UNIVERSITY OF CALIFORNIA
RIVERSIDE

Towards A Robust Deep Neural Network Architecture for Malware Classification

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Wei Song

March 2022

Dissertation Committee:

Dr. Heng Yin, Chairperson
Dr. Chengyu Song
Dr. Zhiyun Qian
Dr. Christian Shelton

The Dissertation of Wei Song is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

I am grateful to my advisor, Dr. Heng Yin, without whose help, I would not have been here. His knowledge, enthusiasm, encouragement and continuous support are the key to my Ph.D study. Besides, my sincere gratitude go to the rest of my PhD disseration committee: Dr. Chengyu Song, Dr. Zhiyun Qian, Dr. Christian Shelton, for their insightful questions and comments. I would like to thank the collaborators Sadia Afroz, Deepali Garg, Dmitry Kuznetsov from Avast Antivirus, for their support and for all of the opportunities I was given to further my research. I also want to thank my fellow Zixiang, Sheng, Zhenxiao, Jinghan, Lian, Jie and Ju, for their help during my hard times. Most importantly, I would like to thank my wife Wenwen Han. This dissertation would not be possible without her love and support. This dissertation includes previously published materials entitled "MAB-Malware: A Reinforcement Learning Framework for Blackbox Generation of Adversarial Malware" published in the ACM ASIA Conference on Computer and Communications Security, 2022, and "DeepMem: Learning Graph Neural Network Models for Fast and Robust Memory Forensic Analysis" published in the ACM SIGSAC Conference on Computer and Communications Security, 2018.

To my wife for all the support.

ABSTRACT OF THE DISSERTATION

Towards A Robust Deep Neural Network Architecture for Malware Classification

by

Wei Song

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, March 2022
Dr. Heng Yin, Chairperson

Modern commercial antivirus systems increasingly rely on machine learning to keep up with the rampant inflation of new malware. However, it is well-known that machine learning models are vulnerable to adversarial examples. Previous works have shown that ML malware classifiers are fragile to the white-box adversarial attacks. However, ML models used in commercial antivirus products are usually not available to attackers and only return hard classification labels. Therefore, it is more practical to evaluate the robustness of ML models and real-world AVs in a pure black-box manner. The next question is how to create a new deep neural network architecture to make the malware classifiers robust by design. Compared to image recognition, there are few studies on improving the robustness of models in the field of malware classification. We found that in the malware domain, its threat model is quite different from the image domain. Existing methods have limited improvement in the robustness of malware classifiers, and the accuracy of the model will also decrease. Finally, memory-only malware has become popular in recent years. Since they are not written on disks, it becomes important to recognize their presence in memory.

These malware samples may hide their process information in the system, so we need a way to identify them fast and robustly.

This dissertation addresses these problems by presenting insights, methods, and techniques on how to perform attacks and defenses on malware classification in disk and memory. Firstly, a black-box reinforcement learning-based framework called MAB-Malware is developed to generate adversarial examples for PE malware classifiers and AV engines. It has a much higher evasion rate than other off-the-shelf frameworks. Second, a new classification architecture based on selective hierarchical BERT is proposed to automatically select only malicious functions for malware classification, which is robust to different attacks and self-explanatory. Thirdly, a graph-based deep learning approach is presented to automatically generate abstract representations for kernel objects, with which we could recognize the objects from raw memory dumps in a fast and robust way.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Malware attacks continue to be one of the most pressing security issues users face today. Recent research showed that during the first nine months of 2019, at least 7.2 billion malware attacks and 151.9 million ransomware attacks have been reported.[1] The attack rate hit a new high with the COVID-19 pandemic.[2] The traditional signature-based methods cannot keep up with this rampant inflation of novel malware. Hence commercial antivirus companies started using machine learning [12, 165] to keep up with the rampant inflation of new malware. Machine learning-based detectors are scalable and efficient at protecting against the huge influx of malware. Since the first paper in 2001 on detecting malware using machine learning [153], there has been an explosion of academic research papers on predicting malicious content using machine learning. Many of them flaunting high accuracy and being able to detect new malware unseen during training [138, 9, 37, 142, 150].

[1] https://www.msspalert.com/cybersecurity-research/sonicwall-research-malware-attacks-2019/

[2] https://labs.bitdefender.com/2020/04/coronavirus-themed-threat-reports-havent-flattened-the-curve/

But machine learning models are known to be vulnerable to adversarial attacks. However, research has also demonstrated that machine-learning-based detectors can be easily evaded by making even trivial changes to malware [28, 52, 177, 114, 143, 74, 85, 90, 7, 5, 161, 144, 72, 70, 131, 54, 179, 45, 112, 23]. Even commercial antivirus systems, such as Cylance, have been shown to be susceptible to trivial adversarial attacks [11]. Suciu et al. [162] and Kolosnjaji et al. [85] propose to calculate network gradient and modify appended and injected content to conduct whitebox attacks on these models. Demetrio et al. [42] and Xu et al. [177] apply genetic programming to append and inject content to generate adversarial examples. Anderson et al. [7] propose to apply deep reinforcement learning (RL) to generate AE for PE malware to bypass machine learning models. However, we found that these methods have some limitations in problem modeling, which hinder their attack capabilities. Meanwhile, few papers study how robust commercial AV systems are against real-world adversarial attacks, and use adversarial malware to infer how closed-source antivirus engines detect malware.

Our next research question is how to fundamentally improve the robustness of malware classifiers? In the most actively researched field of image recognition, many works have proposed various methods to improve the robustness of the model. In contrast, there is very little research in the field of malware classification. Most existing works in the malware domain directly use the method of the image field to improve the robustness of malware classifiers. We found that the threat model in malware is quite different from the image. Existing methods have limited improvement in the robustness of malware classifiers, and the accuracy of the model will also decrease. In this dissertation, we try to find the

limitations of existing defense approaches and propose a new architecture that is robust to different additive attacks, code randomization. At the same time, it does not sacrifice detection accuracy on original samples.

We also find that memory-only malware has become popular in recent years. Since they are not written on disk, it becomes important to first identify their presence in memory. Memory forensic analysis [22] extracts live digital evidence of attack footprints from a memory snapshot (or dump) of a running system. For instance, by identifying `_EPROCESS` objects in a Windows memory dump, analysts can figure out what processes are running on the target operating system. Memory forensic analysis is advantageous over traditional disk-based forensics because although stealth attacks can erase their footprints on disk, they would have to appear in memory to run. Existing approaches have several limitations: 1) list-traversal approaches are vulnerable to DKOM (Direct Kernel Object Manipulation) attacks, 2) robust signature-based approaches are not scalable or efficient, because it needs to search the entire memory snapshot for one kind of object using one signature, and 3) both list-traversal and signature-based approaches all heavily rely on domain knowledge of the operating system.

1.1 Thesis Statement

In short, the fundamental research question is how to effectively attack existing state-of-the-art machine learning-based models, understand how they make classification decisions, and then design a new architecture that is fundamentally robust against adversarial attacks. So the thesis statement is that to design a robust machine learning-based

malware classification model, you need to put your feet in the shoes of the attackers and see what they can do in the malware domain. Understanding the uniqueness of the security domain allows us to design an architecture that is fundamentally resistant to these attacks.

To this end, we propose three frameworks as illustrated in Figure1.1.

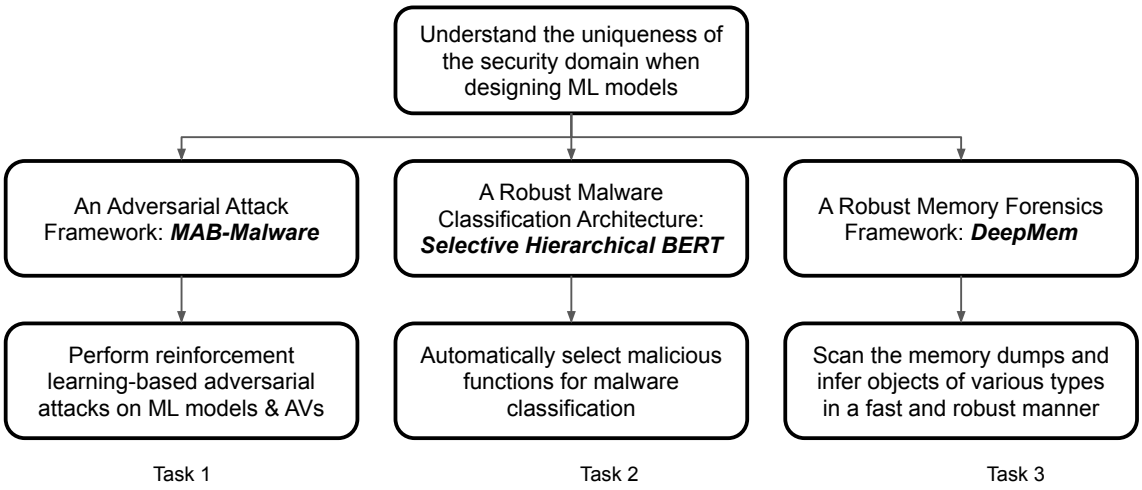


Figure 1.1: Overview of Thesis Work

MAB-Malware. We design a reinforcement learning guided framework MAB-Malware to perform adversarial attacks on state-of-the-art machine learning models for malware classification and top commercial antivirus static classifiers. For each commercial antivirus system, we compute the effectiveness of each action and the key features that cause evasions. Our results show that MAB-Malware largely improves the evasion rate over other reinforcement learning frameworks and that some of the adversarial attacks are transferable between different antivirus systems that are similar to one another.

Selective Hierarchical-BERT. We explore in detail the different constraints of malware classifiers and the properties that need to be satisfied to build a robust classifier. On

this basis, we propose a new robust deep neural network architecture based on selective hierarchical BERT to automatically select malicious functions for malware classification, which is robust to different attacks by design. Compared with other baselines, our model can handle very large samples and can automatically select essential features for malware classification, which fundamentally improves the robustness of the model without sacrificing the accuracy of the model.

DeepMem. We propose a graph-based kernel object detection approach DeepMem. By constructing a whole memory graph and collecting information through topological information propagation, we can scan the memory dumps and infer objects of various types in a fast and robust manner. is advanced in that 1) it does not rely on the knowledge of operating system source code or kernel data structures, 2) it can automatically generate features of kernel objects from raw bytes in memory dump without manual expert analysis, 3) it utilizes deep neural network architectures for efficient parallel computation, and 4) it extracts robust features that are resistant to attacks like pool tag manipulation, DKOM process hiding.

Chapter 2

An Adversarial Attack Framework: MAB-Malware

2.1 Abstract

Modern commercial antivirus systems increasingly rely on machine learning (ML) to keep up with the rampant inflation of new malware. However, it is well-known that machine learning models are vulnerable to adversarial examples (AEs). Previous works have shown that ML malware classifiers are fragile to the white-box adversarial attacks. However, ML models used in commercial antivirus (AV) products are usually not available to attackers and only return hard classification labels. Therefore, it is more practical to evaluate the robustness of ML models and real-world AVs in a pure black-box manner. We propose a black-box Reinforcement Learning (RL) based framework to generate AEs for PE malware classifiers and AV engines. It regards the adversarial attack problem as a multi-

armed bandit problem, which finds an optimal balance between exploiting the successful patterns and exploring more varieties. Compared to other frameworks, our improvements lie in three points: 1) limiting the exploration space by modeling the generation process as a stateless process to avoid combination explosions, 2) reusing the successful payload in modeling; and 3) minimizing the changes on AE samples to correctly assign the rewards in RL learning (which also helps identify the root cause of evasions). As a result, our framework has much higher evasion rates than other off-the-shelf frameworks. Results show it has over 74%–97% evasion rate for two state-of-the-art ML detectors and over 32%–48% evasion rate for commercial AVs in a pure black-box setting. We also demonstrate that the transferability of adversarial attacks among ML-based classifiers is higher than that between ML-based classifiers and commercial AVs.

2.2 Introduction

Adversarial attacks against static malware classifiers are not new. Researchers have proposed a variety of techniques to generate evasive samples (the terms "evasive samples" and "adversarial examples" are used exchangeably in this thesis), including genetic programming [177, 42], Monte Carlo tree search [137], and deep Q-learning [7]. Although some of these attempts [177, 137] are dealing with PDF malware and source code authorship respectively, the general algorithms can be applied to PE malware.

Although these techniques have been demonstrated to be effective, we have identified several limitations. First, the existing techniques model the AE generation in a stateful manner. However, it is hard to train a stateful model given that the search space is huge.

Therefore, we chose a stateless modeling approach, which can significantly reduce the learning difficulty and result in more productive AE generation. Second, most of the existing techniques only learn a decision-making policy that decides what action to take in the next step and randomly picks content if needed. We found that contents are as important as actions. If the content associated with certain action has proved to be useful in generating one AE, the same action-content pair will likely be useful for some other samples as well. Third, when an AE is successfully generated, these techniques will assign rewards to all the actions involved. In our evaluation, we observe that when AEs are generated, only a small number of actions applied to these AE are essential. The rest are redundant and can be removed. Assigning rewards to these redundant actions will confuse the learning process.

Based on these insights, we propose a reinforcement learning framework, called MAB-Malware to generate AEs for PE malware. Its name comes from our modeling of the AE generation problem as a classic multi-armed bandit (MAB) problem. In summary, the contributions of this chapter are as follows:

- We examine the existing algorithms in blackbox AE generation and provide key insights for stateful vs. stateless modeling, content-aware vs. content-agnostic modeling, and redundant vs. essential actions.
- We argue that a stateless and content-aware modeling is more suitable for generating adversarial PE malware, and an action minimization process is essential.
- To meet these design choices, we propose and implement a novel MAB-based reinforcement learning framework for generating adversarial PE malware.
- We conduct an extensive evaluation on two popular machine learning models and

three commercial AV engines. MAB-Malware outperforms the existing blackbox AE generation algorithms by large margins.

- Based on our action minimization, we further look into the root cause of these evasions. Our experiment results suggest the static classifiers in the commercial AV engines are vulnerable to trivial changes to malware samples.

To facilitate the follow-up research on this topic, we released the source code of our framework in a GitHub repository[1].

2.3 Motivation

In this section, we first discuss the existing reinforcement learning-based and genetic programming-based approaches on AE generation and their limitations, and then we present our insights that motivate our MAB-based approach.

2.3.1 Existing Approaches

Deep Q-learning. Anderson et al. [7] propose to apply deep reinforcement learning (RL) to generate adversarial examples of PE malware to bypass machine learning models. They first define a set of actions (binary mutations), including changing fields in PE header, appending overlay bytes, packing, and unpacking. Then the agent selects the next action based on a policy and an environmental state. When an evasive sample is generated, all applied actions (including early actions that produce no immediate reward) get promoted for a given state.

[1] `https://github.com/bitsecurerlab/MAB-malware.git`

Monte Carlo Tree Search. Quiring et al. [137] propose a Monte Carlo Tree Search (MCTS) based approach to mislead the classification of source code authorship. They define a set of actions (code transformation) for changing stylistic patterns. Then they create a Monte Carlo search tree, in which each node represents a variant of the code and each edge represents an action. Then the task of AE generation is converted to a path search problem. The goal is to find a path on the tree that leads to misclassification.

Genetic Programming. Demetrio et al. [42] propose a genetic programming-based approach to generate AEs of PE malware in a black-box attack manner. It formalizes the problem as a constrained minimization problem, to trade-off between the probability of evasion and injected payload size. The fitness function is defined as the sum of confidence scores and injected payload size. In each iteration, it selects variants with the lowest fitness score. Another paper from Xu et al. [177] also uses a genetic programming-based approach to generate adversarial PDF malware.

2.3.2 Our Insights

While these existing techniques have demonstrated their effectiveness more or less, we observe several key insights, which can motivate us to develop a better technique for AE generation.

Stateful vs. Stateless Modeling. Existing reinforcement learning techniques [7, 42, 177, 137] model the AE generation problem in a stateful way. They try to build a search tree, each node representing a state. The original sample is the root state. Each state has multiple actions to choose from. After applying one action, the sample enters the next

state. Existing works attempt to learn a policy to choose the next action in different states. For different states, the action selection strategy is different. However, as the search tree grows, the total number of states increases exponentially. It makes the training of action selection strategies difficult or sometimes impossible.

Our first insight is that we don't need to learn a stateful model for the malware AE generation problem. A model or AV engine classifies a sample as malicious may simply because it matches a specific signature, such as a section name or byte sequence. Before we find the correct action to change that signature, no matter how many irrelevant actions are applied, the sample will be detected for the same reason. Therefore, we don't need to construct the search tree to model different states. It makes the learning task unnecessarily difficult. Instead, we should keep the sample in only one state: non-evasive. Then our job becomes much easier: how to choose the correct action to jump from the same non-evasive state to the desired evasive state. We call it stateless modeling because there is only one state before evasion.

We propose to utilize a classic reinforcement learning model, the multi-armed bandit (MAB) [109] model, to solve the malware adversarial example generation problem. The MAB problem is formally equivalent to a one-state Markov decision process. It has just one state. From that state, it selects candidate actions to apply for the following iterations. Through these selections, it gradually learns the reward probability of each action. It tries to maximize the total rewards by finding the optimal tradeoff between exploration (learning the reward probabilities of unfamiliar actions) and exploitation (applying actions with high average rewards).

Content Modeling. Many actions used for manipulating PE need to be associated with some contents. For instance, when adding a new section, we need to specify what content to be filled in that section. When renaming a section, we need to provide a new section name. Our second insight is that these contents are as important as the actions. If content associated with one action has proved to be useful in generating one AE, the same action-content pair is likely to be useful for other samples.

Most existing works do not take contents into account. They only learn a decision-making policy to decide what action to take in the next step and take random content if required. For example, if the next action to take is "Section Add" according to the policy, they will fill the new section with random content. Our MAB-based framework treats an $< action, content >$ tuple as an integral unit (a slot machine in MAB) for modeling. If the new content is discovered to be useful to generate an adversarial example, it will be saved to be reused for other samples.

Precise Reward Assignment. Reward assignment is essential to all the existing AE generation techniques described above. When an AE is successfully generated, a positive reward is assigned to the corresponding sequence of actions. However, not all actions are essential to the generation of this AE. According to our evaluation in Section 2.6.4, in most cases, only one or two actions are essential. Therefore, assigning rewards to all the actions involved in an AE generation will lead to a less accurate reinforcement learning model. Hence, our third insight is that we should precisely assign rewards only to the essential actions.

2.4 Problem

2.4.1 Threat Model

We follow the study by Carlini et al. [19] to describe our threat model, from three aspects: adversarial goal, adversarial capabilities, and adversarial knowledge.

Adversarial Goal. The adversary's goal is to manipulate malware samples to evade the detection of static PE malware classifiers. Other types of malware like PDF malware or Android malware are not within the scope of this study. This is an untargeted attack because we only consider a binary classification (benign or malicious) not specific malware families in this classification task and we are only interested in causing the malicious samples to be classified as benign.

Adversarial Capabilities. In this work, we assume that the adversary does not have access to the training phase of the malware classifiers. For instance, the adversary cannot inject poisonous data into the training dataset. Also, the adversary cannot arbitrarily change the input data. In most scenarios of adversarial attacks, such as image recognition, the adversary is required to make only "small" changes to the original sample to keep the manipulation visually imperceptible. However, when attacking malware classification, the restriction is not on the number or size of changes, but on the preservation of malicious functionality. If "small" changes on a malware sample indeed confuse a malware classifier but prevent the malware from acting maliciously, this manipulation is not considered successful.

Adversarial Knowledge. Based on the knowledge an adversary can obtain, an attack can be divided into two types: 1) whitebox attacks where the adversary has unlimited access to

the model; and 2) blackbox attacks where the adversary has no knowledge about the model and can obtain the classification results only through a limited number of attempts. A classification result can be a score or simply a label. In this work, we consider an adversary with only blackbox access. The adversary does not know anything about the internals of the deployed classifiers, can perform a limited number of attempts to the classifiers, and can observe the classifiers' actions when the samples are considered malicious.

2.4.2 Problem Definition

In this chapter, we focus on three state-of-the-art machine learning classifiers and the static classifiers of 3 top commercial antivirus products. We aim to automatically generate adversarial examples for malware classifiers and explain the root cause of the evasions. The problem can be divided into two sub-problems: adversarial example generation and feature interpretation.

We aim to manipulate a malware sample such that malware classifiers misclassify it as benign, and do not break its malicious functionalities. For whitebox attacks in the image domain, changes to original images are bounded with L_2 and L_∞ norms. It ensures that the pixel changes are imperceptible to humans. However, in the malware classification domain, as long as the binary behaviors remain the same, normal users are unlikely to notice the differences between the original sample and the modified one. That is why previous blackbox attacks [25, 57, 7] on malware do not try to minimize changes when generating AEs. However, we find that the minimal change requirement is still crucial for three main reasons: 1) it reveals which actions and the corresponding payloads are essential to generate evasive samples that can be applied to other samples to create successfully evasive samples;

2) it unveils which feature changes caused the evasion to ensure that the classifier does not rely on superficial features; and 3) it reduces the chance of creating broken binaries. In the blackbox setting, instead of minimizing added noises in feature space, we minimize action sequences applied to generate AEs. It includes removing redundant actions and replacing actions that cause large changes to the features used for detection.

Let $\mathcal{X}$ be a malware dataset, f be a malware classifier that maps a sample $x \in \mathcal{X}$ to a classification label $y \in \{0, 1\}$ (0 represents benign, 1 represents malicious). We implement an action set $\mathcal{A} = \{a_1, a_2, \dots a_n\}$ that can be used to perturb malware samples. We define an objective function for adversarial example generation in (2.1). An adversarial example $x' = t(x)$ is generated by applying a transformation function t, which is a sequence of actions sampled from set $\mathcal{A}$. $\mathcal{L}(f(t(x)), \bar{y})$ measures the difference between the predicted label of $f(t(x))$ and benign label $\bar{y}$. The transformation function t subjects to the constraint that $t(x)$ does not change the functionality of x, i.e. the functionality difference $\delta(x, t(x))$ before and after transformation equals to 0.

$$\begin{aligned} &\underset{t}{\operatorname{argmin}}\, \mathcal{L}(f(t(x)), \bar{y}), \\ &s.t.\ \ \delta(x, t(x)) = 0, y \neq \bar{y} \end{aligned} \tag{2.1}$$

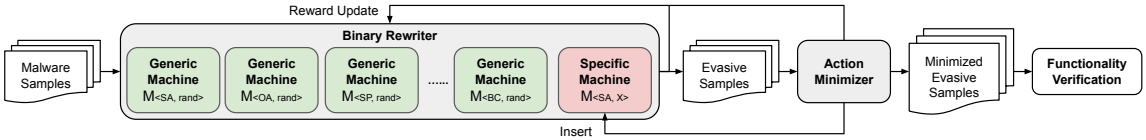


Figure 2.1: Workflow

2.5 Methodology

2.5.1 Adversarial Attack as a Multi-armed Bandit Problem

The Multi-Armed Bandit problem [109] is a classic reinforcement learning problem that embodies the exploration-exploitation trade-off dilemma. It is about how to maximize the total reward by allocating limited resources to multiple competing choices. The property of each choice is gradually learned in the process of resource allocation.

Slot machines in a typical casino are also called one-armed bandits [158] because the early machines have large mechanical levers attached to the sides, and they can empty the player's pockets like a thief. When the lever of the machine is pulled, its reward probability is θ. Therefore, a multi-armed bandit can be viewed as multiple slot machines with different reward probabilities. The reward probabilities are unknown to players. A player can observe each machine's reward probability by pulling it. However, the player has limited money. The goal is to maximize the sum of the rewards obtained through a series of lever pulls. The multi-armed bandit can be viewed as a tuple of $< \mathcal{M}, \mathcal{R} >$. $\mathcal{M}$ is a set of slot machines. $\mathcal{R}$ is a set of reward distributions $\{\theta_1, \ldots, \theta_K\}$, each distribution is associated a machine.

In the malware domain, we have many actions that can change the features of a PE binary without altering its functionality. Many actions require payload content to work.

For example, adding a new section requires benign content as the content of the new section. Content plays a vital role in attacking machine learning models, because the added content can largely change the byte entropy of the original malware sample in a certain direction. Adversarial attacks can be viewed as a problem of how to choose a serial of action and content pairs to maximize the probability of generating adversarial examples.

We treat the tuple $< action, content >$ as a slot machine M. When M is selected, it will apply *action* to the target binary file using the payload *content*. In our framework, we have two kinds of machines: generic machines and specific machines.

Generic Machine. A generic machine $M_{<action,rand>}$ is a machine, when selected, applies *action* to the target malware sample, with a random content extracted from benign binary files. For example, the OA (overlay Append) generic machine $M_{<OA,rand>}$ extracts a random section content from a random benign binary and appends the content to the target malware sample as overlay data. The reason for creating a generic machines is that at the beginning of the attack, we do not know which content is effective. By choosing a generic machine, we can explore different benign content for a certain action.

Specific Machine. A specific machine $M_{<action,X>}$ is a machine that, when selected, applies *action* to the target malware sample, with specific content X. After we generate an adversarial example x' by pulling a generic machine $M_{<action,rand>}$, if M machine is essential to the evasion (see details in Section 2.5.3), we will create a specific machine $M_{<action,X>}$. The content X is the specific content used in generating x'. When $M_{<action,X>}$ is selected by other malware samples, the specific content X is exploited to generate more adversarial examples.

The workflow of our framework MAB-Malware is shown in Figure 2.1. It consists of two main modules: the Binary Rewriter and the Action Minimizer. The Binary Rewriter utilizes Thompson sampling to select machines from the machine set $\mathcal{M}$ and rewrites original malware sample x to generate adversarial example x'. The Action Minimizer removes *redundant machines* to generate adversarial samples x'_{min} with minimal feature changes. *Redundant machines* are machines selected by the Rewriter in the generation of adversarial example x', but later we find that without them, the rest machines can still generate an adversarial example. That is because, at the beginning of the attack, the property of each machine is unclear. Rewriter needs to select these redundant machines to infer their reward probability. The rest necessary machines are called *essential machines*.

Our problem can be viewed as a tuple of $< \mathcal{M}, \mathcal{R} >$. $\mathcal{M}$ is a set of machines (including generic machines and specific machines), each machine M_i refers to pulling one slot machine $< action_i, content_i >$. $\mathcal{R}$ is a set of reward distributions $\{\theta_1, \ldots, \theta_K\}$ (suppose we have K slot machines), each distribution is associated an action. The reward distribution θ_i of each machine is unknown. We have a limited number of attempts to pull these machines. The goal is to maximize the reward through a series of pulls.

Thompson Sampling. In our task, we face a delayed feedback problem. When evaluating the static modules of commercial antivirus systems, we need to copy the generated sample to the virtual machine with antivirus and wait for the scanning result. This process takes seconds, even minutes for certain AVs. If we adopt a deterministic algorithm, such as upper confidence bounds, it will always select the one with the highest values before the result returns. It causes inefficient trials because of outdated information. To address this

issue, we use Thompson sampling algorithm [168], which is more robust than deterministic algorithms in the delayed feedback environment [26].

We assume the reward follows a beta distribution [1] specific to that machine. The beta distribution is a continuous probability distribution parameterized by two positive parameters, denoted by α and β, i.e. $M \sim \text{Beta}(\alpha, \beta)$. α and β correspond to the counts of success or fail respectively. At each action selection iteration, for each machine, we sample a value from its $Beta(\theta; \alpha, \beta)$ distribution and select the machine with the highest value as the next machine. When the α and β values of a machine are small, the uncertainty of M is high. Even if this average reward is lower than other machines, it still has a relatively high possibility to get a large value. In this way, new machines are more likely to be selected for exploration. After several trials, the α and β value of that machine becom large, and the uncertainty decreases. In this way, machines with high average rewards are selected for exploitation.

Reward Propagation. When a machine is created, we set α=1, β=1 for each machine. For every machine that is selected by Rewriter but fails to generate the adversarial example, we increase its β by 1. When an adversarial example is generated and minimized, for every essential machine, we increase the α by 1. If the machine is a generic machine, we also create new specific machines using its specific content (with α=1, β=1). If an essential machine is a specific machine, we also increase the α of its corresponding generic machine that it derives from, to encourage the exploration for certain types of actions.

2.5.2 Binary Rewriter

Action Set and Features

Table 2.1: Action Set.

Type	Abbr	Name	Description
Macro	OA	Overlay Append	Appends benign contents at the end of a binary.
	SP	Section Append	Appends random bytes to the unused space between sections.
	SA	Section Add	Adds a new section with benign contents.
	SR	Section Rename	Change the section name to a name in benign binaries.
	RC	Remove Certificate	Zero out the signed certificate of a binary.
	RD	Remove Debug	Zero out the debug information in a binary.
	BC	Break Checksum	Zero out the checksum value in the optional header.
	CR	Code Randomization	Replace instruction with semantically equivalent one.
Micro	OA1	Overlay Append 1 Byte	Appends 1 byte at the end of a binary.
	SP1	Section Append 1 Byte	Appends 1 byte to the unused space between a section.
	SA1	Section Add 1 Byte	Adds a new section with 1 byte content.
	SR1	Section Rename 1 Byte	Change 1 byte of a section name.
	CP1	Code Section Append 1 Byte	Appends 1 byte to the unused space of code section.

Table 2.2: Affected Features by Actions.

		CR	OA	SP	SA	SR	RC	RD	BC	OA1	SP1	SA1	SR1	CP1
Hash-Based Signatures	F_1: File Hash	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	F_2: Section Hash	✓		✓				✓			✓			✓
Rule-based Signatures	F_3: Section Count				✓							✓		
	F_4: Section Name					✓							✓	
	F_5: Section Padding			✓										
	F_6: Debug Info							✓						
	F_7: Checksum								✓					
	F_8: Certificate						✓							
	F_9: Code Sequence	✓												
Byte Entropy	F_{10}: Byte Entropy		✓		✓									

We implemented 13 actions in Table 2.1. Each action manipulates a set of features that a classifier may use to detect malware (shown in Table 2.2).

Macro-actions. We reimplemented actions proposed by Anderson et al. [7] using the pefile library and fix many corner cases that may break the functionality. We also adopt a code randomization action (CR) from Pappas et al. [125]. It is a defense method originally

proposed to prevent Return Oriented Programming (ROP) attacks. It provides binary code transformations that can safely randomize stripped binaries, without changing their semantics.

Micro-actions. If action a affects k features $\{f_1, f_2, \dots f_k\}$ of the malware sample, then an action that affects only a subset of these features is the micro-action of a. We have implemented 5 micro-actions: OA1, SP1, SA1, SR1, and CP1. They are similar to the corresponding macro-actions but with minimized feature changes. Take SA1 as an example. Similar to SA, SA1 also adds a new section entry in the header, but it only adds a 1-byte section. SA adds a lot of benign content, and greatly changes the byte entropy of the original binary, while SA1 does not. Therefore, SA affects features $\{F_1, F_3, F_{10}\}$, SA1 only affect features $\{F_1, F_3\}$. By looking up Table 2.2, you can see that SA's micro-actions also include $OA1$ and OA, affecting features $\{F_1\}$ and $\{F_1, F_10\}$ respectively. OA is also considered a macro-action during the attack. So, we can see that micro-action is a relative concept.

Workflow of Binary Rewriter

Algorithm 1 summarizes the workflow of Binary Rewriter. For a set of malware samples X, our goal is to generate a set of adversarial samples X_a. First, we initialize $\mathcal{M}$ by creating 8 generic machines, and each machine's α value and β value are set to 1. To select the next machine, for each machine, we sample a value from its β distribution and select the machine with the highest value. Then we apply the corresponding machine to x. We apply a serial of machines iteratively until we get an evasive sample or exceed the total number of attempts N. When an evasive sample is generated, we further use Action

Algorithm 1 Adversarial Attack

Input: malware sample set X
Output: adversarial example set X_a

initialize($\mathcal{M}$)
$X_a \leftarrow []$
for all $x \in X$ **do**
  $list\_M \leftarrow []$
  for all $attempt\_idx \leftarrow 1$ to N **do**
    $M \leftarrow \max(\text{betaSampling}(\mathcal{M}))$
    $x' \leftarrow \text{apply}(x, M)$
    $list\_M$.add(M)
    if isEvasive(x') **then**
      $x'_{min}, list\_M_{min} \leftarrow \text{minimize}(x, list\_M)$
      X_a.add(x'_{min})
      for all $M' \in list\_M_{min}$ **do**
        incAlpha(M')
        if isGeneric(M') **then**
          $M'_s \leftarrow$ createSpecificMachine(M')
          $\mathcal{M}$.add(M'_s)
        else
          $M'_g \leftarrow$ getParentGeneric(M')
          incAlpha(M'_g)
        end if
      end for
      break
    else
      incBeta(M)
    end if
  end for
end for
return X_a

Minimizer to remove redundant machines. For the remaining machines $list\_M_{min}$, we first increase their α by 1. If it is a generic machine, we create a new specific machine M'_s. If it is a specific machine, we increase the α value of its parent generic machine, which has the same action type but with random content. For failed machine, we increase the value of β by 1.

Algorithm 2 Minimize

Input: malware sample x, applied machines $list\_M$
Output: minimized AE x'_{min}, minimized machines $list\_M_{min}$

$list\_M_{min} \leftarrow list\_M$
for all $M \in List\_M$ **do**
  $list\_M' \leftarrow List\_M_{min} - M$
  $x' \leftarrow \text{apply}(x, list\_M')$
  if isEvasive(x') **then**
    $list\_M_{min} \leftarrow List\_M'$
    $x'_{min} \leftarrow x'$
  else
    $list\_micro \leftarrow$ get_micro_actions(M)
    for all $M_{mic} \in list\_micro$ **do**
      $list\_M' \leftarrow List\_M_{min} - M + M_{mic}$
      $x' \leftarrow \text{apply}(x, list\_M')$
      if isEvasive(x') **then**
        $list\_M_{min} \leftarrow List\_M'$
        $x'_{min} \leftarrow x'$
        break
      end if
    end for
  end if
end for
return $x'_{min}, list\_M_{min}$

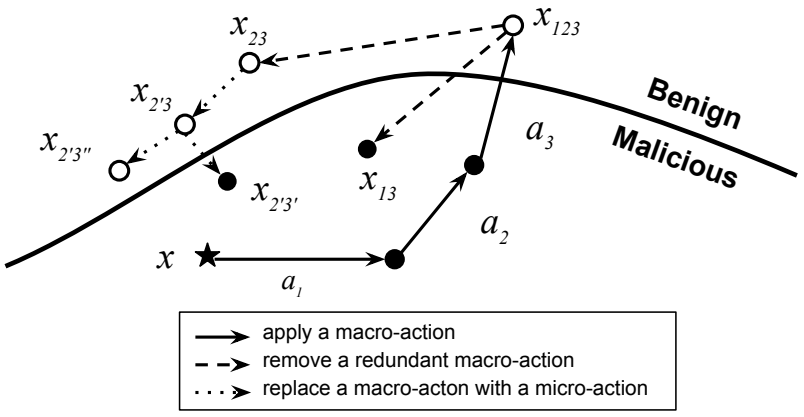


Figure 2.2: An example of action minimization.

2.5.3 Action Minimizer

The Action Minimizer removes redundant actions and uses micro-actions to replace macro-actions, to produce a "minimized" evasive sample that only changes minimal features.

As shown in Figure 2.2, the original malware sample x resides in the malicious region of the feature space. During the attack, we perform a sequence of actions a_1, a_2 and a_3 until the generated sample x_{123} successfully reaches the benign region, and becomes an adversarial example. In the minimization phase, we first remove redundant actions. The action a_2 is essential, because by removing action a_2, the generated sample x_{13} is no longer evasive anymore. The action a_1 is useless because by removing action a_1, the generated sample x_{23} has no effect in the classifier's decision. Then we disentangle these actions into micro ones (i.e., actions that cause smaller changes). a_2 can be replaced with micro-actions a_2'. Action a_3 can be replaced with micro-actions a_3' or a_3''. We generate three samples $x_{2'3}$, $x_{2'3'}$ and $x_{2'3''}$. Finally, we have an adversarial sample $x_{2'3''}$ with a minimized action sequence (a_2', a_3''). So a positive reward can be precisely assigned to these essential actions a_2' and a_3''.

As shown in Algorithm 2, for each machine M in the applied machines $list\_M$, we try to remove it and apply the new sequence $list\_M'$ to the original sample x to generate x'. If x' is still evasive, it means that the machine M action is redundant. We can permanently delete M from $list\_M$. Otherwise, we will find that all micro-actions $list\_micro$ that only change the subset of features changed by M. Then we try to replace M with each micro-action M_{mic} in $list\_micro$ and apply the new sequence to generate x'. If x' is still evasive, then we use M_{mic} to permanently replace M. If we find that M cannot be removed or replaced, it means that machine M is essential. In this way, we can delete redundant feature changes and find the essential actions.

For example, to generate an evasive sample x' for x, Binary Rewriter has applied 5 actions: CR, OA, SP, BC, SA. Action Minimizer will check every machine to determine if it can be removed. It finds that the first 4 actions are redundant, only the last action SA cannot be removed. SA (add a new section) changes 3 features of the original binary file. It changes the file hash, creates a new entry in the section table, and adds a content block to the end of the file. Correspondingly, SA has 3 micro-actions: OA1, SA1, and OA. Each of them only changes one feature. If we replace SA with any micro-action, we will remove redundant feature changes. In this way, we can generate the minimized adversarial example x'_{min}.

From a defender's point of view, we also would like to understand how an evasion happens, where the weakest point of the classifiers is. The action minimization of evasive samples provides a good opportunity to infer that information. Figure 2.3shows how we break macro-actions into micro-actions. Take the action Section Append (SP) as an ex-

ample. First, by looking up Table 2.2, SP changes feature $F = \{F_1, F_2, F_5\}$ (File Hash, Section Hash and Section padding). The actions that only change a subset of F are OA1 that changes $\{F_1\}$ and SP1 that changes $\{F_1, F_2\}$. Starting from the minimum change, we try to replace SP with OA1 and check if the file is still evasive. If so, we can conclude that the evasion is caused by the change of file hash (F_1). If not, we continue to replace SP with SP1. If successful, the evasion is caused by the change of section hash (F_2). Otherwise, the evasion is caused by the change of signatures in section padding content (F_5). This way we can generate a minimized adversarial example and find out what is causing the evasion.

2.6 Evaluation

2.6.1 Experiment Setup

Dataset: In this chapter, we generate adversarial examples for Windows PE binaries. To ensure the executability and functionality of the generated samples, the format and constraints of PE files must remain intact. To guarantee the quality of malware samples, we randomly select 5000 samples from VirusTotal that meet the following requirements: 1) more than 80% antivirus engines of VirusTotal label them malicious; and 2) the execution of those samples in a Cuckoo sandbox shows malicious behavior.

Setup: The experiments are performed on 20 virtual machines of the Microsoft Azure cloud platform. The configuration of each virtual machine is Standard D2s v3 (2 vcpus, 8 GiB memory). For all the antivirus software under testing, free versions and default settings are used. We choose three top commercial antivirus products for blackbox testing,

If SP ← OA1, then feature = **File Hash**
Else If SP ← SP1, then feature = **Section Hash**
Else feature = **Section Padding**

(a) SP

if SA ← OA1, then feature = **File Hash**
Else If SA ← SA1, then feature = **Section Count**
Else If SA ← OA, then feature = **Byte Entropy**
Else feature = **Section Count** & **Byte Entropy**

(b) SA

If CR ← OA1, then feature = **File Hash**
Else If CR ← CP1, then feature = **Section Hash**
Else feature = **Code Sequence**

(c) CR

If RD ← OA1, then feature = **File Hash**
Else If RD ← CP1, then feature = **Section Hash**
Else feature = **Debug Information**

(d) RD

If SR ← OA1, then feature = **File Hash**
Else If SR ← SR1, then feature = **Section Name**
Else feature = Part of **Secton Name**

(e) SR

If OA ← OA1, then feature = **File Hash**
Else feature = **Byte Entropy**

(f) OA

If BC ← OA1, then feature = **File Hash**
Else feature = **Checksum**

(g) BC

If RC ← OA1, then feature = **File Hash**
Else feature = **Certificate**

(h) RC

Figure 2.3: Decision rules are used to map actions to feature space

which are anonymized as AV1, AV2, and AV3. Each antivirus is installed on an Azure virtual machine with Windows 7. To ensure the malware will not infect other machines in the network and the stability and reproducibility of our experiments, all network traffic is routed to an InetSim instance on the host machine to provide simulated network services.

We choose the following models as our target models:

- **EMBER** [9] is an open-source machine-learning-based classifier that uses a tree-based classifier model LightGBM to detect malware. It generates a 2350-dimensional feature vector for each sample consisting of two main types of features: raw features (e.g. ByteHistogram, ByteEntropyHistogram, Strings) and parsed features (e.g. GeneralFileInfo, HeaderFileInfo, SectionInfo, ImportsInfo, ExportsInfo). We use the model provided in MLSEC2019 (Machine Learning Security Evasion Competition) [118].

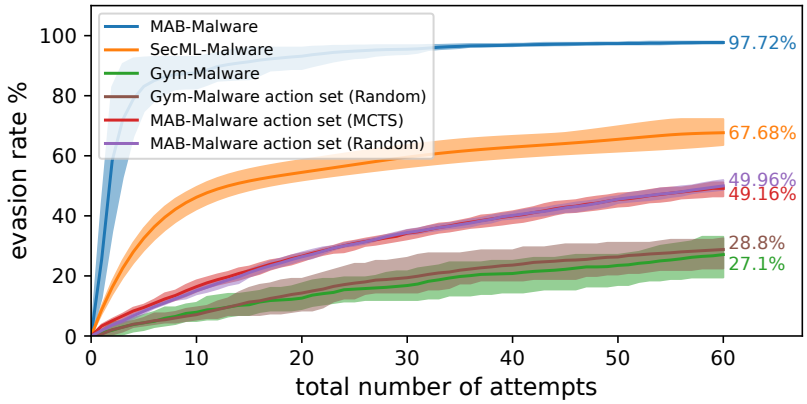
- **MalConv** [138] is a malware detection model that uses a convolutional neural network to learn knowledge directly on the raw bytes of malware samples. We also use the model provided in MLSEC2019 [118].

- **Commercial AVs.** We also test the static classifiers of 3 top commercial antivirus systems.

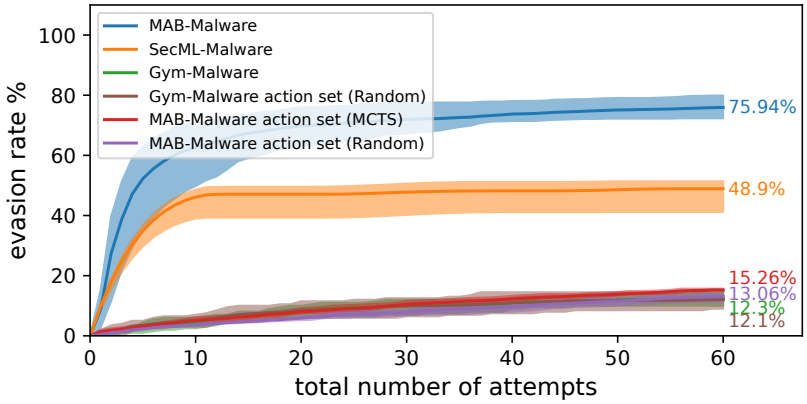
2.6.2 Adversarial Example Generation

Comparison with Other Off-the-Shelf Frameworks. We compare our MAB-Malware with other two off-the-shelf attack frameworks: SecML-Malware[2] and Gym-Malware[3]. SecML-Malware is a plugin for the SecML Python library. It contains many kinds of attacks, including black-box attacks with hard labels. We utilize its genetic programming-based black-box attack (GAMMA) in this experiment. Gym-Malware is a reinforcement learning-based malware manipulation environment using OpenAI's gym. Its agents learn how to manipulate PE files to bypass AV based on a reward provided by taking specific manipulation actions.

[2] `https://github.com/zangobot/secml_malware.git`
[3] `https://github.com/endgameinc/gym-malware.git`



(a) MalConv



(b) EMBER

Figure 2.4: Evasion Results

We measure the evasion rate for two machine learning-based models, MalConv and EMBER. Evasion Rate is defined as: $R_e = N_e/N_d$, where N_e is the total number of successful evasive samples, and N_d is the total number of original samples that can be detected by the target model. For a fair comparison, we use the same dataset (5000 samples from VirusTotal) and MalConv and EMBER models (from the Machine Learning Static Evasion Competition 2019 [118].) We run each experiment five times to calculate an average.

From Figure 2.4, we can see that MAB-Malware performs much better than the other approaches. It can generate AEs for 97.72% samples to evade MalConv, 74.4% samples to evade EMBER. The evasion rate of SecML-Malware (GAMMA-hard label) is 63.6% and 50.0% respectively. Gym-Malware has the lowest evasion rate (27.1% and 12.3%). The evasion rate is almost identical to random action selection using its own action set (28.8% and 12.1%). This indicates that this deep Q-learning model does not learn meaningful knowledge to guide the evasion. The reason is that the problem modeling creates an exponentially large search space. And without action minimization, the reward assignment is chaotic. Within 60 trials, it cannot explore enough in such a large space and learn meaningful policy to select the correct action and corresponding content.

Comparison with Other Algorithms. The action sets of these three frameworks are different. SecML-Malware only uses benign content injection and appending. Gym-Malware's operation set is similar to ours, but it also includes packing and unpacking. As a result, we cannot see the effectiveness of the MAB-based action selection algorithm.

So in this experiment, we only use our own action set and change the action selection algorithms. The baseline is random selection. Then we compare our method with the other reinforcement learning algorithm. In the experiment above, we have already shown that the Q-learning models cannot directly improve the evasion rate over random selection. In this experiment, we further implement another MCTS-based reinforcement learning algorithm. Quiring et al. [137] propose an MCTS-based approach to mislead the classification of source code authorship. Because their code cannot be directly applied to malware classifiers, we borrowed their idea and reimplemented it for malware classification.

It can be seen from Figure 2.4 that in the same action set, our MAB algorithm greatly improves the evasion rate compared to random action selection, while the MCTS algorithm hardly provides any improvement. Existing frameworks model AE generation in a stateful way and try to find the best state path leading to escape. This makes it difficult to train in a large search space. In addition, the existing framework does not have a mechanism to effectively reuse the successful payload.

Table 2.3: Evasion Result on Antivirus.

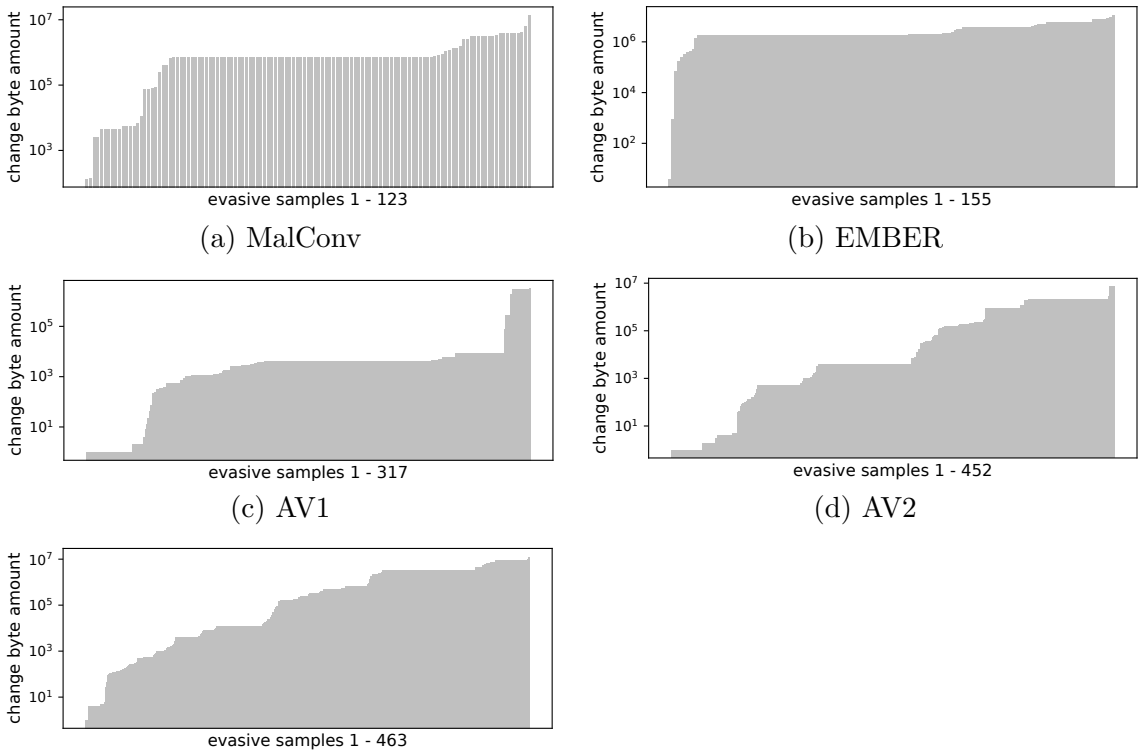
Antivirus	**Frameworks**	
	SecML-Malware	MAB-Malware
AV1	5.61%	**31.99%**
AV2	11.40%	**46.2%**
AV3	12.75%	**48.3%**

Attacking Commercial Antivirus. We also test our framework on three commercial antivirus engines. The throughputs of commercial AV engines are much lower than machine learning classifiers. We need to copy a lot of generated samples into the virtual machine with a particular AV installed and wait for AV engines to scan them to get labels. It usually takes seconds or even minutes to get the result. As a result, we only use 1000 samples for this experiment. Also, we do not compare Gym-malware since it cannot finish the experiment within a reasonable time frame.

As shown in Table 2.3, SecML-Malware only achieves 5% - 12% evasion rate for all AVs, while MAB-Malware achieves 31% - 48% evasion rate. It shows the advantages of MAB-Malware in generating adversarial examples in a pure blackbox setting.

Number of bytes changed. The Action Minimizer ensures that the minimized evasive samples only change minimal content to flip the classification label. So by checking how

many bytes we need to change, we can infer the robustness of different malware classifiers. To measure the difference between the minimized evasive example and the original malware, we compute the total number of bytes appended or modified by our framework.



(a) MalConv (b) EMBER

(c) AV1 (d) AV2

(e) AV3

Figure 2.5: Number of Changed Bytes of Adversarial Examples.

By positioning the samples in a line sorted by byte changes (Figure 2.5), we notice that:

- By only changing one byte of the original malware, we can generate 33 for AV1, 32 for AV2, 3 for AV3.
- Machine learning models are not vulnerable to small changes. However, it does not mean that ML models are more robust than commercial AVs. From the previous

evasion rate results, we can see that using our framework, ML models are easier to evade than commercial AVs.

2.6.3 Testing Functionality Preservation

We found that the action set in Gym-Malware, which is implemented using LIEF [99] library, is not safe. According to our experiment result in Table 2.4, more than 60% of the generated binaries after a single action cannot be executed, or behave differently. To solve this problem, we carefully reimplement most actions using the pefile [21] library to avoid many corner cases that may lead to a broken binary. For example, before adding a new section, we check whether there is enough space between the last section header entry and the first section.

We implement our own action set using the pefile library whereas the Gym-Malware rewrites binaries using the LIEF library. We noticed that rewriting a binary with the LIEF library can cause unnecessary changes to the binary that can sometimes result in broken files, thus destroying the functionality of the original malware samples. To compare our actions with the actions from Gym-Malware, we randomly select 50 malware samples from our dataset, create adversarial samples by applying different actions, analyze all variants in the Cuckoo sandbox, and compare the behaviors with the original samples.

From Table 2.4 we can see that except for the Overlay Append action, most actions in the Gym-Malware framework cause 63.24% of the rewritten samples to lose functionality. In contrast, only less than 8% of the rewritten samples using our actions create broken binaries.

Table 2.4: Functionality Preservation Rate of the Actions

Actions	**Functional Rate**	
	Gym-Malware Actions	MAB-Malware Actions
(OA) Overlay Append	45/48 (93.75%)	46/48 (95.83%)
(SP) Section Append	11/47 (23.40%)	42/43 (97.67%)
(SA) Section Add	11/47 (23.40%)	39/42 (92.86%)
(SR) Section Rename	11/47 (23.40%)	42/43 (97.67%)
(RC) Remove Certificate	1/3 (33.33%)	3/3 (100.00%)
(RD) Remove Debug	5/13 (38.46%)	13/13 (100.00%)
(BC) Break Checksum	9/48 (18.75%)	32/33 (96.97%)
Average	**93/253 (36.76%)**	**217/225 (96.44%)**

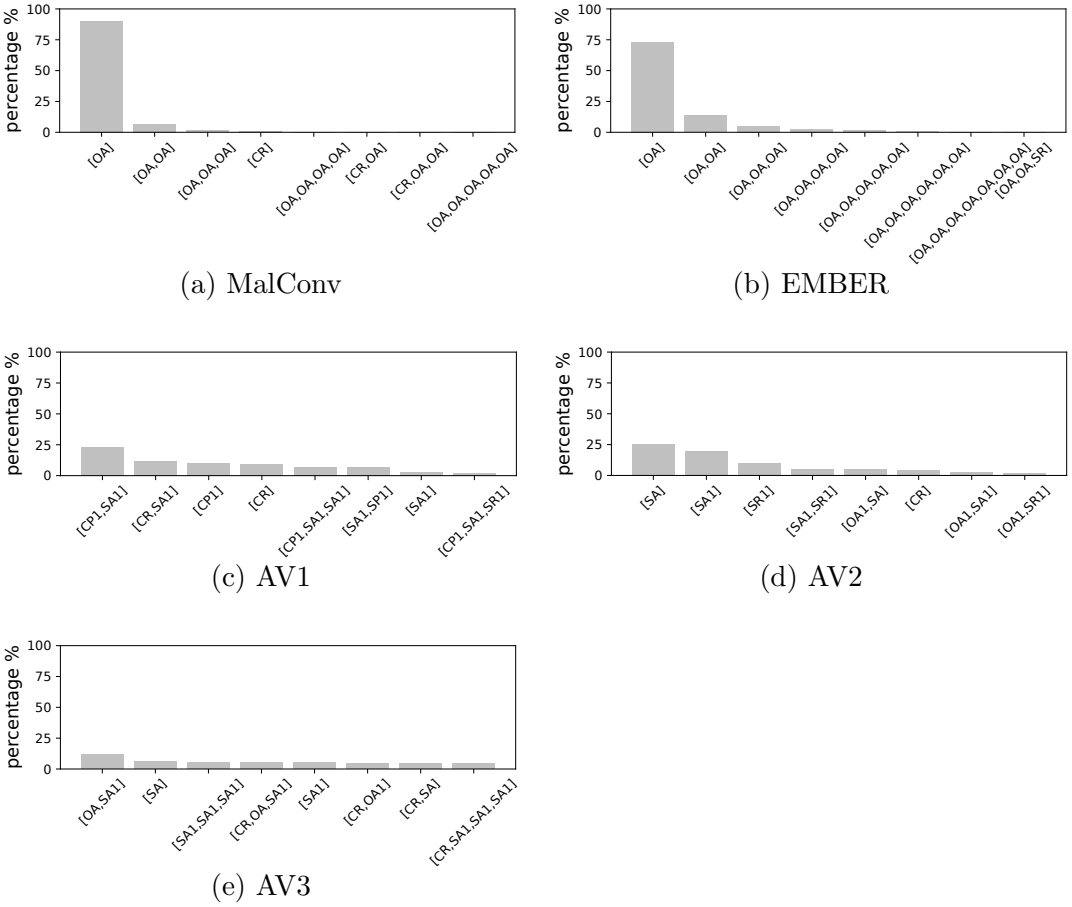
2.6.4 Explanation

Understanding why an evasion happens can help improve the robustness of a classifier against adversarial attacks. For each evasive sample, the Action Minimizer first removes all redundant actions and uses micro-actions to replace the macro-actions. We summarize the most frequent action sequence combination is Figure 2.6. According to the rules in Figure 2.3, we can infer the root cause of each evasion, shown in Figure 2.7. We found that:

- For two machine learning-based classifiers, the most important action is Overlay Append (OA). Other actions that only change a few bytes have almost no effect on them. It shows that the change in byte entropy is the root cause of the evasions.

- The Section Add 1 Byte (SA1) action plays a significant role in evading all AVs. It indicates that all AVs utilize section count as an important feature for detecting malware.

- Comparing to AV2 and AV3, AV1 is also vulnerable to the Code Section Append 1 Byte (CP1) action. CP1 alters the hash of the code section. It indicates AV1 uses



(a) MalConv (b) EMBER

(c) AV1 (d) AV2

(e) AV3

Figure 2.6: Action Sequences for Adversarial Examples

code section hash as an important feature for detecting malware.

- The Section Rename 1 Byte (SR1) action itself can generate many adversarial examples for AV2. SR1 changes one byte of one section name. It indicates that AV2 relies heavily on the section name for detecting malware.

- Comparing with AV2 and AV3, the Section Add (SA) action and the Overlay Append (OA) action have almost no effect on AV1. SA and OA greatly change the byte entropy of the original malware samples. It indicates that AV2 and AV3 integrate

Figure 2.7: Feature changes that cause evasion.

some machine learning models in static detection. And AV1 mainly uses the signature-based approach to detect malware.

2.6.5 Transferability

Transferability refers to the property that allows an adversarial sample that can evade one model can also evade other similar models. If the adversarial malware samples are transferable, then evading one malware detector would be enough to evade all malware detectors.

Figure 2.8 shows the percentage of evasive samples generated for one classifier that can also evade other classifiers. The number in the cell (model A, model B) shows the

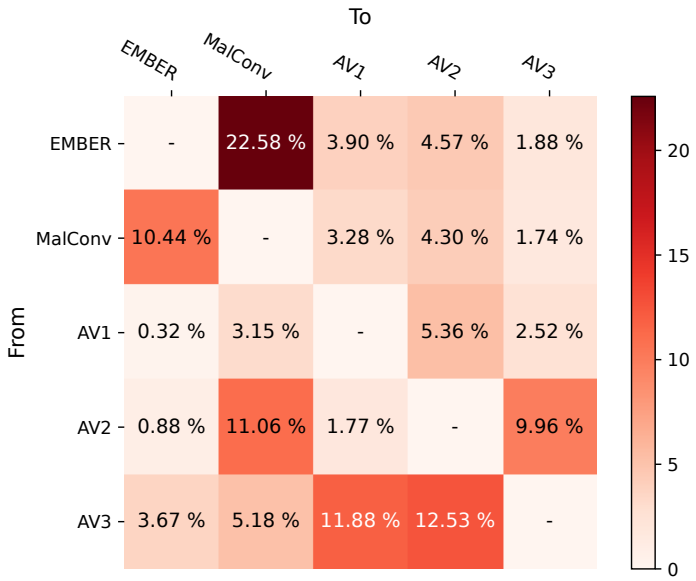


Figure 2.8: Transferability of Adversarial Samples

percentage of evasive samples generated for model A can also evade model B. We noticed:

- The transferability between machine learning models is quite high, although EMBER and MalConv are trained on different features (2350-dimensional extracted features vs. raw bytes), and the architecture is different (decision tree vs. neural network).
- Both AV2 and AV3 utilize machine learning models and consider Section count as an important feature. So the transferability between AV2 and AV3 is relatively higher than others.

2.7 Discussions

Triviality of Defense. The triviality of the defense depends on the type of attack. To defend the overlay append attack, the defender can ignore the overlay data when training

models. To defend against the SA attack, the defender can lower the importance of benign features in models, and only consider malware features. To defend against RD, SR, and BC attacks, defenders should avoid using fragile patterns as malware features. However, completely ignoring the trivial features can reduce the accuracy of a malware detector. The code randomization (CR) attack is hard to defend because the defender cannot locate the small snippet of binary that is randomized.

Recommendation for Researchers. We demonstrate how adversarial examples can be used to explain a complex blackbox system. When training malware classifiers, researchers should use explanation techniques to understand the behavior of the classifiers and check if the learned features are fragile features that can be easily evaded or if they conflict with expert knowledge. We also argue that for security applications, demonstrating harm to real users is crucial to understanding the real ramification of an attack.

The generality of our evasive techniques. First, our framework conducts blackbox attacks against classifiers. Unlike whitebox attacks, blackbox attacks do not require knowledge of the architecture and parameters of the target classifier. Theoretically, our approach can be used on any malware classifier, as long as the classifier returns a label for testing samples. Second, we attacked 5 representative malware detectors of diverse techniques, including a decision tree-based model (EMBER), a deep learning model (MalConv), and 3 commercial AV engines from top-level AV companies. The significant evasion rate improvement of these detectors proves the generality of our method.

Mitigation using Dynamic Detection. Our solution cannot bypass dynamic detectors, but we argue that dynamic evasion is another research topic. Static evasion itself is an

important research direction because it provides a defense before users execute potentially dangerous programs. This is why ML-based static classifiers, such as EMBER and MalConv, increasingly attract attention in the security community.

2.8 Discussions

Triviality of Defense. The triviality of the defense depends on the type of attack. To defend the overlay append attack, the defender can ignore the overlay data when training models. To defend against the SA attack, the defender can lower the importance of benign features in models, and only consider malware features. To defend against RD, SR, and BC attacks, defenders should avoid using fragile patterns as malware features. However, completely ignoring the trivial features can reduce the accuracy of a malware detector. The code randomization (CR) attack is hard to defend because the defender cannot locate the small snippet of binary that is randomized.

Recommendation for Researchers. We demonstrate how adversarial examples can be used to explain a complex blackbox system. When training malware classifiers, researchers should use explanation techniques to understand the behavior of the classifiers and check if the learned features are fragile features that can be easily evaded or if they conflict with expert knowledge. We also argue that for security applications, demonstrating harm to real users is crucial to understanding the real ramification of an attack.

The generality of our evasive techniques. First, our framework conducts blackbox attacks against classifiers. Unlike whitebox attacks, blackbox attacks do not require knowledge of the architecture and parameters of the target classifier. Theoretically, our approach

can be used on any malware classifier, as long as the classifier returns a label for testing samples. Second, we attacked 5 representative malware detectors of diverse techniques, including a decision tree-based model (EMBER), a deep learning model (MalConv), and 3 commercial AV engines from top-level AV companies. The significant evasion rate improvement of these detectors proves the generality of our method.

Mitigation using Dynamic Detection. Our solution cannot bypass dynamic detectors, but we argue that dynamic evasion is another research topic. Static evasion itself is an important research direction because it provides a defense before users execute potentially dangerous programs. This is why ML-based static classifiers, such as EMBER and MalConv, increasingly attract attention in the security community.

2.9 Conclusion

In this chapter, We design a reinforcement learning guided framework MAB-Malware to perform adversarial attacks on state-of-the-art machine learning models for malware classification and top commercial antivirus static classifiers. We model the action selection problem as a multi-armed bandit problem. During the attack, MAB-Malware infers the property of actions and dynamically adding new machines with unseen successful content. It finds an optimal balance between exploitation and exploration to maximize the evasion rate within limited trials. The Action Minimization module of MAB-Malware filters out the actions that are ineffective for adversarial sample generation and only change minimal features, so our framework can also be used to explain why evasion occurs. For each commercial antivirus system, we compute the effectiveness of each action and the key

features that cause evasions. Our results show that MAB-Malware largely improves the evasion rate over other reinforcement learning frameworks and that some of the adversarial attacks are transferable between different antivirus systems that are similar to one another.

Chapter 3

A Robust Malware Classification Architecture: Selective Hierarchical BERT

3.1 Abstract

In the most actively researched field of image recognition, many works have proposed various methods to improve the robustness of the model. In contrast, there is very little research in the field of malware classification. Most existing works directly use the method of the image field to improve the robustness of malware classifiers. We found that in the malware domain, its threat model is quite different from the image domain. Existing methods have limited improvement in the robustness of malware classifiers, and the accuracy of the model will also decrease. In this chapter, we explore in detail the different constraints

of malware classifiers and the properties that need to be satisfied to build a robust classifier. On this basis, a new robust deep neural network architecture based on selective hierarchical BERT is proposed to automatically select malicious functions for malware classification, which is robust to different attacks by design. Compared with other baselines, our model can handle very large samples and can automatically select essential features for malware classification, which fundamentally improves the robustness of the model without sacrificing the accuracy of the model.

3.2 Introduction

In the malware domain, existing state-of-the-art models, such as MalConv, EMBER, are very vulnerable in the presence of adversarial attacks. We used a multi-armed bandit to model the problem in the last chapter and optimally attacked them, we can achieve an evasion rate of 74%–97%.

Some works such as Demetrio1[43] use saliency map analysis to explain what MalConv has learned. The article found that MalConv only learned some very superficial features. For example, it found that most of the features that MalConv considered most important came from certain fields in the PE header. And these fields can be easily modified by attackers. And these modifications will not affect the normal execution of the program. This paper also proposes an attack method that only modifies the content of the PE header. The results show that very high evasion rates can be achieved without modifying the internal code of the malware. Malware samples are malicious because their internal code features do malicious things, other than co-occurrences and artifacts of cer-

tain superficial features in the training dataset. A robust malware classifier should exclude easily changeable, superficial features.

MalConv's recognition of features is location-sensitive. We find that if one random byte is inserted at the beginning of the malware sample, the accuracy of MalConv drops from 95% to about 50%. Although arbitrarily inserting bytes will affect the executable of the malware sample, this shows that the recognition of features by MalConv is not shift-invariant. Attackers can do code reallocation to change the location of malicious code. It will not affect the sample's functionality. Moreover, if attackers have the source code, they can easily insert code to shift the location of the existing code in the compiled binary.

In summary, the contributions of this chapter are as follows:

- We examine existing methods for building robust classifiers and identify different constraints between the fields of image identification and malware classification.
- We discuss and list the properties that need to be satisfied to build a robust malware classifier.
- To satisfy these design choices, we propose and implement a novel selective hierarchical BERT-based robust deep neural network architecture to automatically select malicious functions for malware classification, aiming at robustness against different attacks.
- We conduct extensive evaluations of baseline models and attacks on our architecture to show that our method is more robust and accurate than existing methods.

The remainder of this chapter is structured as follows. Section 3.3 provides a background of existing approaches and our insights. Section 3.4 gives an overview of our

new robust architecture, followed by design details of each component. Section 3.5 presents implementation details and evaluation results.

3.3 Motivation

3.3.1 Existing Approaches

In the field of image recognition, to defend against adversarial attacks, lots of works have proposed various methods to improve the robustness and generality of deep learning models.

Adversarial Training. Madry et al. [111] points out that adversarial training with one-step perturbations is still vulnerable to sophisticated adversaries. They introduce a multiple-step method that utilizes projected gradient descent (PGD) to generate AEs and train the model iteratively. Xie et al. [176] find out that the non-smooth nature of the ReLU activation function weakens the process of adversarial training. The reason is that adversarial training needs more computations for the inner maximization step to generate the perturbation δ.

Regularization. Several papers, such as [79, 181] propose different regularization techniques to minimize features learned by their machine learning models. The goal of regularization is to make the model generalize to unseen examples. including AEs. However, this kind of defense cannot resist more advanced adversarial attacks [91]. It also harms the precision rate on clean samples.

Defensive Distillation. Papernot et al. [123] propose to train a second DNN model using soft labels instead of hard class labels used by the original model. However, Carlini et al. citecarlini2016defensive point out that defensive distillation is not effective since attackers

can slightly modify their attack to evade it. Meanwhile, it is generally used for multi-classification tasks. And our target classifier is the binary classifier, which only distinguishes whether it is malware or not.

Detect & Drop Adversarial Examples. Wang et al. [174] propose X-Ensemble, an interpreter-based ensemble framework to detect and rectify adversarial examples. It utilizes the gradient discrepancy in multiple interpreters between clean samples and adversarial examples to train multiple CNN models (detector) to differentiate them.

Some of these methods have been proven ineffective, such as defensive distillation. The attacker only needs to modify the loss function of the attack to generate stronger adversarial examples. Among the rest methods, the most effective method is adversarial training. It is widely used in the image recognition domain to defend against adversarial attacks. Our first attempt was also to use our MAB-Malware framework (introduced in Chapter 2) to generate adversarial examples and add them to the MalConv training dataset to continue training. We quickly found that adversarial training reduced the accuracy of MalConv.

To deal with similar problems in the image domain, Cai et al. [16] propose the Curriculum Adversarial Training (CAT) method to develop a curriculum of adversarial examples with an increasing range of attack strengths. It learns from easier adversarial examples first, then gradually learns from stronger and stronger adversarial examples. It can help to make the training easier and do not overfit adversarial examples in the image domain. We also tried this approach in the beginning, but later we found that it does help to keep MalConv's accuracy from dropping too much. In the end, we found the curriculum

adversarial trained MalConv is still vulnerable to our MAB-Malware attacks, as well as white-box attacks [85] against MalConv.

The reason for this is that the constraints for attacking images and attacking malware are different. In the field of pictures, if there are too many modifications to the picture, the human eye can notice it. Therefore, attackers usually need to satisfy that the modification of the picture is within the range that a human eye cannot detect, or only a small area of the picture can be modified. But in the malware field, since the malware sample is just a file, ordinary users cannot see whether the file has been modified with the naked eye. Therefore, the attacker can often append a large amount of new section or overlay data to the original sample, and even the newly added data is many times larger than the original data. The newly added data is usually extracted from the benign binaries and contains many benign features. If malware classifiers see benign features are many times more than malicious ones, misclassifications inevitably occur. In the Machine Learning Security Evasion Competition 2020 [118], the winner's method is to keep adding benign sections at the end of the original sample until the model's score falls below a threshold. This seemingly simple method achieves a 100% success rate.

Regarding this issue, some works try to make the model using only malicious features for classification, thus making the appending attack invalid. Fleshman et al. [56] propose to modify the internal data of MalConv to obtain a monotonic model. A monotonic model means that the values of all input features of the model can only be increased, not decreased. In this way, if the input is modified, the score of the final output of the resulting model will only increase. If the malware classifier has this feature, if the attacker appends

a lot of benign content after the malware sample, it will only increase the score of the sample, not reduce it. Since the output of MalConv is 0 for benign and 1 for malicious, all modifications will only make the model feel that the sample is more malicious, so that adversarial examples will not be generated. Specifically, the training process of nonnegative MalConv adds a clamp operation. After each step of training, all negative numbers in weights and bias matrics of all layers are set to 0. set all the weights in the network to be non-negative after each training step. Since the input to the model is a sequence of 8-dimensional vectors, each vector is 8 integers from 0 to 255. So the inputs are all non-negative numbers. If the entire network is also non-negative inside, the final output of the network is also non-negative. Since the output of MalConv is 0 for the benign and 1 for malicious, for a benefit sample, after inputting it to the network, benefit features should not excite any neuron, so that the final output will be a small number close to 0. For a malware sample, the malicious feature will stimulate the corresponding neuron. It forces the network to focus on malicious content and does not rely on benign content. The disadvantage of this method is that its precision on clean samples is much lower than the original model. The nonnegative-malconv paper found that the model obtained after such training is very resistant to appending attacks, but its accuracy is lower than the original MalConv. The accuracy of the original MalConv is 95%, while the nonneg-MalConv is only about 89%. Such a drop in accuracy is unbearable for a malware classifier.

3.3.2 Our Insights

To sum up, our insight is that to build a robust malware classification model by design, the architecture needs to meet the following requirements:

1. **Exclude superficial features.** The new robust architecture needs to base on the intrinsic functionality features of the binary. A robust model should explicitly exclude easily mutable superficial features in the input.

2. **Shift invariance.** This model needs to achieve the characteristics of shift invariance. The attacker can reallocate the code of the malware sample to a new location through each code displacement method. A truly robust model should be shift-invariant, so as long as malicious features exist, they should be recognized by the model regardless of whether their position has changed.

3. **Only focus on malicious functions.** The new robust architecture should only focus on malicious features, and will not lead to a decrease in model accuracy. When this model is doing classification, it should only focus on the malicious features, not the benign features. Since the attacker can add benign features almost unlimitedly, if the model focuses on two aspects of features, the added more benign feature will continuously reduce the model's score.

3.4 Approach

Since we have drawn various characteristics that the malware classifier of robust should have, we will discuss how to implement it in turn.

Exclude superficial features. We choose to use functions in binary as input features. We chose to use DeepDi[180] as the disassembly tool. It uses a novel graphical representation called an "instruction flow graph" to model the different relationships between instructions, and then uses Relational-GCN to reason and classify the instruction flow graph to accurately

classify the instructions. DeepDi matches or outperforms state-of-the-art disassemblers in terms of accuracy. DeepDi is robust against binary obfuscations and adversarial attacks. Moreover, it is orders of magnitude more efficient than other methods.

Shift Invariance. We directly used function embedding as the input of Reasoner and did not use BERT's embedding layer, which contains positional embeddings. So the final output of CLS is only related to function embedding alone, not related to its position.

Only focus on malicious functions.. Nonnegative-MalConv[56] is a good attempt in this direction. It does not modify the network architecture of the original MalConv model, it only sets negative parameters to 0 after each training step. Because the input is non-negative values (from 0 to 255), all parameters in the network are also non-negative, if attackers append more bytes at the end of the original sample, it can only increase the output score. It forces the network to only pick up malicious features to do the classification. However, it causes the accuracy of the model to drop significantly compared to the original MalConv (from 95% to 89%). It shows that non-negative models are hard to train. We need a new architecture that allows models to focus only on malicious functions without hurting accuracy.

Before describing our new robust architecture for malware classification, let's look at some closely related work in the field of natural language processing. Recently, there is a very popular attention model called BERT (Bidirectional Encoder Representation from Transformers) that achieves state-of-the-art results on various NLP tasks. However, BERT cannot handle very long data because the attention mechanism occupies a large amount of video memory, resulting in the input length being limited to a maximum of 512 tokens.

Many subsequent works have tried to solve this problem. For example, Pappagari et al. [124] propose to use hierarchical-BERT, which used two different levels of BERT models to deal with this problem. The first BERT model accepts token-level input and converts the token sequence into an embedding. The BERT of the second layer receives the embedding sequence as inputs. Theoretically, it can receive the input of 512*512 tokens. Another solution is the CogLTX (Cognize Long TeXts) model proposed by Ding et al. [46]. The core insight is that not any sentence of the input is related and necessary to the classification task. For example, if a human reads a book and then answers a question related to the book, he/she usually does not need to remember every sentence in the book at the same time. Instead, in the process of reading, a human finds out the sentences related to this question and highlights them. After reading, only focus on these highlighted sentences, and then answer the questions accordingly. Train a classifier is similar. If the input text is super long, instead of feeding all inputs, we can first extract the sentences that are most relevant to the classification task in the original input to form a shorter input. This way you don't have to feed the entire input into one BERT model (called Reasoner) for classification. To find the most relevant sentences, CogLTX also needs to train a Judge model to estimate the relevance of each sentence. Since usually there is no corresponding relevance label for every sentence, they generate an occlusion-based saliency map for the Reasoner model. If a sentence is deleted from the input, and the value of Reasoner's loss increases beyond a certain threshold, that sentence is considered is important and relevant to classification tasks. Otherwise, it is considered irrelevant. In this way, a training dataset can be generated for the Judge model.

Back to the problem of malware classification. Because the length of the malware sample can be very long, usually far more than 512*512 bytes. If we directly treat each byte in the malware as a token to train a regular BERT or hierarchical BERT model, it can only handle samples up to 256 KB. We also run into the problem of too long input. Naturally, we first tried to integrate CogLTX into the hierarchical-BERT model, by training a Judge model to find the most relevant parts of the input samples. Since the method of CogLTX is orthogonal to hierarchical-BERT, it is not difficult to employ both approaches in one architecture. Next, we revisit the insights of CogLTX and go one step further. CogLTX uses the saliency map of Reasoner to select the most relevant segments for the classification task. If we only select segments that are relevant and have a positive impact on the classification score, then in the task of malware classification, we can select the most malicious segments in the malware sample. Then even if the attacker appends a lot of benign content at the end of a malicious sample, those appending benign segments will not be selected, and eventually not affect the final classification score. It shares the same idea as nonnegative-MalConv, which is to only focus on malicious features. It makes the model robust against appending attacks by design, and much easier to train.

3.4.1 Reasoner

Similar to CogLTX, our basic assumption is that "a few key malicious functions in malware samples store sufficient and necessary information for the task of malware classification". Just like security experts manually analyzing a malware sample, he also doesn't need to check every function one by one. Often they just need to find some candidate functions that contain sensitive system calls and observe what malicious behavior these func-

tions do when chained together. Formally, for a malware sample $\mathbf{x} = [f_1, f_2, f_3, f_4, ..., f_n]$ that contains n functions, we assume there exists a subset $\mathbf{s} = [f_{s_1}, f_{s_2}, f_{s_3}, f_{s_4}, ..., f_{s_N}]$ of $N(N <= n)$ functions, satisfying

$$Reasoner(\mathbf{x}) \approx Reasoner(\mathbf{s}) = sigmoid(MLP(BERT(\mathbf{s}))) \in (0, 1) \tag{3.1}$$

Training. Let us first assume that we have known each function's score, which represents the possibility that the function has a malicious behavior. We sort all functions according to the scores, and select the Top N functions to form $\mathbf{s}_{mal}$. We also create another $\mathbf{s}_{rand}$ with randomly selected functions for exploration, because only functions in $\mathbf{s}$ will be used to infer function labels in the next step. Then we create a BERT model (Reasoner) that accepts $\mathbf{s}_{mal}$ and $\mathbf{s}_{rand}$ as input, and the target label is the original label of x. We train this Reasoner until its loss stops decreasing.

Inference function label. To get the labels of functions about whether they contain malicious functionality, for each function $f \in \mathbf{s}$, we try to remove it and feed the rest functions $\mathbf{s} - f$ to Reasoner. If the Reasoner's output score decreases by more than a certain threshold, the function f will get a malicious label, otherwise, it will get a benign label. (Reasoner outputs 0 for benign and 1 for malicious.) A large reduction in the score means that the removed features can activate neurons representing malicious features in Reasoner.

3.4.2 Judge

Traing. In the previous step, we have generated input f and its corresponding label for each function we have selected as the training dataset for Judge. Then we create a simple 4-layers MLP network model (Judge). We train this Reasoner until its loss stops decreasing.

$$Judge(f) = sigmoid(MLP(f)) \in (0, 1) \tag{3.2}$$

Generate Estimation. For each function f in all samples of the dataset $\mathcal{D}$, we use the trained Judge model to estimate its maliciousness scores, which can be used in generating $\mathbf{s}_{mal}$ in the next iteration. We will alternate training the Reasoner model and the Judge model for *num_epoch* times

More details can be found in Algorithm 3

3.4.3 Workflow

The workflow of our selective hierarchical BERT is shown in Figure 3.1.

1. For the malware samples, we first use DeepDi to find all functions in each sample and generate embeddings for them ($S = \{F1, F2, F3, ...Fn\}$). (Red functions represent functions with malicious behavior.)

2. For each function embedding, use the Judge model to generate an estimated malicious score. Because the Judge model has not been trained in the beginning, this step can be skipped. the estimated maliciousness scores of all functions are all 0s.

Algorithm 3 The Training Algorithm

Input: Traing set $\mathcal{D} = [(x_0, y_0), ..., (x_n, y_n)], num\_epoch, mode, thresh$

for all $epoch \leftarrow 1$ to num_epoch **do**
  {# train Reasoner}
  for all $\mathbf{x}, y \in \mathcal{D}$ **do**
    $\mathbf{s}_{mal} \leftarrow build\_malicious\_input(\mathbf{x})$
    $\mathbf{s}_{rand} \leftarrow build\_random\_input(\mathbf{x})$
    $loss_{rand} \leftarrow BCEWithLogitsLoss(Reasoner(\mathbf{s}_{rand}), y)$
    $loss_{mal} \leftarrow BCEWithLogitsLoss(Reasoner(\mathbf{s}_{mal}), y)$
    Update Reasoner by descending $\nabla_{\varphi}(loss_{rand} + loss_{mal})$
    {# infer function label}
    for all $f \in \mathbf{s}$ **do**
      $\Delta_{score} \leftarrow Reasoner(\mathbf{s}) - Reasoner(\mathbf{s} - f)$
      if $\Delta_{score} > thresh$ **then**
        $label[f] \leftarrow 1$
      else
        $label[f] \leftarrow 0$
      end if
    end for
  end for
  {# train Judge}
  for all $\mathbf{x}, y \in \mathcal{D}$ **do**
    for all $f \in \mathbf{x}$ **do**
      if $f \in label$ **then**
        $loss = MSELoss(f, label[f])$
        Update Judge by descending $\nabla_{\theta}(loss)$
      end if
    end for
  end for
  {# generate estimation}
  for all $\mathbf{x}, y \in \mathcal{D}$ **do**
    for all $f \in \mathbf{x}$ **do**
      $f.estimation = Judge(f)$
    end for
  end for
end for

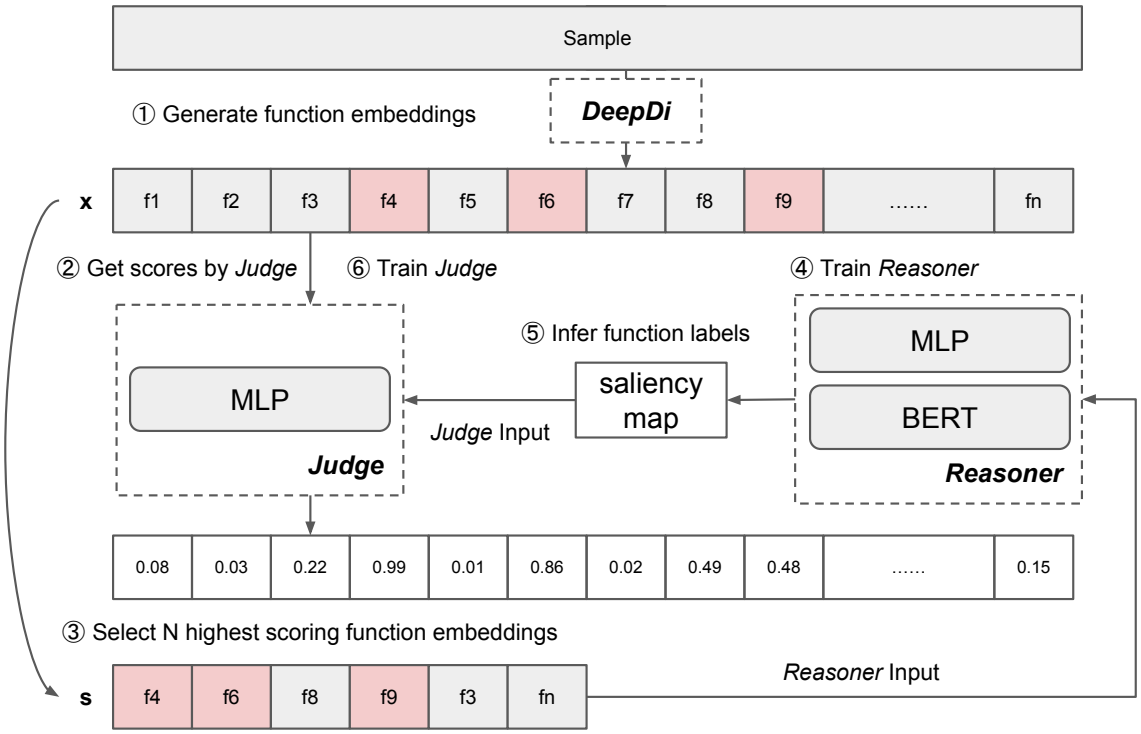


Figure 3.1: Workflow

3. The Top N functions with the highest malicious scores are chosen to form the input $S^{+} = \{F1^{'}, F2^{'}, F3^{'}, ...Fn^{'}\}$ for each sample. (In the beginning, since all functions have malicious scores of 0, it actually chooses N functions at random.)

4. Train the Reasoner model. The input to the model is S^{+} and the corresponding labels are the labels of the original samples S (1 for malicious, 0 for the opposite).

5. After the training of Reasoner is complete, we infer function labels (1 for functions with malicious functionality, 0 for the opposite) by intervening: testing whether a function has a positive effect on the classification score. For each function $Fx^{'}$, we try to remove it from S^{+}. If the Reasoner's score drops more than the threshold T after $Fx^{'}$ is removed, $Fx^{'}$ is considered to contain some malicious behavior. its label

is assigned 1. Otherwise, if the score remains the same or even increases, its label is assigned 0.

6. We use the training dataset generated in the previous step to train the Judge model.

7. Go to step 2). This time we start to use the Judge model to generate estimated maliciousness scores for all functions.

3.5 Evaluation

3.5.1 Experiment Setup

In this section, we first describe the experiment setup, the dataset collection . Section 3.5.2 provides details about training. In the end, we present the evaluation results with respect to accuracy, robustness.

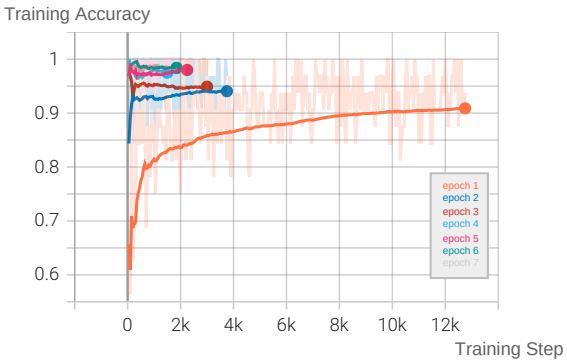
Dataset: In the experiments in this chapter, we use a dataset containing 10,000 samples, including 5,000 malware samples collected by VirusTotal in the last two years, and 5,000 benign samples from CUM.

Setup: The experiments are performed on a machines with Intel(R) Core(TM) i9-10850K CPU @ 3.60GHz, 64GB RAM, GeForce RTX 2080 Ti. The deep neural network models are all implemented using the open-source deep learning framework PyTorch [136]. The remaining codes of data processing, statistics are programmed in Python.

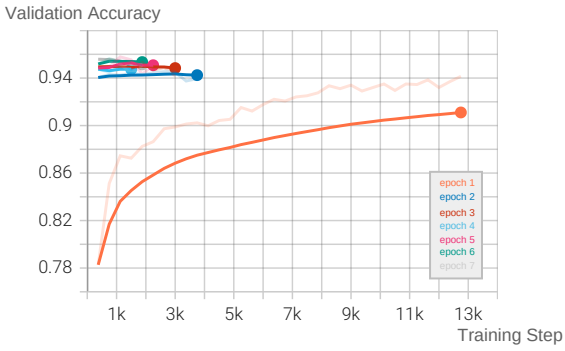
3.5.2 Model Training

Figure 3.2 (a) and (c) show the training accuracy of the Reasoner model and the Judge model. The lowermost orange curve in (a) shows the training accuracy changes of Reasoner at epoch 1. Each epoch stops early when the validation accuracy stops increasing by 3 steps. The lowermost red curve in (b) shows the training accuracy changes of Judge at epoch 1. Since the beginning, no function has estimated maliciousness score. At this time, we can only randomly select functions for input $\mathbf{s}$, instead of selecting the functions with the best score. Therefore, Reasoner can only achieve 91% accuracy at the highest. At the same time, Judge does not have enough function labels, and its highest accuracy can only reach 79%. As the training of Reasoner and Judge alternates, the two are also promoting each other's training, and the accuracy is gradually improving. In the end, Reasoner achieved 98% accuracy, and Judge achieved 89% accuracy. Figure 3.2 (b) shows that after training, Reasoner's accuracy on the validation dataset reached 95.5%, similar to the state-of-the-art models, such as MalConv.

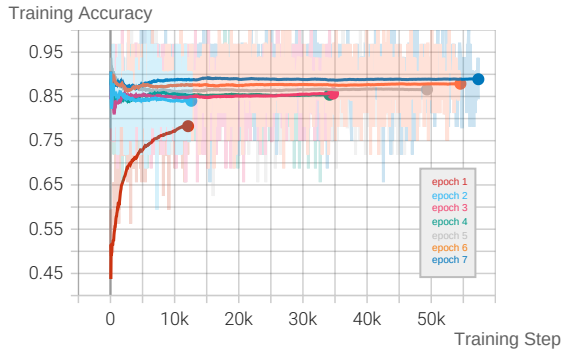
Figure 3.3 shows the distribution of scores for all functions in the dataset. Overall, 6.65% of all functions are considered by Judge to contain malicious behavior. Breaking down by datasets, 84% of the functions in the malware dataset were marked as malicious by Judge. Only 7% of functions in the benign software dataset were marked as malicious by Judge. The functions of benign software may also invoke sensitivity system calls, and Reasoner will comprehensively determine whether these selected suspicious functions indeed conduct malicious activities.



(a) Training Accuracy of Reasoner



(b) Validation Accuracy of Reasoner



(c) Training Accuracy of Judge

Figure 3.2: Training of Reasoner

3.5.3 Robustness under Code Randomization Attack

To verify the robustness of our model against binary randomization attacks, we randomly sample 300 malware samples from the test data and use a method called Malware Makeover[1] to randomize these malware samples for 10 iterations, including in-place randomization and code displacement. Figure 3.4 shows the number of bytes changed of each sample during randomization. On average, each sample is changed by more than 21000

[1] `https://github.com/pwwl/enhanced-binary-diversification`

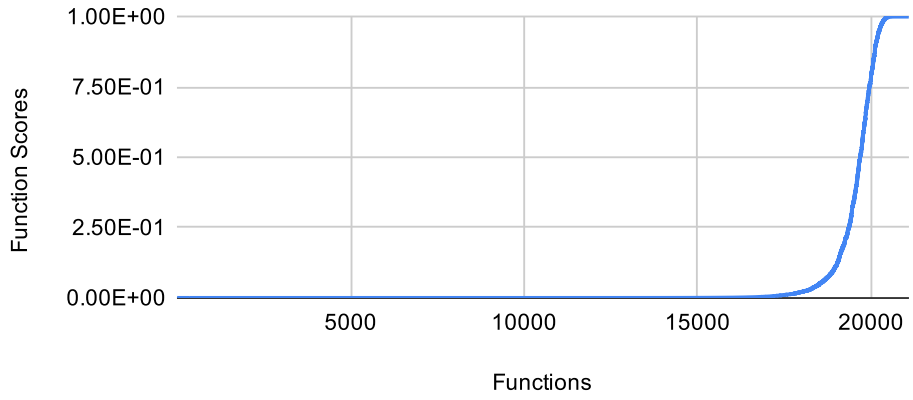


Figure 3.3: Distribution of Function Scores

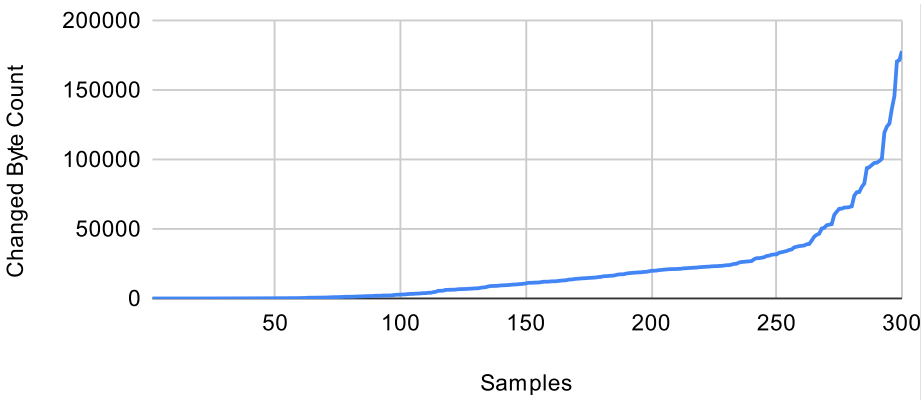


Figure 3.4: Changed Byte Count

bytes.

Table 3.1: Detection Rate of Original and Randomized Malware.

Classifier	**Detection Rate**	
	Original	Randomized
Selective Hierachical-BERT	99.28%	99.28%

From Table 3.1, we can see that even after 10 iterations of binary randomization, the detection rate of our selective hierarchical-BERT model is still as high as 99.28%. And its accuracy on the original sample is not compromised.

3.5.4 Robustness under MAB-Malware Attack

Table 3.2: Accuracy on Original Samples and Robustness against MAB-Malware.

Classifier	**Accuracy on Original**	**Attack Succ Rate**
EMBER	97.2%	75.94%
MalConv	97.1%	97.72%
Nonneg-MalConv	85.45%	0.22%
Adversarial Trained MalConv	88.4%	67.0%
Selective Hierachical-BERT	95.8%	0%

We still use the MAB-Malware framework proposed in our last chapter to attack various models including selective Hierarchical-BERT. Table 3.2 shows that the original MalConv is very vulnerable, and our attack success rate can reach 97.72%. After adversarial training with adversarial examples, our attack success rate can still reach 67%. It can be seen that due to the different constraints of the attack, the attack can append almost unlimited data, and adversarial training cannot effectively improve the robustness of MalConv, and it also affects the model's accuracy on original samples.

For our Selective Hierarchical-BERT model, if we directly use MAB-Malware to attack it, the attack success rate is 0%. Because most actions in MAB-Malware will not affect the function embeddings. To find the upper bound of the robustness of our model, we conduct a gray-box attack on our model. We assume attackers 1) can get embeddings of all functions. 2) know the output scores of both the Reasoner and Judge, 3) can inject as many functions as possible, and 4) the injection will not break the functionality of the original malware sample. The process of grey box attack is as follows: 1) Collect a large number of functions in benign binaries, and get their embeddings, 2) Get scores for benign functions through Judge, and only keep high-scoring functions, 3) Choose a malware sample and add

a high-scoring function benign to it, 4) Observe whether the score of Reasoner decreases, if so, keep it, otherwise discard it, 5) Use multi-armed bandit to modeling function selection, use reinforcement learning to select the next best benign function. 6) Go back to step 3 until the malware sample is misclassified.

Table 3.3: Robustness against Graybox Attack.

Classifier	**Attack Succ Rate**
Selective Hierachical-BERT	2.0%

As shown in Figure 3.3, under this attack, the upper bound of the attack success rate on our selective hierarchical BERT model is 2%, which is much lower than adversarial trained MalConv, and the accuracy of the original samples is not affected.

3.6 Conclusion

In this chapter, we explore in detail the different constraints of malware classifiers and the properties that need to be satisfied to build a robust classifier. On this basis, a new robust deep neural network architecture based on selective hierarchical BERT is proposed to automatically select malicious functions for malware classification, which is robust to different attacks by design. Compared with other baselines, our model can handle very large samples and can automatically select essential features for malware classification, which fundamentally improves the robustness of the model without sacrificing the accuracy of the model.

Chapter 4

A Robust Memory Forensics Framework: DeepMem

4.1 Abstract

Kernel data structure detection is an important task in memory forensics that aims at identifying semantically important kernel data structures from raw memory dumps. It is primarily used to collect evidence of malicious or criminal behaviors. Existing approaches have several limitations: 1) list-traversal approaches are vulnerable to DKOM attacks, 2) robust signature-based approaches are not scalable or efficient, because it needs to search the entire memory snapshot for one kind of objects using one signature, and 3) both list-traversal and signature-based approaches all heavily rely on domain knowledge of operating system. Based on the limitations, we propose DeepMem, a graph-based deep learning approach to automatically generate abstract representations for kernel objects, with which we

could recognize the objects from raw memory dumps in a fast and robust way. Specifically, we implement 1) a novel memory graph model that reconstructs the content and topology information of memory dumps, 2) a graph neural network architecture to embed the nodes in the memory graph, and 3) an object detection method that cross-validates the evidence collected from different parts of objects. Experiments show that DeepMem achieves high precision and recall rate in identify kernel objects from raw memory dumps. Also, the detection strategy is fast and scalable by using the intermediate memory graph representation. Moreover, DeepMem is robust against attack scenarios, like pool tag manipulation and DKOM process hiding.

4.2 Introduction

Generally speaking, the existing memory forensic tools fall into two categories: signature scanning and data structure traversal, all based on certain rules (or constraints), either on values, points-to relations, or both. Signature scanning tools (e.g., `psscan`) in Volatility [172] rely only on value constraints on certain fields to identify memory objects in the OS kernel, whereas SigGraph [101] relies on points-to relations as constraints to scan kernel objects. Data structure traversal tools (e.g., `pslist`) in Volatility and KOP [18] start from a root object in a known location, traverse its pointers to discover more objects, and further traverse pointers in the discovered objects to reach more objects. However, there exist several intertwining challenges in the existing rule-based memory forensic analysis:

(1) **Expert knowledge needed**. To create signatures or traversing rules, one needs to have expert knowledge on the related data structures. For a closed-source operating

system (like Windows), obtaining such knowledge is nontrivial if not impossible.

(2) **Lack of robustness.** Attackers may directly manipulate data and pointer values in kernel objects to evade detection, which is known as DKOM (Direct Kernel Object Manipulation) attacks [47]. In this adversarial setting, it becomes even more challenging to create signatures and traversing rules that cannot be easily violated by malicious manipulations, system updates, and random noise.

(3) **Low efficiency.** High efficiency is often contradictory to high robustness. For example, an efficient signature scan tool (like `psscan`) simply skips large memory regions that are unlikely to have the relevant objects (like `_EPROCESS`) and relies on simple but easily tamperable string constants as constraints. In contrast, a robust signature scan tool would have to scan every single byte and rely on more sophisticated constraints (such as value ranges, points-to relations) that are more computation-intensive to check.

In this work, we are inspired by the successful adoption of deep learning in many domains, such as computer vision, voice, text, and social networks. We treat this memory object recognition problem as a deep learning problem. Instead of specifying deterministic rules for a signature scan and data structure traversal, we aim to learn a deep neural network model to automatically recognize memory objects from raw memory dumps. Since the model is trained in an end-to-end manner, no expert knowledge is required. The learned deep neural network model is also more robust than rule-based search schemes because it comprehensively evaluates all memory bytes and thus can tolerate perturbations to some extent. A deep neural network model also excels in efficiency, as vector and matrix computations can be largely parallelized in modern GPUs.

More specifically, in order to take into account adjacency relations between data fields within an object as well as points-to relations between two objects, we choose to build a graph neural network model [151], in which each node represents a segment of contiguous data values between two pointers, and each directed edge represents an adjacency relation or a points-to relation between two nodes. We then conduct supervised learning on this model: we collect a large number of diverse memory dumps, and label the objects in them using existing memory forensic tools like Volatility, and train the classification model using this labeled dataset.

We implement a prototype called DeepMem and conduct the extensive evaluation with respect to accuracy, efficiency, and robustness. Experimental results show that it achieves high precision and recall rate at above 99.5% for important kernel objects, like `_EProcess` and `_EThread`. For efficiency, it scans a memory dump of 1GB in size only once to build the memory graph in about 80 seconds. Then, for each type of object, the detection time is about 13 seconds per type on a moderate desktop computer (Core i7-6700, 16GB RAM, and no GPU). Moreover, in the attack scenarios, like pool tag manipulation and DKOM process hiding, signature-based memory forensics tool (e.g. Volatility) fail to correctly report kernel objects while DeepMem can tolerate those attacks.

In summary, the contributions of this chapter are as follows:

- **A graph representation of raw memory.** We devise a graph representation for a sequence of bytes, taking into account both adjacency and points-to relations, to better model the topological information in memory dumps.
- **A graph neural network architecture.** We propose a graph-based deep learn-

ing architecture with two jointly-trained networks: embedding network and classifier network. This deep neural network architecture captures both internal patterns of memory bytes as well as topological information in the memory graph and infers node properties in the graph.

- **A weighted voting scheme for object detection.** We propose a weighted voting scheme for object detection, which summarizes and cross-validates the evidence collected from multiple parts of an object to infer its location and type.

The remainder of this chapter is structured as follows. Section 4.3 provides a background of memory object detection. Section 4.4 gives an overview of the DeepMem, followed by design details of each component. Section 4.5 presents implementation details and evaluation results.

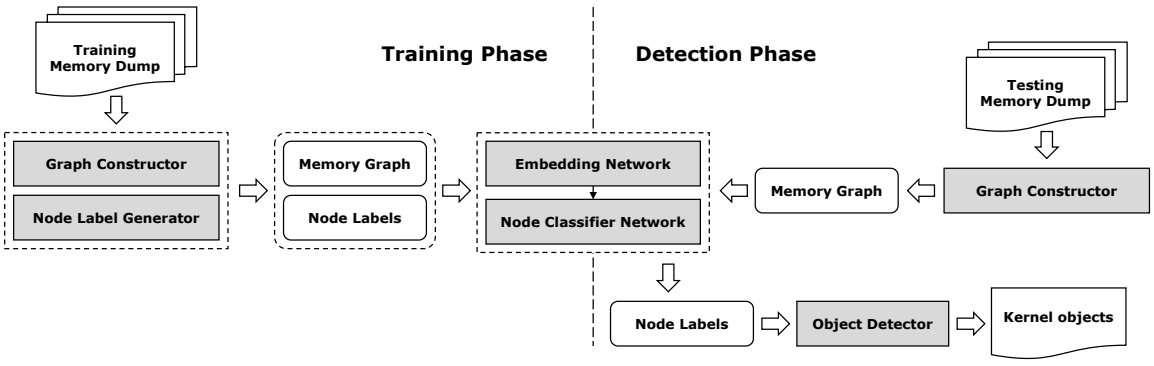


Figure 4.1: The overview of the DeepMem architecture

4.3 Memory Object Detection

In this section, we first give a formal problem statement for memory object detection, and then describe the existing techniques and their limitations.

4.3.1 Problem Statement

If we treat a memory dump as a sequence of bytes, an object in this dump are treated as a sub-sequence in this memory dump. Naturally, we can define the object detection problem as a sub-sequence labeling problem in a large sequence.

Our goal is to search and identify kernel objects in raw memory images dumped from running operating systems. Let $C = \{c_1, c_2, ...\}$ be the set of kernel data structure types in operating system. Given a raw memory dump as input, the output is defined as a set of kernel objects $O = \{o_1, o_2, ...\}$, where each object in the set is denoted as a pair $o_i = (addr_i, c_i), c_i \in C$. Here, $addr_i$ is the address of the first byte of the object in kernel space, and c_i is the type of the kernel object.

We would like to achieve the following goals:

- **No reliance on source code.** Unlike MAS [36] and KOP [18], which rely on the kernel source code to compute a complete kernel object graph, we do not assume the access to such information. Instead, we resort to learn from real memory dumps.

- **Automatic feature selection.** We do not rely on human experts to define signatures or traversing rules for various kernel objects. We aim to automatically learn a detection model in an end-to-end manner.

- **High robustness.** Our method should tolerate content and pointer manipulation of attackers in DKOM attacks.

- **High efficiency.** We would like to design a scanning approach to examine every byte in the memory, and at the same time, achieve high efficiency and scalability.

4.3.2 Existing Techniques

There are two approaches to utilize the knowledge of data structures for memory analysis.

The first one is data structure traversal. We can first identify a root object based on the data structure definition and then follows the pointers defined in this object to find more objects. In particular, Volatility [172], a well-known memory forensic tool, provides a set of tools for listing running processes, modules, threads, network connections, by traversing the relevant data structures. Since data structure definitions in C/C++ are often vague and incomplete (due to the presence of generic pointers), the completeness of this approach is affected. To address this problem, KOP [18] and MAS [36] perform points-to analysis on the C/C++ source code to resolve the concrete types for the generic pointers, and thus produce complete data structure definitions. This approach is efficient (as we can quickly find more objects by just following pointers), but not robust because attackers may modify the pointers to hide important objects, known as Direct Kernel Object Manipulation (DKOM) attacks.

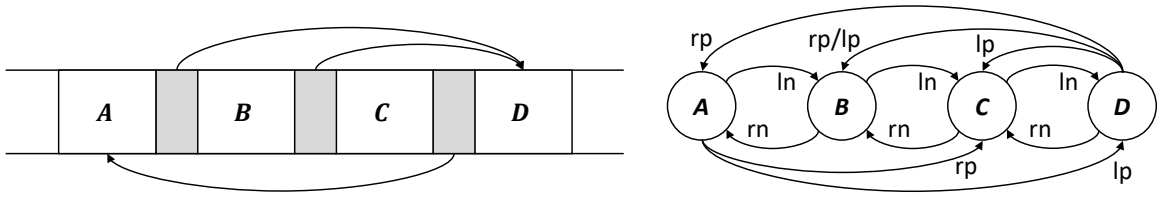
The second approach is signature scan. We can scan the entire memory snapshot for objects that satisfy a unique pattern (called signature). Volatility [172] provides a set of scan tools as well to scan for processes, modules, etc. To improve search accuracy, SigGraph [101] automatically constructs graph-like signatures by taking into account points-to relations in data structure definitions, at the price of even lower search efficiency. In general, the signature scan is more resilient against DKOM attacks, because it does not depend so much on pointers. However, it is very inefficient and not scalable, because it has

to search the entire memory snapshot for one kind of objects using one signature. To further improve the robustness of signatures, Dolan-Gavitt et al. [50] propose to perform fuzz testing to mutate each data structure field and eliminate from the signature the constraints that can be easily violated by attackers. However, this will likely lead to the increase of false positives.

Both data structure traversal and signature scan require precise knowledge of data structures and also heavily depend on specific versions of the software or the operating system, because data structures change from one version to another. Therefore, to use these tools, a data profile must be extracted from each unique operating system version, which is clearly not convenient or scalable. To address this problem, researchers propose to reuse the code already existed in the memory snapshot to interpret the memory snapshot itself [58, 49, 149]. These techniques avoid creating data profiles and implementing traversal algorithms, but they still heavily rely on the knowledge of specific operating systems to understand what code to reuse and how to reuse the code. Moreover, this approach is still subject to DKOM attacks. In terms of efficiency, code reuse is better than signature scan, but worse than data structure traversal.

4.3.3 Our Insight

We believe that the bottleneck for these memory analysis approaches is the rule-based search scheme. They search and traverse memory objects based on pre-defined rules. The rules can be hard to construct in the first place, and moreover, the rules cannot easily adapt to an unknown operating system and a new version and tolerate malicious attackers that attempt to deliberately violate these rules. To address these limitations, a "learning"



(a) Raw Memory (b) Memory Graph

Figure 4.2: Generate a memory graph from raw memory

ability becomes essential. A new memory analysis approach should automatically learn the intrinsic features of an object that are stable across operating system versions and resilient against malicious modifications, and at the same time is able to detect these objects in a scalable manner. In this work, we resort to deep learning to tackle this problem.

4.4 Design of DeepMem

In this section, we first present an overview of DeepMem, and then delve into three important components respectively.

4.4.1 Overview

Figure 4.1 illustrates the overview of DeepMem. Generally speaking, we divide DeepMem into two separate stages: training and detection.

Training Stage

In this stage, DeepMem automatically learns the representation of kernel objects from raw bytes. First, memory dumps are fed into a graph constructor to generate a graph for each memory dump (which is called "memory graph"), where each node is a segment

between two pointers, and each edge represents either an adjacency relation or a points-to relation between two nodes.

Second, a node label generator will assign a label for each node in the memory graph. We can use any existing tools (such as Volatility [172], or dynamic binary analysis tool DECAF [68]) for this purpose. This seems a little contradictory: we rely on an existing analysis tool to build a new analysis tool. This is reasonable because the existing tool only serves as an offline training purpose, so it does not need to be efficient and robust. It only needs to have reasonable accuracy in terms of labeling. After training, our detection model is expected to achieve good efficiency, robustness, and accuracy simultaneously.

Third, a memory graph is fed into a graph neural network architecture. By propagating information from neighboring nodes after several iterations, this graph neural network carries a latent numeric vector (called embedding) for each node in the memory graph.

Finally, all nodes' embedding vectors will go through a neural network classifier to get the predicted labels. The predicted labels will be compared with the expected labels to compute the loss of the classifier and update the weights of our neural network.

Detection Stage

In this stage, DeepMem accepts an unlabeled raw memory dump and detects kernel objects inside it. First, it follows the same procedure to generate a memory graph for this memory dump. Second, the memory graph is fed into the Graph Neural Network (GNN) model obtained from the training stage to generate embeddings of all the nodes and then predict node labels using the neural network classifier. At last, DeepMem performs an object detection process. This is because the labels predicted from the last step are

for segments, and an object may consist of one or several segments. Therefore, the object detection process takes segment labels as input and uses a voting mechanism to detect objects, for which most of their segment labels agree upon the same object label.

In the remainder of this chapter, we will discuss the definition of memory graph and its construction in Section 4.4.2, the graph neural network model for computing memory segments' embeddings as well as the segment classification network in Section 4.4.3, and object detection scheme in Section 4.4.4.

4.4.2 Memory Graph

A memory graph is a directed graph $G = (N, E_{ln}, E_{rn}, E_{lp}, E_{rp})$, where:

- N is a node set, and each $n \in N$ represents a segment of contiguous memory bytes between two pointer fields.
- E_{ln} is an edge set, and each $e \in E$ represents a directed edge from n_i to n_j, and n_i is left neighbor of n_j.
- E_{rn} is an edge set, and each $e \in E$ represents a directed edge from n_i to n_j, and n_i is right neighbor of n_j.
- E_{lp} is an edge set, and each $e \in E$ represents a directed edge from n_i to n_j, and n_i is pointed by a pointer on the left boundary of n_j.
- E_{rp} is an edge set, and each $e \in E$ represents a directed edge from n_i to n_j, and n_i is pointed by a pointer on the right boundary of n_j.

In other words, a memory graph is a directed graph with four sets of edges, which capture both the adjacency and points-to relations of memory segments, on both left-hand-side and right-hand-side of each segment.

Figure 4.2 illustrates an example of how to construct a memory graph from raw memory. Figure 4.2(a) shows a part of raw memory, in which three pointer fields split this part of memory into four segments: A, B, C, and D, each of which may have one or more contiguous memory bytes. As a result, A, B, C, and D become vertices in the corresponding memory graph. These vertices are connected by four kinds of edges. For instance, since A is the left neighbor of B, we have $A \xrightarrow{\text{ln}} B$. Conversely, since B is the right neighbor of A, we have $B \xrightarrow{\text{rn}} A$. Moreover, since the pointer field left to C points to D, and the pointer field right to C points to A, we then have $D \xrightarrow{\text{lp}} C$ and $A \xrightarrow{\text{rp}} C$. Note that these two edges are reverse to the actual points-to directions. This is because an edge in the memory graph represents an information flow. For instance, the pointer field left to C points to D, which means determining D's label can help label C. Therefore, from the information flow point of view, there is an edge from D to C.

A special case is that there are multiple consecutive pointers. Assume there are two consecutive pointers between C and D, pointing to A and B respectively, we then create four edges $A \xrightarrow{\text{rp}} C$, $B \xrightarrow{\text{rp}} C$, $A \xrightarrow{\text{lp}} D$ and $B \xrightarrow{\text{lp}} D$.

A careful reader might suggest adding edges for the points-to directions as well. For instance, the pointer field left to C points to D, and it might make sense to have $C \rightarrow D$, because identifying C also helps to identify D. We choose not to do so, because an adversary can easily create a pointer in an arbitrary address outside of a kernel object and

make it point to the object, then the topology of the object in memory graph is changed if we add edges for point-to directions. This will adversely affect the detection. On the other hand, compared to the above case, it is more difficult to create a fake pointer or manipulate an existing pointer within a legitimate object that he/she tries to hide, without causing system crashes or other issues.

4.4.3 Graph Neural Network Model

The GNN (Graph Neural Network) model will accept the memory graph generated in Section 4.4.2 as input, and then output the labels of all nodes in the graph. The goal of the GNN model is to first extract a low-dimensional internal representation of nodes from raw bytes of a memory dump, and then infer the properties of nodes. As such, the GNN model should consist of two consecutive subtasks: a representation learning task and an inference task.

We represent the GNN model as $\mathcal{F}$. It consists of two jointly-trained subnetworks. The first subnetwork is an embedding network which is responsible for node representation abstraction. We denote it as ϕ_{w_1}. The second subnetwork is a classifier network, which is responsible for node label inference. We denote it as ψ_{w_2}. The formal definition of $\mathcal{F}$ is defined as follows.

$$\mathcal{F} = \psi_{w_2}(\phi_{w_1}(\cdot)) \tag{4.1}$$

The input of the embedding network ϕ_{w_1} is a vector representation of a node, denoted as $\boldsymbol{v}_n$, and the output is embedding vector, denoted as $\boldsymbol{\mu}_n$. The classifier network

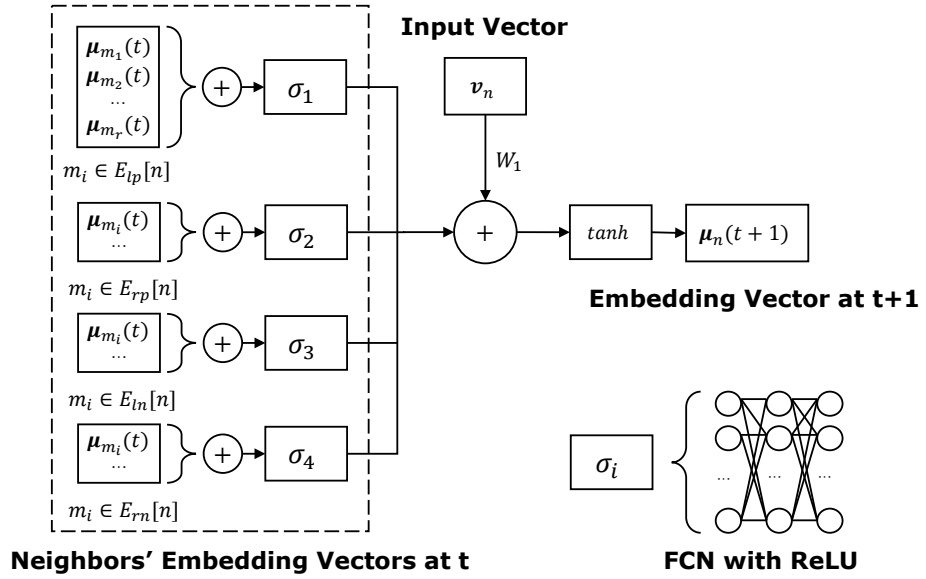


Figure 4.3: Node embedding computation in each iteration

ψ_{w_2} takes the output of the embedding network as input, and then output the node label, denoted as $\boldsymbol{y}_n$.

More specifically, let $\boldsymbol{v}_n$ be a d-dimensional vector of node n derived from its actual memory content, then the embedding vector $\boldsymbol{\mu}_n$ is computed as follows:

$$\boldsymbol{\mu}_n = \phi_{w_1}(\boldsymbol{v}_n, \boldsymbol{\mu}_{E_{ln}[n]}, \boldsymbol{\mu}_{E_{rn}[n]}, \boldsymbol{\mu}_{E_{lp}[n]}, \boldsymbol{\mu}_{E_{rp}[n]}) \tag{4.2}$$

In other words, each node's embedding is computed from its actual content and the embeddings of its four kinds of neighboring nodes. We use a simple method to derive a d-dimensional vector for each node: we treat each dimension as one memory byte. If this memory segment is longer than d bytes, we truncate it and only keep d bytes; if it is shorter than d bytes, we fill the remaining bytes with 0.

Then the output vector $\boldsymbol{y}_n$ is computed as follows.

$$\boldsymbol{y}_n = \psi_{w_2}(\boldsymbol{\mu}_n) \tag{4.3}$$

In the following paragraphs, we will describe how embedding network and classifier network are defined and how they work.

Embedding Network

For each node n in the memory graph G, the embedding network ϕ_{w_1} integrates input vector $\boldsymbol{v}_n$ and the topological information from its neighbors, both adjacent neighbors and point-to neighbors, into a single embedding vector $\boldsymbol{\mu}_n$.

Inspired by Scarselli et al. [151], we implement the embedding vector as a state vector that gradually absorbs information propagated from multiple sources over time. To add a time variable into embedding vector computation, we transform Equation (4.2) into Equation (4.4). The total iterations needed to calculate the embedding vector is denoted as T. The embedding vector of time $t+1$ depends on the neighbor embedding vectors at time t, as shown in Figure 4.3.

$$\begin{aligned}\boldsymbol{\mu}_n(t+1) = \phi_{w_1}(\boldsymbol{v}_n, \boldsymbol{\mu}_{E_{ln}[n]}(t), \boldsymbol{\mu}_{E_{rn}[n]}(t), \\ \boldsymbol{\mu}_{E_{lp}[n]}(t), \boldsymbol{\mu}_{E_{rp}[n]}(t))\end{aligned} \tag{4.4}$$

For each node n, the embedding network collects the information about neighbor nodes in a BFS (Breadth First Search) fashion. In each iteration, it traverses one layer of neighbor nodes and integrates the neighbors' states into the state vector $\boldsymbol{\mu}_n$ of node n.

We name the neighbors expanded in the first layer as 1-hop neighbors, in the same way, the neighbors expanded in the k-th layer as k-hop neighbors. In each layer expansion, we collect information from four types of neighbors, which are left neighbor, right neighbor, left pointer neighbor and right pointer neighbor. The more iterations we run, the information of farther neighbors are collected into embedding vector $\boldsymbol{\mu}_n$. At time $t = T$, $\boldsymbol{\mu}_n(t)$ stores the information of the node sequence n itself and the information of neighbor nodes within T hops.

We implement embedding vector $\boldsymbol{\mu}_n$ as Equation (4.5).

$$\boldsymbol{\mu}_n(t+1) = tanh(W_1 \cdot \boldsymbol{v}_n + \beta(n,t)) \tag{4.5}$$

$$\begin{aligned} \beta(n,t) =& \sigma_1(\sum_{m \in E_{pt}[n]} \boldsymbol{\mu}_m(t)) + \sigma_2(\sum_{m \in E_{rn}[n]} \boldsymbol{\mu}_m(t)) + \\ & \sigma_3(\sum_{m \in E_{lp}[n]} \boldsymbol{\mu}_m(t)) + \sigma_4(\sum_{m \in E_{rp}[n]} \boldsymbol{\mu}_m(t)) \end{aligned} \tag{4.6}$$

The weight matrix W_1 is the weight parameters of the node content, which is a matrix of shape $|\boldsymbol{\mu}| \times d$. Neighbor state weight parameters are a set of weight matrices in multiple layered neural networks. Note that there are four separate sets of weight matrices for σ_1, and σ_2, and σ_3, and σ_4, such that the embeddings of different kinds of neighbors are propagated differently. The architecture of each σ network is a feed-forward neural network, each layer is a fully connected layer with ReLU activation function. The pseudo code of embedding network is shown in Algorithm 4.

Algorithm 4 Information Propagation Algorithm of Embedding Network ϕ_{w_1}

Input: Memory Graph $G = (N, E_{ln}, E_{rn}, E_{lp}, E_{rp})$, iteration time T
Output: Graph Embedding $\boldsymbol{\mu}_n$ for all $n \in N$

Initialize $\boldsymbol{\mu}_n(0) = 0$, for each $n \in N$
for all $t \leftarrow 1$ to T **do**
  for all $n \in N$ **do**
    $\beta = \sigma_1(\sum_{m \in E_{rn}[n]} \boldsymbol{\mu}_m(t-1))$
    $\beta += \sigma_2(\sum_{m \in E_{ln}[n]} \boldsymbol{\mu}_m(t-1))$
    $\beta += \sigma_3(\sum_{m \in E_{lp}[n]} \boldsymbol{\mu}_m(t-1))$
    $\beta += \sigma_4(\sum_{m \in E_{rp}[n]} \boldsymbol{\mu}_m(t-1))$
    $\boldsymbol{\mu}_n(t) = tanh(W_1 \cdot \boldsymbol{v}_n + \beta)$
  end for
end for

All of the mentioned weight parameters of embedding network are learned using supervised learning on a labeled training dataset. Since the weights are learned jointly with the weights in the classifier network, we will leave the training details after introducing the classifier network in the section below. The embedding vector obtained in this section is just an intermediate representation of the whole supervised training. To perform an end-to-end training from raw bytes to labels, we need the classifier network to generate the final node label for training.

Classifier Network

Let l be a node label, and L be the set of all node labels. Node classifier network is used to map embedding vector to a node label: $\psi_{w_2} : \boldsymbol{\mu}_n \rightarrow l$, where $n \in N, l \in L$.

In order to facilitate object detection that will be discussed in Section 4.4.4, we choose to label each node as a 3-tuple of the object type, offset and length. For example, a node with label T_16_24 means the node is part of a `_ETHREAD` object and it is located at offset 16 from the beginning of the `_ETHREAD` object, the length of it is 24 bytes. As

illustrated in Figure 4.4, three nodes are labeled as T_16_24, T_52_12 and T_84_28. These labels all agree upon a single fact that a _ETHREAD object is located at the same address. Similar labeling methods are adopted in the linguistics domain to solve word segmentation tasks [182, 183]. In particular, they label the characters at the start, in-between and at the end of a word, in order to split words from streams of free texts.

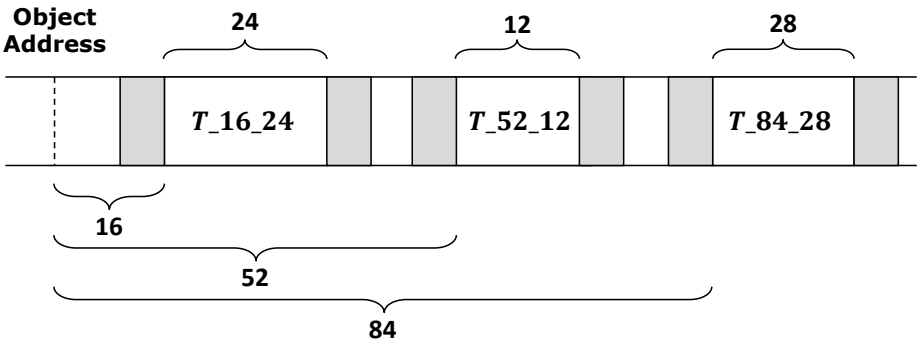


Figure 4.4: Node Labeling of a _ETHREAD Object

An object type may have many node labels. However, some rare and invariant node labels have low occurrences in type c. To get a robust model, we should not fit these outliers node labels. Hence, we just keep the node labels with high frequency in type c, denoted as key node label set $L(c)$. The node labeling method is described in detail in experiment evaluation Section 4.5.2.

With node labels of each object type, we then build a multi-class classifier to classify the nodes into one of the labels in that object type. For example, there will be a _ETHREAD classifier, a _EPROCESS classifier, etc. The node classifier takes an embedding vector $\boldsymbol{\mu}_n$ as input and produces a predicted node label as output. To implement the classifier, we choose to use FCN (Fully Connected Network) model that has multi-layered hidden neurons with ReLU activation functions, following by a softmax layer.

After introducing the embedding network ϕ_{w_1} and the classifier network ψ_{w_2}, we will show how to train them together. During training, training samples are fed into the embedding network for contextual information collection. After propagating several iterations, the final embedding vectors are fed into the classifier network to generate the predicted output labels. To train the weights in the GNN model, we compute the cross-entropy loss between the predicted label and annotated label, and update weights in the process of minimizing the loss.

We adopt the BP (Back Propagation) [152] strategy to pass the loss error from output layer back to previous layers to update the weights along the way. In the next loop of training, the classification is performed using newly-updated weights. After several training loops, the loss will stabilize to a small value and the model is fully trained. Specifically, we use Adam (Adaptive Moment Estimation) [83] algorithm, a specific implementation of BP strategy, as the weight parameter optimizer of the GNN deep model.

Formally, let training dataset $D = \{d_1, d_2, ...\}$ be a set of node samples, where each sample $d_i = (\boldsymbol{v}^{(i)}, \boldsymbol{y}^{(i)})$ is a pair of node vector and associated node label vector. The optimization goal is to compute the solution to Equation (4.7). $\mathcal{L}$ is the cross-entropy loss function that estimates the differences between classifier outputs and annotated labels. The parameters of embedding network w_1(including weights of $W_1, \sigma_1,\ \sigma_2,\ \sigma_3,\ \sigma_4$) and parameters of classifier network w_2 are updated and optimized in training.

$$\arg\min_{w_1,w_2} \sum_{i=1}^{|D|} \mathcal{L}(\boldsymbol{y}^{(i)}, \mathcal{F}(\boldsymbol{v}^{(i)}) \tag{4.7}$$

4.4.4 Object Detection

The basic idea behind object detection is that if several nodes indicate that there exists an object of certain type c at the same address s in the memory dump, we can conclude with a high confidence that we have detected an object of type c at that address, $c \in C$. Thus, a node label can be considered as a voter that votes for the presence of an object. For example, a node with a T_16_24 label means the node votes for the address, 16 bytes before the node address, to be the address of a `_ETHREAD` object. Each node in the memory indicates the presence of an object. Thus with all the node labels, we can generate a set of candidate object addresses $S = \{s_1, s_2, ...\}$ and corresponding voters for each address.

Next, we need to determine whether an address $s \in S$ is indeed a start address of an object. Ideally, if all the key nodes of type c vote for s to be an object of type c, for example T_16_24, T_52_12, T_84_28... all suggest the presence of an `_ETHREAD` at the same address s, we can confidently report a `_ETHREAD` object is detected at s. It is also likely that only a fraction of the key node labels votes for address s, then our confidence to report address s will be lower. We use $L(s, c)$ to denoted the voter set, which is all the key node labels of type c that vote for address s.

Specifically, we design a weighted voting mechanism. It gives different node labels (or in other words voters) different vote weights. Since the voter with higher frequency in a certain object type better indicates the presence of the objects of that type, and thus is assigned with a larger weight. The weights are calculated from a large real-world labeled dataset.

Finally, we introduce the prediction function $f(s, c)$ in Equation (4.8). It measures the difference between the prediction confidence and a pre-defined threshold δ. When the value of $f(s, c)$ exceeds the threshold, we draw a conclusion that an object with type c is detected at address s.

$$f(s,c) = \begin{cases} 1, & \sum_{l_i \in L(s,c)} \frac{\rho(c,l_i)}{\rho(c)} + \gamma(s,c)) > \delta \\ 0, & otherwise \end{cases} \tag{4.8}$$

Here, ρ is a counting function, $\rho(c)$ counts the number of objects of type c in the dataset, and $\rho(c, l)$ counts the number of objects of type c that has node label l in the dataset, $l \in L(c)$. Then, we divide $\rho(c, l)$ by $\rho(c)$ to estimate the weights of node label l in predicting objects of type c , which is a decimal value in $(0, 1]$. Since the weight values of voters range in $(0, 1]$, it is possible that weighted combination of multiple small-weighted voters is less than that of a large-weighted single voter (e.g. weight sum of two small voters $0.4 + 0.3$ ¡ weight value of a single large voter 0.8). In fact, the evidence from multiple voters is more persuasive than a single voter with a large weight, because it is less likely that two different voters both make errors and vote for the same address of the same type in a large and arbitrary memory space. So, we add a function $\gamma(s, c)$ to reward the cross-validated addresses voted by multiple voters.

In the implementation, the threshold δ is determined using a searching method in the validation dataset. We run the experiment by tuning the value of threshold δ to get the one that yields the highest F-score [134], and set it as the default threshold. The reward function is devised as $\gamma(s, c) = |L(s, c)| - 1$.

4.5 Evaluation

In this section, we first describe the experiment setup in Section 4.5.1. Then, we discuss the dataset collection and labeling approach in Section 4.5.2. Section 4.5.3 provides details about training. In the end, we present the evaluation results with respect to accuracy, robustness, and efficiency in Section 4.5.4, 4.5.5, and 4.5.6 respectively.

4.5.1 Experiment Setup

Our experiment uses two settings of configurations. 1) The training experiment is performed on a high-performance computing center with each worker node equipped with 32 cores Intel Haswell CPUs, 2 x NVIDIA Tesla K80 GPUs and 128 GB memory. 2) The detection experiment is performed on a moderate desktop computer with Core i7-6700, 16GB, no GPU. We use powerful GPUs on the computing center for training, which is a one-time effort. Once the model is trained, it is loaded on a desktop computer to conduct the kernel object detection.

The deep neural network models in DeepMem, like embedding network and classifier network, are all implemented using the open-source deep learning framework TensorFlow [166]. The remaining codes of data processing, statistics, plotting are programmed in Python.

4.5.2 Dataset

Memory Dumps Collection

While DeepMem can analyze any operating system versions in principle, it is limited by the object labeling tool used in training. In the evaluation, we choose to evaluate DeepMem on Windows 7 X86 SP1 rather than the latest Windows 10, mainly because the object labeling tool we used, Volatility [172], was unable to consistently parse Windows 10 images or memory dumps, but worked very stable for Windows 7 images.

To automatically collect a large number of diverse memory dumps for training and detection, we developed a tool with two functionalities: 1) simulating various random user actions, and 2) forcing the OS to randomly allocate objects in the memory space between consecutive memory dumps.

To simulate various user actions, the memory collecting tool first starts the guest Windows 7 SP1 virtual machine which is installed in the VirtualBox [170]. When the virtual machine is started, guest OS automatically starts 20 to 40 random actions, including starting programs from a pool of the most popular programs, opening websites from a pool of the most popular websites, and opening random PDF files, office documents, and picture files. Next, the memory collecting tool waits for 2 minutes and then dumps the memory of the guest system to a dump file. When the dump is saved to the hard disk of the host system, it restarts the virtual machine and repeats until we collect 400 memory dumps, each of which is 1GB in size.

To ensure kernel objects to be allocated at random locations, we enabled KASLR when generating our dataset and restarted the virtual machine after each dump. We found

out that the address allocations of objects are different among different memory dumps. Only 1.32% `_EPROCESS` objects in a memory dump are located at the same virtual address of `_EPROCESS` objects in another dump. The ratio is 4.7% for `_ETHREAD`, 0.68% for `_FILE_OBJECT`, 15.9% for `_DRIVER_OBJECT`. The basic statistics of memory dumps and memory graphs are shown in Table 4.1.

Kernel Object Type	**Mean Count**	**Std Dev**
_EPROCESS	85	7.47
_ETHREAD	1,216	112.25
_FILE_OBJECT	3,639	918.06
_DRIVER_OBJECT	109	0.22
_LDR_DATA_TABLE_ENTRY	141	0.59
_CM_KEY_BODY	1,921	953.76
Memory Graph Statistics	**Mean Count**	**Std Dev**
Nodes	1,334,822	134,564.24
Edges	5,325,214	513,624.71

Table 4.1: Statistics of memory dumps and memory graphs.

Kernel Object Types	**Length**	**#TP**	**#FP**	**#FN**	**Precision%**	**Recall%**	**F-Score**
_EPROCESS	704	82.834	0.017	0.303	99.979%	99.635%	0.99807
_ETHREAD	696	1211.476	5.514	0.7	99.547%	99.942%	0.99744
_DRIVER_OBJECT	168	108.938	0.255	0.024	99.766%	99.978%	0.99872
_FILE_OBJECT	128	3621.007	67.545	23.045	98.169%	99.368%	0.98765
_LDR_DATA_TABLE_ENTRY	120	139.093	0.0	2.4	100.0%	98.304%	0.99145
_CM_KEY_BODY	44	1979.207	94.621	0.414	95.437%	99.979%	0.97655

Table 4.2: Object Detection Results on Memory Image Dumps.

Memory Graph Construction

To generate a memory graph, we first read and scan all available memory pages in the kernel virtual space of memory dumps. Then, we locate all the pointers in the pages by finding all fields whose values fall into the range of kernel virtual space. For each segment

between two pointers, we create a node in the memory graph. For each node, we find its neighbor nodes in the memory dump according to the neighbor definitions in Section 4.4.2, and create an edge in the memory graph.

Node Labeling

The node labeling process takes four steps: 1) utilize Volatility to find out the offset and length information of 6 kernel object types (i.e. `_EPROCESS`, `_ETHREAD`, `_DRIVER_OBJECT`, `_FILE_OBJECT`, `_LDR_DATA_TABLE_ENTRY`, `_CM_KEY_BODY`) in memory dumps; 2) for each node in the memory graph, determine if it falls into the range of any kernel object, and if so, calculate the offset and length of that node in that kernel object and give the node a label; 3) select the top 20 most frequent node labels across all kernel objects of type c as key node label set $L(c)$ for type c; and 4) label the rest nodes in the memory graph as *none*.

Sample Balancing

Inside a large memory dump, kernel objects only take up a small portion of the memory space. Thus, the key nodes of kernel objects in the memory graph are very sparse. Also, the key nodes of a certain object type are not evenly distributed. To accelerate the training process and achieve better detection results, we need to balance samples in the training dataset.

The principle of balancing is to preserve the topologies of the key nodes in the memory graph after the balancing process. Specifically, 1) to reduce non-key nodes, we remove the nodes that are k-hops away from key nodes in memory graph (k is a predefined value), 2) to increase key nodes and balance between different node types, we duplicate

the key nodes to the same amount, and also duplicate the edges between nodes in edge matrix. Since the embedding vector is calculated using inward edges only, such duplication does not create new neighbors for the original key nodes, so it does not affect the topology propagation of the original key nodes.

4.5.3 Training Details

We split the collected 400 memory dumps into 3 subsets. We randomly select 100 images as the training dataset, 10 images as the validation dataset and the remaining 290 images as the testing dataset. The validation dataset and testing dataset will not be used in the training phase, and this guarantees that the detection model never sees the testing set in the training phase.

In each training iteration, we randomly select an image from the training dataset for training. To determine whether the model is fully trained, we monitored the loss and accuracy on the validation dataset during the training process. When the loss reaches a relatively small and stable value, we deem the model as fully trained or it reaches its learning capacity. Dropout layers [159] are added to prevent the over-fitting problem. We set the keep probability to 0.8 in the training phase, and to 1 in the evaluation phase and testing phase.

By default, the experiments are all performed under the same parameter setting as described in Table 4.3.

Parameters	Value
Layers of σ	3
Layers of ψ	3
Optimizer	Adam Optimizer
Learning Rate	0.0001
Propagation Iteration T	3
Input Vector Dimension	64
Embedding Vector Dimension	64
keep_prob	0.8

Table 4.3: Default Parameters of Experiments.

4.5.4 Detection Accuracy

We measured the accuracy using a number of different metrics, including precision, recall, and F-score [134]. For each object type, precision calculates the correctly classified samples against all detected samples. Recall calculates the correctly classified samples against all labeled samples in this type. F-score is the harmonic mean of precision and recall.

Table 4.2 shows the detection results of various kernel object types on raw memory images by training for 13 hours. We can see from the result, the overall recall rate is satisfactory, ranging from 98.304% to 99.979%. Most large kernel objects ($\geq$ 120 bytes) have over 98% precision rate. Important kernel object types `_EPROCESS`, `_ETHREAD` both achieve over 99.6% recall rate, and over 99.5% precision rate. Also, we observed a tendency that larger objects achieve better recognition results. The reason is that for small objects, there are fewer nodes and pointers inside them. Then, the chance of obtaining stable key nodes is lower.

4.5.5 Robustness

For the evaluation of robustness, we performed three experiments. The first experiment is pool tag manipulation, with the aim to evaluate its impact on signature scanning tools and DeepMem. The second experiment is pointer manipulation, with the aim to evaluate if DeepMem is still effective in DKOM process hiding attacks. The third experiment is a general yet more destructive attack which is to randomly mutate arbitrary bytes in memory, with the aim to see whether our approach is resistant to various attack scenarios, and to what extent it can tolerate random mutations.

Pool Tag Manipulation

To perform pool tag manipulation, we change the 4 bytes pool tags [155] of each object to random values in the memory dump file. Using the manipulated dump, we then test the effectiveness of our approach and Volatility plugin.

In our experiment, we randomly select 10 memory dumps as the testing set, and take scanning `_FILE_OBJECT` object as an example. As shown in Table 4.4, the `filescan` plugin of Volatility cannot correctly report `_FILE_OBJECT` objects. Its recall rate drops to a small value of 0.0082%. The reason is that `filescan` first needs to search for the pool tag of `_FILE_OBJECT` in the entire memory dump. As a result, most of the objects are not reported.

As a comparison, DeepMem works normally in evaluation results. It can achieve a recognition precision of 99.1% and recall of 99.05%. The reason is that DeepMem examines every byte of a memory dump to detect objects, rather than merely rely on pool tag con-

straints to locate objects. Hence, without valid pool tags, DeepMem can still detect objects in the memory dump. This indicates that approaches based on hard constraint matching are not robust. In contrast, our approach is based on soft features automatically learned from raw object bytes, which can capture a more robust representation of an object.

Method	**Avg. #TP**	**Avg. #FP**	**Avg. #FN**	**Precision%**	**Recall%**
filescan	0.3	0.0	3661.8	100%	0.0082%
DeepMem	3627.2	32.9	34.9	99.1%	99.05%

Table 4.4: Results of `_FILE_OBJECT` Pool Tag Manipulation

DKOM Process Hiding

This DKOM attack is to hide a malicious process by unlinking its connections to precedent and antecedent processes in a double linked list. In this case, list traversal related tools, like the `pslist` plugin in Volatility, will fail to discover the hidden process through this broken link list.

In our experiment, we randomly choose 20 memory dumps as a testing set, and then manipulated the value of the forward link field in each `_EPROCESS` object to random value. In Table 4.5, we can see that the Volatility plugin `pslist` fails to discover most `_EPROCESS` objects except the first one in each dump. Since the `_EPROCESS` list is broken by the manipulation, it cannot traverse through the double linked list to find other processes. In contrast, DeepMem can still find 99.77% `_EPROCESS` objects with 100% precision.

Method	**Avg. #TP**	**Avg. #FP**	**Avg. #FN**	**Precision%**	**Recall%**
pslist	1.05	0.0	85.7	100%	1.21%
DeepMem	86.55	0.0	0.2	100%	99.77%

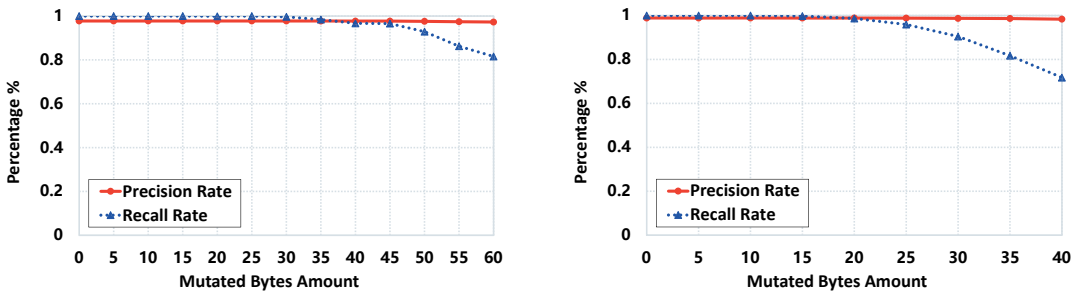
Table 4.5: Results of DKOM Process Hiding Attacks

Random Mutation Attack

It is hard to simulate all kinds of DKOM attacks. Therefore, we take a simple approach to find out how much DeepMem can tolerate DKOM attacks: we gradually increase the number of bytes to be manipulated in random positions of kernel objects, including the pointer and non-pointer fields, and evaluate the precision and recall rate at different mutation levels. In Section 4.5.5 and Section 4.5.5, we have already demonstrated how DeepMem works on memory dumps with small changes. In this section, we will show how DeepMem perform when large bytes are changed.

Even if an attacker largely changes the contents and topologies in kernel objects of the operating system, DeepMem can be used in this scenario without retraining the detection model with the samples from that attack. We just need to lower the prediction threshold δ. However, in extreme case, if the threshold is set to a very small value, then most addresses in candidate address set S will be reported, causing many false positives and low precision. To guarantee a high precision while getting a recall as high as possible, it is better to report the objects cross-validated by at least two voters. This can be achieved by setting the threshold δ of prediction function $f(s, c)$ to 1 (If there are more than two voters, the reward function $\gamma(s, c) = |L(s, c)| - 1 \geq 1$, the prediction confidence ¿ 1. See Equation (4.8)).

We evaluate the detection results by mutating different amount of bytes in objects for `_EPROCESS` and `_ETHREAD` objects, with threshold δ set to 1. We can see from Figure 4.5, as the number of mutated bytes increases, the precision rate remains stable at around 97% - 98% with tiny perturbations. Recall rate curve stays at a high rate at first, then drops down



(a) Random Mutation Attack (_EPROCESS) (b) Random Mutation Attack (_ETHREAD)

Figure 4.5: Random Mutation Attack

as the number of mutated bytes further increases. Specifically, for _EPROCESS, it achieves over 97% precision rate at all mutation levels, and 100% recall rate before 20 bytes are changed. Our model can tolerate up to 50 bytes random mutation, without causing the precision and recall rate drop significantly. For _ETHREAD, our model can tolerate up to 30 bytes random mutation. We can see when we set the threshold δ to a low value 1, the precision rate does not drop significantly.

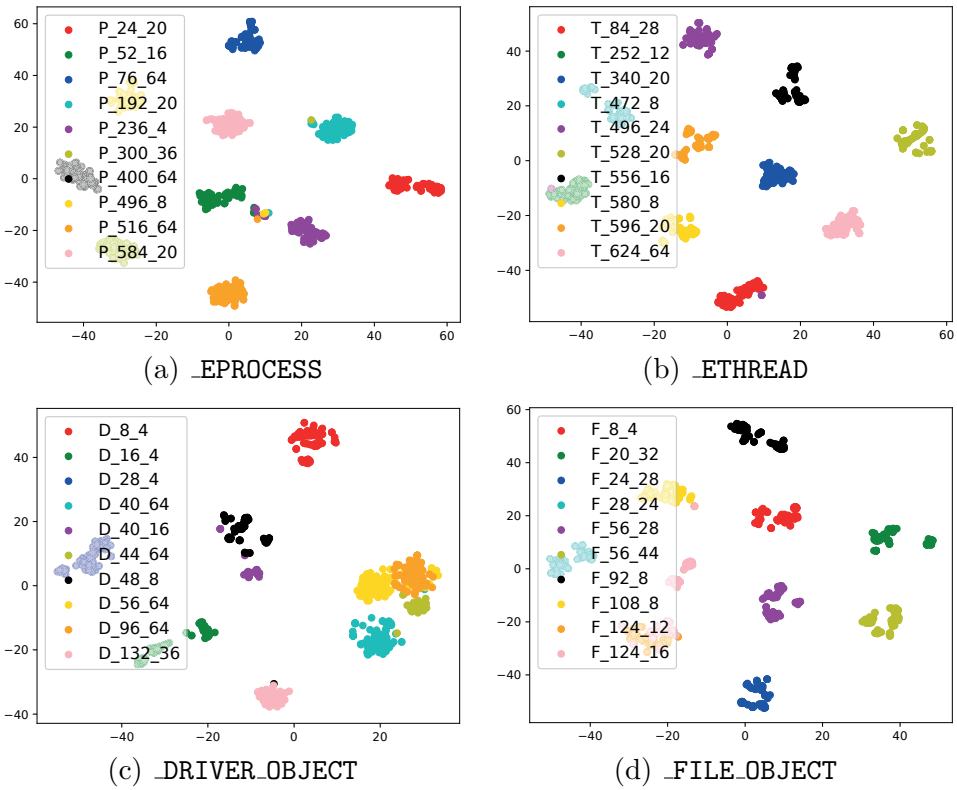
The causes of the high precision and recall rate are twofold. First, the neural network itself can inherently tolerate small mutations due to the robust features it learns from the training data. Second, even when deep model incorrectly predicts the labels of some nodes of an object, the remaining nodes can make cross-validation and collectively conclude the presence of an object. The recall rate indeed drops significantly with larger mutations. However, these larger mutations will likely cause system crashes or instability, and therefore might be rarely seen in real-world attacks.

4.5.6 Efficiency

To investigate the efficiency of DeepMem, we measure the time allocations in different phases. We consider three types of time consumption: GNN model training time T_t, memory graph construction time T_g and object detection time T_d. 1) The training time T_t measures the time from inputting raw labeled training dataset dumps to obtaining a fully trained model with a small and stable prediction loss. 2) The memory graph construction time T_g measures the time from inputting a raw memory dump to obtaining matrix representation of the memory graph. 3) The object detection time T_d measures the time from inputting a memory graph matrix to obtaining detected kernel object set of a certain object type. The experiment settings of training and detection are described in Section 4.5.1.

In the training phase, we utilize the GPU in the computing center to train the model because the major computation of training is matrix-based and GPU can accelerate the matrix computation. We train the model for 13 hours for one object type. After training, the model can be saved to disk and deployed in a desktop computer(with or without GPU). In our detection experiment, we copy the model to a moderate desktop computer without GPU. On average, it takes 79.7 seconds to construct the whole memory graph for one memory dump of 1GB size, and 12.73 seconds to recognize the objects of a certain type in it, as shown in Figure 4.6. This detection time can be accelerated by using GPU. In our computing center, the detection time can be reduced to about 7.7 seconds.

DeepMem is efficient for two reasons. First, it turns a memory dump into a graph structure denoted as large node matrices and edge matrices, which is especially suitable for fast GPU parallel computation. Second, since it converts the memory dump into an



(a) _EPROCESS (b) _ETHREAD

(c) _DRIVER_OBJECT (d) _FILE_OBJECT

Figure 4.6: Node Embedding Visualization using t-SNE

intermediate representation (memory graph), and performs the detection of various object types on this graph, there is no need to scan the raw memory multiple times to match the various set of signatures for different object types.

	Time Measurements	**Mean**	**Std Dev**
Training	Training T_t (per object type)	13 Hours	N/A
Detection	Graph Construction T_g (per dump)	79.7 Sec	6.64
	Object Detection T_d (per type)	12.73 Sec	1.24

Table 4.6: Time Consumption at Different Phases.

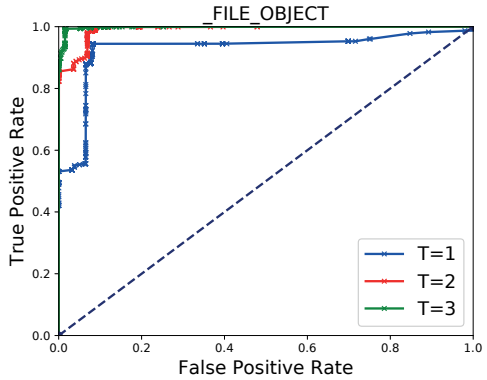
4.5.7 Understanding Node Embedding

We plot the embedding vectors of nodes using t-SNE visualization technique [108] in Figure 4.6. Each node embedding vector in multi-dimensional space is mapped as a point in two-dimensional space. We collect embedding vectors of different object types at the output layer of the embedding network before they are fed into the classifier network. Figure 4.6 shows the distribution of embedding vectors in 2D space, where different colors are used to denote different types of node labels. To clearly show plenty of embeddings of different types, we only plot the first 10 key nodes for each object type. We expect to observe that points of the same colors locate near each other, and different colors locate far from each other. From the figure, we can see that the visualized results meet that expectation. These embeddings can capture the intrinsic characteristics of nodes, and different types of nodes are well separated.

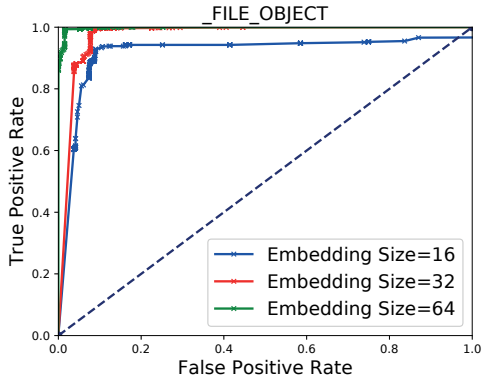
4.5.8 Impact of Hyperparameters

We plot ROC curves [67] of detection results to show the impact of the different hyperparameters of our model. We adjust three parameters: the propagation iteration times T, the embedding vector size, and the embedding depth of embedding network σ. ROC curve shows the trade-off between sensitivity (true positive rate) and specificity (false positive rate) of the object detector.

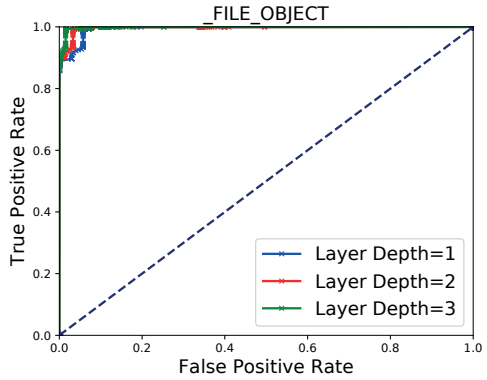
Figure 4.7(a) shows the performance of the `_FILE_OJBECT` detector by tuning the iteration parameter T of node embedding network ϕ. We can see that the ROC curve of $T = 3$ is nearest to the upper left corner, followed by the curves of $T = 2$ and $T = 1$.



(a) ROC versus Iterations T



(b) ROC versus Embedding Vector Size



(c) ROC versus Layer Depth of σ

Figure 4.7: ROC Curves by Tuning Parameters

The trend demonstrates the importance of topological information propagation in object detection. With more information collected through propagation, the prediction ability of the object detector is further improved.

Figure 4.7(b) shows the performance of `_FILE_OBJECT` detector by tuning embedding vector size of node embedding network ϕ. In the figure, the ROC curve with larger embedding size is closer to the upper left corner. It shows that larger embedding vector size is more expressive and better approximate the data intrinsic characteristics. However, this is also a trade-off between learning ability and training time. In practical usages, for the

same level of learning ability, a smaller embedding size is preferred for faster training and testing. The determination of such embedding size should be a combined consideration of the task complexity and training effort.

Figure 4.7(c) shows the performance of _FILE_OBJECT detector by tuning embedding layers depth of σ. In the figure, the ROC curve with more layers is closer to the upper left corner. It indicates that the learning ability of deeper neural network is stronger than shallower networks. Enlarging the number of layers and embedding size is a preferred solution for training complex object types.

4.6 Discussion

Small Objects. DeepMem may not perform well for small objects with few or no pointers, like many other pointer-based approaches [100]. Our approach model objects based on both content of objects and topological relations between objects. Small objects lacking pointers are not informative enough and also have weak or no relations with other nodes in the memory. Thus very little information could be gathered from others nodes to make inference on the objects. Fortunately, important kernel objects like _EPROCESS, _ETHREAD and _DRIVER_OBJECT are long enough for our approach to achieve over 99.6% recall and over 99.5% precision rate, which is sufficient for general memory forensic purposes.

Data Diversity and Validity To generate diverse dumps, we try to simulate random user actions and allocate kernel objects in random positions in the memory, as described in the evaluation section. Even with these efforts, our dataset may not be diverse enough. To make it more diverse, researchers can use different physical machines, load different drivers, etc.

Nevertheless, our evaluation on the dataset at least demonstrates the feasibility of DeepMem in a homogeneous environment (e.g., an enterprise network in which all computers have the same configuration and in a cloud environment where VMs are instantiated from the same base image). We use Volatility to label memory dumps as ground truth. According to the paper [135], Volatility achieves zero FPs and FNs for most of their plugins for non-malicious dumps. So our training set labeling should not be affected. Plus, we can use other solutions to label memory dumps as suggested in this chapter, such as using DECAF [68].

Cross Operating System Versions. In the evaluation phase, we have already demonstrated the robustness of our approach in scenarios like pool tag attack, DKOM process hiding and random bytes mutation. It shows that our approach tolerates well for small changes and manipulations of the memory. This feature is useful in real-world applications. For example, our approach will adapt to systems changes across versions and patches. We leave this for future work.

4.7 Conclusion

In this chapter, we propose a graph-based kernel object detection approach DeepMem. By constructing a whole memory graph and collecting information through topological information propagation, we can scan the memory dumps and infer objects of various types in a fast and robust manner. DeepMem is advanced in that 1) it does not rely on the knowledge of operating system source code or kernel data structures, 2) it can automatically generate features of kernel objects from raw bytes in memory dump without manual expert analysis, 3) it utilizes deep neural network architectures for efficient parallel

computation, and 4) it extracts robust features that are resistant to attacks like pool tag manipulation, DKOM process hiding. The experimental result shows that it performs well in terms of accuracy, robustness, and efficiency. For accuracy, it reaches above 99.5% recall and precision rate for important kernel objects like `_EPROCESS` and `_ETHREAD`. In terms of robustness, DeepMem's recognition results remain stable across different attack scenarios, such as manipulating pool labels, pointers, and even random byte mutations.In terms of efficiency, DeepMem converts a memory dump into an intermediate memory graph representation and efficiently uses the GPU for detection of different types of objects on this graph.

Chapter 5

Conclusions

In a nutshell, attacking state-of-the-art machine learning-based models effectively helps us understand how ML models make classification decisions and why they fail on adversarial examples. Then a new architecture is proposed that is fundamentally robust to adversarial attacks. To address the problem of memory-only malware attacks, DeepMem is proposed to scan memory dumps and infer various types of objects in a fast and robust manner.

MAB-Malware utilizes reinforcement learning to perform adversarial attacks on state-of-the-art machine learning models for malware classification and top commercial antivirus static classifiers. It finds an optimal balance between exploitation and exploration to maximize the evasion rate within limited trials. It filters out the actions that are ineffective for adversarial sample generation, so our framework can also be used to explain why evasion occurs. Our results show that MAB-Malware largely improves the evasion rate over other reinforcement learning frameworks.

Selective Hierarchical BERT is proposed to automatically select malicious functions for malware classification, which is robust to different attacks by design. Compared with other baselines, our model can handle very large samples and can automatically select essential features for malware classification, which fundamentally improves the robustness of the model without sacrificing the accuracy of the model.

DeepMem constructs a whole memory graph and collects information through topological information propagation, we can scan the memory dumps and infer objects of various types in a fast and robust manner. DeepMem is advanced in that 1) it does not rely on the knowledge of operating system source code or kernel data structures, 2) it can automatically generate features of kernel objects from raw bytes in memory dumps, 3) it utilizes DNN architectures for efficient parallel computation, and 4) it extracts robust features that are resistant to attacks like pool tag manipulation, DKOM process hiding.

5.1 Final Thoughts and Future Works

As machine learning becomes more and more widely used in various fields, we are facing more challenges while enjoying the convenience it brings. It is especially true in the field of computer security. It requires researchers to keep security in mind when designing the DNN model. As the models' complexity grows, it is increasingly difficult to know whether the model works as expected. That's why deep model explanation gains more and more interest in the ML community. As many works point out, interpretability and robustness are two sides of the same coin. If the interpretability of the deep model is increased, its robustness is also increased. Our work Selective Hierarchical BERT shows that

explanation can be directly used to improve the robustness of deep models. (We interpret the Reasoner model to get function labels to train the Judge model, which in turn helps to increase the robustness of the Reasoner model.) Our work MAB-Malware shows that it is possible to explain blackbox models, like the commercial antivirus engines.

My future work will focus on the analysis and interpretation of the Judge model. I will do some case studies to explain how the Judge model identifies malicious functions. I plan to try more techniques to improve the accuracy of the Judge model. It will help to further improve the interpretability and robustness of the whole malware classification architecture.

Bibliography

[1] Beta distribution. `https://en.wikipedia.org/wiki/Beta_distribution`.

[2] Mohammed Abuhamad, Tamer AbuHmed, Aziz Mohaisen, and DaeHun Nyang. Large-scale and language-oblivious code authorship identification. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 101–114. ACM, 2018.

[3] Ahmed Abusnaina, Aminollah Khormali, Hisham Alasmary, Jeman Park, Afsah Anwar, Ulku Meteriz, and Aziz Mohaisen. Examining adversarial learning against graph-based iot malware detection systems. *arXiv preprint arXiv:1902.04416*, 2019.

[4] Amir Afianian, Salman Niksefat, Babak Sadeghiyan, and David Baptiste. Malware dynamic analysis evasion techniques: A survey. *arXiv preprint arXiv:1811.01190*, 2018.

[5] Abdullah Al-Dujaili, Alex Huang, Erik Hemberg, and Una-May O'Reilly. Adversarial deep learning for robust detection of binary encoded malware. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 76–82. IEEE, 2018.

[6] Abdullah Al-Dujaili, Alex Huang, Erik Hemberg, and Una-May O'Reilly. Adversarial deep learning for robust detection of binary encoded malware. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 76–82. IEEE, 2018.

[7] Hyrum S Anderson, Anant Kharkar, Bobby Filar, David Evans, and Phil Roth. Learning to evade static pe machine learning malware models via reinforcement learning. *arXiv preprint arXiv:1801.08917*, 2018.

[8] Hyrum S Anderson, Anant Kharkar, Bobby Filar, and Phil Roth. Evading machine learning malware detection. *Black Hat*, 2017.

[9] Hyrum S Anderson and Phil Roth. Ember: an open dataset for training static pe malware machine learning models. *arXiv preprint arXiv:1804.04637*, 2018.

[10] Cosimo Anglano. Forensic analysis of whatsapp messenger on android smartphones. 2014.

[11] Adi Ashkenazy and Shahar Zini. Cylance, i kill you! https://skylightcyber.com/2019/07/18/cylance-i-kill-you/, 2019.

[12] Ai & machine learning. https://www.avast.com/en-us/technology/ai-and-machine-learning, 2018.

[13] Arati Baliga, Vinod Ganapathy, and Liviu Iftode. Automatic inference and enforcement of kernel data structure invariants. In *Computer Security Applications Conference (ACSAC)*, 2008.

[14] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, 2005.

[15] Chris Betz. MemParser. https://sourceforge.net/p/memparser/wiki/Home/, 2018.

[16] Qi-Zhi Cai, Min Du, Chang Liu, and Dawn Song. Curriculum adversarial training. *arXiv preprint arXiv:1805.04807*, 2018.

[17] Aylin Caliskan-Islam, Richard Harang, Andrew Liu, Arvind Narayanan, Clare Voss, Fabian Yamaguchi, and Rachel Greenstadt. De-anonymizing programmers via code stylometry. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 255–270, 2015.

[18] Martim Carbone, Weidong Cui, Long Lu, Wenke Lee, Marcus Peinado, and Xuxian Jiang. Mapping kernel objects to enable systematic integrity checking. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 555–565. ACM, 2009.

[19] Nicholas Carlini, Anish Athalye, Nicolas Papernot, Wieland Brendel, Jonas Rauber, Dimitris Tsipras, Ian Goodfellow, and Aleksander Madry. On evaluating adversarial robustness. *arXiv preprint arXiv:1902.06705*, 2019.

[20] Curtis Carmony, Xunchao Hu, Heng Yin, Abhishek Vasisht Bhaskar, and Mu Zhang. Extract me if you can: Abusing pdf parsers in malware detectors. In *NDSS*, 2016.

[21] Ero Carrera. pefile. https://github.com/erocarrera/pefile, 2016.

[22] Andrew Case and Golden G Richard III. Memory forensics: The path forward. 2016.

[23] Raphael Labaca Castro, Corinna Schmitt, and Gabi Dreo. Aimed: Evolving malware with genetic programming to evade detection. In *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pages 240–247. IEEE, 2019.

[24] Raphael Labaca Castro, Corinna Schmitt, and Gabi Dreo Rodosek. Poster: Training gans to generate adversarial examples against malware classification.

[25] Fabrício Ceschin, Marcus Botacin, Heitor Murilo Gomes, Luiz S Oliveira, and André Grégio. Shallow security: on the creation of adversarial variants to evade machine learning-based malware detectors. In *Proceedings of the 3rd Reversing and Offensive-oriented Trends Symposium*, pages 1–9, 2019.

[26] Olivier Chapelle and Lihong Li. An empirical evaluation of thompson sampling. In *Advances in neural information processing systems*, pages 2249–2257, 2011.

[27] Li Chen. Understanding the efficacy, reliability and resiliency of computer vision techniques for malware detection and future research directions. *arXiv preprint arXiv:1904.10504*, 2019.

[28] Lingwei Chen, Yanfang Ye, and Thirimachos Bourlai. Adversarial machine learning in malware detection: Arms race between evasion attack and defense. In *2017 European Intelligence and Security Informatics Conference (EISIC)*, pages 99–106. IEEE, 2017.

[29] Xiao Chen, Chaoran Li, Derui Wang, Sheng Wen, Jun Zhang, Surya Nepal, Yang Xiang, and Kui Ren. Android hiv: A study of repackaging malware for evading machine-learning detection. *IEEE Transactions on Information Forensics and Security*, 15:987–1001, 2019.

[30] Yizheng Chen, Shiqi Wang, Dongdong She, and Suman Jana. On training robust pdf malware classifiers. *arXiv preprint arXiv:arXiv:1904.03542*, 2019.

[31] Clamav. `https://www.clamav.net/`.

[32] James C. Corbett. Using shape analysis to reduce finite-state models of concurrent java programs. In *Proceedings of the International Symposium on Software Testing and Analysis*, 1998.

[33] Scott E Coull and Christopher Gardner. Activation analysis of a byte-based deep neural network for malware classification. *arXiv preprint arXiv:1903.04717*, 2019.

[34] Anthony Cozzie, Frank Stratton, Hui Xue, and Samuel T. King. Digging for data structures. In *the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, 2008.

[35] Cuckoo Sandbox. `https://cuckoosandbox.org/`.

[36] Weidong Cui, Marcus Peinado, Zhilei Xu, and Ellick Chan. Tracking rootkit footprints with a practical memory analysis system. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 601–615, 2012.

[37] George E Dahl, Jack W Stokes, Li Deng, and Dong Yu. Large-scale malware classification using random projections and neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 3422–3426. IEEE, 2013.

[38] Hanjun Dai, Bo Dai, and Le Song. Discriminative embeddings of latent variable models for structured data. In *International Conference on Machine Learning*, pages 2702–2711, 2016.

[39] Anusha Damodaran, Fabio Di Troia, Corrado Aaron Visaggio, Thomas H Austin, and Mark Stamp. A comparison of static, dynamic, and hybrid analysis for malware detection. *Journal of Computer Virology and Hacking Techniques*, 13(1):1–12, 2017.

[40] Hung Dang, Yue Huang, and Ee-Chien Chang. Evading classifiers by morphing in the dark. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 119–133. ACM, 2017.

[41] Park Daniel, Khan Haidar, and Yener Bülent. Generation & evaluation of adversarial examples for malware obfuscation. *arXiv preprint arXiv:1904.04802*, 2019.

[42] Luca Demetrio, B. Biggio, Giovanni Lagorio, F. Roli, and A. Armando. Functionality-preserving black-box optimization of adversarial windows malware. *arXiv: Cryptography and Security*, 2020.

[43] Luca Demetrio, Battista Biggio, Giovanni Lagorio, Fabio Roli, and Alessandro Armando. Explaining vulnerabilities of deep learning to adversarial malware binaries. *arXiv preprint arXiv:1901.03583*, 2019.

[44] Ambra Demontis, Marco Melis, Battista Biggio, Davide Maiorca, Daniel Arp, Konrad Rieck, Igino Corona, Giorgio Giacinto, and Fabio Roli. Yes, machine learning can be more secure! a case study on android malware detection. *IEEE Transactions on Dependable and Secure Computing*, 2017.

[45] Ambra Demontis, Marco Melis, Maura Pintor, Matthew Jagielski, Battista Biggio, Alina Oprea, Cristina Nita-Rotaru, and Fabio Roli. Why do adversarial attacks transfer? explaining transferability of evasion and poisoning attacks. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 321–338, 2019.

[46] Ming Ding, Chang Zhou, Hongxia Yang, and Jie Tang. Cogltx: Applying bert to long texts. *Advances in Neural Information Processing Systems*, 33:12792–12804, 2020.

[47] FU rootkit. `https://www.blackhat.com/presentations/win-usa-04/bh-win-04-butler.pdf`, 2018.

[48] Brendan Dolan-Gavitt. The vad tree: A process-eye view of physical memory. In *Digital Investigation, Volume 4, Supplement 1*, 2007.

[49] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, May 2011.

[50] Brendan Dolan-Gavitt, Abhinav Srivastava, Patrick Traynor, and Jonathon Giffin. Robust signatures for kernel data structures. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, pages 566–577, 2009.

[51] Brendan Dolan-Gavitt, Abhinav Srivastava, Patrick Traynor, and Jonathon Giffin. Robust signatures for kernel data structures. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 566–577. ACM, 2009.

[52] Saeed Ehteshamifar, Antonio Barresi, Thomas R Gross, and Michael Pradel. Easy to fool? testing the anti-evasion capabilities of pdf malware scanners. *arXiv preprint arXiv:1901.05674*, 2019.

[53] EvadeML. `https://github.com/uvasrg/EvadeML`, 2016.

[54] Aurore Fass, Michael Backes, and Ben Stock. Hidenoseek: Camouflaging malicious javascript in benign asts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1899–1913, 2019.

[55] Qian Feng, Aravind Prakash, Heng Yin, and Zhiqiang Lin. Mace: high-coverage and robust memory analysis for commodity operating systems. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC'14)*, pages 196–205. ACM, 2014.

[56] William Fleshman, Edward Raff, Jared Sylvester, Steven Forsyth, and Mark McLean. Non-negative networks against adversarial attacks. *arXiv preprint arXiv:1806.06108*, 2018.

[57] William Fleshman, Edward Raff, Richard Zak, Mark McLean, and Charles Nicholas. Static malware detection & subterfuge: Quantifying the robustness of machine learning and current anti-virus. In *2018 13th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 1–10. IEEE, 2018.

[58] Yangchun Fu and Zhiqiang Lin. Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In *2012 IEEE symposium on security and privacy*, pages 586–600. IEEE, 2012.

[59] Hisham Shehata Galal, Yousef Bassyouni Mahdy, and Mohammed Ali Atiea. Behavior-based features model for malware detection. *Journal of Computer Virology and Hacking Techniques*, 12(2):59–67, 2016.

[60] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in c. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, 1996.

[61] Yoav Goldberg and Omer Levy. word2vec explained: Deriving mikolov et al.'s negative-sampling word-embedding method. 2014.

[62] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[63] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. Adversarial perturbations against deep neural networks for malware classification. *arXiv preprint arXiv:1606.04435*, 2016.

[64] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. Adversarial examples for malware detection. In *European Symposium on Research in Computer Security*, pages 62–79. Springer, 2017.

[65] Wenbo Guo, Dongliang Mu, Jun Xu, Purui Su, Gang Wang, and Xinyu Xing. Lemna: Explaining deep learning based security applications. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 364–379. ACM, 2018.

[66] gym-malware. `https://github.com/endgameinc/gym-malware`, 2018.

[67] James A Hanley and Barbara J McNeil. The meaning and use of the area under a receiver operating characteristic (roc) curve. 1982.

[68] Andrew Henderson, Aravind Prakash, Lok Kwong Yan, Xunchao Hu, Xujiewen Wang, Rundong Zhou, and Heng Yin. Make it work, make it right, make it fast: building a platform-neutral whole-system dynamic binary analysis platform. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014.

[69] Greg Hoglund and James Butler. Rootkits: subverting the windows kernel. 2006.

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

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

[72] Weiwei Hu and Ying Tan. Black-box attacks against rnn based malware detection algorithms. In *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[73] Weiwei Hu and Ying Tan. Black-box attacks against rnn based malware detection algorithms. In *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[74] Alex Huang, Abdullah Al-Dujaili, Erik Hemberg, and Una-May O'Reilly. On visual hallmarks of robustness to adversarial malware. *arXiv preprint arXiv:1805.03553*, 2018.

[75] Yonghong Huang, Utkarsh Verma, Celeste Fralick, Gabriel Infantec-Lopez, Brajesh Kumar, and Carl Woodward. Malware evasion attack and defense. *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, Jun 2019.

[76] IDA Pro Disassembler. `http://www.hex-rays.com/idapro/`.

[77] Inigo Incer, Michael Theodorides, Sadia Afroz, and David Wagner. Adversarially robust malware detection using monotonic classification. In *Proceedings of the Fourth ACM International Workshop on Security and Privacy Analytics*, pages 54–63. ACM, 2018.

[78] Kyriakos K. Ispoglou and Mathias Payer. malwash: Washing malware to evade dynamic analysis. In *10th USENIX Workshop on Offensive Technologies (WOOT 16)*, Austin, TX, August 2016. USENIX Association.

[79] Daniel Jakubovitz and Raja Giryes. Improving dnn robustness to adversarial attacks using jacobian regularization. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 514–529, 2018.

[80] ElMouatez Billah Karbab, Mourad Debbabi, Abdelouahid Derhab, and Djedjiga Mouheb. Android malware detection using deep learning on api method sequences. *arXiv preprint arXiv:1712.08996*, 2017.

[81] Aminollah Khormali, Ahmed Abusnaina, Songqing Chen, DaeHun Nyang, and Aziz Mohaisen. Copycat: Practical adversarial attacks on visualization-based malware detection. *arXiv preprint arXiv:1909.09735*, 2019.

[82] 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.

[83] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. 2014.

[84] Pang Wei Koh and Percy Liang. Understanding black-box predictions via influence functions. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1885–1894. JMLR. org, 2017.

[85] Bojan Kolosnjaji, Ambra Demontis, Battista Biggio, Davide Maiorca, Giorgio Giacinto, Claudia Eckert, and Fabio Roli. Adversarial malware binaries: Evading deep learning for malware detection in executables. In *2018 26th European signal processing conference (EUSIPCO)*, pages 533–537. IEEE, 2018.

[86] Hyungjoon Koo and Michalis Polychronakis. Juggling the gadgets: Binary-level code randomization using instruction displacement. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 23–34. ACM, 2016.

[87] Hyungjoon Koo and Michalis Polychronakis. Juggling the gadgets: Binary-level code randomization using instruction displacement. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 23–34. ACM, 2016.

[88] Alex Kouzemtchenko. Defending malware classification networks against adversarial perturbations with non-negative weight restrictions. *arXiv preprint arXiv:1806.09035*, 2018.

[89] Felix Kreuk, Assi Barak, Shir Aviv-Reuven, Moran Baruch, Benny Pinkas, and Joseph Keshet. Adversarial examples on discrete sequences for beating whole-binary malware detection. *arXiv preprint arXiv:1802.04528*, 2018.

[90] Felix Kreuk, Assi Barak, Shir Aviv-Reuven, Moran Baruch, Benny Pinkas, and Joseph Keshet. Deceiving end-to-end deep learning malware detectors using adversarial examples. *arXiv preprint arXiv:1802.04528*, 2018.

[91] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial machine learning at scale. *arXiv preprint arXiv:1611.01236*, 2016.

[92] Raphael Labaca-Castro, Battista Biggio, and Gabi Dreo Rodosek. Poster: Attacking malware classifiers by crafting gradient-attacks that preserve functionality. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2565–2567. ACM, 2019.

[93] Chris Lattner, Andrew Lenharth, and Vikram Adve. Making Context-Sensitive Points-to Analysis with Heap Cloning Practical For The Real World. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*, San Diego, California, June 2007.

[94] Quan Le, Oisín Boydell, Brian Mac Namee, and Mark Scanlon. Deep learning at the shallow end: Malware classification for non-domain experts. *Digital Investigation*, 26:S118–S126, 2018.

[95] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. 2015.

[96] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.

[97] Deqiang Li, Ramesh Baral, Tao Li, Han Wang, Qianmu Li, and Shouhuai Xu. Hashtran-dnn: A framework for enhancing robustness of deep neural networks against adversarial malware samples. *arXiv preprint arXiv:1809.06498*, 2018.

[98] Deqiang Li, Qianmu Li, Yanfang Ye, and Shouhuai Xu. Enhancing robustness of deep neural networks against adversarial malware samples: Principles, framework, and aics'2019 challenge. *arXiv preprint arXiv:1812.08108*, 2018.

[99] LIEF. `https://github.com/lief-project/LIEF`.

[100] Zhiqiang Lin, Junghwan Rhee, Chao Wu, Xiangyu Zhang, and Dongyan Xu. Dimsum: Discovering semantic data of interest from un-mappable memory with confidence. In *Proc. NDSS*, 2012.

[101] Zhiqiang Lin, Junghwan Rhee, Xiangyu Zhang, Dongyan Xu, and Xuxian Jiang. Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures. In *Proceedings of the Network and Distributed System Security Symposium*, February 2011.

[102] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 11th Annual Information Security Symposium*, 2010.

[103] Xiang Ling, Shouling Ji, Jiaxu Zou, Jiannan Wang, Chunming Wu, Bo Li, and Ting Wang. Deepsec: A uniform platform for security analysis of deep learning model. In *IEEE S&P*, 2019.

[104] Xiaolei Liu, Xiaojiang Du, Xiaosong Zhang, Qingxin Zhu, Hao Wang, and Mohsen Guizani. Adversarial samples on android malware detection systems for iot systems. *Sensors*, 19(4):974, 2019.

[105] Xinbo Liu, Jiliang Zhang, Yaping Lin, and He Li. Atmpa: Attacking machine learning-based malware visualization detection methods via adversarial examples. In *2019 IEEE/ACM 27th International Symposium on Quality of Service (IWQoS)*, pages 1–10. IEEE, 2019.

[106] Yanpei Liu, Xinyun Chen, Chang Liu, and Dawn Song. Delving into transferable adversarial examples and black-box attacks. *arXiv preprint arXiv:1611.02770*, 2016.

[107] Keane Lucas, Mahmood Sharif, Lujo Bauer, Michael K Reiter, and Saurabh Shintre. Malware makeover: Breaking ml-based static analysis by modifying executable bytes. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, pages 744–758, 2021.

[108] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. 2008.

[109] Multi-armed bandit. `https://en.wikipedia.org/wiki/Multi-armed_bandit`.

[110] Holger Macht. Live memory forensics on android with volatility. *Friedrich-Alexander University Erlangen-Nuremberg*, 2013.

[111] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083*, 2017.

[112] Davide Maiorca, Davide Ariu, Igino Corona, Marco Aresu, and Giorgio Giacinto. Stealth attacks: An extended insight into the obfuscation effects on android malware. *Computers & Security*, 51:16–31, 2015.

[113] Davide Maiorca, Battista Biggio, Maria Elena Chiappe, and Giorgio Giacinto. Adversarial detection of flash malware: Limitations and open issues. *CoRR*, abs/1710.10225, 2017.

[114] Davide Maiorca, Battista Biggio, and Giorgio Giacinto. Towards robust detection of adversarial infection vectors: Lessons learned in pdf malware. *arXiv preprint arXiv:1811.00830*, 2018.

[115] MCTS. `https://gist.github.com/qpwo/c538c6f73727e254fdc7fab81024f6e1`, 2019.

[116] Marco Melis, Davide Maiorca, Battista Biggio, Giorgio Giacinto, and Fabio Roli. Explaining black-box android malware detection. In *2018 26th European Signal Processing Conference (EUSIPCO)*, pages 524–528. IEEE, 2018.

[117] Xiaozhu Meng, Barton P Miller, and Somesh Jha. Adversarial binaries for authorship identification. *arXiv preprint arXiv:1809.08316*, 2018.

[118] Machine Learning Static Evasion Competition 2019. https://github.com/endgameinc/malware_evasion_competition.

[119] Machine Learning Static Evasion Competition 2020. https://github.com/Azure/2020-machine-learning-security-evasion-competition.

[120] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, 1999.

[121] Nicolas Papernot, Patrick McDaniel, and Ian Goodfellow. Transferability in machine learning: from phenomena to black-box attacks using adversarial samples. *arXiv preprint arXiv:1605.07277*, 2016.

[122] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning. In *Proceedings of the 2017 ACM on Asia conference on computer and communications security*, pages 506–519. ACM, 2017.

[123] Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. Distillation as a defense to adversarial perturbations against deep neural networks. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 582–597. IEEE, 2016.

[124] Raghavendra Pappagari, Piotr Zelasko, Jesús Villalba, Yishay Carmiel, and Najim Dehak. Hierarchical transformers for long document classification. In *2019 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*, pages 838–844. IEEE, 2019.

[125] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *2012 IEEE Symposium on Security and Privacy*, pages 601–615. IEEE, 2012.

[126] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *2012 IEEE Symposium on Security and Privacy*, pages 601–615. IEEE, 2012.

[127] Daniel Park, Haidar Khan, and Bülent Yener. Short paper: Creating adversarial malware examples using code insertion. *CoRR*, abs/1904.04802, 2019.

[128] Jithin Pavithran, Milan Patnaik, and Chester Rebeiro. D-time: Distributed threadless independent malware execution for runtime obfuscation. In *13th USENIX Workshop on Offensive Technologies (WOOT 19)*, Santa Clara, CA, August 2019. USENIX Association.

[129] The best antivirus protection. https://www.pcmag.com/picks/the-best-antivirus-protection, 2020.

[130] Nick L Petroni, Aaron Walters, Timothy Fraser, and William A Arbaugh. Fatkit: A framework for the extraction and analysis of digital forensic data from volatile system memory. 2006.

[131] Fabio Pierazzi, Feargus Pendlebury, Jacopo Cortellazzi, and Lorenzo Cavallaro. Intriguing properties of adversarial ml attacks in the problem space. *2020 IEEE Security and Privacy*, 2020.

[132] Robert Podschwadt and Hassan Takabi. Effectiveness of adversarial examples and defenses for malware classification. *arXiv preprint arXiv:1909.04778*, 2019.

[133] Michael JD Powell. An efficient method for finding the minimum of a function of several variables without calculating derivatives. 1964.

[134] David Martin Powers. Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation. 2011.

[135] Aravind Prakash, Eknath Venkataramani, Heng Yin, and Zhiqiang Lin. On the trustworthiness of memory analysis-an empirical study from the perspective of binary execution. 2015.

[136] Pytorch. `https://pytorch.org/`.

[137] Erwin Quiring, Alwin Maier, and Konrad Rieck. Misleading authorship attribution of source code using adversarial learning. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 479–496, 2019.

[138] 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.

[139] 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.

[140] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. Droidchameleon: evaluating android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 329–334. ACM, 2013.

[141] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. Why should i trust you?: Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1135–1144. ACM, 2016.

[142] Konrad Rieck, Philipp Trinius, Carsten Willems, and Thorsten Holz. Automatic analysis of malware behavior using machine learning. *Journal of Computer Security*, 19(4):639–668, 2011.

[143] Ishai Rosenberg, Asaf Shabtai, Yuval Elovici, and Lior Rokach. Query-efficient gan based black-box attack against sequence based machine and deep learning classifiers. *arXiv preprint arXiv:1804.08778*, 2018.

[144] Ishai Rosenberg, Asaf Shabtai, Lior Rokach, and Yuval Elovici. Generic black-box end-to-end attack against state of the art api call based malware classifiers. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 490–510. Springer, 2018.

[145] Ishai Rosenberg, Asaf Shabtai, Lior Rokach, and Yuval Elovici. Generic black-box end-to-end attack against state of the art api call based malware classifiers. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 490–510. Springer, 2018.

[146] Brendan Saltaformaggio, Rohit Bhatia, Zhongshu Gu, Xiangyu Zhang, and Dongyan Xu. Guitar: Piecing together android app guis from memory images. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 120–132. ACM, 2015.

[147] Brendan Saltaformaggio, Rohit Bhatia, Zhongshu Gu, Xiangyu Zhang, and Dongyan Xu. Vcr: App-agnostic recovery of photographic evidence from android device memory images. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[148] Brendan Saltaformaggio, Rohit Bhatia, Xiangyu Zhang, Dongyan Xu, and Golden G Richard III. Screen after previous screens: Spatial-temporal recreation of android app displays from memory images. In *USENIX Security Symposium*, 2016.

[149] Brendan Saltaformaggio, Zhongshu Gu, Xiangyu Zhang, and Dongyan Xu. Dscrete: Automatic rendering of forensic information from memory images via application logic reuse. In *USENIX Security Symposium*, 2014.

[150] Joshua Saxe and Konstantin Berlin. Deep neural network based malware detection using two dimensional binary program features. In *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 11–20. IEEE, 2015.

[151] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. pages 61–80, 2009.

[152] Jürgen Schmidhuber. Deep learning in neural networks: An overview. 2015.

[153] Matthew G Schultz, Eleazar Eskin, F Zadok, and Salvatore J Stolfo. Data mining methods for detection of new malicious executables. In *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*, pages 38–49. IEEE, 2000.

[154] Andreas Schuster. Searching for processes and threads in microsoft windows memory dumps. 2006.

[155] Andreas Schuster. The impact of microsoft windows pool allocation strategies on memory forensics. In *Digital Investigation, Volume 5*, 2008.

[156] Alexander G Schwing and Raquel Urtasun. Fully connected deep structured networks. 2015.

[157] PV Shijo and A Salim. Integrated static and dynamic analysis for malware detection. *Procedia Computer Science*, 46:804–811, 2015.

[158] Slot machine. https://en.wikipedia.org/wiki/Slot_machine.

[159] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. 2014.

[160] Jack W Stokes, De Wang, Mady Marinescu, Marc Marino, and Brian Bussone. Attack and defense of dynamic analysis-based, adversarial neural malware classification models. *arXiv preprint arXiv:1712.05919*, 2017.

[161] Jack W Stokes, De Wang, Mady Marinescu, Marc Marino, and Brian Bussone. Attack and defense of dynamic analysis-based, adversarial neural malware detection models. In *MILCOM 2018-2018 IEEE Military Communications Conference (MILCOM)*, pages 1–8. IEEE, 2018.

[162] Octavian Suciu, Scott E Coull, and Jeffrey Johns. Exploring adversarial examples in malware detection. In *2019 IEEE Security and Privacy Workshops (SPW)*, pages 8–14. IEEE, 2019.

[163] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 3319–3328. JMLR. org, 2017.

[164] Rahim Taheri, Reza Javidan, Mohammad Shojafar, Zahra Pooranian, Ali Miri, and Mauro Conti. On defending against label flipping attacks on malware detection systems. *arXiv preprint arXiv:1908.04473*, 2019.

[165] Microsoft Defender ATP Research Team. New machine learning model sifts through the good to unearth the bad in evasive malware. https://www.microsoft.com/security/blog/2019/07/25/new-machine-learning-model-sifts-through-the-good-to-unearth-the-bad-in-evasive-malware/, 2019.

[166] Tensorflow. https://www.tensorflow.org.

[167] Shun Tobiyama, Yukiko Yamaguchi, Hajime Shimada, Tomonori Ikuse, and Takeshi Yagi. Malware detection with deep neural network using process behavior. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, volume 2, pages 577–582. IEEE, 2016.

[168] Thompson Sampling. https://en.wikipedia.org/wiki/Thompson_sampling.

[169] Upx packer. https://upx.github.io.

[170] VirtualBox. https://www.virtualbox.org/, 2018.

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

[172] Volatility: Memory Forencis System. `https://www.volatilityfoundation.org/`, 2018.

[173] Daixin Wang, Peng Cui, and Wenwu Zhu. Structural deep network embedding. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, 2016.

[174] Jingyuan Wang, Yufan Wu, Mingxuan Li, Xin Lin, Junjie Wu, and Chao Li. Interpretability is a kind of safety: An interpreter-based ensemble for adversary defense. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 15–24, 2020.

[175] Zhi Wang, Xuxian Jiang, Weidong Cui, and Peng Ning. Countering kernel rootkits with lightweight hook prevention. In *Proceedings of the 16th ACM Conference on Computer and Communication Security (CCS'09)*, 2009.

[176] Cihang Xie, Mingxing Tan, Boqing Gong, Alan Yuille, and Quoc V Le. Smooth adversarial training. *arXiv preprint arXiv:2006.14536*, 2020.

[177] Weilin Xu, Yanjun Qi, and David Evans. Automatically evading classifiers. In *Proceedings of the 2016 network and distributed systems symposium*, pages 21–24, 2016.

[178] 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. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[179] Wei Yang, Deguang Kong, Tao Xie, and Carl A Gunter. Malware detection in adversarial settings: Exploiting feature evolutions and confusions in android apps. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 288–302, 2017.

[180] Sheng Yu, Yu Qu, Xunchao Hu, and Heng Yin. Deepdi: Learning a relational graph convolutional network model on instructions for fast and accurate disassembly.

[181] Hongyang Zhang, Yaodong Yu, Jiantao Jiao, Eric P Xing, Laurent El Ghaoui, and Michael I Jordan. Theoretically principled trade-off between robustness and accuracy. *arXiv preprint arXiv:1901.08573*, 2019.

[182] Yue Zhang and Stephen Clark. Joint word segmentation and pos tagging using a single perceptron. 2008.

[183] Hai Zhao, Chang-Ning Huang, and Mu Li. An improved chinese word segmentation system with conditional random field. In *Proceedings of the Fifth SIGHAN Workshop on Chinese Language Processing*, 2006.

[184] Zhengli Zhao, Dheeru Dua, and Sameer Singh. Generating natural adversarial examples. *arXiv preprint arXiv:1710.11342*, 2017.

[185] Fan Zhou, Yitao Yang, Zhaokun Ding, and Guozi Sun. Dump and analysis of android volatile memory on wechat. In *Communications (ICC), 2015 IEEE International Conference on*, 2015.