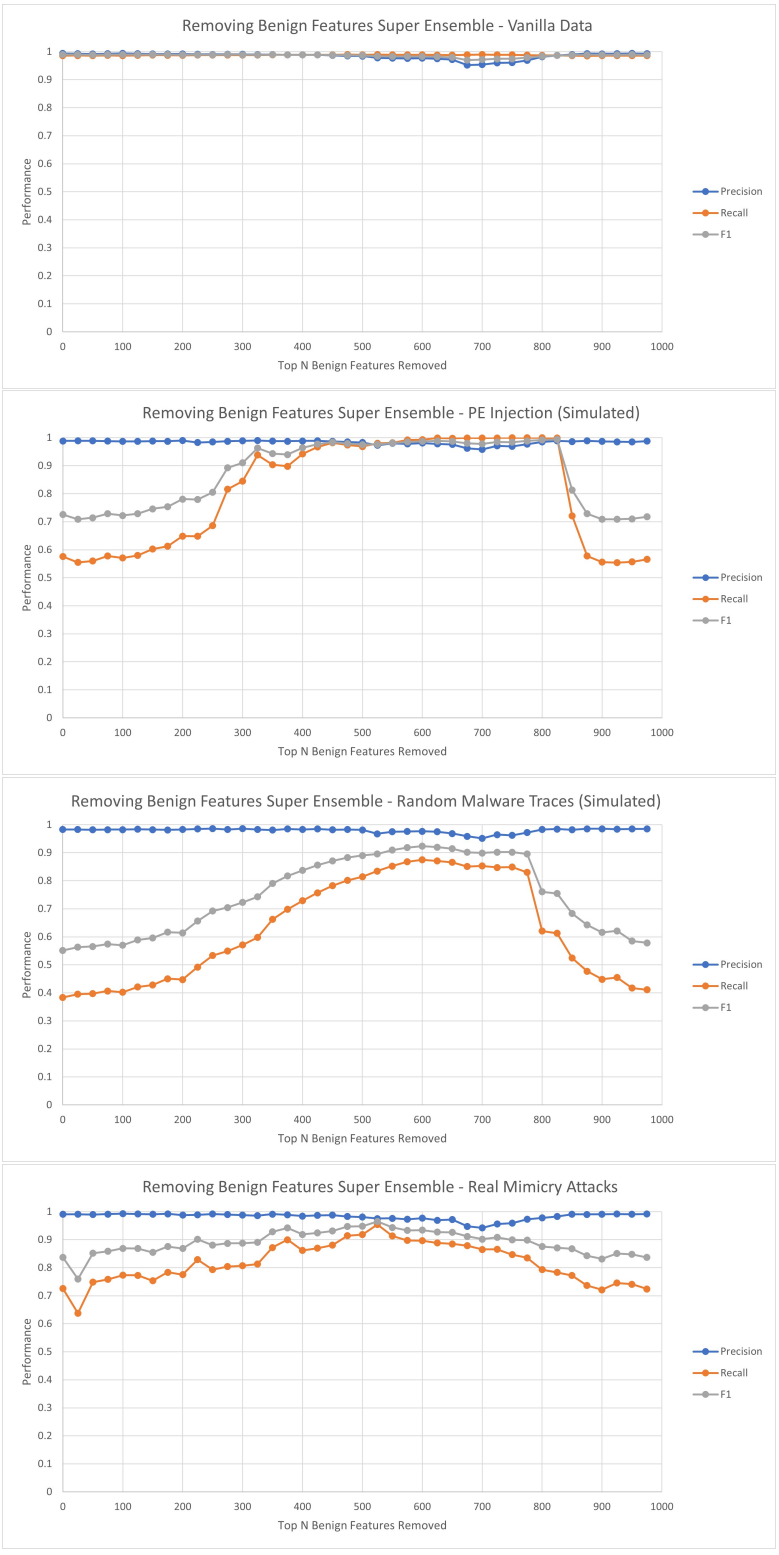


Figure 8.5: Precision, Recall, and F1 scores graphed versus N for the three attacks and vanilla data when removing benign features in a Super Ensemble.

precision. Therefore, these Super Ensembles are our proposed detection models that we will we run further evaluations on. However, even with Super Ensembles, choosing the right value for N is important, because recall still decreases for some N values, just not as much as when using a single model. As mentioned above, more on choosing the appropriate N value will be discussed in Section 9.

8.3 Comparison with Other Techniques

Next, we compare our proposed methods with other common defense techniques on the mimicry attacks. Our goal is to evaluate the additional benefit of our behavior summary, as well as the effectiveness of our Super Ensembles (both monotonic and feature removed) as compared with other adversarial defenses. One aspect we kept constant in all experiments is the underlying classifier (LGBM with the same hyperparameters), so we can directly compare different defense techniques on top of the same model architecture. Note that in this section, we focus on works and techniques that are designed with robustness against mimicry attacks in mind or are likely to be effective against mimicry attacks, rather than any API-based malware detection method, because we do not believe it is a fair or meaningful comparison to show the effectiveness of these detection methods on attacks that they are not designed to defend against. The large variety of feature sizes and models also make direct comparisons difficult and any difference in results less meaningful.

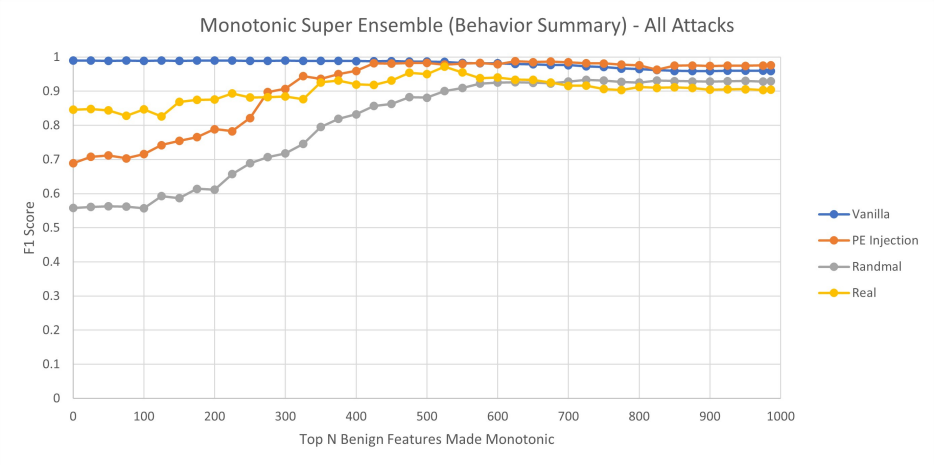
8.3.1 API 2-Grams

To evaluate the additional benefit of our behavior summary, we implemented API 2-grams as a simple method of extracting sequential information without a behavior model. We chose 2-grams as opposed to any other n-grams because they can be encoded in a similar manner as the behavior summary (see Section 6.1), where each feature represents a pair of APIs. This results in a vector with the most similar size as the behavior summary vector,

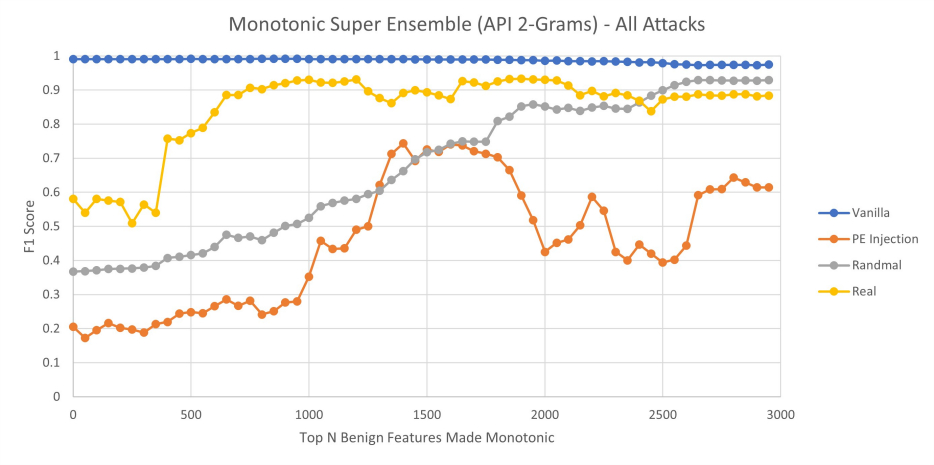
allowing for a more direct comparison (although the 2-grams vectors still has more features than behavior summary vectors, but any other n-grams would have even more). We kept the resource tag information because it can be extracted in the same sliding window algorithm used to generate the 2-grams, but buffer information that indicates changes (e.g. a change from ASCII text to no ASCII text) are not included because they require the saved states of a behavior summary. We still use the Super Ensemble for both the behavior summary and API 2-grams experiments to evaluate the additional contributions of the behavior summary.

See Figure 8.6 for the performance of Super Ensembles with monotonic features on the three attacks and vanilla data at different N values for both the behavior summary and API 2-grams. Note that API 2-grams has about three times as many features as the behavior summary, so the X-axis has a larger range, but the graphs are scaled for a more direct comparison. The most noticeable difference between the two graphs is that the lines for the experiments are more tightly clustered and do not drop as much in the behavior summary than in API 2-grams. This gives the defender more leeway when choosing an N value because there is a large range of N values where detection of the different attacks are all very good, while for API 2-grams, an N value that is good for one attack may be very poor for another. This means that in a real world setting, unknown future attacks are have a better probability of evading detection again API 2-grams because a new attack is more likely to have a different optimal N value for detection than the N value used for the detector, and a suboptimal N value is more likely to result in worse performance in API 2-grams than in the behavior summary.

Another major difference between the behavior summary and API 2-grams is that the lines in the behavior summary graph are much smoother than in API 2-grams, i.e., small changes in the value of N do not result in as much of a change in performance (especially large drops in performance). This is also important for choosing an N value because this means detection rates will be more consistent for any chosen N value if there are any unexpected changes, e.g., if there is a small change in the number of features, if there is a mismatch



(a) Behavior Summary Super Ensemble



(b) API 2-grams Super Ensemble

Figure 8.6: A comparison of the performance (F1 scores) of the proposed Super Ensemble on all three mimicry attacks and vanilla data at different N values for both the behavior summary (top) and API 2-grams (bottom). The Super Ensemble enforces monotonicity on features in one of its submodels.

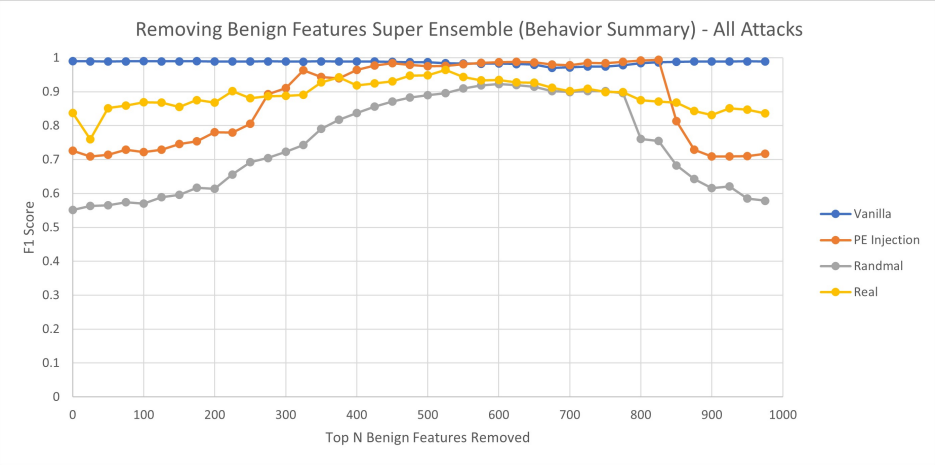
between experimental and real world datasets, or even if there is some random variation in the data or training process. Finally, Figure 8.6 shows a higher overall performance in behavior summary versus API 2-grams. This is most clear in simulated mimicry attacks with PE injection, but the difference is noticeable in real mimicry attacks as well, especially in the peak performances. Simulated mimicry attacks with random malware traces shows similar performances for the two methods at their peaks, but for API 2-grams this is only achieved at very large values of N, where the other two attacks are less effectively detected.

When we look at Figure 8.7, we see similar patterns for the performance of Super Ensembles that remove features. The lines in the behavior summary graph converge better, are smoother, and achieve higher overall scores than the lines in the API 2-grams graph. Thus, when removing benign features, the behavior summary also provides better and more consistent results as well as more leeway when choosing an N value. As compared to enforcing monotonicity, using API 2-grams when removing features results in even more jagged lines, especially during the large drop in performance at larger values of N, while the behavior summary when removing features is more similar to enforcing monotonicity, and thus the behavior summary provides an even greater benefit for removing features than for enforcing monotonicity.

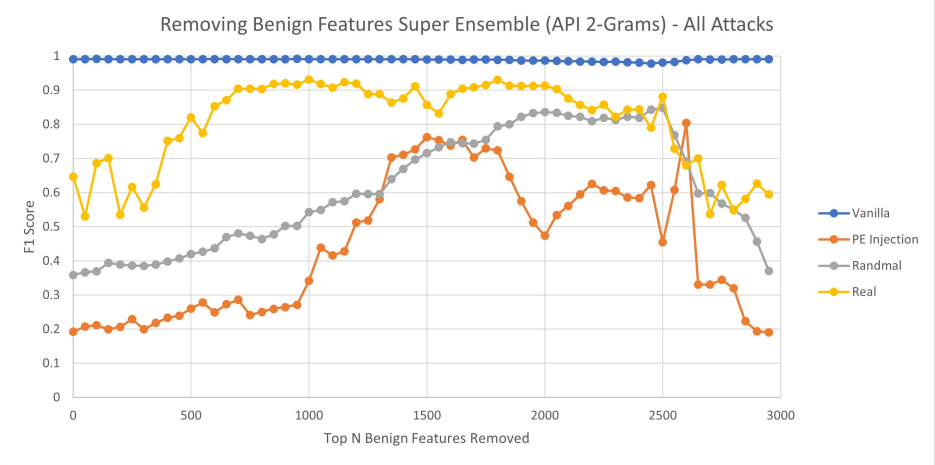
Overall, these results show that using our proposed behavior summary and vectorization method provides a noticeable boost to accuracy and consistency over not using it, and it does so with around one third the number of features, which has speed, storage, and training efficiency benefits as well.

8.3.2 Other Robustness Techniques

Next, we compare our proposed Super Ensembles with other popular robustness techniques: metamorphic detection and adversarial training.



(a) Behavior Summary Super Ensemble



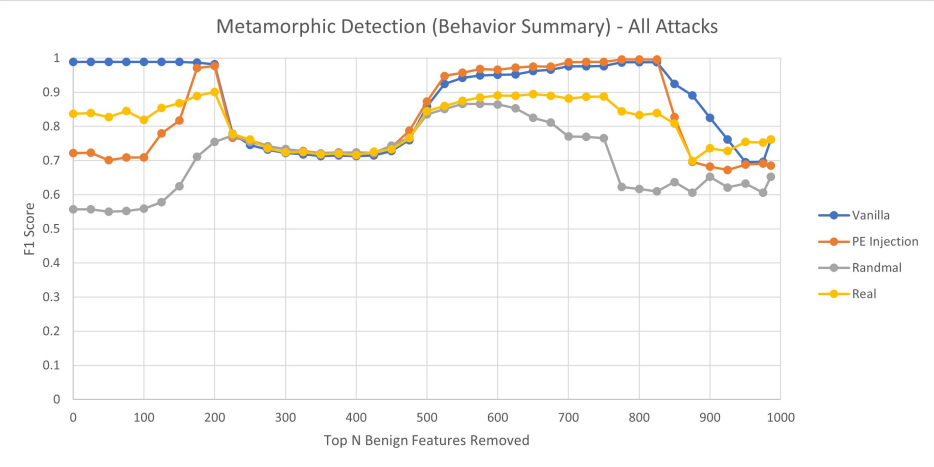
(b) API 2-grams Super Ensemble

Figure 8.7: A comparison of the performance (F1 scores) of the proposed Super Ensemble on all three mimicry attacks and vanilla data at different N values for both the behavior summary (top) and API 2-grams (bottom). The Super Ensemble removes benign features in one of its submodels.

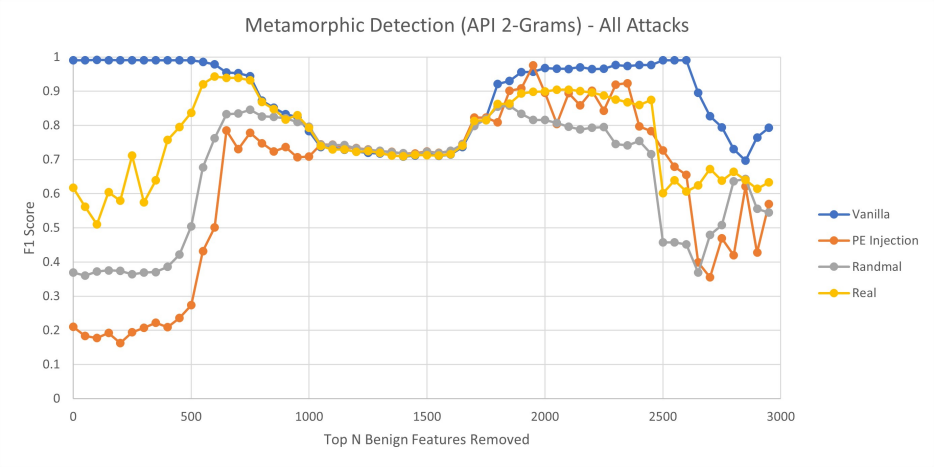
Metamorphic Detection

We implemented metamorphic detection from Singh et al. [38], but with some modifications. Like Singh et al., we trained a classifier on the vanilla dataset, and during inference, any samples classified as benign would have its N most benign features removed and inputted into the *same* classifier again (unlike our Super Ensemble, which trained a *separate* classifier on data with those features removed). However, rather than using an interpretable model, we calculated the top N most benign features using our statistical method from Section 6.3. This allowed metamorphic detection to be applied to any model, not just interpretable ones, and allows for a more direct comparison with our work. Additionally, they used the same model and threshold on the input after removing the benign features, but we found that doing this created too many false positives. Instead, we adjusted the threshold for the classifier scores that separates benign and malicious samples. During training, we removed the top N most benign features on all training samples and inputted them into the classifier again. We then calculated the midpoint between the the average classifier scores for benign and malicious samples and used that as the threshold. This resulted in a significant improvement, so we kept these improvements in our evaluations.

See Figure 8.8 for the graphs of metamorphic detection on all three mimicry attacks and vanilla data at different N values for both the behavior summary and API 2-grams. For small values of N (approximately $N < 200$ for the behavior summary and $N < 600$ for API 2-grams), we see some of the same patterns as the Super Ensemble graphs (Figures 8.6 and 8.7); the attacks start at varying F-1 scores that tend to increase as N increases. This is expected because for small values of N, metamorphic detection is very similar to the Super Ensemble when removing features because they both use an unmodified model and then remove features on a second model. For the same reason, we also see a similar drop in performance when N is large, and too many features are removed for the model to be able to effectively separate malicious and benign samples. Also like the Super Ensembles, we can see that the behavior summary leads to smoother lines and higher overall performance, so



(a) Behavior Summary Metamorphic Detection



(b) API 2-grams Metamorphic Detection

Figure 8.8: A comparison of the performance (F1 scores) of metamorphic detection on all 3 attacks and vanilla data at different N values for both the behavior summary (top) and API 2-grams (bottom). For each sample, if it is classified as benign, the sample has its N most benign features removed and inputted into the same model again. The sample is classified as malicious if the model produces a malicious classification at any time (either originally or after removing benign features).

the behavior summary and vectorization method also has important benefits in metamorphic detection.

However, the most noticeable new pattern we can see in these graphs are the "valleys", where performance decreases and then stays consistent for around the middle values of N in the graphs (approximately $200 < N < 500$ for the behavior summary and $700 < N < 2000$ for API 2-grams) before increasing again. These valleys are interesting because all four lines converge very closely during the valley, more closely than any of the previous experiments. The fact that these patterns occur in the middle of the graphs are also interesting; whatever causes the sharp decrease in performance goes away when N becomes large enough, which we also do not see in any of the other experiments. To further analyze the details and develop possible explanations for these results, we will break down the F1 scores into precision and recall for all four experiments, for both the behavior summary and API 2-grams.

See Figure 8.9 for the precision, recall, and F1 scores graphed versus N for the three attacks and vanilla data when using metamorphic detection with the behavior summary. We can immediately see that for all four experiments, the valley from Figure 8.8 is caused by a decrease in precision (to approximately $0.55-0.6$), while for the same values of N, recall increases rapidly to almost 1. These precision and recall numbers indicate that at those N values, the model (correctly) classifies almost every malicious sample as malicious, but it also classifies a large portion of the benign samples as malicious as well, which means even with our improvement of the metamorphic detection algorithm (adjusting model thresholds during training), removing benign features can cause benign samples to be classified as malicious.

For all four experiments, precision increases again for values of N past the valley. For vanilla data and simulated mimicry attacks with PE injection, precision increases without a corresponding decrease to recall, so there are N values past the valley where precision and recall are both very high, approaching the performance of the base LGBM model on vanilla data. However, in simulated mimicry attacks with random malware traces and real mimicry attacks, this increase in precision is matched by a corresponding decrease in recall. Overall,

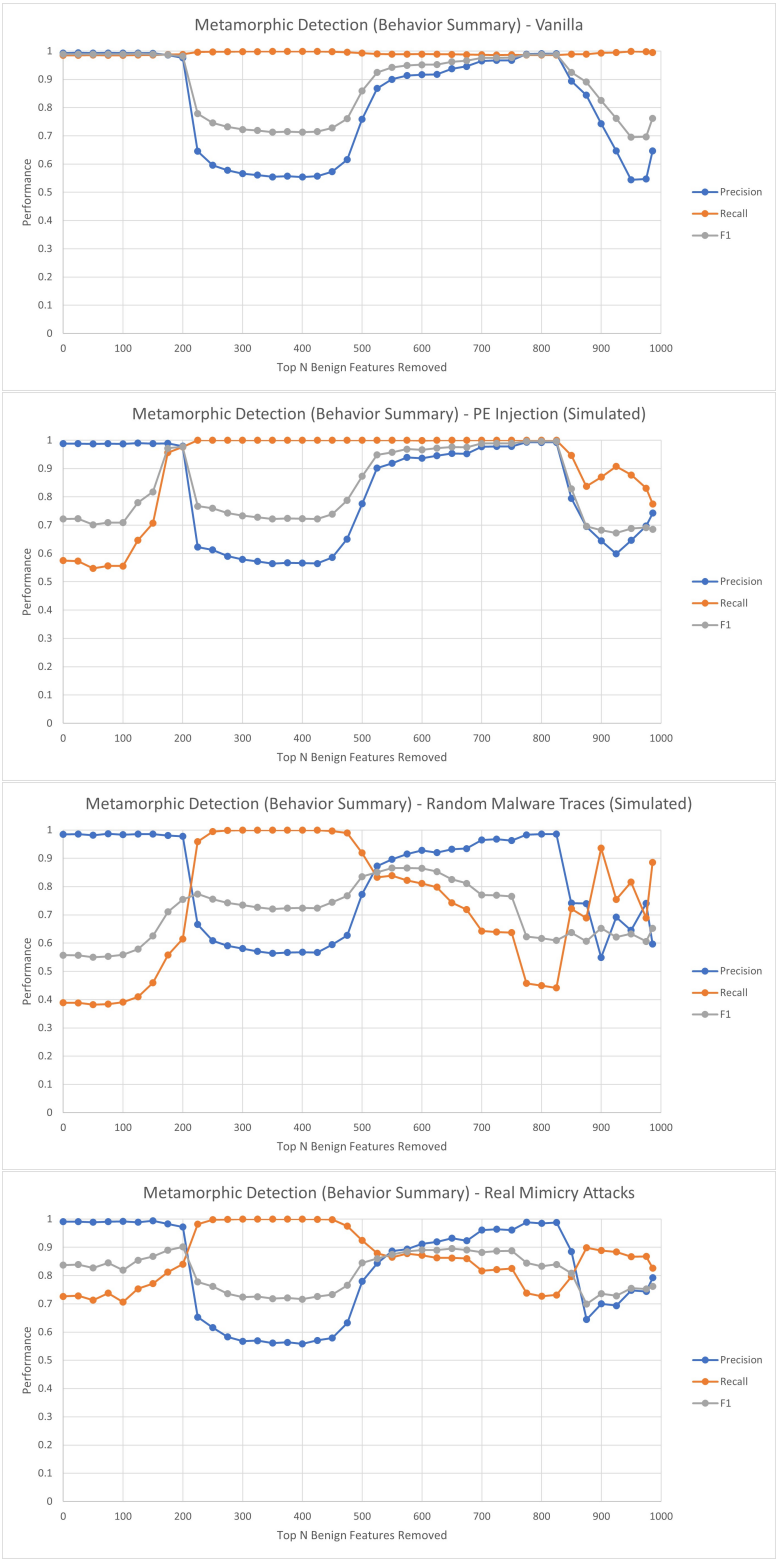


Figure 8.9: Precision, Recall, and F1 scores graphed versus N for the three attacks and vanilla data when using metamorphic detection with the behavior summary.

for these two attacks, whenever precision is high, recall is low, and vice versa, so there are almost no N values where both precision and recall are high, which means metamorphic detection is unable to effectively detect these attacks (a result reflected in the low F1 scores for all N values).

See Figure 8.10 for the precision, recall, and F1 scores graphed versus N for the three attacks and vanilla data when using metamorphic detection with API 2-grams. Like with the Super Ensembles (Figure 8.6 and Figure 8.7), metamorphic detection using the behavior summary results in much smoother lines (and thus more consistent results) than when using API 2-grams. Also like with the Super Ensembles, the behavior summary provides a benefit to detection for simulated mimicry attacks with PE injection, since the API 2-grams graph for that attack no longer shows values of N where both precision and recall are high like the behavior summary graph. However, overall, metamorphic detection with API 2-grams is very similar to the results with the behavior summary: we see the same valley where precision decreases drastically for N values around the middle of the graph; we see no N values with high precision and recall for simulated mimicry attack with random malware traces and real mimicry attacks; and we see N values with high precision and recall in the vanilla data. In conclusion, these results show that although the behavior summary is better than API 2-grams when used with metamorphic detection, neither can achieve good F1 scores on simulated mimicry attacks with random malware traces and real mimicry attacks, and thus metamorphic detection is not an effective method for detecting mimicry attacks.

Adversarial Training

Another technique we implemented is adversarial training, where potential adversarial samples are generated and added to the training set. As discussed in Section 3.2.1, adversarial training requires an accurate prediction of future attacks to be effective, but we believe perfectly predicting future attacks is difficult. Thus, for this experiment, we train on adversarial samples, but not ones from the exact same distribution as the attacks. For experiments with

Figure 8.10: Precision, Recall, and F1 scores graphed versus N for the three attacks and vanilla data when metamorphic detection with API 2-grams.

simulated PE injection malware, we train on the simulated evasive malware generated using randomly selected malware traces, to simulate a situation where the defender knows generally evasive attacks may occur, but not the exact malicious APIs. Conversely, in experiments with randomly selected malware traces and real evasive malware, we train on the PE injection attacks to simulate a situation where the defender targets some specific malware type, but is not able to predict the variety of malware possible. In both scenarios, the defender knows the attack method (inserting APIs), but not the exact form the attack will take, which we believe is more realistic than a scenario with perfect knowledge. Note that adversarial training can be applied on top of other techniques (e.g., our Super Ensembles), but we evaluated them separately to compare the individual contributions of these techniques.

Since there is no N parameter for adversarial training, we do not create graphs for these results. Instead, we compare the results from adversarial training with results from other methods in the table below.

Comparison Table

See Table 8.2 for a table comparing the performances of all the techniques discussed so far. For conciseness, we abbreviated "behavior summary" as *BeS*, "API 2-grams" as *(2-G)*, "adversarial training" as *adv. training*, "metamorphic detection" as *meta. detection*, "Super Ensemble" as *Super Ens.*, "enforcing monotonicity" as *mono*, and "removing features" as *rmv*. The left half of the table shows the results of the analyzed robustness techniques with the behavior summary while the right half of the table shows the results using API 2-grams. Results for the base LGBM model, adversarial training, metamorphic detection, and the two Super Ensembles are listed for each of the three attacks and for vanilla data. For techniques that require an N parameters, they are chosen so the models perform the best across all attacks, with more importance given to more realistic attacks, i.e., real mimicry attacks are the most realistic, while simulated mimicry attacks with random malware traces are the next, and simulated mimicry attacks with PE injection are the least realistic. The N values

Table 8.2: Model Performance on Mimicry Attacks. The N values used are the ones that performed the best across all attacks: Meta. - 600, Meta. (2-G) - 700, Super Ens. (Mono, BeS) - 525, Super Ens. (Mono, 2-G) - 1950, Super Ens. (Rmv, BeS) - 525, and Super Ens. (Rmv, 2-G) - 1950. For clarity, results for our proposed methods (both Super Ensembles with the Behavior Summary) are bolded. The performance metrics reported are accuracy (Acc), precision (Prec), recall (Rec), F1 score, and false positive rate (FPR) as a percentage.

Model	**Behavior Summary (BeS)**					**API 2-Grams (2-G)**				
	Performance Metrics									
	Acc	Prec	Rec	F1	FPR(%)	Acc	Prec	Rec	F1	FPR(%)
					Vanilla					
LGBM	0.990	0.993	0.986	0.989	0.7	0.991	0.995	0.987	0.991	0.5
Adv. Training	0.988	0.985	0.990	0.988	1.5	0.990	0.991	0.989	0.990	0.9
Meta. Detection	0.952	0.920	0.989	0.954	8.4	0.948	0.911	0.992	0.950	9.6
Super Ens. (Mono)	**0.986**	**0.983**	**0.990**	**0.986**	**1.7**	0.987	0.983	0.990	0.987	1.7
Super Ens. (Rmv)	**0.985**	0.**981**	**0.990**	**0.985**	**1.9**	0.988	0.986	0.991	0.988	1.4
					PE Injection (Simulated)					
LGBM	0.762	0.988	0.532	0.688	0.7	0.552	0.956	0.108	0.193	0.5
Adv. Training	0.992	0.987	0.998	0.993	1.3	0.967	0.989	0.944	0.966	1.0
Meta. Detection	0.944	0.929	0.962	0.942	7.3	0.814	0.896	0.705	0.781	7.7
Super Ens. (Mono)	**0.978**	**0.982**	**0.974**	**0.978**	**1.8**	0.676	0.955	0.369	0.526	1.6
Super Ens. (Rmv)	**0.980**	**0.978**	**0.982**	**0.98**0	**2.2**	0.671	0.966	0.354	0.513	1.2
					Random Malware Traces (Simulated)					
LGBM	0.689	0.979	0.386	0.553	0.8	0.611	0.981	0.227	0.368	0.4
Adv. Training	0.724	0.988	0.454	0.622	0.6	0.685	0.988	0.375	0.543	0.5
Meta. Detection	0.865	0.930	0.790	0.853	6.0	0.857	0.902	0.801	0.848	8.8
Super Ens. (Mono)	**0.906**	**0.976**	**0.832**	**0.899**	**2.0**	0.876	0.983	0.765	0.861	1.3
Super Ens. (Rmv)	**0.906**	**0.974**	**0.835**	**0.899**	**2.2**	0.853	0.978	0.723	0.831	1.6
					Real Mimicry Attacks					
LGBM	0.859	0.992	0.725	0.837	0.6	0.709	0.992	0.422	0.578	0.3
Adv. Training	0.851	0.992	0.712	0.825	0.6	0.818	0.989	0.643	0.777	0.7
Meta. Detection	0.888	0.907	0.867	0.886	9.1	0.943	0.919	0.970	0.944	8.5
Super Ens. (Mono)	**0.971**	**0.981**	**0.960**	**0.971**	**1.8**	0.935	0.986	0.883	0.932	1.3
Super Ens. (Rmv)	**0.971**	**0.982**	**0.958**	**0.970**	**1.7**	0.917	0.983	0.849	0.911	1.4

used are as follows:

1. **Super Ensemble (Monotonic) - Behavior Summary**: **N=525**. $N = 525$ achieves the best results on real mimicry attacks, while $N = 550$ slightly reduces performance against real mimicry attacks but slightly increases performance against simulated mimicry attacks with random malware traces. Performance on vanilla data and simulated mimicry attacks with PE injection are good for both N values, so they are not important factors. While both N values are good choices, we choose $N = 525$ to optimize for performance on real mimicry attacks. See Figure 8.6a for reference.

2. **Super Ensemble (Monotonic) - API 2-Grams: N=1950**. The best performance on real mimicry attacks are achieved with N values around 1000 and N values around 1900. Around those values, the best performance on simulated mimicry attacks with random malware traces are N values around 1900. The best performance on simulated mimicry attacks with PE injection are N values around 1500, but even then, the best recall achieved is low at around 0.6. Thus, we optimize only for real mimicry attacks and simulated mimicry attacks with random malware traces and choose an N value around 1900. We use $N = 1950$ because it has the best performance on simulated mimicry attacks with random malware traces with no difference on real mimicry attack performance. See Figure 8.6b for reference.

3. **Super Ensemble (Removing Features) - Behavior Summary: N=525**. Like the monotonic super ensemble, $N = 525$ achieves the best performance on real mimicry attacks, but $N = 550$ has a larger drop than in the monotonic super ensemble. Thus, $N = 550$ is less viable here than enforcing monotonicity, so we use $N = 525$ to optimize for performance on real mimicry attacks. See Figure 8.7a for reference.

4. **Super Ensemble (Removing Features) - API 2-Grams: N=1950**. Like the monotonic Super Ensemble, N values around 1900 achieve the best performance on real mimicry attacks and simulated mimicry attacks with random malware traces.

Unlike the monotonic Super Ensemble however, $N = 1800$ is the peak in performance for real mimicry attacks, although it is a relatively small difference compared to its surrounding points. We choose $N = 1950$ because the small drop in performance on real mimicry attacks results in a much larger performance increase on simulated mimicry attacks with random malware traces. Additionally, choosing an N value similar to the monotonic Super Ensemble also allows us to make more direct comparisons between the two. Again, we do not consider simulated mimicry attacks with PE injection because of its low recall. See Figure 8.7b for reference.

5. **Metamorphic Detection - Behavior Summary: N=600**. The possible options for good performance are either before the valley at $N \approx 200$ or after the valley at $N \approx 600$. Peak performances on real mimicry attacks are similar for both, but the performance on simulated mimicry attacks with random malware traces is much higher after the peak, so we choose $N = 600$, where performance on real mimicry attacks is close to a peak but before the performance on simulated mimicry attacks with random malware traces decreases. See Figure 8.8a for reference.

6. **Metamorphic Detection - API 2-Grams: N=700**. Like metamorphic detection with the behavior summary, the possible options for good performance are either before the valley at $N \approx 700$ or after the valley at $N \approx 2000$. However, unlike metamorphic detection with the behavior summary, the performance on real mimicry is noticeably higher before the valley, while the performance on simulated mimicry attacks with random malware traces is about the same before or after the valley, so in this case we choose before the valley at $N = 700$. Performance on simulated mimicry attacks with PE injection is better after the valley than before, but the performance after the valley is very inconsistent, and as discussed above, we place more importance on real mimicry attacks because they are the most realistic attacks. See Figure 8.8b for reference.

Note that by choosing an N value, some techniques may be shown to be slightly less ef-

fective in this table than their maximum potential. However, we do this to simulate the decision a defender will need to make in a real world setting: choosing an N value based on training data to optimize for some attacks over others. This allows us to look at concrete performance numbers comparing these methods and techniques in the most realistic setting. For performance trends against different attacks across the entire range of N values, refer to their respective graphs (linked above in the discussion of choosing N values). From Table 8.2, we make the following observations about these techniques:

Finding 1: The behavior summary offers substantial benefits over API 2-grams, even without constraining features. Across all 3 mimicry attack experiments and for various models and robustness techniques, the behavior summary achieves better results than API 2-grams. Even without any additional robustness techniques, we can see that all 3 attacks are highly effective against LGBM using API 2-grams but much less so against LGBM using the behavior summary, especially for simulated mimicry attacks using PE injection and real mimicry attacks. Across all models and techniques, the difference is most dramatic for simulated mimicry attacks using PE injection and less so but still noticeable for simulated mimicry attacks using random malware traces. This is unsurprising, since PE injection is a particular sequence of API calls, so when these calls are inserted into benign processes, API 2-gram features are modified by the additional calls, whereas a behavior summary can group those calls together and extract the necessary sequential information. For simulated mimicry attacks using random malware traces, since they are likely to contain many more indicators of maliciousness, there are enough malicious features for both the behavior summary and API 2-grams to be effective, although the behavior summary does still provide additional information, as seen in the increases to recall (approximately 7-16%) for all models except metamorphic detection, where the performances for both are similar, but the behavior summary still has a slightly higher F1 score.

For real mimicry attacks, there is a large recall difference in the behavior summary vs.

API 2-grams for LGBM (almost as high as PE injection), but the difference is less (at around the same as simulated mimicry attacks using random malware traces) for the Super Ensembles. One possible explanation is that since real mimicry attacks are more restricted (and thus less powerful) than simulated mimicry attacks, the behavior summary is effective enough to achieve a decent detection rate even without additional robustness techniques. However, when features are removed or made monotonic, API 2-grams become more effective as well, and the additional benefit of a behavior summary is reduced. Nevertheless, the behavior summary still provides a significant increase in detection for both Super Ensembles and most of the other models. The only outlier to this pattern is metamorphic detection on real mimicry attacks, where API 2-grams performs better than behavior summary. We have found that metamorphic detection is quite inconsistent (large differences in detection rate for small changes of N, see Figure 8.8), so this can be simply be a result of the N values chosen or even due to randomness. Overall however, these results are consistent with the finding in Section 8.3.1.

Finding 2: Super Ensembles that remove features achieve similar results to Super Ensembles enforcing monotonicity in most cases. For all 4 experiments, removing features achieves almost identical recall and precision to enforcing monotonicity when using the behavior summary. This is mostly true for API 2-grams as well, although there is a small difference for simulated mimicry attacks using random malware traces and real mimicry attacks. This is likely because for API 2-grams, when there is a variety of malicious features (i.e., in random malware traces and real mimicry attacks), removing features will inevitably result in some malicious features also being removed from some samples, thus reducing overall recall. However, the drop in recall is still small (around 4% at most), so removing benign features is still a viable method.

Finding 3: Adversarial training can be effective, but only if the malware behaviors can be accurately predicted. For simulated mimicry attacks using PE injection,

adversarial training on simulated mimicry attacks using random malware traces is effective both for the behavior summary and for API 2-grams. Since the attack generation method is the same for both (inserting benign traces into malicious traces), as long as there are sufficient samples exhibiting behavior similar to PE injection in the malware set, it is expected that adversarial training will be effective. The behavior summary also performs slightly better ($\sim$ 5% more recall) than API 2-grams in this scenario, likely for the same reasons as discussed in Finding 1. However, for simulated mimicry attacks using random malware traces and real mimicry attacks, training on PE injection is not effective, although it is still more effective than the base LGBM model. This likely means that training on PE injection is effective at detecting similar malware, but since there are many other malware types in the test set, the improvement is small. This technique not being effective against simulated mimicry attacks using random malware traces also indicates that even if the attack generation method is identical, the malware used to generate adversarial samples lacks sufficient variety, adversarial training is not effective. This finding is largely consistent with the literature (see Section 3.2.1) .

Finding 4: Metamorphic detection is less effective than our methods. For all three mimicry attack experiments, our methods (behavior summary with Super Ensemble) achieve higher F1 scores than metamorphic detection, even with our improvements over the original work [38]. For PE injection, metamorphic detection when using the behavior summary achieves really good recall but at the cost of precision (and thus also FPR). For simulated mimicry attacks using random malware traces and real mimicry attacks, both recall and precision are lower than in our methods when using the behavior summary. For these two attacks when not using the behavior summary, recall is better than the Super Ensembles, but the precision is still lower, which results in similar F1 scores overall. For all experiments, metamorphic detection requires a lower precision to achieve good recall, which results in worse overall performance than our proposed methods (that use both the behavior

summary and Super Ensembles).

These results show that training on the vanilla dataset but removing benign features during inference (at least when removing enough to detect mimicry attacks) results in too many false positives, which means the authors' central assumption that removing benign features from benign samples does not result in a significant change to its classification score due to the presence of other benign features is not correct in this case with our API features. Thus, first removing those features from the data and then training on them (like in our proposed Super Ensemble models) is required to get good results.

Finding 5: Our methods perform the best overall, but does have lower precision and a higher false positive rate than some other methods. Our proposed methods, removing features and enforcing monotonicity with the behavior summary, achieves the most consistently good results of all the methods tested. Some other methods do perform slightly better on select attacks, but on the other attacks, they perform much worse. However, the cost of achieving this performance is a slightly higher FPR (by $\sim 0.5\%$ to 1%) than the other models, likely due to both our vectorization algorithm (which combines many features) and robustness techniques (monotonocity and removing features). As discussed in Section 8.2.1, the FPR can be decreased by decreasing N, but at a cost to recall. The defender will need to determine the acceptable FPR for the desired recall.

8.4 Malware Mimicry Attack Case Study

To gain a better understanding of why the combination of the behavior summary and a Super Ensemble that constrains features are effective against mimicry attacks in API traces, we will do a case study of a specific malware sample. The sample we will study has the MD5 hash `e119511f9b92ec688f4d23b3b5417f7d` and is classified most commonly as a generic backdoor trojan by VirusTotal vendors [68]. This classification is used by Cuckoo vendors whenever there is not a more specific category to assign to it, and we did not observe any

opened ports or attempted network activity that indicates an actual backdoor, although it is possible the sample determined it is in an environment disconnected from the internet and did not attempt any network activity. When run in our Cuckoo sandbox, this malware sample performs mostly benign-looking actions: opening/reading files, opening/reading registry keys, etc., but it uses PE injection (or another similar process injection technique) to launch a second process, which also performs mostly benign-looking actions like opening/reading files and opening/reading registry keys. See Figure 8.11 for the APIs captured by Cuckoo demonstrating the PE injection taking place. Similar to the example used in Section 7.2, this malware sample runs the PE injection code as part of an extracted payload, so we use the same IAT and EAT hooking techniques described in Section 7.2 to output the functions called by the malware (red boxes) before they are captured by Cuckoo (blue boxes).

In this case study, we will first evaluate the models on the vanilla sample to establish a baseline, and then we will evaluate the same models on the sample modified to perform mimicry attacks to see how mimicry attacks degrade model performance. The vanilla sample is chosen from the malware dataset, and mimicry attack sample is the corresponding sample from the real mimicry attack dataset described earlier, in Section 8.2. To see the individual contributions of both the behavior summary and the Super Ensemble, we will compare the predicted maliciousness scores of this sample produced by classifiers with and without the Super Ensemble, as well as with and without the behavior summary. Thus, we will be evaluating an LGBM model without the behavior summary, an LGBM model with the behavior summary, a Super Ensemble without the behavior summary, and a Super Ensemble with the behavior summary. For the Super Ensemble experiments, we evaluate both enforcing monotonicity and removing features, and the N values used for the Super Ensembles are the same values used in the previous section. Because the training process is randomized, there may be variation in maliciousness scores, so we repeat each experiment 100 times and plot a histogram of the scores. For the Super Ensemble graphs, the maliciousness score used is the highest score of its submodels because a sample is classified as malicious if any submodel

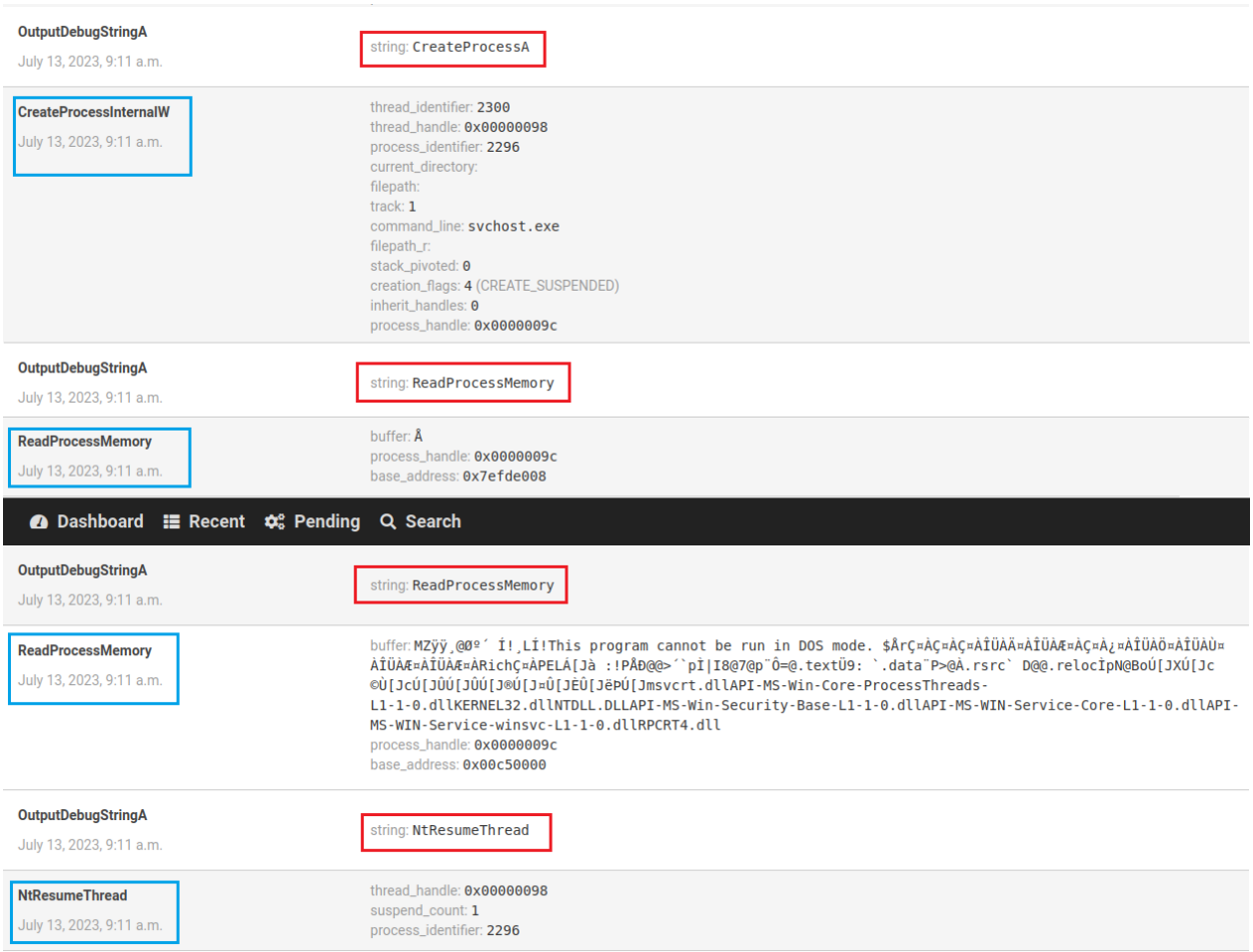


Figure 8.11: API trace of the malware sample that shows PE injection taking place. The red boxes show the functions called by the malware, while the blue boxes show the APIs that those function calls are captured as by Cuckoo.

Figure 8.12: Histograms of maliciousness scores produced by various models on the vanilla malware sample. As expected, all models produce very high scores and classify this sample as malicious 100% of the time.

classifies it as such.

See Figure 8.12 for the histograms of maliciousness scores produced by the models on the vanilla malware sample. Like previously in Table 8.2, we use the base LGBM model to represent a model that is not a Super Ensemble, and we use API 2-grams as a method to generate sequential features without the behavior summary. The label for each bin represents the lower bound of the range of scores placed in that bin, up to the next bin label. For example, the bin labeled 0.1 contains all scores ranging from 0.1 to 0.2. Since no score lies exactly on the edge of the bin ranges (as most scores are floating point numbers with 15 decimal places), the results are identical whether these ranges are inclusive or exclusive.

A score > 0.5 produces a malicious classification, while a score < 0.5 produces a benign classification. When we look at Figure 8.12, we see that all models solidly classify the sample as malicious, producing scores in the highest bin for all 100 runs, and when we look at the actual scores, all scores are > 0.999. This is expected because the sample we used is a vanilla malware sample with no modifications, and the models are all specifically trained to classify these samples like this.

Next, we evaluate the same sample modified to perform mimicry attacks. Recall from Section 7.2 that this includes using IAT and EAT hooking to insert additional benign calls, as well as using DLL injection to hook any spawned processes. The mimicry attacks reduce the effectiveness of detection models in two ways. First, they "disrupt" malicious sequences by inserting benign calls between malicious APIs, producing a new, unknown, and more benign-seeming sequence that still performs the malicious actions. In this malware sample, this means inserting benign calls like `GetForegroundWindow`, `OutputDebugStringA`, and `GetCursorPos` before and after the PE injection calls: `CreateProcessInternalW`, `ReadProcessMemory`, and `NtResumeThread`. Additionally, mimicry attacks add more benign calls to all parts of the malicious sample, increasing benign features in the sample overall. The behavior summary and the Super Ensemble each address one of these aspects of mimicry attacks; the behavior summary, by grouping calls based on related resources, prevents API sequences from being modified by inserted APIs if the resources are different, and even if the sequences are modified (i.e., when the inserted APIs have the same resource as the sequence), the vectorization algorithm prevents malicious features from being removed; the Super Ensemble, by removing features or enforcing monotonicity, constrains the additional benign features to limit their impact on the final output.

Figure 8.13 shows the histograms of maliciousness scores produced by the models on the malware sample modified to perform mimicry attacks. In contrast with the vanilla sample (Figure 8.12), these graphs show a clear impact of the mimicry attacks. Only the two Super Ensembles with the behavior summary remain effective at classifying the sample

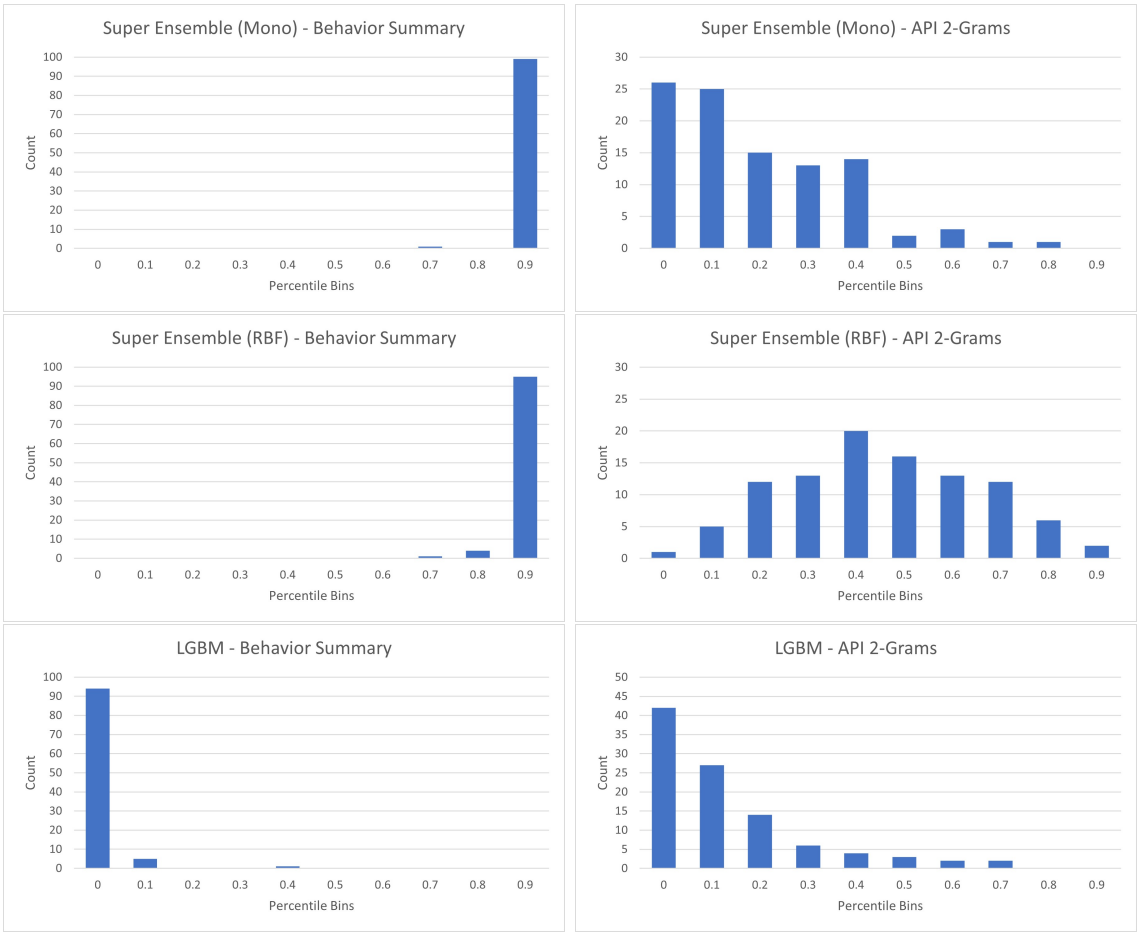


Figure 8.13: Histograms of maliciousness scores produced by various models on the malware sample modified to perform mimicry attacks. Only Super Ensembles that use the behavior summary remain effective at detecting the sample with mimicry attacks.

as malicious, while the other methods all show a degradation in performance to varying degrees. The monotonic Super Ensemble with API 2-grams, the LGBM model with the behavior summary, and the LGBM model with API 2-grams all produce a benign score (< 0.5) more often than a malicious score (> 0.5), which means they are more likely to classify this sample as benign rather than malicious, while the Super Ensemble that removes features produces a benign score equally as likely as producing a malicious score, better than the others but still not a good detection rate.

These results show that missing either the behavior summary or the Super Ensemble drastically degrades performance. Using the behavior summary without the Super Ensemble achieves the worst performance out of all the models, which likely indicates that even when preserving all the malicious features from the vanilla sample, the addition of many benign features causes a benign classification. The Super Ensembles that do not use the behavior summary achieve moderately better performance, with the monotonic Super Ensemble having a significant number of scores close to or over the threshold (in the 0.4 bin or higher), while the Super Ensemble that removes benign features classifies the sample as malicious about as often as it classifies it as benign. This likely means that even with the PE injection sequence modified by inserting benign APIs, those malicious APIs (and their new API 2-gram features) still remain, and they can have an effect on the classification scores when benign features are constrained in the Super Ensembles. Finally, the LGBM model with API 2-grams (that uses neither the behavior summary nor the Super Ensemble) performs slightly better than the LGBM model with the behavior summary, but it still outputs a benign score the vast majority of the time. It is unclear the reason for this, but it is possible that since API 2-grams uses more features overall, the proportion of benign features added is less than in the behavior summary because there are more malicious features overall as well. It is also possible that with more features, there is more variance in the training process, which leads to higher scores in some runs simply due to this variance.

It is clear from Figure 8.13 that both the Super Ensemble and the behavior summary are

```
Group 26 - {'pid_2624', 'svchost.exe'}

* CreateProcessInternal{1-1} -> ReadProcessMemory{2-2} ->
NtResumeThread{1-1} -> NtOpenProcess{1-1} -> WriteProcessMemory{1-1} ->
CreateRemoteThreadEx{1-1} -> CreateThread{1-1}
<|0|> <{00000000000000000000000000000000100000}>
<[<(0,; 0,; 0,; 0,; 0,; 0,; 0)> diffs: ]> 1 times,
```

Figure 8.14: Behavior summary group for the PE injection API calls from the malware sample. Even when performing mimicry attacks, these APIs are grouped together in a sequence by the behavior summary.

needed to reliably detect this sample. To better understand their contributions, we can look at the relevant sections of the behavior summary for this sample. As discussed above, the most malicious behavior in this sample is the PE injection, and Figure 8.14 shows the group from the behavior summary corresponding to the PE injection APIs. The first three APIs, `CreateProcessInternal`, `ReadProcessMemory`, and `NtResumeThread`, are the APIs used to perform the PE injection, while the next three APIs, `NtOpenProcess`, `WriteProcessMemory`, and `CreateRemoteThreadEx` are used to perform DLL injection as part of the mimicry attack to hook the newly created process. The last API, `CreateThread`, is called in the newly created process and is not related to an attack, so it can be ignored here. Thus, the behavior summary not only groups together the APIs of the PE injection attack, but it also groups together other APIs that affect the same process like the DLL injection APIs, resulting in many malicious API relative ordering features (refer to Section 6.1 for a description of API relative ordering features). In the API trace, each of these malicious APIs is surrounded by benign APIs, so the behavior summary is crucial to extracting the malicious features from the API trace. When combined with Super Ensembles that constrain any additional benign features, these malicious features have a large impact on the final score, leading to a malicious classification of this sample.

We can confirm the importance of this API sequence by removing it from the sample and graphing the maliciousness scores again (see Figure 8.15), and when we do, we can see

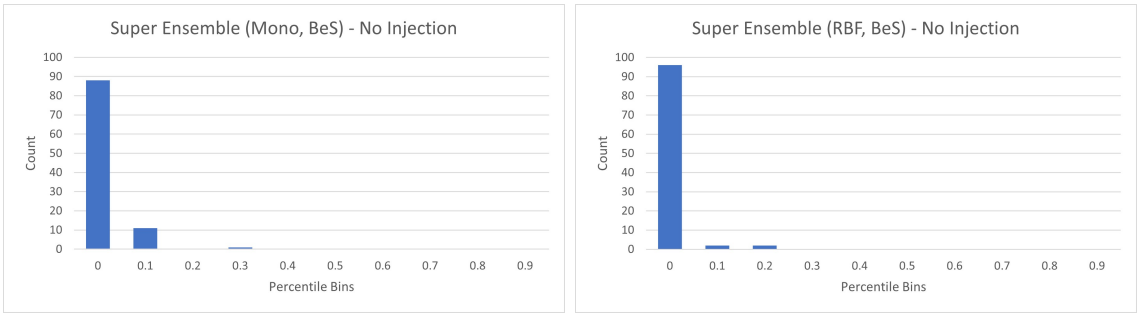


Figure 8.15: Histograms of maliciousness scores produced by various models on the malware sample when the PE injection sequence is removed. The scores produced are overwhelmingly benign scores, which means the PE injection APIs are the most important (and perhaps only) malicious API calls in the sample.

that both Super Ensembles now produce overwhelmingly benign scores, meaning that this sequence is a very important indicator of maliciousness for this sample. Thus, by looking at how mimicry attacks affect this one sequence and how Super Ensembles with the behavior summary can extract the necessary malicious features despite the mimicry attacks, we gain a better understanding of how a Super Ensemble with the behavior summary can output a malicious classification for samples when other methods do not.

8.5 Adversarial Attack Experiments

In addition to mimicry attacks, attackers can also use adversarial attacks, so we evaluate our methods (Super Ensembles with behavior summary) on their robustness to adversarial attacks as well. Because adversarial attacks and defenses have been well studied in the literature, the main goal of this section is to evaluate the robustness of our methods rather than compare with other defense techniques. Recall that we assume the only feasible method of attack is to perform a transfer attack by training a surrogate model (since our models' binary output and binary vector input are not suitable for black-box attacks, see Section 7.3). Also recall from Section 7.3 that we modified the attack algorithm so that input vectors can only be perturbed by changing features from 0 to 1 because our threat model assumes APIs can only be added to API traces in attacks, and the behavior summary and vectorization

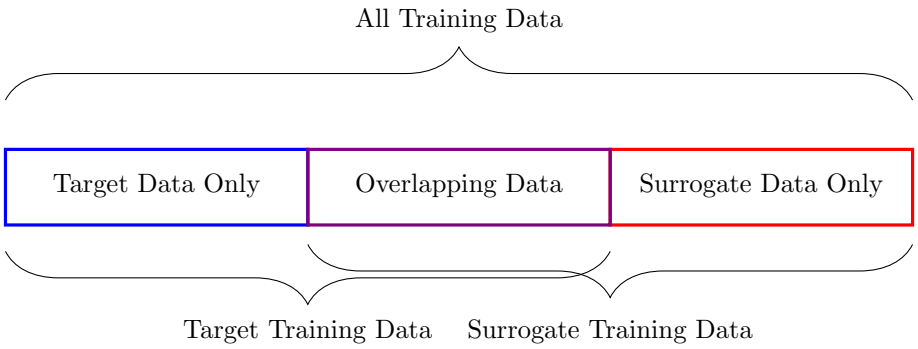


Figure 8.16: Illustration of the division of the data used to train the surrogate models versus the target models. The entire dataset is divided into thirds, and the surrogate models are trained on the first two thirds of the data, while the target models are trained on the last two thirds. Thus, for each set of models, half of their training data is the same as the other set while the other half is different.

algorithms are designed so that adding APIs does not remove features. For simplicity, we also assume that the attacker can generate a corresponding API trace from an input vector (a reasonable assumption if the attacker is aware of the behavior summary and vectorization algorithms), so an attack is successful if a misclassified input vector can be found.

Since it is unreasonable for the attacker to have perfect knowledge of the samples used to train the target model, in these experiments, we simulate an attacker that has good knowledge of the training samples but not the exact samples. Therefore, we split the training data into thirds, and the target models are trained on the first two thirds of the data while the surrogate models are trained on the last two. Thus, half of each set of models' training data are exactly the same, while the remaining half are different but are still from the same distribution. This data division is done before splitting the data for k-fold cross-validation, so the surrogate and target datasets are kept separate for all folds of the same experiment. See Figure 8.16 for an illustration of this data division. To more closely emulate the target model, the labels used to train each surrogate model are the predicted outputs of the target model rather than the true labels of the samples. A new surrogate model is trained for each fold of 10-fold cross validation.

Since adversarial training is a common defense technique against adversarial attacks, we

Algorithm 12 Generating Randomized Adversarial Attack Samples for Training

Input: malicious sample vector - mal_vec
Input: maximum perturbation ratio - $max\_perturb\_ratio$

$max\_perturbed\_features \leftarrow mal\_vec.length \times max\_perturb\_ratio$
$num\_features\_perturbed \leftarrow$ **rand_int**$(0, max\_perturbed\_features)$
$features\_to\_perturb \leftarrow$ **rand_choice**$(\{0, mal\_vec.length\}, num\_features\_perturbed)$
for all $feature \in features\_to\_perturb$ **do**
    $mal\_vec[feature] \leftarrow 1$
end for
return mal_vec

also evaluate the target model after adversarial training. Note that the adversarial training in this section trains against adversarial attacks rather than mimicry attacks like in the previous section. However, like the previous section, we assume the defender is not able to perfectly predict the attack, so we do not train on samples generated with the same adversarial attack algorithm. Instead, we choose a random sample of the malicious training data, and we set a random number of features (up to some maximum) to 1 for each sample. These are the adversarial samples that we add to the training data to train on. We choose 50% of the malicious samples to generate adversarial samples from, and we perturb up to a maximum of 20% of the features in each vector. See Algorithm 12 for the pseudocode of this method. It uses the **rand_int** subroutine to generate a random integer between the two given arguments (inclusive) to determine the number of features to perturb, and it uses the **rand_choice** subroutine to randomly choose $num\_features\_perturbed$ integers from the range denoted by the curly braces {} as the features to perturb. These features in the input sample vector are then set to 1. Overall, this algorithm aims to simulate when the defender generally knows that APIs will be inserted, but not the exact attack.

The results are shown in Table 8.3. The Surrogate, Target and Target (AT) columns show the effectiveness of the adversarial samples on the surrogate model, target model, and target model after adversarial training, respectively. A lower recall indicates a more effective attack because fewer of the adversarial samples are detected, and precision is included to

Table 8.3: Model Performance on Adversarial Attacks

	Surrogate		Target		Target (AT)	
Model	Prec	Rec	Prec	Rec	Prec	Rec
LGBM	0.806	0.125	0.987	0.755	0.990	0.964
SE Mono	0.887	0.331	0.977	0.975	0.977	0.988
SE RBF	0.897	0.361	0.977	0.979	0.976	0.989

show the effect of adversarial training on benign samples. We evaluated the base LGBM as well as our proposed Super Ensembles (using the same N values as the previous section). As we can see, the adversarial attack is highly effective on the surrogate LGBM (around 87.5% effectiveness since only 12.5% of the adversarial samples are classified as malicious), but much less effective on the target model (with a recall of around 75%), and even less so with adversarial training.

The results are similar for the two Super Ensembles. The attack is still effective at creating adversarial samples (67% and 64% of the samples can evade detection for the two surrogate Super Ensembles respectively), albeit less so than LGBM. The attack is even less effective on the target model, which achieves almost as high recall as vanilla data (see Table 8.2 from Section 8.3), and adversarial training further increases recall to as high as vanilla data. First, these results show that the Super Ensemble architecture is more difficult to directly attack than the base LGBM model, which is expected because with two models using different N parameter values, it is more difficult to find a sample that evades both models rather than just one. For example, an adversarial sample effective against the model with no features constrained may not be effective against the model with a large number of features constrained.

Additionally, these results also show that while the attack remains effective against the surrogate model for Super Ensembles, the degree of transferability between the surrogate and target models is lower than that of the base LGBM model. For the LGBM model, Since 87.5% of the samples evade detection in the surrogate model while 24.5% of the samples evade

detection in the target model, 28% (i.e., $\frac{24.5}{87.5}$) of the adversarial samples remain effective when transferred. Using the same percentage, we expect around $18-19\%$ of the adversarial samples to remain effective against the target model for the two Super Ensembles, but we only see 2.5% or less. We see these results even when a large portion (50%) of the training data is shared between the surrogate and target models, and when the surrogate model is trained on the outputs of the target model. Thus, the Super Ensemble architecture provides resistance to both the attack success chance on the surrogate model as well as the chance of a successful transfer between the surrogate and target models. As a result, the Super Ensemble models achieve almost as high recall as in vanilla data against these adversarial attacks, which means only a small number of adversarial samples are effective.

Finally, these results also show that adversarial training, even with very imprecise adversarial samples generated without knowing the attack algorithm, is effective at increasing robustness in the target model. The LGBM model achieves almost as good recall as in vanilla data, while the Super Ensemble models achieve the same recall as in vanilla data. Additionally, for all three models, adversarial training does not seem to affect precision much, and thus the amount of adversarial training we are doing would not likely affect vanilla accuracy to any significant degree while providing a noticeable boost to robustness against adversarial attacks. Overall, the results of these experiments show that the Super Ensemble design is quite robust to adversarial attacks, and even with some imprecise adversarial training, this robustness can be increased even further without negatively affecting precision (and likely accuracy on vanilla data as well).

Chapter 9

Discussion

9.1 Choosing an N Value

In our mimicry attack evaluations, we showed that both removing the top N benign features and enforcing monotonicity on them can achieve good results for both the vanilla dataset and mimicry attacks. Since the features removed or made monotonic are done so starting from the "most benign" features, the remaining features are features that occur the most frequently in malicious samples and least frequently in benign samples. For the N values we used, every remaining feature appears more often in malicious samples than in benign samples. As a result, we have constructed a classifier that determines the maliciousness of a sample based mostly on the existence or absence of certain malicious features. When removing benign features, this is true because these malicious features are the only features remaining after the benign features have been removed, while when enforcing monotonicity, this is true because the final maliciousness score can only be lowered with the absence of these malicious features rather than the presence of benign features. This property in a classifier is needed to detect mimicry attacks because the addition of benign features can be used by the attacker to induce a misclassification, but the existence of malicious features cannot be changed (or else the attack may no longer work). Additionally, when enforcing monotonicity, since the

benign features are not removed, the absence of certain benign features can indicate a more malicious sample. Theoretically, this can lead to some benign samples that lack those benign features to be classified as malicious, thus leading to a higher FPR and lower precision, but we largely do not see this in our results for the N values we used.

Additionally, as mentioned above, choosing the appropriate N can be difficult. Choosing N is more forgiving when enforcing monotonicity because there is no steep drop in recall for large values of N like there is when removing features, but for both strategies the vanilla performance closely matches the performance on more realistic experiments (simulated mimicry attacks using random malware traces and real mimicry attacks, see Figure 8.1) if N is large enough. Note that this is the case only for large N values in a single LGBM model because in a Super Ensemble, the model with $N = 0$ helps maintain good accuracy on vanilla data no matter the N value on the other model. Overall, increasing N generally increases recall at a cost to precision (and FPR), although past a certain point recall also decreases. Thus, the defender should choose the largest N within acceptable FPR limits but before recall drops too steeply on vanilla data, and then use that value in a Super Ensemble. This does not guarantee the best performance for all mimicry attacks, but it will likely result in performance that is close to the performance on vanilla data. This also means that N can be adjusted based on requirements of the defender. Note that because we see this only for simulated mimicry attacks using random malware traces and real mimicry attacks, this is likely only true for mimicry attacks that use similar malicious sequences to ones present in the vanilla malware data, so the defender should strive to acquire as comprehensive a malware dataset as possible to do this analysis.

For this work, the N values that achieved the highest results are values that are slightly above half of the total number of features. As discussed above, when enforcing monotonicity, this means adding benign features cannot be used to lower the maliciousness score. Ideally however, all the features would be made monotonic (i.e., N would be equal to the number of features) because this means no features can be added to a malicious sample to make it more

benign, thus exhibiting guaranteed robustness against not only mimicry attacks but also any attack that adds features to evade detection, including adversarial attacks. We were unable to achieve this because if N values are too high, then both precision and recall are reduced in vanilla data, simulated mimicry attacks using random malware traces, and real mimicry attacks (see Figure 8.2). If all features are made monotonic, then a precision of ~ 0.92 and a recall of ~ 0.89 can be achieved for vanilla data and those two mimicry attacks. We did not think these results are sufficient for a good malware detector, especially the low precision (which leads to a high false positive rate of around $5 - 8\%$), but for some use cases, the benefits of the guarantees may outweigh the drawbacks of the lower performance, e.g., if this is used in conjunction with other detection methods as well as whitelisting of known benign software to keep the false positive rate low. A possible direction for future research may be to develop a feature set (e.g., as a result of a different behavior summary or vectorization algorithm) where good performance can be achieved even if all features are made monotonic.

This work also only used Super Ensembles with two submodels, one with $N = 0$ and the other with N set to a higher value. However, it is also possible to use several more submodels, each with a different N value to optimize for different types of attacks. Again, with our results, the number of false positives would be too high even with a third submodel (especially because one of the models will need a relatively high value of N for good coverage, where precision is the lowest), so we did not pursue this avenue, but if future research can reduce the number of false positives, then this may become viable. Another option may be to change how the Super Ensemble aggregates the results of its submodels; currently, a sample is classified as malicious if any submodel is classified as malicious, but some sort of voting to decide the final classification may reduce the false positive rate. However, this voting system will need to be carefully designed to avoid one submodel classifying the sample as malicious, but it is outvoted by other submodels with N values not optimized for the attack. Nevertheless, this may be another area of fruitful research.

9.2 Other Potential Benefits of the Behavior Summary

The evaluations in this document focuses entirely on classification performance, but another potential benefit of the behavior summary, as the name suggests, is that it can provide a summary of program behavior for engineers or analysts to manually inspect. The behavior summary achieves a balance between the API trace and a feature vector in terms of conciseness and readability; it provides more readable information (API sequences, resources, buffer information, etc.) than a feature vector (i.e., a list of floating point numbers or integers), but it uses much less storage space and is easier to read than an entire API trace, which can be tens or hundreds of thousands of APIs long. We did not do a user study on the readability benefits of the behavior summary, but anecdotally, we found it very useful when looking for specific behaviors in our samples. In particular, when looking for examples to illustrate IAT and EAT hooking as well as an example to do our case study, we were able to search through all of our behavior summaries with a search tool like grep for the specific API sequences that we are looking for. Then, once we have some candidates, it was easy to manually look through the behavior summaries and get a good idea of the program behavior and select samples to further analyze in more detail. In a corporate or industrial setting, if there are many malware samples in a database, then the same steps can be taken to find samples that the exhibit a specific behavior and identify them for further analysis. Even though we did not do a rigorous study to test this, we believe this can be another good use case for behavior summaries, in additional to the detection benefits it offers.

9.3 Performance Costs and Feasibility

In this document, most of the experiments evaluated the effectiveness of our detection pipeline, but another important aspect is its performance costs, so we can determine its feasibility to be implemented in a real system. It is important to note that we wrote our code entirely in Python for ease of development, so an implementation designed for real

deployment would likely use a much faster language like C or C++. The library modules we used, however, are likely written in C++ or assembly, so they are likely to be more similar to a real deployment situation; this applies mostly to the machine learning code (training and classification) due to the machine learning libraries we used.

By far the most costly part of the detection pipeline is generating the behavior summary because the code needs to build trees from and summarize 10's or 100's of thousands of APIs per sample. To quantify this performance cost, we measured how long it took to generate the behavior summary versus how long the sample ran for. This is to evaluate whether a security program monitoring programs (say, on a separate core), can generate behavior summaries fast enough for real-time detection. We calculated how long a sample ran for by subtracting the timestamp of the first API call in the sample from the timestamp of the last call. Note that this is slightly different than the VM duration, which is how long the Cuckoo VM is running for. For 99% of the data, generating the behavior summary took less time than running the sample, and for 98% of the data, generating the behavior summary took less than a quarter of the time it took to run the sample. Furthermore, for 90% of the data, generating the behavior summary took less than 5% of the time it took to run the sample. For the few samples where generating the behavior summary took longer than running the sample, the reason is likely because of the large number of APIs with buffers, so searching them for patterns to assign buffer tags likely takes up most of the time. Thus, a way to speed this up even more (other than using a faster language), is to optimize searching buffers for patterns; perhaps some sort of sampling can be used along with a timeout to stop buffer tags if generating the behavior summary is taking too long. Nevertheless, the vast majority of the samples do not take that long, so real-time monitoring can be feasible.

The other parts of the detection pipeline are trivial performance-wise. Per sample (i.e., program), vectorization of a behavior summary takes around 0.02s, and it takes 0.05s to run around 2000 samples batched through the machine learning model. Thus, it is feasible to run detection several times per the lifetime of a program to do something close to real-time

detection. The limiting factor is generating the behavior summary, so detection can be done as fast as they are generated. Because of the efficiency of LGBM, even training our Super Ensembles take less than 10 seconds for the amount of data we used. Thus, it is quite feasible for a security company to do, say, daily retraining of models to update them, even with much more data. The ability for fast retraining not only allows for quick updates when new threats are discovered, but it also makes it harder for attackers to perform adversarial attacks since models are quickly replaced even if attacks are successful.

The only aspect of the detection method we did not evaluate performance for is capturing the API trace. Because we ran all the samples in the virtual environment of Cuckoo, rerunning them in a virtual environment without capturing APIs does not give a good indication of the performance cost of capturing APIs in a bare metal machine. It it likely that capturing APIs can significantly slow down a process because so many calls are made during its execution. However, this is an issue with all detection systems that monitor API or system calls (including systems with other purposes like intrusion detection), so it is not unique to our detection system. Additionally, some commercial antivirus products do capture APIs client-side, in addition to running more thorough sandbox analyses in the cloud [76], so there is precedence to using APIs in real security products, which indicates that our method can be viable, either on the client or in the cloud. More work is needed to evaluate the efficiency of capturing APIs in real-time.

Overall, these performance results indicate that with more engineering and optimizations, this system is fast enough to be deployed as a real-time detection system. If the detection code is offloaded to a separate CPU core or possibly even a separate server protecting an entire group of machines, then the only task the host machine needs to do is capture the API calls and send them to the detection system. As discussed above, this relies on being able to efficiently capture API calls, so this is an important research direction to making this detection system feasible.

Finally, the work described here is evaluated on software, both benign and malicious,

that were run in a VM with a definitive start and end point. In a real-time monitoring system, the detection system will need to continuously monitor a system for new programs and processes and run detection on them with no indication when they are going to end, so evaluating in a similar experimental setup can give better indication of how a hypothetical real-time system would perform. Other than the speed and efficiency issues discussed above, this environment would also be more challenging as there are so many benign processes that even a low false positive rate can produce many false positives. Turning this working into a real-time detection system will also be a fruitful area of future research.

9.4 Additional Limitations and Further Research

There are some limitations to this work. First, since the goal of this work is to demonstrate the overall effectiveness of a behavior summary with constrained features, some details were determined through manual inspection of the data, like which APIs and resource tags to use, etc. While we believe they should generalize well, it will be useful to evaluate them on different malware datasets and develop more systematic ways of choosing them if they don't generalize well. Similarly, more work can be done to further improve our feature vectors, either removing unnecessary features through more rigorous analysis of feature contributions or adding other potentially useful features, such as a more sophisticated measure of similarity in buffers to determine potential modifications by ransomware. Since we ran Cuckoo without internet access, malware that require an internet access to fully execute are not well represented in our data, so replicating the evaluations on those types of malware can be useful.

Additionally, the APIs and resources used in this work are targeted towards Windows malware, but theoretically these methods can also apply to other operating systems (like Linux), with its own set of APIs (or system calls) and resource tags. It would be useful to evaluate these methods on other operating systems to see how well this method transfers

to those other operating systems. Finally, in this work, all top N benign features calculated using our statistical method are constrained. Perhaps being more discerning on which features to constrain (with, say, an additional metric like the impact each feature has on producing a benign classification) may lead to better results. We leave these ideas open for future research.

Chapter 10

Conclusion

In this document, I have described our work in developing a hierarchical, resource-based behavior summary that, in combination with feature constraining models, has demonstrated robustness against mimicry attacks and adversarial attacks. We summarize the API trace, transform it into a feature vector, and constrain the most benign features to limit the impact of these attacks. Our evaluation shows that this method can more effectively and consistently classify evasive malware using these attacks than other common robustness methods in a variety of scenarios.

As discussed in the previous chapter, there is still more work to be done to use these methods to build a real-time detection system, as well as to expand to other operating systems. However, if it can be done, we believe that this system will be an effective tool against malware, both by itself and in conjunction with other types of detection systems. Signature-based detection systems can detect known threats with a low false positive rate, so our work complements it well by detecting unknown threats. Our detection system should also work well with an anomaly-based detection system because an attacker must mimic benign behaviors to avoid triggering anomaly-based alerts, but mimicry attacks are the attacks our system is designed to detect, so an attacker will need to create samples that evade the complementary detection characteristics of both systems, which makes evasion

much more difficult. Thus, we believe that our system will be a valuable addition in the fight against the proliferation of malware.

Appendix A

APIs Considered

Table A.1 lists the APIs considered in this work, grouped by category.

Table A.1: APIs considered in this work

Category	APIs
File	NtQueryAttributesFile, NtReadFile, NtWriteFile, NtCreateFile, DeleteFileW, DeleteFileA, NtOpenFile, FindFirstFileExW, NtQueryInformationFile, CopyFileA, MoveFileWithProgressW, NtOpenDirectoryObject, NtSetInformationFile, SetFileAttributesW, SetFileAttributesA
Registry	RegSetValueExA, RegSetValueExW, RegOpenKeyExW, RegOpenKeyExA, NtOpenKey, NtOpenKeyEx, RegCreateKeyExA, RegCreateKeyExW, NtQueryValueKey, RegQueryValueExW, RegQueryValueExA
Process/Thread	CreateProcessInternalW, RtlCreateUserProcess, ShellExecuteExW, ReadProcessMemory, WriteProcessMemory, Process32FirstW, Process32NextW, NtQueueApcThread, NtSetContextThread, NtResumeThread, NtOpenProcess, CreateThread, NtCreateThreadEx, CreateRemoteThread, CreateRemoteThreadEx, NtOpenThread
Network	gethostbyname, connect, getaddrinfo, InternetConnectA, InternetConnectW, HttpOpenRequestA, HttpOpenRequestW, HttpSendRequestA, HttpSendRequestW
System	GetComputerNameA, GetUserNameW, GetSystemInfo, GetSystemDirectoryW, GetSystemMetrics, NtQuerySystemInformation
Service	CreateServiceA, CreateServiceW
Windows	FindWindowA, FindWindowW, GetForegroundWindow
Console	SendNotifyMessageA, MessageBoxTimeoutW, WriteConsoleW, WriteConsoleA, DrawTextExA, DrawTextExW, OutputDebugStringA
Hooks	SetWindowsHookExA, SetWindowsHookExW, GetCursorPos, GetKeyState
Mutant	NtCreateMutant, NtOpenMutant
Exception	__exception__

Appendix B

Tags Considered

Table B.1 contains the buffer tags used. The first tag (Is PE) is applied to buffers from individual APIs, while the rest are applied to buffers of the same resource that change between APIs, e.g., for a particular file, there is a change from what is read from it to what is written to it. Table B.2 lists the resource tags used along with example strings.

Table B.1: Buffer tags considered and descriptions of them.

Buffer Tag	Description
Is PE	This tag indicates whether the buffer is a PE file. For Windows, this means it starts with the characters "MZ".
Is PE to Not PE	This tag indicates whether the buffer is a PE file and then changes to not a PE file (does not start with "MZ")
Not PE to Is PE	This tag indicates whether the buffer is not a PE file and then changes to be a PE file.
Contains PE to No PE	This tag indicates whether the buffer contains a PE (contains APIs like LoadLibraryA) and then changes to not contain a PE
No PE to Contains PE	This tag indicates whether the buffer does contain a PE changes to contain a PE
Is ASCII to Not ASCII	This tag indicates whether the buffer first contains only ASCII characters and then changed to contain other characters.
Not ASCII to Is ASCII	This tag indicates whether the buffer first contains non-ASCII characters and then changed to contain only ASCII characters.

Table B.2: Resource tags considered in this work and example strings

Resource Tag	Example Strings
Network	“Internet Settings”, “Network”
Browser	“Chrome”, “Internet Explorer”, “Firefox”
File Explorer	“Explorer”
System	“System32”, “SysWOW64”
Startup	“Startup”, “Winlogon”, “userinit”, “Run”
Service	“services”
System Update	“WindowsUpdate”
System Restore	“SystemRestore”
Crypography	“SystemCertificates”, “Cryptography”
Windows Security	“Security Center”, “WinDefend”
Other Security	“McAfee”, “AVAST”,
Firewall	“Firewall”
Hardware	“Disk”, “HARDWARE”
Temporary	“Temp”
FTP Software	“FileZilla”, “FTP”
Email/Messenging	“Outlook”, “Windows Mail”
Critical Info	“password”, “accounts”, “ProfileList”
Cryptocurrency	“Bitcoin”, “wallet”
Installed Software	“Program Files”
User	“User”, “Users”
Recycle Bin	“$Recycle.Bin”, “RECYCLER”
Command line	“cmd.exe”
Backups/Shadows	“vssadmin”, “wbadmin”
Boot Config	“bcdedit”
Registry Editor	“regedit”
Task Scheduler	“schtasks”
Power Shell	“PowerShell”
WScript	“WScript”
Hidden	“+h”, “/Quiet”, “HideIcons”
Remove/Disable	“Delete”, “Disable”
Trays	”window_Shell_TrayWnd”
Keyboard	“WH_KEYBOARD”
Mouse	”WH_MOUSE”
Connects to IP	“38.124.60.223” (contains only digits and periods)
Dynamic DNS	“no-ip”, “dyndns” (list of known DDNS domains)
Exe Extensions	”exe”, ”dll”, ”vbs”
Suspicious Files	Any with with an extension not from a known set
Suspicious TLD	Any domain excluding “.com”, “.net”, “.org”, and “.us”

Appendix C

Example Summary

Figure C.1 is an example of a group in the behavior summary. It is the 31st group in the summary, and each summary consists of many such groups. The first line contains the group number, and the root resource of that group (the resource that's the ancestor of all other resources in this group) is "C:\\Windows\\".

There are two API sequences in this group, and they are denoted by the star (*) at the beginning of the line. The curly braces after each call indicates the number of repetitions of that call with the left number being the minimum and the right number being the maximum. The number between < | and | > indicate the number of times a failed status occurred for that API sequence, but it is not used in the feature vector. The binary string between < { and } > represent the resource tags of that sequence's resources, and the buffer tags are between < [and] >, separated by ",;". Diffs refers to similarity between different buffers in an API sequence, but it is not used in the feature vector. Lastly, the number of times refers the number of times that particular API sequence has appeared in that group (also not used in the feature vector).

```
Group 31 - {'C:\\Windows\\'}
  * NtOpenFile{1-1} <|0|> <{00010000000000000000000000000000000000000}>
   1 times,
  * NtOpenFile{1-1} -> NtQueryInformationFile{1-1} -> NtReadFile{1-1} <|0|>
   <{00001000000000000000000000000000000000}>
   <[<(0,; 0,; 0,; 0,; 0,; 0,; 0)> diffs: ]> 1 times,
```

Figure C.1: An example group in the behavior summary.

Appendix D

Malicious API Sequence

The malicious API sequence used to simulate PE Injection is:

- `NtCreateFile`, `NtReadFile`, `CreateProcessInternalW`, `NtAllocateVirtualMemory`, `WriteProcessMemory`($\times 5$), `NtSetContextThread`, and `NtResumeThread`.

In this API sequence, an executable is opened (`NtCreateFile`), read (`NtReadFile`) and written to the memory of (`WriteProcessMemory`) a newly created suspended process (`CreateProcessInternalW`). The relevant registers are set (`NtSetContextThread`) and the process is resumed (`NtResumeThread`).

Appendix E

Benign API Sequences Used to Make Real Mimicry Attacks

The following are benign API sequences inserted into malware samples to perform real mimicry attacks in malware. Note that only API calls considered by the detection methods are included here, even though some additional API calls may be made for these API calls to work. Also note that we use the names of the API calls made directly from the software here, so usually the APIs listed here are the Windows APIs rather than the Native APIs that are called by the Windows APIs (e.g., we use `CreateMutex` rather than `NtCreateMutant`). However, sometimes we do make Native API calls directly, so those Native calls are listed here. Finally, these API sequences are not all made with equal probability. The shorter sequences are more likely to be called, so there is a smaller chance of a timeout from inserting too many APIs.

1. `CreateFile`, `OpenFile`, `ReadFileEx`, `GetFileAttributesA`
2. `GetFileAttributesA`
3. `RegGetValue` (this makes multiple open registry and get registry value API calls)
4. `FindWindowA`, `GetForegroundWindow`, `DrawTextEx`, `GetComputerName`, `DrawTextEx`,

DeleteFileA, OpenFile, FindFirstFileEx

5. FindWindowA, DrawTextEx, GetComputerName, CreateMutex, GetComputerName, CreateMutex, OpenMutex, GetKeyState, OpenMutex, GetSystemMetrics

6. GetSystemDirectory, NtQuerySystemInformation, GetUserName, GetSystemMetrics, GetUserName($\times 2$), GetSystemDirectory, GetUserName, GetSystemMetrics,
GetSystemInfo, GetSystemMetrics

7. GetSystemMetrics, GetCursorPos($\times 2$), OutputDebugStringA, GetCursorPos, GetForegroundWindow, GetCursorPos, GetKeyState($\times 2$), GetSystemMetrics($\times 2$)

8. GetSystemMetrics, FindWindowA($\times 2$), GetComputerName, GetSystemMetrics($\times 2$), GetForegroundWindow, GetKeyState, GetForegroundWindow, GetSystemDirectory, FindWindowA, GetForegroundWindow, OutputDebugStringA($\times 2$),
GetSystemDirectory, GetForegroundWindow, GetSystemMetrics

9. FindWindowA, DrawTextEx, SendNotifyMessageA, DrawTextEx, DeleteFileA, DrawText

10. CreateFileW, DeleteFileW, GetFileInformationByHandle

11. FindWindowA, DrawTextEx, GetKeyState, DrawTextEx, GetCursorPos, DrawTextEx

12. CreateFGetKeyStateileW

Bibliography

[1] Daniel Snyder. The very first viruses: Creeper, Wabbit and Brain., May 2010. Section: Malware Info. URL: https://infocarnivore.com/the-very-first-viruses-creeper-wabbit-and-brain/.

[2] Sam Cook. Malware Statistics in 2023: Frequency, impact, cost & more. URL: https://www.comparitech.com/antivirus/malware-statistics-facts/.

[3] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM conference on Computer and communications security*, CCS '02, pages 255–264, New York, NY, USA, November 2002. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/586110.586145, doi:10.1145/586110.586145.

[4] Weiwei Hu and Ying Tan. Black-Box Attacks against RNN based Malware Detection Algorithms, May 2017. URL: https://arxiv.org/abs/1705.08131v1.

[5] Ishai Rosenberg, Asaf Shabtai, Yuval Elovici, and Lior Rokach. Query-Efficient Black-Box Attack Against Sequence-Based Malware Classifiers. In *Annual Computer Security Applications Conference*, ACSAC '20, pages 611–626, New York, NY, USA, December 2020. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/3427228.3427230, doi:10.1145/3427228.3427230.

[6] Fenil Fadadu, Anand Handa, Nitesh Kumar, and Sandeep Kumar Shukla. Evading API Call Sequence Based Malware Classifiers. In Jianying Zhou, Xiapu Luo, Qingni Shen,

and Zhen Xu, editors, *Information and Communications Security*, Lecture Notes in Computer Science, pages 18–33, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-41579-2_2.

[7] Jeff Pracht. Hide and Seek: Evasive Malware, December 2022. URL: https://www.mainstream-tech.com/hide-and-seek-evasive-malware/.

[8] Cross-entropy, August 2023. Page Version ID: 1170369413. URL: https://en.wikipedia.org/w/index.php?title=Cross-entropy&oldid=1170369413.

[9] Gradient boosting, September 2023. Page Version ID: 1174466888. URL: https://en.wikipedia.org/w/index.php?title=Gradient_boosting&oldid=1174466888.

[10] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL: https://papers.nips.cc/paper_files/paper/2017/hash/6449f44a102fde848669bdd9eb6b76fa-Abstract.html.

[11] Precision and recall, May 2021. Page Version ID: 1025279870. URL: https://en.wikipedia.org/w/index.php?title=Precision_and_recall&oldid=1025279870.

[12] Windows API, September 2023. Page Version ID: 1176886591. URL: https://en.wikipedia.org/w/index.php?title=Windows_API&oldid=1176886591.

[13] Windows Native API, June 2023. Page Version ID: 1162354727. URL: https://en.wikipedia.org/w/index.php?title=Windows_Native_API&oldid=1162354727.

[14] Cybersecurity Spotlight – Signature-Based vs Anomaly-Based Detection. URL: https://www.cisecurity.org/spotlight/cybersecurity-spotlight-signature-based-vs-anomaly-based-detection/.

[15] Candid Wueest and Himanshu Anand. Living off the land and fileless attack techniques. *Internet Security Threat Report*, 2017. URL: https://docs.broadcom.com/doc/istr-living-off-the-land-and-fileless-attack-techniques-en.

[16] Davide Canali, Andrea Lanzi, Davide Balzarotti, Christopher Kruegel, Mihai Christodorescu, and Engin Kirda. A quantitative study of accuracy in system call-based malware detection. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 122–132, New York, NY, USA, July 2012. Association for Computing Machinery. doi:10.1145/2338965.2336768.

[17] Fei Xiao, Yi Sun, Donggao Du, Xuelei Li, and Min Luo. A Novel Malware Classification Method Based on Crucial Behavior. *Mathematical Problems in Engineering*, 2020, March 2020. Publisher: Hindawi. URL: https://www.hindawi.com/journals/mpe/2020/6804290/, doi:10.1155/2020/6804290.

[18] Fei Xiao, Zhaowen Lin, Yi Sun, and Yan Ma. Malware Detection Based on Deep Learning of Behavior Graphs. *Mathematical Problems in Engineering*, 2019, February 2019. Publisher: Hindawi. URL: https://www.hindawi.com/journals/mpe/2019/8195395/, doi:10.1155/2019/8195395.

[19] Yi Sun, Ali Kashif Bashir, Usman Tariq, and Fei Xiao. Effective malware detection scheme based on classified behavior graph in IIoT. *Ad Hoc Networks*, 120, September 2021. URL: https://www.sciencedirect.com/science/article/pii/S1570870521001049, doi:10.1016/j.adhoc.2021.102558.

[20] Ömer Aslan, Refik Samet, and Ömer Özgür Tanrıöver. Using a Subtractive Center Behavioral Model to Detect Malware. *Security and Communication Networks*, 2020, February 2020. Publisher: Hindawi. URL: https://www.hindawi.com/journals/scn/2020/7501894/, doi:10.1155/2020/7501894.

[21] Xiaohui Chen, Zhiyu Hao, Lun Li, Lei Cui, Yiran Zhu, Zhenquan Ding, and Yongji Liu. CruParamer: Learning on Parameter-Augmented API Sequences for Malware Detection. *IEEE Transactions on Information Forensics and Security*, 17:788–803, 2022. Conference Name: IEEE Transactions on Information Forensics and Security. URL: https://ieeexplore.ieee.org/document/9715123, doi:10.1109/TIFS.2022.3152360.

[22] Ce Li, Zijun Cheng, He Zhu, Leiqi Wang, Qiujian Lv, Yan Wang, Ning Li, and Degang Sun. DMalNet: Dynamic malware analysis based on API feature engineering and graph learning. *Computers & Security*, 122:102872, November 2022. URL: https://www.sciencedirect.com/science/article/pii/S0167404822002668, doi:10.1016/j.cose.2022.102872.

[23] Lorenzo Maffia, Dario Nisi, Platon Kotzias, Giovanni Lagorio, Simone Aonzo, and Davide Balzarotti. Longitudinal Study of the Prevalence of Malware Evasive Techniques, December 2021. arXiv:2112.11289 [cs]. URL: http://arxiv.org/abs/2112.11289, doi:10.48550/arXiv.2112.11289.

[24] Nicola Galloro, Mario Polino, Michele Carminati, Andrea Continella, and Stefano Zanero. A Systematical and longitudinal study of evasive behaviors in windows malware. *Computers & Security*, 113:102550, February 2022. URL: https://www.sciencedirect.com/science/article/pii/S0167404821003746, doi:10.1016/j.cose.2021.102550.

[25] Kymie Tan, John McHugh, and Kevin Killourhy. Hiding Intrusions: From the Abnormal to the Normal and Beyond. In Fabien A. P. Petitcolas, editor, *Information Hiding*, Lecture Notes in Computer Science, pages 1–17, Berlin, Heidelberg, 2003. Springer. doi:10.1007/3-540-36415-3_1.

[26] Nedim Šrndić and Pavel Laskov. Practical Evasion of a Learning-Based Classifier: A Case Study. In *2014 IEEE Symposium on Security and Privacy*, pages 197–211, May

2014. ISSN: 2375-1207. URL: https://ieeexplore.ieee.org/document/6956565, doi:10.1109/SP.2014.20.

[27] Igino Corona, Davide Maiorca, Davide Ariu, and Giorgio Giacinto. Lux0R: Detection of Malicious PDF-embedded JavaScript code through Discriminant Analysis of API References. In *Proceedings of the 2014 Workshop on Artificial Intelligent and Security Workshop*, AISec '14, pages 47–57, New York, NY, USA, November 2014. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/2666652.2666657, doi:10.1145/2666652.2666657.

[28] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. Adversarial Examples for Malware Detection. In Simon N. Foley, Dieter Gollmann, and Einar Snekkenes, editors, *Computer Security – ESORICS 2017*, Lecture Notes in Computer Science, pages 62–79, Cham, 2017. Springer International Publishing. doi:10.1007/978-3-319-66399-9_4.

[29] Ishai Rosenberg, Asaf Shabtai, Lior Rokach, and Yuval Elovici. Generic Black-Box End-to-End Attack Against State of the Art API Call Based Malware Classifiers. In Michael Bailey, Thorsten Holz, Manolis Stamatogiannakis, and Sotiris Ioannidis, editors, *Research in Attacks, Intrusions, and Defenses*, Lecture Notes in Computer Science, pages 490–510, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-030-00470-5_23.

[30] Chetan Parampalli, R. Sekar, and Rob Johnson. A practical mimicry attack against powerful system-call monitors. In *Proceedings of the 2008 ACM symposium on Information, computer and communications security*, ASIACCS '08, pages 156–167, New York, NY, USA, March 2008. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/1368310.1368334, doi:10.1145/1368310.1368334.

[31] Liang Tong, Bo Li, Chen Hajaj, Chaowei Xiao, Ning Zhang, and Yevgeniy Vorobeychik. Improving Robustness of ML Classifiers against Realizable Evasion Attacks Using Conserved Features. In *Proceedings of the 28th USENIX Security Symposium*, pages 285–302, 2019. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/tong.

[32] Xiao Chen, Chaoran Li, Derui Wang, Sheng Wen, Jun Zhang, Surya Nepal, Yang Xiang, and Kui Ren. Android HIV: A Study of Repackaging Malware for Evading Machine-Learning Detection. *IEEE Transactions on Information Forensics and Security*, 15:987–1001, 2020. Conference Name: IEEE Transactions on Information Forensics and Security. URL: https://ieeexplore.ieee.org/document/8782574, doi:10.1109/TIFS.2019.2932228.

[33] Íñigo Íncer Romeo, Michael Theodorides, Sadia Afroz, and David Wagner. Adversarially Robust Malware Detection Using Monotonic Classification. In *Proceedings of the Fourth ACM International Workshop on Security and Privacy Analytics*, IWSPA '18, pages 54–63, New York, NY, USA, March 2018. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/3180445.3180449, doi:10.1145/3180445.3180449.

[34] Yizheng Chen, Shiqi Wang, Dongdong She, and Suman Jana. On Training Robust PDF Malware Classifiers. In *Proceedings of the 29th USENIX Security Symposium*, pages 2343–2360, 2020. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/chen-yizheng.

[35] Yizheng Chen, Shiqi Wang, Yue Qin, Xiaojing Liao, Suman Jana, and David Wagner. Learning Security Classifiers with Verified Global Robustness Properties. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, pages 477–494, New York, NY, USA, November 2021. Association for

Computing Machinery. URL: https://dl.acm.org/doi/10.1145/3460120.3484776, doi:10.1145/3460120.3484776.

[36] Md Shohidul Islam, Behnam Omidi, and Khaled N. Khasawneh. Monotonic-HMDs: Exploiting Monotonic Features to Defend Against Evasive Malware. In *2021 22nd International Symposium on Quality Electronic Design (ISQED)*, pages 97–102, April 2021. ISSN: 1948-3287. URL: https://ieeexplore.ieee.org/document/9424310, doi:10.1109/ISQED51717.2021.9424310.

[37] Sicco Verwer, Azqa Nadeem, Christian Hammerschmidt, Laurens Bliek, Abdullah Al-Dujaili, and Una-May O'Reilly. The Robust Malware Detection Challenge and Greedy Random Accelerated Multi-Bit Search. In *Proceedings of the 13th ACM Workshop on Artificial Intelligence and Security*, AISec'20, pages 61–70, New York, NY, USA, November 2020. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/3411508.3421374, doi:10.1145/3411508.3421374.

[38] Shirish Singh and Gail Kaiser. Metamorphic Detection of Repackaged Malware. In *2021 IEEE/ACM 6th International Workshop on Metamorphic Testing (MET)*, pages 9–16, June 2021. doi:10.1109/MET52542.2021.00009.

[39] Cuckoo Sandbox - Automated Malware Analysis. URL: https://cuckoosandbox.org/.

[40] Welcome to LightGBM's documentation! — LightGBM 4.0.0.99 documentation. URL: https://lightgbm.readthedocs.io/en/latest/.

[41] Karl-Bridge-Microsoft. PE Format - Win32 apps, March 2023. URL: https://learn.microsoft.com/en-us/windows/win32/debug/pe-format.

[42] dzzie. Understanding the Import Address Table. URL: http://sandsprite.com/CodeStuff/Understanding_imports.html.

[43] Matt Pietrek. Inside Windows: An In-Depth Look into the Win32 Portable Executable File Format, Part 2, October 2002. URL: https://learn.microsoft.com/en-us/archive/msdn-magazine/2002/march/inside-windows-an-in-depth-look-into-the-win32-portable-executable-file-format-part-2.

[44] Russ Osterlund. Windows 2000 Loader: What Goes On Inside Windows 2000: Solving the Mysteries of the Loader, March 2002. URL: https://learn.microsoft.com/en-us/archive/msdn-magazine/2002/march/windows-2000-loader-what-goes-on-inside-windows-2000-solving-the-mysteries-of-the-loader.

[45] stevewhims. Load-Time Dynamic Linking - Win32 apps, January 2021. URL: https://learn.microsoft.com/en-us/windows/win32/dlls/load-time-dynamic-linking.

[46] stevewhims. Run-Time Dynamic Linking - Win32 apps, January 2021. URL: https://learn.microsoft.com/en-us/windows/win32/dlls/run-time-dynamic-linking.

[47] Mega's Computer Land Adventures: Export Address Table Hooking: What, Why, and How, March 2012. URL: http://megamandos.blogspot.com/2012/03/export-address-table-hooking-what-why.html.

[48] SATYAJIT DAULAGUPHU. Understanding Concepts of VA, RVA and File Offsets, October 2019. Section: Malware Analysis. URL: https://tech-zealots.com/malware-analysis/understanding-concepts-of-va-rva-and-offset/.

[49] Dennis A. Babkin. Deep Dive Into Assembly Language - Windows Shellcode - GetProcAddress, September 2020. URL: https://dennisbabkin.com/blog/?t=how-to-implement-getprocaddress-in-shellcode.

[50] Pieter Arntz. Explained: Packer, Crypter, and Protector | Malwarebytes Labs, March 2017. URL: https://www.malwarebytes.com/blog/news/2017/03/explained-packer-crypter-and-protector.

[51] Vadim. 3 Effective DLL Injection Techniques for Setting API Hooks | Apriorit, May 2020. URL: https://www.apriorit.com/dev-blog/679-windows-dll-injection-for-api-hooks.

[52] Ghidra. URL: https://ghidra-sre.org/.

[53] hasherezade. IAT patcher, April 2023. original-date: 2015-01-02. URL: https://github.com/hasherezade/IAT_patcher.

[54] stevewhims, alexbuckgit, jasonepperly, v kents, drewbatgit, DCtheGeek, GrantMeStrength, AlexGuteniev, mijacobs, and msatranjr. DllMain entry point (Process.h) - Win32 apps, September 2022. URL: https://learn.microsoft.com/en-us/windows/win32/dlls/dllmain.

[55] ghorsington. EAT and IAT hook. URL: https://gist.github.com/ghorsington/93ea22c1f4e79e68466a26cbfc58af05.

[56] Jimster480. EAT Hooking On DLL's. URL: https://www.unknowncheats.me/forum/c-and-c-/50426-eat-hooking-dlls.html.

[57] karl-bridge microsoft. CreateRemoteThreadEx function (processthreadsapi.h) - Win32 apps, November 2022. URL: https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createremotethreadex.

[58] Andrew Ilyas, Logan Engstrom, Anish Athalye, and Jessy Lin. Black-box Adversarial Attacks with Limited Queries and Information, July 2018. arXiv:1804.08598 [cs, stat]. URL: http://arxiv.org/abs/1804.08598, doi:10.48550/arXiv.1804.08598.

[59] Eric Balkanski, Harrison Chase, Kojin Oshiba, Alexander Rilee, Yaron Singer, and Richard Wang. Adversarial Attacks on Binary Image Recognition Systems, October 2020. arXiv:2010.11782 [cs]. URL: http://arxiv.org/abs/2010.11782, doi:10.48550/arXiv.2010.11782.

[60] Wieland Brendel, Jonas Rauber, and Matthias Bethge. Decision-Based Adversarial Attacks: Reliable Attacks Against Black-Box Machine Learning Models, February 2018. arXiv:1712.04248 [cs, stat]. URL: http://arxiv.org/abs/1712.04248, doi:10.48550/arXiv.1712.04248.

[61] Nicolas Papernot, Patrick McDaniel, and Ian Goodfellow. Transferability in Machine Learning: from Phenomena to Black-Box Attacks using Adversarial Samples, May 2016. arXiv:1605.07277 [cs]. URL: http://arxiv.org/abs/1605.07277, doi:10.48550/arXiv.1605.07277.

[62] VirusShare.com. URL: https://virusshare.com/.

[63] Software Informer - Windows software downloads and editorial reviews. URL: https://software.informer.com/.

[64] Free Software Downloads and Reviews for Windows, Android, Mac, and iOS – CNET Download. URL: https://download.cnet.com/.

[65] Softpedia - Free Downloads Encyclopedia. URL: https://www.softpedia.com/.

[66] Latest entries - The Portable Freeware Collection. URL: https://www.portablefreeware.com/.

[67] Aaron Walker, Muhammad Faisal Amjad, and Shamik Sengupta. Cuckoo's Malware Threat Scoring and Classification: Friend or Foe? In *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 0678–0684, January 2019. doi:10.1109/CCWC.2019.8666454.

[68] VirusTotal - Home. URL: https://www.virustotal.com/gui/home/upload.

[69] Shuofei Zhu, Jianjun Shi, Limin Yang, Boqin Qin, Ziyi Zhang, Linhai Song, and Gang Wang. Measuring and Modeling the Label Dynamics of Online Anti-Malware Engines.

In *Proceedings of the 29th USENIX Security Symposium*, pages 2361–2378, 2020. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/zhu.

[70] Silvia Sebastián and Juan Caballero. AVClass2: Massive Malware Tag Extraction from AV Labels, October 2020. arXiv:2006.10615 [cs]. URL: http://arxiv.org/abs/2006.10615, doi:10.48550/arXiv.2006.10615.

[71] Emotet Malware | CISA, October 2020. URL: https://www.cisa.gov/news-events/cybersecurity-advisories/aa20-280a.

[72] Malpedia (Fraunhofer FKIE). URL: https://malpedia.caad.fkie.fraunhofer.de/.

[73] Threat Encyclopedia. URL: https://www.trendmicro.com/vinfo/us/threat-encyclopedia/.

[74] Cyberthreats, viruses, and malware - Microsoft Security Intelligence. URL: https://www.microsoft.com/en-us/wdsi/threats.

[75] Process Injection: Portable Executable Injection, Sub-technique T1055.002 - Enterprise | MITRE ATT&CK. URL: https://attack.mitre.org/techniques/T1055/002/.

[76] Randy Treit. Detonating a bad rabbit: Windows Defender Antivirus and layered machine learning defenses, December 2017. URL: https://www.microsoft.com/en-us/security/blog/2017/12/11/detonating-a-bad-rabbit-windows-defender-antivirus-and-layered-machine-learning-defenses/.