DeepReflect: Discovering Malicious Functionality through Binary Reconstruction

Evan Downing
Georgia Institute of Technology

Yisroel Mirsky*
Georgia Institute of Technology & Ben-Gurion University

Kyuhong Park*
Georgia Institute of Technology

Wenke Lee
Georgia Institute of Technology

Abstract

Deep learning has continued to show promising results for malware classification. However, to identify key malicious behaviors, malware analysts are still tasked with reverse engineering *unknown* malware binaries using static analysis tools, which can take hours. Although machine learning can be used to help identify important parts of a binary, supervised approaches are impractical due to the expense of acquiring a sufficiently large labeled dataset.

To increase the productivity of static (or manual) reverse engineering, we propose DEEPREFLECT: a tool for localizing and identifying malware components within a malicious binary. To localize malware components, we use an unsupervised deep neural network in a novel way, and classify the components through a semi-supervised cluster analysis, where analysts incrementally provide labels during their daily work flow. The tool is practical since it requires no data labeling to train the localization model, and minimal/noninvasive labeling to train the classifier incrementally.

In our evaluation with five malware analysts on over 26k malware samples, we found that DEEPREFLECT reduces the number of functions that an analyst needs to reverse engineer by 85% on average. Our approach also detects 80% of the malware components compared to 43% when using a signature-based tool (CAPA). Furthermore, DEEPREFLECT performs better with our proposed autoencoder than SHAP (an AI explanation tool). This is significant because SHAP, a state-of-the-art method, requires a labeled dataset and autoencoders do not.

1 Introduction

Reverse engineering malware statically can be a manual and tedious process. Companies can receive up to 5 million portable executable (PE) samples per week [13]. While most organizations triage these samples ahead of time to reduce the amount of malware to analyze (i.e., checking VirusTotal [12] for antivirus (AV) engine results, executing the sample in a controlled sandbox, extracting static and dynamic signatures, etc.), at the end of the day there will still be malware samples which require static reverse engineering. This is due to the fact that there will always be new malware samples which no antivirus company has analyzed before or no signature which has been crafted to identify these new samples. Finally, there is a possibility that the sample will refuse to execute within the analyst's dynamic sandbox [42].

Current solutions exist in the form of creating signatures [33, 45, 72], classification [14, 30, 36, 41], and clustering [18, 25, 52] for malware samples. However, these solutions only predict the class of the samples (e.g., benign vs. malicious, or a particular malware family). They cannot localize or explain the behaviors within the malware sample itself, which an analyst needs to perform to develop a report and improve their company's malware detection product. In fact, there has been burnout reported in the field due to excessive amounts of workload [27, 55].

To identify their needs, we consulted with four reverse engineer malware analysts (one from an AV company and three from the government sector). We found that malware analysts would be more productive in their work if they had a tool which could (1) identify where malicious functionalities are in a malware and (2) label those functionalities. The challenges in developing such a tool are that (1) one would need to be able to distinguish between what is benign and what is malicious and (2) understand the semantics of the identified malicious behaviors. For the first challenge, distinguishing between what is benign and what is malicious is difficult because the behaviors of malware and benign software often overlap at a high level. For the second challenge, automatically labeling and verifying these behaviors is difficult because there are no published datasets of individually labeled malware functions (unlike malware detection and classification systems which use open datasets like antivirus labels).

To solve these challenges we developed DEEPREFLECT, a novel tool which uses (1) an unsupervised deep learning model which can locate malicious functions in a binary and (2) a semi-supervised clustering model which classifies the identified functions using very few labels obtained from

*These authors are co-2nd authors.

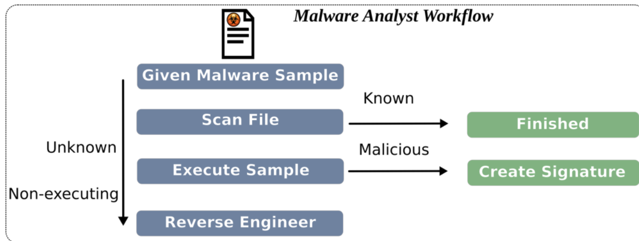

Figure 1: The general workflow of a malware analyst. DEEPREFLECT assists the analyst when they must statically reverse engineer an unknown malware sample.

analyst's regular daily workflow.

To locate the malware components in a binary, we use an autoencoder (AE). An AE is a neural network based machine learning model whose task is to reconstruct its input as its output. Since there is compression in the network's inner layers, the AE is forced to learn key concepts in the training distribution. Our intuition is that if we train the AE on benign binaries, it will have difficulty reconstructing malicious binaries (i.e., the samples we did not train it on). Naturally, the AE will not be able to reconstruct regions of the binary which contain malicious behaviors (which are unseen or rare in benign samples). Thus, the reconstruction errors can be used to identify the malicious components in a malware. Additionally, since AEs are trained in an unsupervised manner, we do not need millions of labeled samples and companies can utilize their own internal datasets of malware binaries.

To classify the located malware components, we (1) perform clustering on all of the identified functions in the malware samples and (2) label clusters using the analyst's annotations made during his or her regular daily workflow. This approach is semi-supervised since only a few labels (e.g., three) are needed per cluster to assign the majority label to the entire cluster. Over time, we can predict the class (e.g., C&C, privilege escalation, etc.) of functions identified by the AE by mapping them to the clustering model. This, in turn, saves the analyst time as they are not forced to reverse engineer the same code again and again.

We note that the unsupervised AE provides immediate utility to malware analysts without training or using the semi-supervised clustering model. This is because it (1) draws the attention of the analyst to the most relevant functions by ranking them (by their reconstruction error) and (2) filters out functions which would have cost the analyst hours or potentially days to interpret.

DEEPREFLECT was designed and revised with feedback from our four malware analysts. Then five different malware analysts were recruited to evaluate DEEPREFLECT's effectiveness and utility. Overall, we evaluate the tool's performance on (1) identification of malicious activities within a malware, (2) clustering related malware components, (3) focusing the analyst's attention to what is important, (4) revealing insights into shared behaviors between different malware families, and (5) handling adversarial attacks involving obfuscation.

Our contributions are as follows:

- A novel tool which can help malware analysts by automatically (1) locating and identifying malicious behaviors within static malware samples and (2) deriving insights by associating functionality relationships between different malware families.
- A novel and practical approach for using machine learning on static analysis where
  1. Training is performed in an unsupervised manner: *an expert does not need to label any samples for the system to yield utility – highlighting the malware's components*; and
  2. Classification is accomplished in a semi-supervised manner with minimal intervention: *annotations from the analyst's regular workflow are used as labels and the majority label in a cluster is used to classify associated malware components*.
- We propose an approach for localizing important parts of a malware with an explanation framework (such our proposed AE or SHAP [40]) by using localized features that can be mapped back to the original binary or control flow graph.

2 Scope & Overview

In this section, we present a motivating scenario and explain the threat model and goals of our system.

2.1 Motivation

As a motivating example, let us assume there exists a malware analyst named Molly. An illustration of her daily workflow can be found in Figure 1. This general workflow is realistic based on descriptions in recent work [69] and of our own discussions with real-world malware analysts. Given a malware sample, Molly is tasked with understanding what the sample does so that she can write a technical report as well as improve her company's current detection system to identify that sample in the future.

She first queries VirusTotal [12] and other organizations to determine if they have seen this particular sample before. Unfortunately, no one has. Thus, she moves onto her next step which is to execute it in a custom sandbox to get an overview the sample's dynamic behaviors. Unfortunately, the sample does not display any malicious or notable behaviors – it is also possible that it has detected the environment and refuses to execute. She runs a few in-house tools to try to coax the malware into performing its hidden behaviors, but to no avail. Exhausting these options, she resorts to unpacking and statically reverse engineering the sample to understand what its potential behaviors are.

Upon opening the unpacked sample in a disassembler (such as IDA Pro [7] or BinaryNinja [1]), Molly is overwhelmed by the thousands of functions that exist within it. She tries

running various static signature detection tools to identify some specific malicious components of the malware, but again to no avail. She must look through each function one-by-one (possibly filtering them by the API calls and strings which exist within them) to try to understand their behaviors (often times resorting to debugging to verify observed behaviors).

After noting its behaviors, she writes her report (composed of basic information like indicators of compromise (IOCs), static signatures, etc.) and passes it along to her superiors. The next day, she repeats the same tasks. Due to this repetitive manual labor, the job becomes tedious and time-consuming for Molly.

DEEPREFLECT aims to alleviate her laborious task by automatically narrowing her focus to the functions which are most likely malicious (out of the thousands she is presented with) and provide labels to those functions she has seen similarly in the past.

2.2 Proposed Solution

We propose DEEPREFLECT, a tool which (1) locates malicious functions within a malware binary and (2) describes the behaviors of those functions. While an analyst may first attempt to identify behaviors statically by searching for specific strings and API calls [69], these can be easily obfuscated or hidden from the analyst. DEEPREFLECT makes no such assumption and seeks to identify these same behaviors through a combination of control-flow graph (CFG) features and API calls.

DEEPREFLECT works by learning what benign binary functionalities look like normally. Thus, any abnormalities would suggest that these functionalities do not appear in benign binaries and could be used to facilitate malicious behaviors. This allows our tool to narrow down the analyst's search space before they open or scan the binary. DEEPREFLECT reduced the number of functions the analyst had to examine (in each malware sample) by 85% on average as shown in Figure 5, illustrating the amount of work required for them to accomplish their task. Additionally, we show that our methodology outperforms signature-based techniques which aim to accomplish the same goal §4.3.

2.3 Threat Model

We assume the malware analyst is performing static analysis. The limitations of static analysis have been discussed in prior work [44]. We do not address dynamic analysis in this paper, though conceptually our tool can be extended to work with dynamic analysis data. We assume the malware given to our system is unpacked, as is similar to prior work [37, 39, 59, 60].

The problem of unpacking has been studied in prior work and solutions have been proposed to address it [21, 58]. Our results are directly dependent on malware being unpacked and thus we rely on prior work [11] to first unpack the binaries for us. We emphasize that our tool is just one step in the analyst's pipeline, and unpacking is the first step as illustrated in Figure 1 and Figure 2.

We assume we can reliably disassemble the malware in order to extract basic blocks and functions. The challenges of accurately disassembling binaries have been discussed in prior work [15, 38].

For our experimentation, we trust that our machine learning models and datasets are reliable (i.e., are not actively attempting to attack or thwart our system). A discussion of the limitations of this assumption (and its solutions) in deployment settings can be found in §5.1.

2.4 Research Goals

As discussed in §1 and §2.1, the analyst needs to locate and describe behaviors of internal functions within malware samples. Therefore, DEEPREFLECT has four primary goals: (*G1*) Accurately identify malicious activities within malware samples, (*G2*) Focus the attention of the analyst when statically analyzing malware samples, (*G3*) Handle new (unseen) malware families, and (*G4*) Give insights into malware family relationships and trends.

3 Design

In this section, we detail the pipeline of DEEPREFLECT as well as the features and models it uses.

3.1 Overview

The goal of DEEPREFLECT is to identify malicious functions within a malware binary. In practice, it identifies functions which are *likely* to be malicious by locating abnormal basic blocks (*regions of interest* – RoI). The analyst must then determine if these functions exhibit malicious or benign behaviors. There are two primary steps in our pipeline, illustrated in Figure 2: (1) RoI detection and (2) RoI annotation. RoI detection is performed using an autoencoder, while annotation is performed by clustering all of the RoIs per function and labeling those clusters.

Terminology. First, we define what we mean by "malicious behaviors." We generate our ground-truth based on identifying core components of our malware's source code (e.g., denial-of-service function, spam function, keylogger function, command-and-control (C&C) function, exploiting remote services, etc.). These are easily described by the MITRE ATT&CK framework [9], which aims to standardize these terminologies and descriptions of behaviors. However, when statically reverse engineering our evaluation malware binaries (i.e., in-the-wild malware binaries), we sometimes cannot for-certain attribute the observed low-level functions to these higher-level descriptions. For example, malware may modify registry keys for a number of different reasons (many of which can be described by MITRE), but sometimes determining which registry key is modified for what reason is difficult and thus can only be labeled loosely as "Defense Evasion: Modify

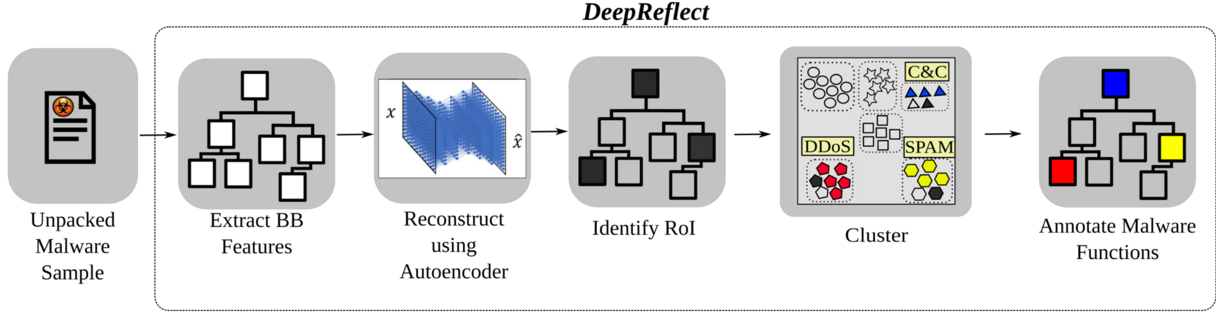

Figure 2: Overview of DEEPREFLECT. Our system takes unpacked malware samples as an input, extracts CFG features from each input (basic block (BB)), applies them to a pretrained autoencoder model to highlight RoI (*regions of interest*). Finally, it clusters and labels these regions.

Registry" in MITRE. Even modern tools like CAPA [3] identify these types of vague labels as well. Thus in our evaluation, we denote "malicious behaviors" as functions which can be described by the MITRE framework.

RoI Detection. The goal of detection is to automatically identify malicious regions within a malware binary. For example, we would like to detect the location of the C&C logic rather than detect the specific components of that logic (e.g, the network API calls `connect()`, `send()`, and `recv()`). The advantage of RoI detection is that an analyst can be quickly pointed to specific regions of code responsible for launching and operating its malicious actions. Prior work only focuses on creating *ad hoc* signatures that simply identify a binary as malware or some capability based on API calls alone. This is particularly helpful for analysts scaling their work (i.e., not relying on manual reverse engineering and domain expertise alone).

RoI Annotation. The goal of annotation is to automatically label the behavior of the functions containing the RoIs. In other words, this portion of our pipeline identifies *what* this malicious functionality is doing. Making this labeling nonintrusive to an analyst's workflow and scalable is crucial. The initial work performed by an analyst for labeling clusters is a long-tail distribution. That is, there is relatively significant work upfront but less work as they continue to label each cluster. The advantage of this process is simple: it gives the analyst a way to automatically generate reports and insights about an unseen sample. For example, if a variant of a malware sample contains similar logic as prior malware samples (but looks different enough to an analyst to be unfamiliar), our tool gives them a way to realize this more quickly.

3.2 RoI Detection

An autoencoder is a neural network M which consists of an encoder $En(x)$, which compresses the input x into an encoding e, and a decoder $De(e)$, which reconstructs x from a given e. When trained with the objective $De(En(x)) = x$, the network learns to summarize the distribution of $x \in X$ where $X \subset \mathbb{R}^m$. In works such as [43], it has been shown that autoencoders can detect malicious (abnormal) behaviors when trained on a benign distribution. This is because M would fail to reconstruct the features in x because m would recall the malicious concepts/patterns.

Given a sample's reconstruction $M(x) = \hat{x}$, a malicious sample is typically identified by computing the mean-squared-error (MSE) and checking if the resulting scalar is above a given threshold ϕ. The MSE is calculated as

$$MSE(x,\hat{x}) = \frac{1}{m}\sum\left(x^{(i)} - \hat{x}^{(i)}\right)^2 \quad (1)$$

where $x^{(i)}$ is the i-th feature in x.

Our assumption is that malware binaries will contain similar, but unique functionalities compared to benign binaries. Given this intuition, we train M on a diverse benign dataset which represents a variety of behaviors and functionalities. In contrast to previous works, which identify an *entire* sample as being malicious, we identify the malicious *regions* in each sample. Concretely, we compute the localized MSE defined as

$$LMSE(x,\hat{x}) = \left(x^{(i)} - \hat{x}^{(i)}\right)^2 \quad (2)$$

and then apply a threshold ϕ to the resulting vector to identify the patterns which M did not recognize or understand. Each block which received a squared error over ϕ is called a region of interest (RoI). We denote the mapped set of RoIs identified in sample x as the set

$$R_x = \left\{ x^{(i)} \middle| \left(x^{(i)} - \hat{x}^{(i)}\right)^2 \geq \phi \right\} \quad (3)$$

The highlights represented by R_x are similar to SHAP [40] explanations of supervised classifiers (e.g., image classification). However, our approach is designed to explain unsupervised neural network anomaly detectors (i.e., trained on *unlabeled* datasets), whereas SHAP is used on supervised classification models (trained on *labeled* datasets).

3.2.1 Features

When given a binary sample, we extract features to summarize the samples as x. There are many static features which have been used in prior work for malware detection (e.g., code section entropy, imported API calls, etc.) [29, 35, 53, 61, 63]. However, for M to localize malicious behaviors within a binary,

our features must be mapped 1-to-1 back into the original sample. Therefore, we represent each binary as an m-by-c matrix which captures the first m basic blocks using c features to summarize each of their activities. Basic blocks are, in general, a series of instructions which end in a control transfer instruction. Of course, basic blocks may be represented differently depending on the disassembler, so this strict definition may not apply to all static malware analysis systems.

Our c features were inspired from those found in prior works, namely attributed control flow graph (ACFG) features [23, 75]. ACFG features were chosen to perform binary similarity in these works because they assume these features (made up of structural and numerical CFG features) will be consistent across multiple platforms and compilers. While an argument can be made that our goals are similar (i.e., identifying similarities and differences across binaries), we tailored these features specifically for studying malware. In particular, we chose our features for the autoencoder to use in order to capture higher-level behaviors. Our features consist of counts of instruction types within each basic block (a more detailed form of those extracted for ACFG features), structural features of the CFG, and categories of API calls (which have been used to summarize malware program behaviors [18]).

In DEEPREFLECT, we set m to be the first 20k basic blocks. We chose this because 95% of our dataset samples have 20k basic blocks or less. We set c to be the 18 features which summarize each basic block as follows:

Structural Characteristics. The structural features we use are the number of offspring and betweenness score of each basic block. These characteristics can represent a control-flow structure commonly used for operations like network communication (e.g., connect, send, recv) and file encryption (e.g., findfile, open, read, encrypt, write, close). An example of this functionality from an actual malware sample can be found in Figure 6.

Arithmetic Instructions. The arithmetic instruction features we use are the number of "basic math", "logic operation", and "bit shifting" instructions contained within each basic block. The features can be used to represent how mathematical operations are carried out for higher level behaviors. They illustrate how numbers are interacted with for the function (e.g., encryption functions likely include lots of xor instructions, obfuscation functions likely include a combination of logic and bit-shifting operations, etc.). We retrieved these instructions from the Intel architectures software developer's manual [26]. Additionally, we provide an example from a malware sample showcasing these types of features in Figure 9.

Transfer Instructions. The transfer instruction features we use are the number of "stack operation", "register operation", and "port operation" instructions within each basic block. The features can be used to represent how transfer operations are carried out for higher level behaviors. They illustrate how arguments provided to the function (and returned values from function calls) interact with the rest of the data within that function. It can be indicative of complex logic and data manipulation (e.g., deobfuscation/decryption will likely involve more move-related instructions and C&C logic will involve more stack-related instructions as it calls more internal/external functions). We similarly retrieved these instructions from the Intel architectures software developer's manual [26].

API Call Categories. The API call features we use are the number of "filesystem", "registry", "network", "DLL", "object", "process", "service", "synchronization", "system information", and "time" related API calls within each basic block. These categories are inspired from prior work for malware clustering [18]. The features can be used to represent high level library operations needed to perform malicious activities such as network communications and filesystem, registry, and process operations. Since these directly represent high-level behaviors, they are crucial to understanding the overall behaviors of a function. Examples of malware functions which utilize these different call types to perform different behaviors can be found in Figure 6 and Figure 8.

We argue that these features are better suited for malware than classical ACFG features because (1) they include API calls which have been used in prior work for malware detection, (2) the instruction categories are finer-grained, allowing for more context into each basic block (as previously described), and (3) they do not rely on strings which are too easily prone to evasion attacks [77]. Of course, given a motivated adversary, any machine learning model can be attacked and tricked into producing an incorrect and unintended outputs. Whilst our features and model are not an exception to this, we argue that they suffice to produce a reliable model (i.e., it behaves as expected) and make it difficult enough such that an adversary would have to work extensively to produce a misleading input (as demonstrated in §4.7). For a discussion of potential attacks against our system, please refer to §5.

3.2.2 Model

To train M, we create a training set X from a variety of benign binaries, where $x \in X$ is an m-by-c feature vector representing one of the binaries. For the autoencoder model architecture, we use a U-Net [57]. U-Nets have been shown to perform well on generative image tasks such as biomedical image segmentation and the creation of fake imagery. The advantage of using a U-Net is that it has skip connections between the En and De which M can use to skip the compression of certain features to retain a higher fidelity in $\hat{x}$.

We train M on X with the goal of minimizing the reconstruction loss. The loss is the common L2 loss between the input and output, and is defines as

$$\mathcal{L}_2(x,\hat{x}) = \sum (x-\hat{x})^2 \tag{4}$$

Once trained, M is given the static features x of an unseen

malware sample. We then highlight the potentially malicious code regions using Equation 2, which is further discussed later in §4, such that any MSE over that value is considered a RoI. After highlighting the RoIs (basic blocks), we cluster the functions they belong to.

3.3 RoI Annotation

Given a new sample x, we want to identify the behavior (category) of each of its functions[1] and report it to Molly. Since it is not practical to label *all* functions, we annotate only a *few* functions and propagate the results using cluster analysis. We will now explain how this process is setup prior to receiving Molly's sample.

3.3.1 Clustering Features

Let x be a feature extracted binary taken from a collection of unpacked malwares. Let F be the set of functions in x found using BinaryNinja. For each $f_i \in F$ we denote the RoIs in f_i as q_i, where $q_i \subset R_x$.

We create a training set D for clustering as follows: Given the malware x_i, For each $q_i \neq \varnothing$, we summarize the behavior of f_i as $\frac{1}{|q_i|}\sum q_i$ and add it to D. This is repeated for all malwares in our collection.

Experimentally, we found that this representation of f_i's RoIs best capture the functions' behaviors in terms of cluster quality (i.e., using Silhouette Coefficient & Davies Bouldin Score).

3.3.2 Clustering Model

To cluster the functions in D, we first reduce the dimensionality from 18 to 5 so that we can scale to 500k functions. The reduction is performed using principle component analysis (PCA).

Next, we cluster the reduced vectors using HDBSCAN [6] and denote the clustering of D as C. HDBSCAN is a variant of the density based clustering algorithm DBSCAN. The reason we chose HDBSCAN is because (1) it can identify non-convex clusters (unlike k-means) and (2) it automatically selects the optimal hyper-parameters for cluster density (unlike classic DBSCAN).

3.4 Deployment

Next, we describe how DEEPREFLECT is deployed and used by a malware analyst.

Initialization. To initialize DEEPREFLECT, Molly begins by unpacking benign and malware binaries. She then passes them to DEEPREFLECT which (1) extracts our static features, (2) trains an autoencoder model M on the benign samples, (3) extracts RoIs R_x from each malware sample, (4) summarizes each function's behavior by averaging their RoIs (q_i) as D, and (5) reduces the summaries with PCA and clusters them as C.

[1]The functions in a binary are heuristically and statically found using a tool such as BinaryNinja on the CFG.

At this point, Molly has now identified groups of behaviors (functions) which are malicious (anomalous) according to M. She can now annotate a small subset of the functions or proceed with her regular work routine while adding annotations to D (as mentioned earlier).

Execution. When Molly receives a new sample x, the behaviors are automatically visualized, localized, and labeled for her by DEEPREFLECT as follows: (1) x is unpacked using unipacker [11], (2) x is passed through M and the RoIs R_x are obtained, (3) functions are identified using BinaryNinja and each function is summarized as q by averaging its RoIs, (4) the remaining function summaries are reduced using the PCA model, (5) each function is associated with the cluster that is most similar to it,[2] and (6) assign the majority cluster annotations to the functions and map the result back to Molly's user interface. This workflow is illustrated in Figure 2.

Molly then investigates the highlighted functions, and while doing so she (1) obtains a better perspective on what the malware is doing, (2) annotates any function labeled "unknown" with the corresponding MITRE category (dynamically updating D), and (3) is able to observe shared relationships between other malware samples and families by their shared clusters.

4 Evaluation

In this section, we present our evaluation of DEEPREFLECT. First, we outline our objectives for each evaluation experiment and list which research goals (§2.4) are achieved by the experiment. We evaluate DEEPREFLECT's (1) reliability by running it on three real-world malware samples we compiled and compared it to a machine learning classifier, a signature-based solution, and a function similarity tool, (2) cohesiveness by tasking malware analysts to randomly sample and label functions identified in in-the-wild samples and compare how DEEPREFLECT clustered these functions together, (3) focus by computing the number of functions an analyst has to reverse engineer given an entire malware binary, (4) insight by observing different malware families sharing the same functionality and how DEEPREFLECT handles new incoming malware families, and (5) robustness by obfuscating and modifying a malware's source code to attempt to evade DEEPREFLECT.

4.1 Dataset

Constructing a good benign dataset is crucial to our model's performance. If we do not provide enough diverse behaviors of benign binaries, then everything within the malware binary will appear as unfamiliar. For example, if we do not train the autoencoder on binaries which perform network activities, then *any* network behaviors will be highlighted.

To collect our benign dataset, we crawled CNET [4] in 2018 for Portable Executable (PE) and Microsoft Installer (MSI)

[2]This can be done by measuring centroid distance, using an incremental DBSCAN, or by reclustering D (which is what we do in this paper).

Category	Size	Category	Size
Drivers	6,123	Business Software	1,692
Games	1,567	Utilities	1,453
Education	1,244	Developer Tools	1,208
Audio	1,023	Security	1,000
Communications	994	Design	844
Digital Photo	826	Video	787
Customization	778	Productivity	730
Desktop Enhancements	699	Internet	695
Networking	612	Browsers	440
Home	390	Entertainment	257
Itunes	43	Travel	17

Table 1: Benign Dataset: 22 categories from CNET.

Label	virut	vobfus	hematite	sality	crytex
Size	3,438	3,272	2,349	1,313	914
Label	wapomi	hworld	pykspa	allaple	startsurf
Size	880	720	675	470	446

Table 2: Malware Dataset: Top 10 most populous families.

files from 22 different categories as defined by CNET to ensure a diversity of types of benign files. We collected a total of 60,261 binaries. After labeling our dataset, we ran our samples through Unipacker [11], a tool to extract unpacked executables. Though not complete as compared to prior work [21, 58], the tool produces a valid executable if it was successful (i.e., the malware sample was packed using one of several techniques Unipacker is designed to unpack). Since Unipacker covers most of the popular packers used by malware [67], it is reasonable to use this tool on our dataset. By default, if Unipacker cannot unpack a file successfully, it will not produce an output. Unipacker was able to unpack 34,929 samples. However, even after unpacking we found a few samples which still seemed partially packed or not complete (e.g., missing import symbols). We further filtered PE files which did not have a valid start address and whose import table size was zero (i.e., were likely not unpacked properly). We also deduplicated the unpacked binaries. Uniqueness was determined by taking the SHA-256 hash value of the contents of each file. To improve the quality of our dataset, we only accepted benign samples which were classified as malicious by less than three antivirus companies (according to VirusTotal). In total, after filtering, we obtained 23,307 unique samples. The sizes of each category can be found in Table 1.

To acquire our malicious dataset, we gathered 64,245 malware PE files from VirusTotal [12] during 2018. We then ran these samples through AVClass [62] to retrieve malware family labels. Similar to the benign samples, we unpacked, deduplicated, and filtered samples. Unipacker was able to unpack 47,878 samples. In total, we were left with 36,396 unique PE files from 4,407 families (3,301 of which were singleton families – i.e., only one sample belonged to that family). The sizes of the top-10 most populous families can be found in Table 2.

After collecting our datasets, we extracted our features from each sample using BinaryNinja, an industry-standard binary disassembler, and ordered each feature vector according to its basic block's address location in a sample's binary.

4.2 Model Setup

After extracting our datasets, we trained the autoencoder on 80% of our benign dataset and tested it on the remaining 20%. We used a kernel size of 24 with a stride of 1 and normalized the feature vectors; we found these parameters to improve results empirically. We trained the model for a maximum of 10 epochs and we obtained a training MSE of 2.5090e-07 and testing MSE of 2.1575e-07 – recall that a lower the MSE value means a better reconstruction of the benign samples. It took roughly 40 hours to train the model on an NVIDIA GeForce RTX 2080 Ti GPU. [3]

4.3 Evaluation 1 – Reliability

To evaluate DEEPREFLECT's reliability, we explore and contrast the models' performance in localizing the malware components within binaries.

4.3.1 Baseline Models

To evaluate the localization capability of DEEPREFLECT's autoencoder, we compare it to a general method and domain specific method for localizing concepts in samples: (1) SHAP, a classification model explanation tool [40], (2) CAPA [3], a signature-based tool by FireEye for identifying malicious behaviors within binaries,[4] and (3) FunctionSimSearch [5], a function similarity tool.

Given a trained classifier and the sample x, SHAP provides each feature $x^{(i)}$ in x a contribution score for the classifier's prediction. For SHAP's model, we trained a modified deep neural network VGG19 [64] to predict a sample's malware family and whether the sample is benign. For this model, we could not use our features because the model would not converge. Instead, we used the classic ACFG features without the string or integer features. We call these features *attributed basic block* (ABB) features. We trained this model for classification (on both malicious and benign samples) and achieved a training accuracy of 90.03% and a testing accuracy of 83.91%. In addition to SHAP, we trained another autoencoder on ABB features to compare to our new features as explained in §3.2.1.

4.3.2 Ground-Truth Dataset

For our ground-truth, we statically identified the locations of the malicious components (functions) in the source code of three different malwares. We located these functions in the binary's CFG by matching markers (e.g., strings and API calls) and labeling the corresponding basic blocks as malicious. All other blocks we labeled as benign. We note that we were unable to locate 14% to 30% of the malicious functions

[3]For reproducibility, our source-code and dataset can be found at https://github.com/evandowning/deepreflect.

[4]We used the community and expert rule sets v1.2.0 from https://github.com/fireeye/capa-rules

(depending on the sample), so they were marked as benign. These functions were not found because (1) the functions could not be recognized due to obscured and partial identifiers (calls and strings) in the binary, and (2) they were lost due to a limitation of function identification from a static disassembler such as dynamically resolved functions and anti-static analysis techniques [16]. Note, the omitted functions are reflected in the results as false positives (FPs) (Figure 3) so technically our false positive rate (FPR) is better in reality.

The three malware samples which make up our ground-truth are `rbot`, `pegasus`, and `carbanak`. We chose `rbot` because while it is an older internet relay chat (IRC) botnet from 2004, it still exists in common malware feeds – i.e., it still appears in the wild. We also chose it because it compiles into a single PE file (directly comparable to our PE malware samples from our dataset). We chose `pegasus` because it is a newer banking trojan from 2016 and is composed of multiple payloads (PE files and DLL files). This allows us to evaluate our tool on files which could be captured in memory or elsewhere (i.e., not just assuming that all malware will neatly pack all of its behaviors into a single file). Finally, we chose `carbanak` because it is a recently leaked banking malware from 2014, making it still relatively modern. The diversity in behaviors, code layout and implementation, and malware family types and ages is why we chose these three samples.

4.3.3 Results

The results of this experiment can be found in Figure 3. To obtain values for each function, we summed its corresponding basic block SHAP (setting negative values to 0) or MSE values.

DEEPREFLECT vs SHAP. The goal of SHAP is to identify regions within the model's inputs which affect the model's classification decision. While a malware classifier alone provides the analyst with the input's malware family, SHAP will identify *where* the most important regions of the input are for making that decision. Thus, conceptually it could be used to identify differences between different malware families and benign software (as previously discussed). However, this may not be completely effective. The analyst would have to continuously retrain the model whenever a new class of malware was discovered, and SHAP is inherently slow due to its recursive algorithm (making multiple passes back and forth through the neural network). DEEPREFLECT overcomes these issues by utilizing unsupervised learning and only requiring one pass through the neural network to retrieve the model's output.

DEEPREFLECT vs CAPA. Next, we compared DEEPREFLECT to CAPA [3], a tool which statically identifies capabilities within executables. It accomplished this by using hand-written signatures which describe various behaviors. For example, "connect to HTTP server", "create process", "write file", etc. Since CAPA is signature-based it is possible for it to miss malicious behaviors due to lack of generality, while DEEPREFLECT is trained using unsupervised learning and does not have this limitation. For DEEPREFLECT, we selected the detection threshold ϕ as follows: First, we plotted the ROC curves of all ground-truth samples (Figure 3). Then we identified separate thresholds for each sample which achieved a true positive rate (TPR) of 80%. We chose this TPR because it was large enough to detect a majority of malicious functions while keeping the FPs relatively low (for reviewing individual samples).

An example of where CAPA failed to identify behaviors was when the API call symbol was obfuscated by the malware (e.g., dynamically resolving the API call's name during runtime). Thus, it missed the function `KeyLoggerThread()` which calls various dynamically resolved API calls to log the victim's keystrokes. But since there are no interesting API calls here, CAPA misses it. DEEPREFLECT was able to successfully identify it because it *does not* solely rely on API calls and signatures to discover malicious behaviors.

An example of where DEEPREFLECT was unable to identify a behavior that CAPA (supposedly) did was an internal function which transports sent files to the C&C server. DEEPREFLECT should have conceptually picked up on this, it failed to do so. However, the API calls are all obfuscated, so CAPA should have failed here. Upon further investigation, CAPA believes there is a call here to retrieve a file's size, though in the source code such a call does not exist. Examining a neighboring function, we find it calls `GetFileSize()`. Therefore, we believe this is an example of an inconsistency between disassembler function addresses between CAPA's default disassembler and BinaryNinja (as both likely use different methods for function boundary detection). In this case, DEEPREFLECT discovered all of the malicious functions that CAPA did. While our tool did not succeed at catching the aforementioned malicious function (due to the thresholds we set), it is still more generic and scalable than signature-based tools, like CAPA, which rely on API calls and strings.

DEEPREFLECT vs FunctionSimSearch. FunctionSimSearch is a function similarity tool developed by Google Project Zero [5]. We trained a database on benign functions from our dataset with default parameters. After training their tool on our benign dataset, we queried it with the functions in our ground-truth dataset. We specified for the tool to output the top-1000 most common functions and their similarity scores. We chose 1,000 because of the speed at which it takes for a query to return from the tool (1 hour) and the sheer volume of functions inserted into the database (1,065,331 functions). To use this as an anomaly detector, we would expect that unfamiliar functions (i.e., malicious functions) would result in significantly smaller similarity scores than familiar functions. As seen in Figure 3, it performed poorly. It should be noted that a possible explanation for the poor performance is due to disagreements between function boundaries (as is common with different disassembly tools), but that this should not be drastically different (as seen with CAPA's disassembly tool which performed better).

Sample from the Wild. For verifying DEEPREFLECT's

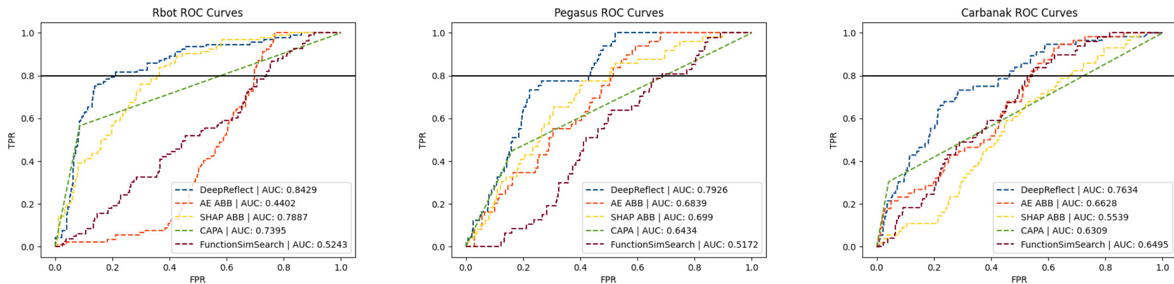

Figure 3: The ROC plot (performance at every threshold) for DEEPREFLECT, AE using ABB features, SHAP using ABB features, CAPA, and FunctionSimSearch on the three ground truth malware samples. The horizontal black bar represents a TPR of 80%.

ability to identify malicious functionality for in-the-wild samples, we randomly selected one binary from the `virut` family. We chose this sample binary to reverse engineer because it was simple (i.e., it was relatively small and were able to reverse engineer every internal function) and `virut` has been a well-studied botnet from 2006 - 2013 and beyond. First, a malware analyst reverse engineered this sample using CAPA and BinaryNinja, manually examining all 39 internal functions and labeling them as either malicious (according to the MITRE framework) or benign. Next, the analyst executed DEEPREFLECT on this sample and it identified 15 RoI's. Comparing this to our analyst's manual analysis, we initially thought that DEEPREFLECT missed one function (logic for comparing an argument that will either lead to terminating the malware's processor or not). Due to differences between CAPA's default disassembler's disassembly and BinaryNinja's disassembly, the function addresses (boundaries) were not identical. In this case, CAPA identified process termination at this internal function, where BinaryNinja contained no such logic at that function location. Because of this discrepancy, DEEPREFLECT essentially caught all malicious functionalities. Additionally we had an analyst use DEEPREFLECT on a malware which he has analyzed in the past. This is discussed more in §A.1.

Summary. We have shown that our autoencoder localization approach in DEEPREFLECT achieves goals G1 and G3 by identifying malicious behaviors in binaries without training on sample malwares or labeled data. Additionally, we have demonstrated its improvement over a popular explanation framework (SHAP) and signature-based method (CAPA). Most importantly, DEEPREFLECT is more practical than SHAP (which is slower and requires labeled dataset) and CAPA (a signature-based solution), because the model does not require the expensive process of having experts label malwares or their components. Lastly, we have shown that our features perform better than the ABB features.

4.4 Evaluation 2 – Cohesiveness

To evaluate DEEPREFLECT's ability to classify the malware components identified by the AE, we explore the semi-supervised clustering model's quality with the help of five experienced malware analysts.

4.4.1 Experiment Setup

First, we used the autoencoder M and identified 593,181 malicious components (functions) in 25,206 malware samples. This is less than the original ~36k samples because some of the samples either (1) never finished extracting features, (2) had no RoIs detected above the selected threshold, or (3) the RoI did not exist in the binary – the result of which perplexes us but could be explained as either data corruption, some issue with automatic upgrading BinaryNinja between extracting features and running clustering, or because the basic block exists in a function we do not consider (i.e., an external function)).

For clustering a large number of malware sample functions, we wanted to keep the FPR at a low level of 5%. In industry and real-world environments, lower FPR is often times more valued than TPR. Using this threshold (which yielded a combined TPR/FPR of 40%/5% on our ground-truth samples), we used DEEPREFLECT to extract and cluster the identified functions as C (§3.3). After running PCA on the function feature vectors, HDBSCAN produced 22,469 clusters using the default hyper-parameters. The largest cluster contained 6,321 functions and the smallest contained 5. There were 59,340 noise points.

In Figure 10, we present the distribution on the clusters' sizes. The figure shows that there is a long-tail distribution (common in density-based clustering) where the top-10 most populous clusters make up 5% of the functions.

The Reverse Engineers. To evaluate the clustering quality we recruited five malware analysts with 2-7 years of experience in reverse engineering.

The five analysts randomly sampled functions and labeled them using the MITRE ATT&CK [9] categorization. If the functions were deemed benign, the analysts labeled them as such. Overall, the analysts randomly sampled 177 functions (for the 176 different types of MITRE ATT&CK labels) each from the 25 largest malware family RoIs (chosen because of their size and diversity of behaviors). Time was a limiting factor to how many functions were selected. While 177 functions is small compared to the 600k extracted, it took between 15-30 minutes (and sometimes longer) to reverse engineer each function. We then selected one analyst to group these functions by hand. Finally, we compared the manual groupings to DEEPREFLECT's clusters performed various

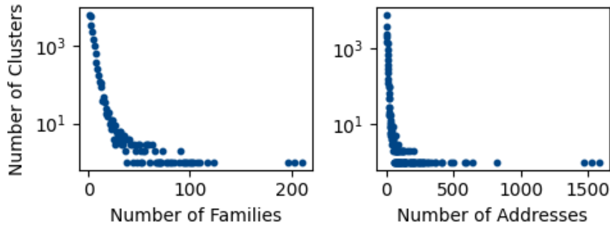

Figure 4: Cluster Diversity. Left: the distribution of families per cluster. Right: the distribution of addresses per cluster to show that there is no bias in function location.

measurements. In Table 3, we present the various MITRE ATT&CK labels the analysts came across in their work.

4.4.2 Results - Cluster Quality

After manually labeling functions, the analysts ended up with 78 malicious clusters. There were 5 cases where the handmade clusters appeared in *different* clusters in C. For brevity, we will only discuss three here. In the first case, the two functions which resided within the handmade cluster were deemed similar by the analyst. They were both small functions which called `SetEvent()`, though were not identical in content. One function had one more instruction that set the subroutine argument's value + 0x40 offset to 0. This was not enough of a difference to the analyst, so they clustered them as the same. However, to HDBSCAN the feature vector contents would have changed and thus (depending on parameters) may separate these two functions. This is a case where HDBSCAN was too sensitive. In the second case, three functions were deemed as similar to the analyst, but were separated into two clusters in C. The differing function contained a precondition `IsProcessorFeaturePresent()`, however both called `TerminateProcess()` on `GetCurrentProcess()` – thus they were close enough in behavior as to label them "Discovery: Virtualization/Sandbox Evasion". These are indicative of sandbox evasion because these techniques look for differences between processes in a sandbox and process on a real host [54]. Normally, one of the only reasons malware will attempt to exit is if they receive a command to do so from the C&C server or if they are in an undesirable environment (either not fit for the malware to infect or is determined to be an analysis environment). In the third case, a handmade cluster contained two functions which were separated in C. Similar to the other cases, these functions were close enough, but not exact, in content. They both performed `GetTickCount()` as well as calling various other internal functions in the same fashion. There were 8 cases where the handmade clusters were merged into the same clusters in C.

Though these errors appeared, 89.7% of the analyst's handmade clustered functions matched what our tool created. Thus, we consider the clustering results trustworthy. In the future, HDBSCAN's parameters could be tuned to correct these discrepancies.

Error Margins. We now evaluate what percentage of the clusters were benign versus malicious. When labeling randomly sampled functions, we look at hand-clusters with *consistent* labels. Sometimes our analysts disagreed with each other on what MITRE label to assign to a function. For consistency, we only consider those on which the analysts agreed. We found that of the 119 functions, 60.5% were malicious and 39.4% were benign, with a margin-of-error of 9.29%. Examining the percentages for all functions (regardless of their cluster) we find similar percentage results. Note that in §4.3.3 the false positive rate was much lower for our ground truth samples. This is because they only selected from the largest malware family RoIs (i.e., not uniformly random for the entire 600k population). This was done to ensure the analysts reviewed the most commonly extracted functions, which gave the analysts a better chance of discovering commonly shared malicious functions like C&C behaviors, anti-analysis behaviors, etc.
Summary. The malware analysts found that the clusters of DEEPREFLECT are consistent (regardless of malware family or the function's location within the binary). Although the amount of selected samples should capture the population, the results may differ on a larger sample size. We also found that the clustering matches 89.7% of an analyst's manually-clustered functions, contributing to goal G1.

4.5 Evaluation 3 – Focus

From prior work [69] and discussions with other analysts, we found that malware analysts' static reverse-engineering workflow begins with forming hypotheses about *where* various functionalities are within a malware binary. This is normally accomplished by observing where suspicious strings (e.g., URLs, domains) or API calls (e.g., `connect` or `send`) exist. However, as demonstrated in §4.3, these indicators cannot be relied upon alone. The benefit of DEEPREFLECT is its ability to focus the attention of the malware analyst, rather than sending them blindly to search through functions within each binary. We evaluate this by (1) calculating the percent of highlighted functions out of all the malware's functions, for each malware binary, (2) analyzing the false positives and a potential ranking scheme for DEEPREFLECT to prioritize which highlighted functions the analyst should look at first, and (3) discussing false negatives and how they might be mitigated in the future.
Workload Reduction. For each malware sample, we extracted each function which contained at least one RoI found by the autoencoder and compare that to the total number of internal functions within the binary. As seen in Figure 5, a large majority of the highlighted functions reduced the amount of functions for the analyst to view by at least 90%. The minimum reduction was 0% (i.e., all functions were highlighted), maximum reduction was over 99.9999%, and the average reduction was 85%.

These percentages by themselves could be misleading if number of functions in a malware sample is small to begin with. In terms of raw numbers, the min/max/average number of *highlighted* functions per malware sample was 1/527/23.53

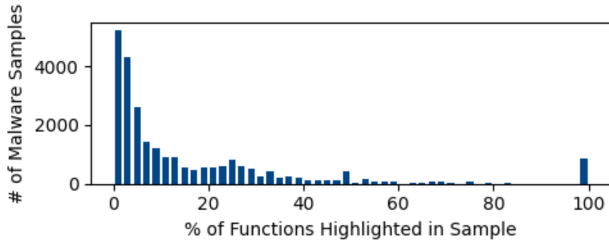

Figure 5: Function Counts. Percentage of functions (per malware sample) the analyst has to review.

respectively. The min/max/average number of *total* functions per malware sample was 1/26,671/663.81 respectively. This demonstrates that, on average, the analyst only has to review 24 functions, compared to 664 functions. However, we need to delve further, as these functions could be small in size and thus likely trivial to reverse engineer.

To answer this, we counted up the number of basic blocks for each function within each malware sample. Basic blocks can be an indicator of the complexity of a function. The min/max/average number of basic blocks within each *highlighted* function was 1/134,734/96.02 respectively. The min/max/average number of basic blocks within each function was 1/22,947/16.51 respectively. This shows that most of the highlighted functions were much more complex compared to the average function, and that if those functions were automatically labeled for an analyst, it would significantly reduce their workload.

False Positives & Prioritization

False positives exist in all security solutions. Reducing them is a never-ending task for those who work in real-world environments. When running DEEPREFLECT on our ground-truth samples using our cluster threshold, `rbot` contained 39 true positives (TPs) and 23 FPs, `pegasus` contained 22 TPs and 80 FPs, and `carbanak` contained 8 TPs and 69 FPs. While the TPs are relatively small (40% TPR), the FPs are much smaller comparatively (5% vs 25% FPR). To further reduce FPs, a solution is to sort the functions identified by DEEPREFLECT according to their MSE (similar to how we determined the threshold for clustering). Intuitively, the higher the MSE, the more malicious the function it should be. When examining the top-100 highest ranked components, DEEPREFLECT/SHAP had a precision of 0.629/0.487 on `rbot`, 0.229/0.138 on `pegasus`, and 0.111/0.01 on `carbanak`. As expected, the precision decreases when adding more top components since model's confidence is less on those with lower MSE (both `pegasus` and `carbanak` have larger code bases as they are more modern malware). These results are also consistent with our analyst's hands-on evaluation on `Mikey` in §A.1 where the false positives were placed into the bottom third in terms of MSE.

Sorting functions by MSE value is not always reliable. In this case, other basic mitigation strategies can be utilized. For example, the analyst can use simple heuristics (like those used in CAPA) on the functions extracted by DEEPREFLECT to get an understanding of what behavior category it may be. They can also prioritize functions by their uniqueness to the other functions in their dataset, finding which functions are potentially new emerging malicious behaviors the analyst has not seen before. For example, sort functions by their associated cluster's size (smaller clusters denoting more unique and less common functionalities).

False Negatives. False negatives are also common in all security solutions. Unknown threats will always exist which evade these systems. Using the same cluster threshold, DeepReflect had 53 FNs (325 TNs) for `rbot`, 27 FNs (407 TNs) for `pegasus`, and 48 FNs (2,111 TNs) for `carbanak`. Next, we discuss three FN cases from our ground-truth sample `rbot`. The first was a function `CaptureVideo()`, which took incremental screenshots of the victim's computer. This function had many calls to external APIs which were obfuscated (as is commonly done in malware). While we demonstrated that our tool is able to capture malicious functions containing obfuscated API calls, it is not always reliable at doing so, and any tool which does not have access to higher-level function calls will suffer because of it. The second is a function `getcdkeys()` which gathers the video game installation keys from the victim's host and sends it to the attacker's C&C server. Again, calls were made to obfuscated registry key API calls, which provides crucial contextual information. It might also be the case that some of the benign software games perform this exact same functionality to check if the user has installed a valid copy of the video game. This illustrates the need for carefully procuring a training dataset (as discussed later in §5). Finally, a third FN is a function `DDOSAttack()` which calls functions `ResolveAddress()`, `SpoofIP()`, and `SendDDoS()` which launches the attack. This function may have been missed because it acts more like a caller function to launch malicious behaviors. However, this caller function gives important contextual information about how the attack is launched. To mitigate this, a simple "guilt by association" heuristic could be used in the future where functions calling suspicious behaviors are identified as suspicious. Additionally, the threshold could be tuned depending on the analyst's goals of whether to increase TPs or reduce FPs.

Finally, we detail concrete examples of malicious functionalities identified by DEEPREFLECT (and labeled via MITRE) in Appendix A. There, we illustrate behaviors such as C&C communication for file dropping (Figure 6), file and data deobfuscation/decoding (Figure 7), and searching for various files to copy the contents of (Figure 8).

Summary. We have demonstrated that DEEPREFLECT has the ability to focus the analyst's attention on a variety of malicious activities within a malware sample. For most samples, it reduces their search space by 90% and 85% on average. This is helpful for when analysts need a high-level understanding of *where* malicious behaviors may exist so they can analyze them more in-depth (e.g., debugging). This satisfies goal G2.

4.6 Evaluation 4 – Insight

To evaluate if DEEPREFLECT provides meaningful insights into the relationships of malware families and their behaviors, we explored the cluster diversity. The left side of Figure 4 plots the number of distinct families per cluster in C. It can be seen that there are many shared malware techniques and variants between the families.

Diversity. Naturally, most of the clusters only have one malware family (explained by the long-tail distribution of our clusters shown in Figure 10). However, 10s to 1000s of clusters include a variety of families – some which even contain over 200 different families. For example, `tiggre` and `zpevdo` families share a "Execution: command and Scripting Interpreter" behavior where they call `GetCommandLineA()` and parse the characters involved (as described by MITRE).

Singleton Samples. These are malware families with only one sample. Since we use an autoencoder, we can capture novel behaviors from singleton samples. To check if DEEPREFLECT can identify malicious functions in a singleton sample, we observed if any singleton samples in our dataset got clustered with other malware families. Indeed, we found that DEEPREFLECT identified 1,763 clusters which contained at least one singleton sample.

Novel Malware Families. Next, we examine what happens when novel families are introduced to DEEPREFLECT. We made a clustering model C_1 on all of our malwares except for four families. Then, we added the families to the set and clustered the set as C_2. When we compared C_1 to C_2, we found that (1) new clusters were created by introducing the new families and (2) that portions of those families' functions were added to old clusters (i.e., the analyst would receive classification information on novel families). For more details, see §A.3.

Summary. We found that DEEPREFLECT provides insight into the relationship of malware behaviors (G4). In deployment, this meta information can be associated to the identified components providing the analysts with immediate insights.

4.7 Evaluation 5 – Robustness

Obfuscation. Given the rise of adversarial machine learning, we must be aware that the adversary may attempt to obfuscate their code to mitigate the productivity of DEEPREFLECT. Therefore, we evaluated DEEPREFLECT against an obfuscation attack scenario. We did not evaluate against packing or cryptors because those are out of scope for our tool. Instead, we utilize Obfuscator-LLVM [31] (denoted as *ollvm*). Using *ollvm* we obfuscated our `rbot` sample's source code using five techniques: (`A`) control-flow flattening, (`B`) instruction substitution, (`C`) bogus control-flow, (`D`) combining techniques (`A`) & (`B`), and (`E`) combining techniques (`B`) & (`C`).

Examining the functions extracted and clustered, DEEPREFLECT was mostly unaffected by the obfuscations. This makes sense because the autoencoder highlights functionalities it does not recognize and our features contain API calls (which were not modified by *ollvm*). For details, see §A.4.

Mimicry-like Attack. Next, we performed a simple mimicry attack where we inserted benign code which directly manipulated our features into malicious functions in our ground-truth samples. The benign code chosen was taken from an open-source repository of basic code for performing integer, string, and file I/O operations [10]. It was chosen because it has been used as a benchmark to test resilience against obfuscations [10]. In particular, we observed how much the MSE values changed for each function when using DEEPREFLECT compared to the AE we trained on ABB features. We targeted 12 functions (4 from each ground-truth sample) from a variety of behaviors (e.g., anti-AV, keylogger, dropper, DDoS, etc.). Using thresholds at TPR 80% from Figure 3 for each sample, we found that DEEPREFLECT outputted significantly larger MSE values (by several orders of magnitude) compared to the threshold for these modified functions (including the original functions) compared to the other AE. This suggests that DEEPREFLECT is more confident in labeling these functions as malicious. While none of these attacks were able to evade either model consistently, we observed that DEEPREFLECT's MSE values do not change drastically enough to cause concern. In addition, we observed that sometimes inserting the function with file I/O operations caused DEEPREFLECT to think a function was *more* anomalous than it originally considered (more so than compared to the AE trained on ABB features – this is reflected by the fact that both average MSE values increased after the attempted mimicry attacks). It also demonstrates the difficulty the attacker is tasked with: not just any benign code can be inserted into the malicious functions to evade it.

To increase the likelihood of bypassing DEEPREFLECT, we tested two more benign functions: (1) adding a network connect/send example hosted by Microsoft's website to the dropper malicious function, (2) adding the same example to the DDoS behavior, and (3) adding a process I/O creation example to a remote code execution where the malware starts a 'cmd.exe' process. The same results were observed, where our features outperformed ABB features in addition to DEEPREFLECT considering them more unfamiliar.

Summary. Although DEEPREFLECT was not significantly affected by *ollvm*'s obfuscation methods or our basic mimicry experiment, we are certain that DEEPREFLECT can be evaded. However, these experiments demonstrate that it is not easily fooled by these basic attacks.

5 Discussion

To summarize, we demonstrated that DEEPREFLECT can reliably identify malicious activities within malware samples (as shown in §4.3 and §4.6), which satisfies G1 from our research goals §2.4. Through other experiments we demonstrated that the system can focus the attention of the analyst and handle new malware families (shown in §4.4 and §4.5) which satisfies goals G2 and G3. It also demonstrates that DEEPREFLECT is

able to identify insights into shared functionalities of malware behaviors, satisfying G4 (the remaining goal). We also show that our tool is better than other baseline approaches such as explainable machine learning or signature-based solutions.

5.1 Limitations

Every system has weaknesses and ours is no exception.

Adversarial Attacks. A motivated adversary could poison the training dataset [46, 51] to cause the autoencoder to create a vulnerable model that would effectively hide the malware's functions. They could also blend in to look like a benign binary [24, 70]. Many papers have explored attacking machine learning models at the architectural level [47, 48, 73]. They could also poison the dataset used to cluster [19]. While these attacks do exist, common countermeasures [49, 65, 71, 74] can be applied to subvert them in the future.

An adversary could also attack our features by manipulating them to thwart our system. However, this could prove to be difficult, as our features are based on characteristics not easily changed. They would have to know how to precisely modify the structure of the CFG, types of instructions, and types of API calls used all without breaking the malware's *dynamic functionality*. This is not trivially done, either pre- or post-compile time.

Training Data Quality. Finally, our autoencoder model heavily depends on the content and quality of the benign dataset. If some functionality is left out of the training set, then the results will become biased. For example, if we were not to include any programs which performed network behaviors, then every network behavior seen would be something considered as malicious. Therefore, one must be careful to select a wide variety of benign software to compliment the malicious behaviors. On the other hand, if we train on too many malicious-like functionalities, our system may miss them in malware. For example, if Remote Desktop Protocol (RDP) behavior was an application in our benign dataset, our system may not label any RDP functionality as malicious. A proper balance needs to be struck to tailor our system to detect malicious functionalities the analyst is interested in exploring.

Human Error. DEEPREFLECT depends heavily on human analyst experience and agreement. There were issues with labeling the `pegasus` ground-truth in the beginning – we were not perfect in our initial source-code labeling. After debugging, we realized that there was a function which removed the history of internet connections via a remote desktop protocol (RDP) which was actually *not* a FP. Another supposed FP spawned a thread to interact with the remote victim's service control manager (SCM) which is certainly a malicious behavior. Thus we needed to update our labels, as there were other examples of this. While this may initially seem like a limitation, we see this as a potential teaching application. That is, experienced analysts can use our tool to provide labeled examples of functions and code from malware samples to facilitate training new or less-experienced analysts.

6 Related Works

Deep Learning and Malware. Recently, deep learning has been adopted by the malware analysis community. A majority of the goals are to classify or detect malware samples using deep learning neural networks [50, 66, 68]. Malconv [53] extracted raw byte values from executables and trained them on a convolutional neural network (CNN). Neurlux [30] extracted features from dynamic sandbox reports. Even Microsoft hosted a Kaggle competition [8, 56] where the goal was to take binaries (without their PE header attached) and classify them accurately according to 9 malware families.

Binary similarity has also been studied using both static and dynamic features [2,17,22,75]. While binary similarity is a similar problem to ours, it differs in an important way: their goal is to compare each binary with every other binary, whereas we encode what a particular type of binary looks like (benign binary) into a CNN and utilize reconstruction errors to tell us what portions it does not recognize. Our goal is not to formally identify similarities between binaries – though we do extend our analysis to identified shared concepts between malware families.

Autoencoders and Security. This paper is not the first to study autoencoders on cybersecurity datasets. [34] used a deep autoencoder to generalize what malware samples look like and provided the results to a generative adversarial network (GAN) in an attempt to thwart static techniques to obfuscate malware (e.g., re-ordering function layout). Other papers [20,28,32,76] use autoencoders to generate inputs to train other malware classifiers as a way to improve generalization. Our work differs significantly, as we train an autoencoder on benign binaries in an attempt to generalize what looks normal and use the reconstruction MSE to identify malicious functionalities in malware binaries. In [43] the authors used an ensemble of autoencoders as an NIDS by detecting abnormal feature vectors (snapshots of network traffic statistics). However, [43] uses Equation 1 to identify the abnormality of the observation as a whole, whereas DEEPREFLECT uses an autoencoder to localize one or more abnormalities within an observation using Equation 2.

To the best of our knowledge, there aren't any related works which statically identify and localize malicious functionalities in malware using machine learning, let alone with an unsupervised approach using autoencoders.

7 Conclusion

In this paper, we introduced DEEPREFLECT: a tool for localizing and identifying malicious components in malware binaries. The tool is practical since it requires no labeled datasets perform localization and a small number of labels for classification – collected incrementally from analysts during their regular workflow. We hope that this tool and published code will help analysts around the world by identifying *where* and *what* malicious functionalities exist in malware samples.

8 Acknowledgments

We thank the anonymous reviewers for their helpful and informative feedback. This material was supported in part by the Office of Naval Research (ONR) under grants N00014-17-1-2895, N00014-15-1-2162, and N00014-18-1-2662, and the Defense Advanced Research Projects Agency (DARPA) under contract HR00112090031. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of ONR or DARPA.

References

[1] Binaryninja. https://binary.ninja/.

[2] Bindiff. https://www.zynamics.com/bindiff.html.

[3] Capa. https://github.com/fireeye/capa.

[4] Cnet. https://download.cnet.com/windows.

[5] Functionsimsearch. https://github.com/googleprojectzero/functionsimsearch.

[6] Hdbscan. https://github.com/scikit-learn-contrib/hdbscan.

[7] Ida pro. https://www.hex-rays.com/products/ida/.

[8] Microsoft malware classification challenge (big 2015). https://www.kaggle.com/c/malware-classification.

[9] Mitre att&ck. https://attack.mitre.org/versions/v7/matrices/enterprise/windows/.

[10] Obfuscation benchmarks. https://github.com/tum-i4/obfuscation-benchmarks.

[11] unipacker: Automatic and platform-independent unpacker for windows binaries based on emulation. https://github.com/unipacker/unipacker.

[12] Virustotal. https://www.virustotal.com/.

[13] Virustotal statistics. https://www.virustotal.com/cs/statistics/, Aug 2020.

[14] Hyrum S. Anderson and Phil Roth. EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models. *arXiv preprint arXiv:1804.04637*, April 2018.

[15] Dennis Andriesse, Asia Slowinska, and Herbert Bos. Compiler-Agnostic Function Detection in Binaries. In *IEEE European Symposium on Security and Privacy*, 2017.

[16] Michael Bailey. Carbanak week part one: A rare occurrence. https://www.fireeye.com/blog/threat-research/2019/04/carbanak-week-part-one-a-rare-occurrence.html, Apr 2019.

[17] Davide Balzarotti, Marco Cova, Christoph Karlberger, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Efficient Detection of Split Personalities in Malware. In *NDSS*, 2010.

[18] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. Scalable, Behavior-Based Malware Clustering. In *NDSS*, volume 9, pages 8–11. Citeseer, 2009.

[19] Battista Biggio, Konrad Rieck, Davide Ariu, Christian Wressnegger, Igino Corona, Giorgio Giacinto, and Fabio Roli. Poisoning behavioral malware clustering. In *Proceedings of the 2014 Workshop on Artificial Intelligent and Security Workshop*, pages 27–36. ACM, 2014.

[20] Omid E. David and Nathan S. Netanyahu. Deepsign: Deep learning for automatic malware signature generation and classification. In *2015 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2015.

[21] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In *ACM Conference on Computer and Communications Security*, pages 51–62. ACM, 2008.

[22] Mohammad Reza Farhadi, Benjamin CM Fung, Philippe Charland, and Mourad Debbabi. Binclone: Detecting code clones in malware. In *2014 Eighth International Conference on Software Security and Reliability (SERE)*, pages 78–87. IEEE, 2014.

[23] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 480–491. ACM, 2016.

[24] Prahlad Fogla, Monirul I. Sharif, Roberto Perdisci, Oleg M. Kolesnikov, and Wenke Lee. Polymorphic Blending Attacks. In *USENIX Security Symposium*, 2006.

[25] Guofei Gu, Roberto Perdisci, Junjie Zhang, Wenke Lee, and others. BotMiner: Clustering Analysis of Network Traffic for Protocol-and Structure-Independent Botnet Detection. In *USENIX Security Symposium*, pages 139–154, 2008.

[26] Part Guide. Intel® 64 and ia-32 architectures software developer's manual. *Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4*, 2020.

[27] Yotam Gutman. Stop the churn, avoid burnout: How to keep your cybersecurity personnel. *SentinelOne*, Mar 2020.

[28] William Hardy, Lingwei Chen, Shifu Hou, Yanfang Ye, and Xin Li. DL4MD: A deep learning framework for intelligent malware detection. In *Proceedings of the International Conference on Data Science (ICDATA)*, page 61. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2016.

[29] Suman Jana and Vitaly Shmatikov. Abusing file processing in malware detectors for fun and profit. In *IEEE Symposium on Security & Privacy*, pages 80–94. IEEE, 2012.

[30] Chani Jindal, Christopher Salls, Hojjat Aghakhani, Keith Long, Christopher Kruegel, and Giovanni Vigna. Neurlux: dynamic malware analysis without feature engineering. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 444–455, 2019.

[31] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-LLVM – Software Protection for the Masses. In *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*, pages 3–9. IEEE, 2015.

[32] Temesguen Messay Kebede, Ouboti Djaneye-Boundjou, Barath Narayanan Narayanan, Anca Ralescu, and David Kapp. Classification of malware programs using autoencoders based deep learning architecture and its application to the microsoft malware classification challenge (big 2015) dataset. In *2017 IEEE National Aerospace and Electronics Conference (NAECON)*, pages 70–75. IEEE, 2017.

[33] Hyang-Ah Kim and Brad Karp. Autograph: Toward Automated, Distributed Worm Signature Detection. In *USENIX Security Symposium*, volume 286. San Diego, CA, 2004.

[34] Jin-Young Kim, Seok-Jun Bu, and Sung-Bae Cho. Zero-day malware detection using transferred generative adversarial networks based on deep autoencoders. *Information Sciences*, 460:83–102, 2018.

[35] Dhilung Kirat, Lakshmanan Nataraj, Giovanni Vigna, and B. S. Manjunath. Sigmal: A static signal processing based malware triage. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 89–98. ACM, 2013.

[36] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiao-yong Zhou, and XiaoFeng Wang. Effective and Efficient Malware Detection at the End Host. In *USENIX Security Symposium*, pages 351–366, 2009.

[37] Deguang Kong and Guanhua Yan. Discriminant malware distance learning on structural information for automated malware classification. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1357–1365. ACM, 2013.

[38] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In *USENIX Security Symposium*, volume 13, pages 18–18, 2004.

[39] Bo Li, Kevin Roundy, Chris Gates, and Yevgeniy Vorobeychik. Large-scale identification of malicious singleton files. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 227–238, 2017.

[40] Scott M. Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In *Advances in Neural Information Processing Systems*, pages 4765–4774, 2017.

[41] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. MAMADROID: Detecting Android Malware by Building Markov Chains of Behavioral Models. *Proceedings of 24th Network and Distributed System Security Symposium (NDSS)*, 2017.

[42] Najmeh Miramirkhani, Mahathi Priya Appini, Nick Nikiforakis, and Michalis Polychronakis. Spotless Sandboxes: Evading Malware Analysis Systems using Wear-and-Tear Artifacts. *IEEE Symposium on Security & Privacy*, 2017.

[43] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. Kitsune: An ensemble of autoencoders for online network intrusion detection. In *The Network and Distributed System Security Symposium (NDSS) 2018*, 2018.

[44] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *Annual Computer Security Applications Conference*, pages 421–430. IEEE, 2007.

[45] James Newsome, Brad Karp, and Dawn Song. Polygraph: Automatically generating signatures for polymorphic worms. In *IEEE Symposium on Security & Privacy*, pages 226–241. IEEE, 2005.

[46] James Newsome, Brad Karp, and Dawn Song. Paragraph: Thwarting signature learning by training maliciously. In *International Workshop on Recent Advances in Intrusion Detection*, pages 81–105. Springer, 2006.

[47] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z. Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning.

In *The ACM Asia Conference on Computer and Communications Security*, pages 506–519. ACM, 2017.

[48] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 372–387. IEEE, 2016.

[49] Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. Distillation as a defense to adversarial perturbations against deep neural networks. *IEEE Symposium on Security & Privacy*, 2016.

[50] Razvan Pascanu, Jack W. Stokes, Hermineh Sanossian, Mady Marinescu, and Anil Thomas. Malware classification with recurrent networks. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1916–1920. IEEE, 2015.

[51] Roberto Perdisci, David Dagon, Wenke Lee, Prahlad Fogla, and Monirul Sharif. Misleading worm signature generators using deliberate noise injection. In *IEEE Symposium on Security & Privacy*, pages 15–pp. IEEE, 2006.

[52] Roberto Perdisci, Wenke Lee, and Nick Feamster. Behavioral Clustering of HTTP-Based Malware and Signature Generation Using Malicious Network Traces. In *NSDI*, pages 391–404, 2010.

[53] Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles K. Nicholas. Malware detection by eating a whole exe. In *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[54] Thomas Raffetseder, Christopher Kruegel, and Engin Kirda. Detecting system emulators. In *Information Security*, pages 1–18. Springer, 2007.

[55] Alison DeNisco Rayome. Cybersecurity burnout: 10 most stressful parts of the job. *TechRepublic*, May 2019.

[56] Royi Ronen, Marian Radu, Corina Feuerstein, Elad Yom-Tov, and Mansour Ahmadi. Microsoft malware classification challenge. *CoRR*, abs/1802.10135, 2018.

[57] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. *MICCAI*, 2015.

[58] Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *Annual Computer Security Applications Conference*, pages 289–300. IEEE, 2006.

[59] Igor Santos, Felix Brezo, Xabier Ugarte-Pedrero, and Pablo G. Bringas. Opcode sequences as representation of executables for data-mining-based unknown malware detection. *Information Sciences*, 231:64–82, 2013. Publisher: Elsevier.

[60] Igor Santos, Javier Nieves, and Pablo G. Bringas. Semi-supervised learning for unknown malware detection. In *International Symposium on Distributed Computing and Artificial Intelligence*, pages 415–422. Springer, 2011.

[61] Matthew G. Schultz, Eleazar Eskin, F. Zadok, and Salvatore J. Stolfo. Data mining methods for detection of new malicious executables. In *IEEE Symposium on Security and Privacy*, pages 38–49. IEEE, 2001.

[62] Marcos Sebastián, Richard Rivera, Platon Kotzias, and Juan Caballero. AVclass: A Tool for Massive Malware Labeling. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 230–253. Springer, 2016.

[63] M. Zubair Shafiq, S. Momina Tabish, Fauzan Mirza, and Muddassar Farooq. Pe-miner: Mining structural information to detect malicious executables in realtime. In *International Workshop on Recent Advances in Intrusion Detection*, pages 121–141. Springer, 2009.

[64] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[65] Microsoft Defender ATP Research Team. New machine learning model sifts through the good to unearth the bad in evasive malware. *Microsoft Security*, Aug 2019.

[66] Microsoft Threat Protection Intelligence Team. Microsoft researchers work with intel labs to explore new deep learning approaches for malware classification. *Microsoft Security*, May 2020.

[67] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G. Bringas. SoK: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers. *IEEE Symposium on Security & Privacy*, 2015.

[68] R. Vinayakumar, K. P. Soman, and Prabaharan Poornachandran. Deep android malware detection and classification. In *International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 1677–1683. IEEE, 2017.

[69] Daniel Votipka, Seth Rabin, Kristopher Micinski, Jeffrey S. Foster, and Michelle L. Mazurek. An Observational Investigation of Reverse Engineers' Processes. In *USENIX Security Symposium*, pages 1875–1892, 2020.

[70] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *ACM Conference on Computer and Communications Security*, pages 255–264, 2002.

[71] Bolun Wang, Yuanshun Yao, Shawn Shan, Huiying Li, Bimal Viswanath, Haitao Zheng, and Ben Y. Zhao. Neural cleanse: Identifying and mitigating backdoor attacks in neural networks. In *IEEE Symposium on Security and Privacy*, 2019.

[72] Ke Wang, Gabriela Cretu, and Salvatore J. Stolfo. Anomalous payload-based worm detection and signature generation. In *International Workshop on Recent Advances in Intrusion Detection*, pages 227–246. Springer, 2005.

[73] David Warde-Farley, Ian Goodfellow, T. Hazan, G. Papandreou, and D. Tarlow. Adversarial perturbations of deep neural networks. In *Perturbations, Optimization, and Statistics*, pages 1–32. 2016.

[74] Weilin Xu, David Evans, and Yanjun Qi. Feature Squeezing: Detecting Adversarial Examples in Deep Neural Networks. In *Network and Distributed Systems Security Symposium*, 2018.

[75] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection. *ACM Conference on Computer and Communications Security*, 2017.

[76] Mahmood Yousefi-Azar, Vijay Varadharajan, Len Hamey, and Uday Tupakula. Autoencoder-based feature learning for cyber security applications. In *International Joint Conference on Neural Networks (IJCNN)*, pages 3854–3861. IEEE, 2017.

[77] Kim Zetter. Researchers easily trick cylance's ai-based antivirus into thinking malware is 'goodware'. https://www.vice.com/en_us/article/9kxp83/researchers-easily-trick-cylances-ai-based-antivirus-into-thinking-malware-is-goodware, Jul 2019.

Appendix A

A.1 Evaluation 1

Hands-on Evaluation. We asked a malware analyst with reverse engineering experience to use DEEPREFLECT on a malware which he has analyzed in the past (`Mikey`). Of the 15 functions which our tool identified in `Mikey`, the analyst found that there were 13 TPs, and 2 FPs. He noted that DEEPREFLECT identified an interesting component, which he had missed and that the two FPs were placed at the bottom third of the component's priority rankings.

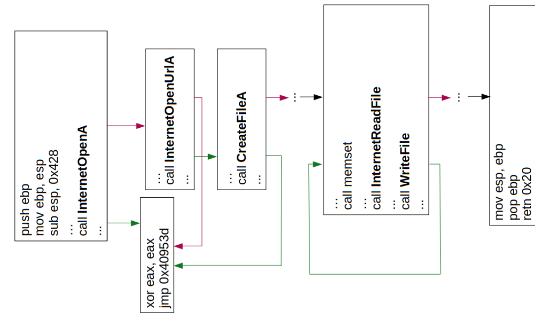

Figure 6: Command and Control: Ingress Tool Transfer. The malware accesses a URL via `InternetOpenUrlA()`, creates a file via `CreateFileA()` and writes data received from the connection to the file via `InternetReadFile()` and `WriteFile()`.

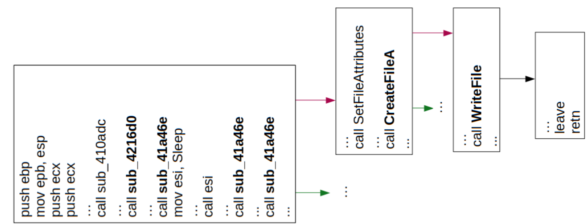

Figure 7: Defense Evasion: Deobfuscate/Decode Files or Information. This function makes many calls to internal functions (bolded) which contain complex bitwise operations on data (similar to that of Figure 9). These complex operations exhibit deobfuscation behavior. After calling these functions, it writes the decoded data to a file via `CreateFileA()` and `WriteFile()`.

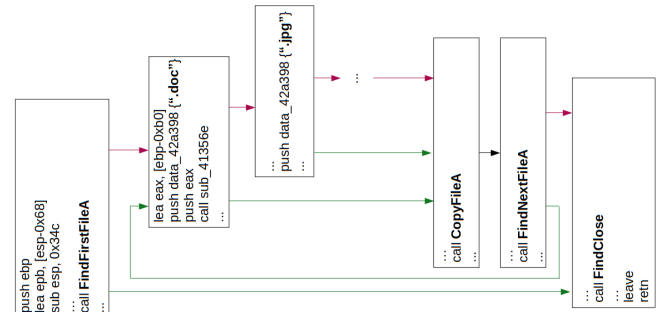

Figure 8: Discovery: File and Directory Discovery. This function searches for various files with specific extensions (i.e., `doc`, `jpg`, etc.). It then copies those files to a separate location. This behavior could be a setup for additional malicious behaviors like data exfiltration or ransom.

A.2 Evaluation 2

Most benign functionalities we discovered were memory allocation, loading a library, loading data from the process file's resources section, terminating a process (without context), etc. What an analyst labels as malicious can be subjective and relies on their experience and ability to match it with descriptions like those in MITRE ATT&CK.

Discovery	59	Defense Evasion	17	Privilege Escalation	4	Execution	11	Command and Control	7
System Information Discovery	16	Deobfuscation/Decode Files or Information	11	Create or Modify System Process	2	Scheduled Task/Job	7	Application Layer Protocol	4
File and Directory Discovery	12	Modify Registry	4	Access Token Manipulation	1	Command and Scripting Interpreter	2	Ingress Tool Transfer	3
Application Window Discovery	9	Hide Artifacts	1	Process Injection	1	System Services	2		
Query Registry	7	Virtualization/Sandbox Evasion	1						
Virtualization/Sandbox Evasion	5								
Process Discovery	4								
System Time Discovery	3								
Domain Trust Discovery	1								
Software Discovery	1								
System Network Connection Discovery	1								
Persistence	**2**	**Impact**	**2**	**Exfiltration**	**1**	**Collection**	**2**		
External Remote Services	1	Data Manipulation	1	Automated Exfiltration	1	Screen Capture	1		
Unknown	1	Network Denial of Service	1						

Table 3: The counts of MITRE ATT&CK categories and subcategories found by the analysts in §4.4.

```
push edi
mov edi, edx
add eax, 0x1
mov edx, 0x89705f41
mul edx
shr eax, 0x1e
mov ecx, edx
and edx, 0x1fffffff
shr ecx, 0x1d
lea edx, [edx+edx*4]
add edx, eax
mov eax, ecx
or eax, 0x30
mov byte [edi], al
mov eax, edx
cmp ecx, 0x1
sbb edi, 0xffffffff
shr eax, 0x1c
and eax, 0xffffffff
or ecx, eax
or eax, 0x30
mov byte [edi], al
lea eax, [edx+edx*4]
lea edx, [edx+edx*4]
cmp ecx, 0x1
sbb edi, 0xffffffff
shr eax, 0x1b
and edx, 0x7fffffff
or ecx, eax
or eax, 0x30
...
mov byte [edi], al
lea eax, [edi+0x1]
pop edi
retn
```

Figure 9: Defense Evasion: Deobfuscate/Decode Files or Information. This function performs various bitwise operations on data. Complex logic like this could be construed as performing some deobfuscation or decoding in an effort to hide data the malware interprets or gathers.

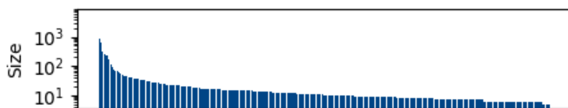

Figure 10: Cluster size distribution on our malware dataset using DEEPREFLECT. The x-axis is each cluster ID.

A.3 Evaluation 4

Novel Malware Families. We chose four well-known malware families: `zbot`, `gandcrypt`, `cosmicduke`, and `wannacry`. For `zbot`, before there were 22,433 clusters and after there were 22,470 clusters. Samples existed in 359 clusters, 4 of which were only `zbot` and the other 355 were mixed. On average, the number of `zbot` samples in the uniform clusters was 5.75 and the number in the mixed clusters was 1.49. That is, there were 4 new concepts not originally in the old clusterings. 320 new clusters (which contained `zbot`) were identical to old clusters. That is, 320 clusters (if labeled) would have provided 320 x 1.49 = 476.8 function labels automatically, leaving the analyst to review the newer clusters (behaviors). There were cases of 18 new clusters which only contained samples which were old noise points. There were 187 new clusters which contained old noise points. Finally, 17 new clusters which contained `zobt` samples were split into two clusters (i.e., were not identical to old clusters). Similar observations were made with the other families. Notably, `cosmicduke` samples did not result in new concepts (i.e., new clusters only composed of that family), and a majority of the new clusters after adding `wannacry` were composed of samples which were old noise points.

A.4 Evaluation 5

Obfuscation. First, we ran all five (plus the original source code compiled with *ollvm* with no obfuscations enabled) through DEEPREFLECT to observe the functions it identified using the threshold chosen for clustering. Our original, unobfuscated sample had 158 functions highlighted, `A` had 118, `B` had 156, `C` had 138, `D` had 118, and `E` had 137. Instead of manually examining 825 functions, we chose a random 10% from each sample to label (we chose 10% because it would ensure that we would have enough statistical significance to rely on our results – we identified 42% benign and 57% malicious with a margin-of-error of 11%). Our unobfuscated sample had 12 benign functions and 4 malicious functions highlighted. Our ground-truth labeling was stricter than our labeling for our evaluation set, and 7 out of the 12 benign functions could have been labeled by MITRE. `A` had 5 benign and 7 malicious functions. However, 2 out of the 5 benign functions could be described by MITRE. `B` had 10 benign and 6 malicious functions. However, 3 out of the 10 benign functions could be described by MITRE. `C` had 4 benign and 10 malicious functions, however 2 out of the 4 benign functions could be described by MITRE. `D` had 2 benign and 10 malicious functions. None of the benign functions could be described by MITRE. Finally, `E` had 3 benign and 11 malicious functions. However, 1 of the 3 benign functions could be described by MITRE. Lastly, we clustered the highlighted functions to observe the effect they have on the other functions. We hypothesized two outcomes: (1) the obfuscated functions look so obscure that they get labeled as noise points, or (2) the obfuscated functions look uniformly obscure, so they get clustered under one large cluster. However, we saw neither of these cases.