STUDYING THE ROBUSTNESS OF MACHINE LEARNING-BASED MALWARE DETECTION MODELS: ANALYSIS, DESIGN, AND IMPLEMENTATION

by

AHMED ABUSNAINA
M.S. University of Central Florida, 2021
B.S. An-Najah National University, Palestine, 2018

A dissertation submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida,
Orlando, Florida

Spring Term
2022

Major Professor: David Mohaisen

© 2022 Ahmed Abusnaina

ABSTRACT

With the rise of the popularity of machine learning (ML), it has been shown that ML-based classifiers are susceptible to adversarial examples and concept drifting, where a small modification in the input space may result in misclassification. The ever-evolving nature of the data, the behavioral and pattern shifting over time not only lessened the trust in the machine learning output but also created a barrier for its usage in critical applications. This dissertation builds toward analyzing machine learning-based malware detection systems, including the detection and mitigation of adversarial malware examples. In particular, we first introduce two black-box adversarial attacks on control flow-based malware detectors, exposing the vulnerability of graph-based malware detection systems. Further, we propose DL-FHMC, fine-grained hierarchical learning technique for robust malware detection, leveraging graph mining techniques alongside pattern recognition for adversarial malware detection. Enabling machine learning in critical domains is not limited to the detection of adversarial examples in laboratory settings, but also extends to exploring the existence of adversarial behavior in the wild. Toward this, we investigate the attack surface of malware detection systems, shedding light on the vulnerability of the underlying learning algorithms and industry-standard machine learning malware detection systems against adversaries in both IoT and Windows environments. Toward robust malware detection, we investigate software pre-processing and monotonic machine learning. In addition, we explore potential exploitation caused by actively retraining malware detection models. We uncover a previously unreported malicious to benign detection performance trade-off, causing the malware to revive and be classified as a benign or different malicious family. This behavior leads to family labeling inconsistencies, hindering the efforts toward malicious families' understanding. Overall, this dissertation builds toward robust malware detection, by analyzing and detecting adversarial examples. We highlight the vulnerability of industry-standard applications to black-box adversarial settings, including the continuous evolution of malware over time.

To my mother.

ACKNOWLEDGMENTS

This work would not have been possible without the support of so many individuals who have enriched my personal and professional experience over the past years whom I would like to take this opportunity to acknowledge.

First and foremost, I would like to dedicate this dissertation to my family: my mother, sister, brother, and father, who all supported me unconditionally, and have always been by my side. In particular, my mother, the person who motivated me to excel and never lost faith in me. A doctoral program in computer science is not an easy endeavor by any means, with countless ups and downs along the way. There has been many times when I faced dead-ends and contemplated dropping out of the program, and it is only thanks to the continued support and belief of my mother that helped me clear my thoughts, collect myself, and continue my course of study. I dedicate this dissertation to my best friend and supporter, my mother. Despite all challenges in the past few years, physically and emotionally, she always reminded me of the light at the end of the tunnel, and words cannot describe my gratitude.

I would like to extend my gratitude to my doctoral advisor, Prof. David Mohaisen, for his continued support and guidance during the past four years. Coming to the University of Central Florida, I had no prior research experience, and working in the Security and Analytics Lab (SEAL) under Dr. Mohaisen's guidance helped me grow professionally thanks to his support, and I am grateful for this opportunity. I would like to also thank my doctoral dissertation committee members, Prof. Cliff Zou, Prof. Gita Sukthankar, Prof. Sung Choi Yoo, Prof. Shibu Yooseph, and Prof. Gary Leavens, for their feedback at the different program's milestones: the candidacy, proposal, and final dissertation exam. Their critiques and feedback have been immensely helpful in improving this work.

While working at SEAL, I had the opportunity to collaborate with, mentor, and learn from a lot of the current and past amazing SEAL members whom I would like to thank (in no particular order):

Hisham, Afsah, Mohammad (Abuhamad), Rhongho, Saad, Ulku, Jinchun, Amin, Sultan, Jabbar, and Mohammed (Alkinoon).

I would like to extend my gratitude to my collaborators inside and outside UCF: Prof. Murat Yuksel (UCF), Prof. Saeed Salem (NDSU), Prof. Songqing Chen (GMU), Prof. DaeHun Nyang (Ewha), and Prof. An Wang (CWRU). Finally, I would like to thank my mentors and collaborators at Visa Research for two wonderful summers that have immensely cultivated my interest in a career in the industry: Hao Yang, Maliheh Shirvanian, Mihai Christodorescu, Sunpreet Arora, Yizhen Wang, and Yuhang Yu.

I would like to finally acknowledge the sponsors of the work reported in this dissertation: UCF's ORC (for a doctoral fellowship that supported my first year at UCF), National Science Foundation (grant # CNS-1809000), National Research Foundation (grant # NRF-2016K1A1A2912757), CyberFlorida (Collaborative Seed Grant), and NVIDIA (GPU Grant).

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

Deep learning techniques are being widely used in various domains, including malware detection [146, 150, 24, 21], computer vision [87, 58, 56, 61], natural language processing [57, 84], and speech recognition [59, 63]. However, an extensive line of research has shown that an attacker can manipulate the prediction of a deep learning-based classification system by adding a small perturbation to the inputs fed to a deep learning model , intermediate embeddings [60, 131, 55], or by inducing distribution shifts [62, 73]. These results highlight a major security issue for deep neural network-based prediction systems, especially the ones deployed in critical applications, such as access control, user authentication, and malware detection [153, 127].

To address this security concern, a variety of defence mechanisms have been proposed. These defense mechanisms can be broadly categorized into two categories. The *proactive* approaches [76, 54, 136], mitigating the effects of known adversarial attack methods on the machine learning frameworks, which increases model robustness to adversarial perturbation. In contrast, the *reactive* approach aim to detect adversarial examples, and distinguish them from legitimate samples. In particular, it builds a detector to detect adversarial examples in the test environment. This approach is more suitable for already deployed systems, as no changes to the operating algorithm is needed. In addition to detecting adversarial examples, analyzing and implementing defense mechanisms can also help to identify security-compromised input sources, providing information regarding the attack surfaces of the designed framework.

In this dissertation, we focus on analyzing the robustness of machine learning-based malware detection frameworks. Bypassing the detection frameworks, via generating adversarial examples, may cause diminishing returns to the implemented framework and environment, such as compromising the system by misclassifying malware as benign. To this end, this dissertation makes three different contributions in four different thrusts:

Thrust 1. *Robustifying graph-based malware detection engines*: We introduce DL-FHMC, a fine-

grained hierarchical learning approach designed for robust IoT malware detection. DL-FHMC utilizes Control Flow Graph (CFG)-based behavioral patterns for adversarial IoT malicious software detection. In particular, we extract a comprehensive list of behavioral patterns from a large dataset of malicious IoT binaries, represented by the shared execution flows, and use them as a modality for malicious behavior detection. Leveraging machine learning and subgraph isomorphism matching algorithms, DL-FHMC provides state-of-the-art performance in detecting malware samples and adversarial examples (AEs). We first highlight the caveats of CFG-based IoT malware detection systems, which showed a detection accuracy of up to 99% in the literature, showing the adversarial capabilities in generating practical functionality-preserving AEs with reduced overhead using Graph Embedding and Augmentation (GEA) techniques. We then introduce Suspicious Behavior Detector, a component that extracts comprehensive behavioral patterns from three popular IoT malicious families, Gafgyt, Mirai, and Tsunami, for AEs detection with high accuracy. The proposed detector operates as a model-independent standalone module, with no prior assumptions about the adversarial attacks or their configurations.

Thrust 2. *Analyzing the robustness of research- and industrial-standard malware detection systems under adversarial settings*: We systematically examine the state-of-the-art IoT malware detection approaches, which utilize various representation and learning techniques, under a range of adversarial settings, including binary padding, stripping, and packing, alongside a set of white-box practical adversarial attacks. Our analyses highlight the instability of the state-of-the-art detectors in learning patterns that distinguish the benign from the malicious software. The results show that software mutations with functionality-preserving operations, such as stripping and padding, significantly deteriorate the accuracy of such detectors. Additionally, our analysis of industry-standard malware detectors shows their instability with respect to the malware mutations. Through experiments, we highlight the gap between the capabilities of the adversary and that of the existing malware detectors. The evaluations and analyses show that the optimal malware detection system is nowhere near, with existing

detection frameworks over-fitting on benign patterns and can be exploited toward misclassification using pattern injection attacks. Toward addressing some of the issues, we explore the threat surface of existing malware detection systems, providing root cause analysis of the ongoing black-box adversarial malware examples, and uncovering the usage of volatile exploitable features by online industry-standard detection engines, resulting in detection performance reduction of up to 30%.

Thrust 3. *Highlighting temporal inconsistencies in malware detection systems and their underlying learning algorithms*: We dive further into the robustness of existing malware detection engines, questioning the effectiveness of malware detection retraining approaches by exposing various limitations in the retraining approaches adopted by detection frameworks. We show that model retraining only provides a marginal performance improvement for malicious samples detection, while simultaneously degrading the benign samples detection accuracy. Our analysis also shows that lapses in periodic retraining may not even provide the marginal performance improvements, while continuously increasing the computation cost. We also highlight the capabilities of out-of-distribution detection models in detecting the emergence of new malware families. Further, inspired by our findings, we investigate the family labeling inconsistencies within online detectors over time, unveiling family renaming inconsistency, which affects thousands of shared malicious capabilities among malware families. While intuitive, the shared capabilities increase the challenge of distinguishing the malicious families among others, and limiting the understanding of their origin and unique capabilities.

In summary, this dissertation addresses the robustness of machine learning-based malware detection frameworks, by both designing robust solutions with adversarial detection capabilities, and analyzing implemented systems highlighting their attack surface and possible exploitation and vulnerabilities.

Organization. This dissertation is organized as follows: We review the literature and outline notable related works in chapter 2. Toward defending against the adversarial attacks, we introduce

DL-FHMC, robust and accurate graph-based malware detection framework, in chapter 3. In chapter 4, we provide a large-scale analysis of attack surfaces existing within IoT malware detection systems, a natural extension of such work is analyzing the threat surface of the attack vectors, mitigating the adversarial attacks effects. In chapter 5, we discuss our efforts on analyzing the robustness of Windows malware detection systems in terms of consistency and accurate learning, including the limitations and caveats of continuous model retraining. Finally, the dissertation concludes in chapter 6, with concluding remarks and future work directions.

CHAPTER 2: LITERATURE REVIEW

In the following, we discuss notable related works relevant to this dissertation. We first discuss the malware behavioral and temporal analysis and detection, followed by the literature on malware detection and adversarial examples.

Malware Analysis & Detection

Malware Behavioral Analysis. While efforts have been put towards malware analysis and detection in general, IoT malware robustness analysis still lacks exploration and investigation. Among the IoT malware studies, efforts towards the analysis and detection of malicious software are limited, particularly, from the lens of CFG-based techniques. ManXu *et al.* [146] proposed a CNN-based malware detection system for Android application from the semantic representation of their control and data flow graphs representations. In addition, Yang *et al.* [150] identified and detected Android malicious behaviors throughout generating two-level behavioral representations built from the CFG graph and call graphs of a program. Allix *et al.* [24] designed multiple machine learning classifiers to detect Android malware using different textual representations extracted from the applications' CFGs. Further, Alasmary *et al.* [21] conducted an in-depth CFG-based comparative study for the Android and IoT malware.

Similarly, Pa *et al.* [108] established the first IoT honeypot and sandbox system, called IoTPOT, that run over eight CPU architectures to capture the IoT attacks running over Telnet protocol. Further, Caselden *et al.* [41] built an algorithm that generates an attack from the representation of the hybrid information and CFG applied to the program binaries. Alam *et al.* [19] proposed a metamorphic malware analysis and detection system that uses two different techniques that match the CFGs of small malware and then address the change in the opcodes frequencies.

Moreover, Tamersoy *et al.* [132] proposed a malware detection algorithm that identifies the executable files of the malware and then computes the similarities between them and the partial dataset

files from the Norton Community Watch. Then, they construct graphs based on the measurement of inter-relationship between these files. In addition, Wuchner *et al.* [142] proposed a graph-based detection system that uses quantitative data flow graphs generated from the system calls, and uses the graph node properties, i.e., centrality metric, as a feature vector for the classification between malicious and benign programs. Moreover, they extended the work by using a compression-based mining technique applied to the quantitative data flow graphs for malware detection [141]. Moreover, Cen *et al.* [42] used Android API calls as features extracted from the decompiled source code of the software, and proposed a probabilistic logistic regression-based model for malware detection.

Furthermore, Qiu *et al.* [114] surveyed existing machine/deep learning-based Android malware detection and classification systems. Their study shows a consistent trend of using neural network-based architectures for extracting deep representations and characteristics of the Android malware for the detection and classification tasks, which provide an improvement in comparison to the handcrafted features.

While the over-time malware analysis is a well-studied domain, the effects of malware evolution on machine learning-based malware detection frameworks is yet to be explored. In this dissertation, we provide a wide-scale analysis of the limitations of machine learning under continuous malware evolution, unveiling malicious practices that may exploit the detection frameworks and hinder their learning process, including, but not limited to, malware revival and re-spreading, and family labeling inconsistencies.

Malware Detection. The prior work in this space explores the potential of machine learning algorithms for building effective malware detection systems [47, 22, 29, 137, 97]. The performance of such systems largely depends on the choice of representations, generated by static and dynamic analysis techniques.

With advances in learning theory, the application of learning algorithms to defend systems against malware attacks has provided remarkable success. In order to apply advances in learning the-

ory, malware binaries are transformed into different representations that are machine learning-compatible [89, 154]. For example, Cui *et al.* [48] introduced a malware detection method using deep learning by transforming the malicious code into grayscale images, achieving an accuracy rate of 94.5% on the Vision Research Lab dataset [104], and showing better performance than static [81] and dynamic [18] feature-based representations. Similarly, Ni *et al.* [105] proposed a malware detection system trained over 10,805 grayscale images associated with nine different Windows malware families with comparable success. Fu *et al.* [65] proposed a malware detector using an RF model trained over colored images generated from malware binaries, achieving a detection accuracy of 97.47% and a family classification accuracy of 96.85%.

Arp *et al.* [30] detected malicious android applications using machine learning models with manually-engineered and statically extracted features. Pajouh *et al.* [72] detected ARM-based malware targeting IoT devices using an LSTM-based architecture to model opcode sequences, achieving detection an accuracy of 98.18%. Furthermore, Wang *et al.* [139] proposed an adversary-aware neural network technique for malware detection by leveraging feature nullification. Li *et al.* [90] leveraged a dataset of billions of program binary files to reliably identify malicious singleton malware samples using static features, with a false positive rate of as low as 1.4%. Shafiq *et al.* [123] achieved a detection rate of 99% with less than 0.5% false alarm rates on Portable Executables.

The aforementioned literature aimed to identify malicious behavior using various representations. In this dissertation, we leverage the aforementioned representations to analyze the robustness of malware detection and classification tasks, particularly the robustness of static analysis-based feature representations.

Malware Temporal Analysis. Malware has significantly been analyzed to build defenses; *e.g.* the aforementioned studies for malware detection and classification. Unfortunately, fewer efforts have been dedicated to understand malware mutations, and even fewer were dedicated for understanding the temporal aspects of malware evolution.

Graziano *et al.* [68] analyzed the malware samples submitted to public dynamic analysis sandboxes

before being used for malware campaigns in the wild. Their analyses show that malware appeared on the public sandboxes months or even years before they are involved in targeted attacks.

Similarly, Bilge and Dumitras [35] noted that antivirus companies collect the malware involved in attacks in advance (i.e., before they are used for such attacks), which allows address zero-day attacks by identifying the malicious files exploiting a vulnerability before its public disclosure.

Mohaisen and Alrawi [101] systematically analyze the inconsistencies among the labeling followed by malware detection vendors and their impact on the overall malware detection rate, accuracy, and consistency. Their study highlights high dependency between the detection rate and the family of the malware. Moreover, Bayer *et al.* [34] analyzed the reports of a public dynamic analysis engine to study the trends and evolution of malicious behavior over a period of two years, unveiling the common malicious behaviors among a diverse range of malicious software across families.

Toward understanding malware temporal behavior, Alrawi *et al.* [26] introduced a large-scale study of the current IoT malware threat landscape. By analyzing a total of 166,000 linux-based IoT malicious software applications, they deduced that the IoT malware evolution follows trends similar to the traditional malware by using exploits for infection. Lindorfer *et al.* [93] further studied the evolution of well-known Windows malware families over time, by studying malware updating process in disassembly. Their study enabled effective monitoring of the evolution of a malware's functional and behavioral components.

In this dissertation, we focus on a different aspect of malware behavior temporal shifting: the resulted vulnerabilities and exploitation. In particular, we analyzed the model retraining in the context of malware detection, a to-go solution for over time malware detection. Further, we studied the reasons behind the family labeling inconsistencies, shedding light on shared malicious patterns and capabilities among malware families, which can be used to understand the malware origin.

Adversarial Machine Learning

Machine/deep learning networks are widely used in security-related tasks, including malware detection [28, 102, 100, 23]. However, it has been shown that deep learning-based models are vulnerable against adversarial attacks [99]. Unfortunately, such a behavior can be a critical issue in malware detection systems, where misclassifying malware as benign may result in compromising the underlying service [12, 15].

Various adversarial machine learning attack methods in the context of image classification have been introduced to generate AEs. For example, Goodfellow *et al.* [67] introduced FGSM, a family of fast method attacks to generate Adversarial Examples (AEs) that forces the model to misclassification. In addition, Carlini *et al.* [39] proposed three L-norm-based adversarial attacks, known as C&W adversarial attacks, to investigate the robustness of neural networks and existing adversarial defenses. Similarly, Moosavi *et al.* [103] proposed DeepFool, an L_2 distance-based adversarial iterative method to generate AEs with minimal perturbation.

Further, a critical application of the AEs is malware detection. Recent studies investigated generating AEs in the context of malware detection [129]. For instance, Grosse *et al.* [70] implemented an augmented adversarial crafting algorithm to generate AEs, misleading a CNN-based classifier to misclassify 63% of the malware samples to benign. Additionally, in the context of Android malware detection, Chen *et al.* [44] proposed a novel approach to evade the Android malware detection systems by applying optimal perturbations onto Android APK using a substitute model, utilizing the transferability characteristics of the AEs. This allows generating AEs to non-differentiable models, such as support vector machines and random forest. Applying perturbation directly onto APK's Dalvik bytecode, they achieved a performance degradation of more than 95% against two state-of-the-art detection approaches, MaMaDroid [107] and Drebin [31].

The detection of the AEs is challenging [38]. While work on detecting AEs in the context of IoT malware detection is very limited, multiple studies attempt to detect them in the context of image classification [144, 92, 98], achieving detection accuracy of 20% to 90%. In this dissertation,

we implemented various approaches for adversarial malware detection and mitigation, including graph mining-based malicious behavior signaturing, and robust software pre-processing for volatile features removal, achieving a state-of-the-art robust malware detection accuracy of up to 99% with adversarial mitigation capabilities.

CHAPTER 3: DL-BASED FINE-GRAINED HIERARCHICAL LEARNING APPROACH FOR ROBUST MALWARE CLASSIFICATION

There has been a large body of research work on the problem of malware analysis using both static and dynamic approaches [102, 66, 78], and a few attempts on analyzing IoT malware in particular. Recently, machine learning algorithms, specifically deep learning techniques, are actively utilized for detecting/classifying malicious software from benign ones. However, it should be noted that the research work on IoT malware analysis has been very limited not only in the size of the analyzed samples, but also in the utilized approaches [32, 50, 125, 119]. Among the static analysis-based approaches, one of the prominent approaches is to use abstract graph structures for IoT malware analysis and detection, such as the control flow graph (CFG) [23, 20]. Previously, it has been shown that the software graph-based analysis can be incorporated with machine learning methods to introduce more powerful analysis tools [151, 28]. For the IoT malware detection, CFGs allow defenders to extract plentiful feature representations that can be used to distinguish those malware from benign, owing to their various properties, such as the degree distribution, centrality measures, radius, etc.. [23]. Those properties can be represented as a feature vector that can be used to enable machine learning algorithms to accurately detect and classify IoT malware samples. Proposed by Alasmary *et al.*, one such application is exploring IoT malware using both graph analysis and machine/deep learning [23]. Their model not only can learn the representative characteristics of the graph, but also can be utilized to build an automatic detection system for predicting the label of the unseen software.

Using machine and deep learning techniques should first address the concerns and challenges related to their security and usability. Recent studies have shown that machine learning-based IoT malware detection methods are prone to adversarial manipulation [110]. Adversarial machine

This work has been published at the IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING, 2021.

learning has shown the fragile nature of those algorithms to perturbation and data poisoning attacks, leading to misclassification. For example, an adversary can introduce a small modification to the input sample to make the classifier misidentify the malware as benign (*i.e.* the adversary introduces an AE). Such modification is usually crafted using small perturbation to make the AE undetectable and very difficult to distinguish for the original sample.

We stress the importance of addressing the threat posed by the machine learning vulnerability to AEs, particularly in security-sensitive applications. We undertake this challenge by (1) showing the high potential of successful detection/classification of IoT malware using deep learning methods; (2) assessing the robustness of such methods to AEs generated by different state-of-the-art CFG-based AEs generation techniques; (3) introducing a fine-grained hierarchical approach to tackle adversarial attacks by leveraging patterns extracted from the basic and elementary structure of the tested software.

To this end, we start by investigating the robustness of DL-SSMC, **D**eep **L**earning-based **S**ingle **S**hot **M**alware **C**lassification approach, for accurate IoT malware detection and classification. DL-SSMC utilizes machine learning techniques for IoT malware detection and classification, taking feature representation as an input, and outputs the classification corresponding to the input, we refer to this process as "single shot", as it requires querying the system once, and no decisions are taken outside of the machine learning model itself. Then, we examine the approach against AEs generated by Graph Embedding and Augmentation (GEA) [13] and Sub-Graph Embedding and Augmentation (SGEA) [12]. The GEA and SGEA are graph-based AEs generation approaches that are proposed to bypass CFG-based malware detection systems.

To cope with adversaries and minimize their effects, we propose DL-FHMC, **F**ine-grained **H**ierarchical Learning for **M**alware **C**lassification, for detecting and classifying malware samples by operating on a fine-grained level of structures and patterns extracted from the malicious software. DL-FHMC utilizes the shared behavioral structures of the malicious IoT software from the same family to create sub-graph signatures representing the execution flows of the malicious behavior. The extracted

signatures are then used to distinguish benign software and AEs with high accuracy. Our experiments show the effectiveness of our proposed approach in detecting malware samples as well as a high-degree of robustness against a variety of adversarial attacks.

Summary of Completed Work

1. *Investigate the robustness of traditional CFG-based IoT malware detection systems*. Through comprehensive experiments, we show the effectiveness of GEA and SGEA in producing successful AEs that can fool the machine learning-based malware detection system.

2. *Propose robust and resilient CFG-based malware detection system*. We propose DL-FHMC, a fine-grained hierarchical learning for malware classification, that extracts potential malicious behavioral patterns of IoT malicious families. We extracted 30,000 behavioral patterns from three IoT malicious families.

3. *Investigate the robustness of DL-FHMC under adversarial configurations.* DL-FHMC operates by investigating the malicious subgraphs within the IoT malware, using subgraph mining and pattern recognition to detect suspicious and malicious behaviors within the tested samples, mitigating the effects of AEs and detecting up to 100% of malicious AEs.

Graph Analysis: A Preliminary Overview

Graph Analysis. The CFG is a graph representation of the program which shows all paths that can be reached during the execution, as shown in Fig. 3.1. In a CFG, the set of nodes means the basic blocks where each block is a straight-line instruction without any *jump* or *jump target*, while the set of directed edges corresponds to the *jump* which traverses from the block to the other block at the branch (*if*), loop (*while, for*), and the end of the function (*return*). Once the first instruction of the basic block is executed, the rest of the instructions in the same block are necessarily executed unless terminated by external interference. In general, CFG is used for the structural analysis of

the program. For example, from the perspective of optimization, the CFG is used to analyze the reachability of each block. By constructing the CFG and evaluating the reachability, the flaws of the program (infinite loop or unreachable codes) can be found and addressed.

CFG-based Analysis. In graph theory, there are various concepts that express the characteristics of a graph. Given $G = (V, E)$, for example, the number of vertices ($|V|$) means the order of G, while the number of edges ($|E|$) corresponds to the size of G. The density of the graph can be defined as $D = |E|/(|V| * (|V| - 1))$ for directed simple graph, which means the ratio of the number of edges in G to the maximal number of edges in the complete graph. The centrality is measured for each node $v \in V$, which shows how important a specific node is. In detail, there are several different kinds of centrality, such as closeness centrality, betweenness centrality, Eigenvector centrality, etc..

These indicators (and further concepts not described above) can be considered the features of the graph G. Moreover, the combination of those metrics can be a more deterministic characteristic of the graph. Considering that a CFG is a kind of graph, it is true that each binary has not only its unique graph representation but also the associated values, such as the order, size, and density of CFG, and centrality for each vertex in CFG. On the other hand, the graph-based analysis can provide the possibility for identifying the malware. Because it is highly likely that the binaries in the same "family" share the structural similarity (even if there is a little difference), the CFG-based features can be combined with the state-of-the-art machine learning technique to determine whether a given binary is malicious or not.

Threat Model

The rapid reliance on machine learning methods in various applications has raised several security and privacy concerns, especially in security-sensitive applications. It has become crucial to understand and assess the robustness of machine learning techniques to several adversarial settings. These adversarial settings include AEs, where an adversary intends to fool or misguide the classification model with malicious inputs that are generated by applying a minimal perturbation to the

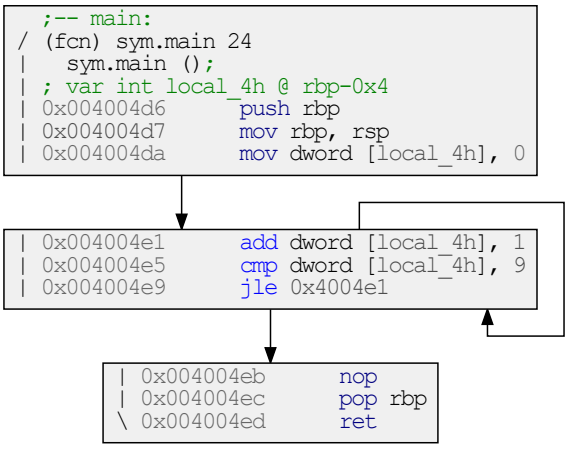


Figure 3.1: The generated CFG for the original sample and used for extracting graph-based features (graph size, centralities, etc.) for graph/program classification and malware detection.

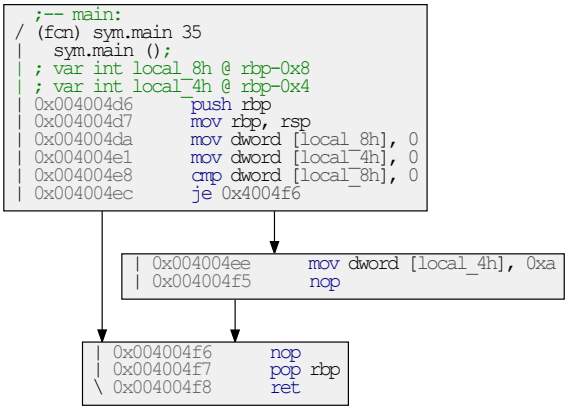


Figure 3.2: The CFG for the selected target sample generated and used for extracting graph-based features (graph size, centralities, etc.) for graph/program classification and malware detection.

original sample [111]. These modifications misclassify the samples of the model from benign to malware and vice versa and even misclassify the malware classes to another class. Such adversarial attacks can be launched under different adversarial capabilities that allow for either black-box and white-box attacks. In a white-box attack, the adversary has full knowledge of the inner networking paradigm of the model, while in a black-box attack, the adversary has only access to the model via an oracle and observes only the output of the model.

The literature on AEs and their effects includes numerous studies where the perturbation is applied

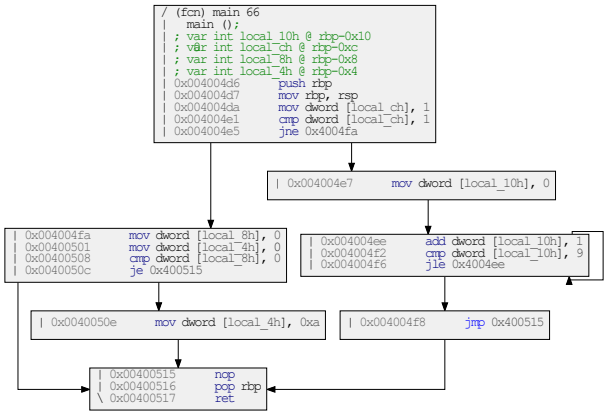


Figure 3.3: The generated adversarial graph using GEA approach. Note that this graph is obtained logically by embedding the graph in Fig. 3.2 into the graph in Fig. 3.1, although indirectly done by injecting the code listings as highlighted in Listings 1, 2, and 3.

to image pixels [111, 110, 138]. Unlike image AEs, the generated AEs from the IoT software must preserve the original sample's functionality and practicality in order to function properly. Adversarial machine learning can be derived from two perspectives: targeted and non-targeted attacks.

Targeted attacks. The focus of this attack is to generate AE x' that forces the classifier f to misclassify into a specific target class t. For instance, the adversary generates a set of malicious IoT software samples, which are classified as benign. That is: $x' : [f(x') = t] \wedge [\Delta(x, x') \leq \epsilon]$, where $f(.)$ represents the classifier's output, $\Delta(x, x')$ denotes the difference between x and the crafted AE x', whereas ϵ is a distortion threshold.

Non-targeted attacks. The focus of non-targeted attack is to generate an AE that forces the classifier f to misclassify to any class other than the original class $f(x)$, where x is the original input. That is: $x' : [f(x') \neq f(x)] \wedge [\Delta(x, x') \leq \epsilon]$, where $f(.)$ shows the classifier's output, $\Delta(x, x')$ represents the difference between x and x', and ϵ is the distortion threshold.

In this study, we generate AEs from the IoT software based on code-level manipulation using GEA [13] and SGEA [12]. In the following, we discuss each attack briefly.

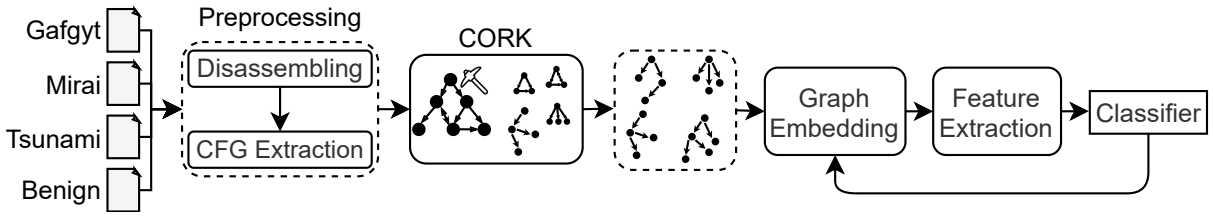


Figure 3.4: SGEA pattern extraction and AE generation process. SGEA uses CORK to extract discriminative subgraphs from each class. Then, the extracted subgraphs are embedded to generate the AEs. The process is terminated and the AE is returned upon successfully misclassifying the model.

Graph Embedding and Augmentation (GEA). GEA generates realistic AEs, where the functionality and practicality of the original binary are maintained. The key idea of GEA is combining an original CFG with a targeted CFG. In the following, we briefly describe GEA using an example.

Practical Implementation. Assume an original sample (software) (x_{org}) and a selected target sample (x_{sel}), GEA combines x_{org} with a x_{sel} while preserving the functionality and practicality of x_{org}. The GEA process combines the two scripts while ensuring that x_{sel} does not affect the process and functionality of x_{org}. Note that the condition is set to execute only the functionality related to x_{org} and preventing the processes of x_{sel} from being executed. Prior to generating the CFG for these algorithms, we compile the code using the GNU Compiler Collection (GCC) command. Afterward, Radare2 [7] is used to extract the CFG from the binaries.

Fig. 3.1 and Fig. 3.2 show the generated CFGs for both x_{org} and x_{sel}, respectively. As shown in Fig. 3.3, the combined CFG consists of the two scripts sharing the same entry and exit nodes. Therefore, the GEA approach adds modifications to the CFG for generating the AE. Given the nature of the extracted features, the applied changes on the CFG are reflected upon the features, regardless of the effects on the functionality and executability of the original sample. Following the adopted approach in [13], we select three different-sized graphs from benign samples as x_{sel}. The selected graphs vary in size, where the size is the number of nodes in the graph. To generate AEs, we selected a graph and connected it with all malicious samples.

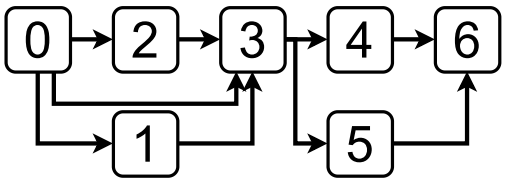


Figure 3.5: Sample of extracted discriminative subgraph from Gafgyt malicious family. Here, the graph size is 7, and the labels are arbitrary. Ideally, connecting this subgraph to a sample should lead the model to misclassify the sample into Gafgyt.

Sub-GEA (SGEA). While GEA combines an original CFG to a selected CFG of the IoT samples to misclassify the machine learning model, the SGEA approach aims to reduce the injection size and achieve the adversarial objectives with minimal perturbation. More specifically, it uses deep discriminative subgraph patterns extracted from the CFGs of each class using a correspondence-based quality criterion (CORK) algorithm, which defines a submodular quality criterion that ensures a solution close to the optimal solution [133]. This is done by using subgraphs that appear more frequently in one class than others, to fool the machine learning model in predicting that class (*e.g.* when launching a targeted attack).

Let D denotes the CFGs of the training samples, $D = \{G_i\}_{i=1}^{n}$ and class labels $C = \{c_i\}_{i=1}^{n}$ where $c_i \in \{+1, -1\}$ is the class label of graph G_i. Also let D^+ and D^- denote the set of graphs in the corresponding classes. For a multi-class dataset, we run the CORK algorithm once for each class where all the graph that belong to the same class are included in D^+ and the other graphs are in D^-. A graph G_i supports another graph S if S is a subgraph of G_i. Let $D_S = \{G_i | S \subseteq G_i \; \forall \; G_i \in D\}$ denote the supporting graphs of a subgraph S. Moreover, let D_S^+ and D_S^-, denote the supporting graphs of the subgraph in the positive and negative graphs, respectively.

CORK defines a submodular quality criterion, q, for a subgraph based on the set of supporting graphs ('hits') and non-supporting graphs ('misses') in the two classes and is calculated as follows: $q(G_s) = -(|D_S^{+\sim}| * |D_S^{-\sim}| + |D_S^+| * |D_S^-|)$. The best quality score is achieved when a subgraph appears in all graphs of one class and not once in the graphs of the other classes. Pruning strategies, as used in the quality criterion of CORK, are integrated into the gSpan algorithm [149] to directly

```c
#include<stdio.h>
void  main(){
    int GEAVar1 = 0; // block 0
    if(GEAVar1 == 1){ // block 1
        GEAVar1 += 1;
    }
    else if(GEAVar1 == 2){ // block 2
        GEAVar1 += 2;
    }
    int GEAVar2 = 0; // block 3
    if(GEAVar2 == 0){ // block 4
        GEAVar2 += 1;
    }
    else{ // block 5
        GEAVar2 += 2;
    }
    int GEAVar3 = 0; // block 6
}
```

Listing 3.1: C script of an example Gafgyt extracted subgraph. Each block is represented as a node in the generated CFG. Appending this code to the source code of a sample will lead to producing the subgraph shown in Fig. 3.5.

mine discriminative subgraphs. Once the set of discriminative subgraphs are mined, we employ gSpan, a graph-based substructure mining pattern for mining frequent subgraphs of size five nodes or higher.

Practical Implementation. SGEA combines x_{org} with the selected discriminative subgraph (x_{sel}). For example, Fig. 3.5 shows the discriminative subgraph extracted from the Gafgyt class and listing 3.1 shows the equivalent C script to generate that subgraph, which can then be combined with the x_{org} to generate an AE. Figure 3.4 shows the overview of patterns extraction and the process of generating AEs in the SGEA approach. While GEA modifies the CFG by connecting the selected graph with the original sample, SGEA connects a carefully generated subgraph with the original sample to generate AE, reducing the injected graph size. To generate the subgraph, we extracted the discriminative subgraph patterns from each class, with a size of five nodes or higher. Then, in order to reduce the graph size needed to be embedded, we connect x_{org} with the subgraph

Table 3.1: Distribution of IoT samples across the classes. We split the dataset into 80% training and 20% testing, with an overall 10,091 IoT samples (7,091 IoT malware and 3,000 benign).

Class		# of Samples			% of Samples
		# Train	# Test	# Total	
Benign		2,400	600	3,000	29.72
Malicious	Gafgyt	2,400	600	3,000	29.72
	Mirai	2,400	600	3,000	29.72
	Tsunami	872	219	1091	10.84
Overall		8,072	2,019	10,091	100

with minimum size. If the generated AE misclassifies, the process succeeds, and the AE will be returned; else, we select the next subgraph in ascending order regarding the number of nodes in the subgraph. In case none of the subgraphs cause misclassification, the original sample will be returned as the process failed.

Constructing an AE. As shown in Figure 3.4, we extract a set of subgraphs from the targeted class. Then, we combine the original sample with the smallest extracted subgraph of the targeted class regarding the number of nodes. If the generated CFG fails to misclassify the model, another subgraph is selected in ascending order with respect to the number of nodes in the set of generated subgraphs and combined with the same original graph to generate another CFG. This process is repeated until a subgraph successfully misclassifies the model. If no existing subgraph from the set of targeted subgraphs causes misclassification, the original sample is returned, hence the process failed in generating AE. In this study, we consider AEs that misclassify malware as benign, as such AEs have huge risk on the users, and render the malware detector systems useless.

Data Representation & Learning

In this section, we discuss the utilized dataset, dataset representation, and learning algorithms, including the experimental setup for DL-SSMC and DL-FHMC.

Table 3.2: The distribution of extracted features. 23 algorithmic features are extracted from the CFGs. These features are categorized into seven groups, including number of nodes and edges, density, shortest path, and centralities. When possible, the minimum, maximum, median, mean, and standard deviation values are extracted, as in the shortest path group.

Feature category	# of features
Betweenness centrality	5
Closeness centrality	5
Degree centrality	5
Shortest path	5
Density	1
# of Edges	1
# of Nodes	1
Total	23

Dataset. In this work, we collected binaries of two categories, IoT malicious and benign samples. The malicious samples are collected from CyberIOCs [6], VirusTotal [8], and VirusShare [4] in the period of January 2018 to late January of 2021, with a total of 7,091 samples that belong to three malware families. Additionally, we assembled a dataset of 3,000 benign IoT samples compiled from the source files on GitHub [52].

Ground Truth Class. The benign and malicious samples in our dataset were validated using the *VirusTotal* [8]. We uploaded the samples on VirusTotal and gathered the scan results corresponding to each sample. We then used AVClass [121] to classify the malicious samples into their corresponding families. We summarize the dataset in table 3.1.

Data Representation . Samples of the IoT benign (3,000 sample) and IoT malware categories (7,091 samples) were reverse-engineered using *Radare2* [7], a reverse engineering framework that provides various analysis capabilities, for obtaining the samples' corresponding CFGs. Using the samples' CFGs extracted by *Radare2*, we represent the CFG using the graph-theoretic features proposed by Alasmary *et al.* [21]. In particular, we extracted 23 different algorithmic features categorized into seven groups. Table 3.2 shows the feature category and the number of features in each category. Except for the number of edges, nodes, and the density of the graph, five features were

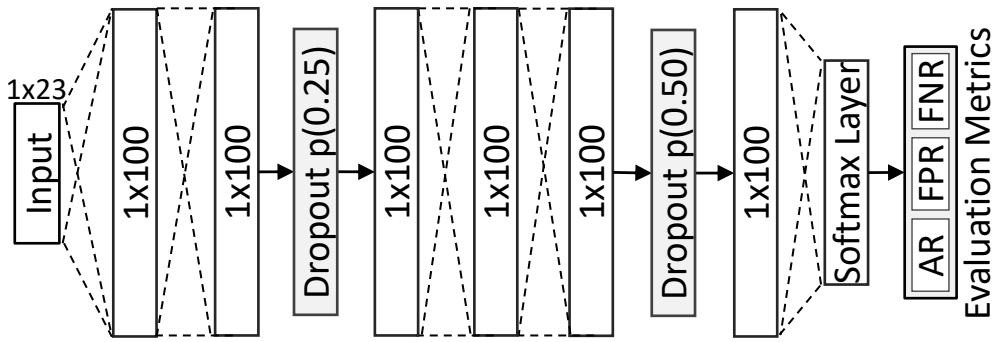


Figure 3.6: The internal architectural of the DNN used for the detection and classification tasks. The design consists of six fully connected layers, with dropout operations and softmax activation function.

extracted from each feature category, including the minimum, maximum, median, mean, and standard deviation values for the observed parameters. To this end, each IoT software is represented as a vector of size 1×23 representing the corresponding CFG-based algorithmic features.

Learning Algorithms. Toward IoT malware detection and classification, we utilize different machine and deep learning algorithms for pattern learning and deep feature extraction. In the following, we briefly describe each learning algorithm.

Random Forest (RF). RF consists of N decision trees, each decision tree is trained on a collection of random features, and finds the non-linear relationships between the features and the output decision. The final prediction of RF classifier with N decision trees is determined by a majority vote over the predictions or by averaging the prediction of all trees, determined as follows:

$$f_{RF} = \frac{1}{N} \sum_{n=1}^{N} f_n(X_s^{'}),$$

where, for a randomly selected feature set, ($X_{\cdot}^{'} \subset X_{\cdot}$), f_n is the n^{th} tree's prediction and $X_s^{'}$ is the segment's s vector.

Deep Neural Networks (DNN). DNN is an artificial neural network with neurons of each layer are fully connected to the neurons of the next layer. It consists of multiple hidden layers between the

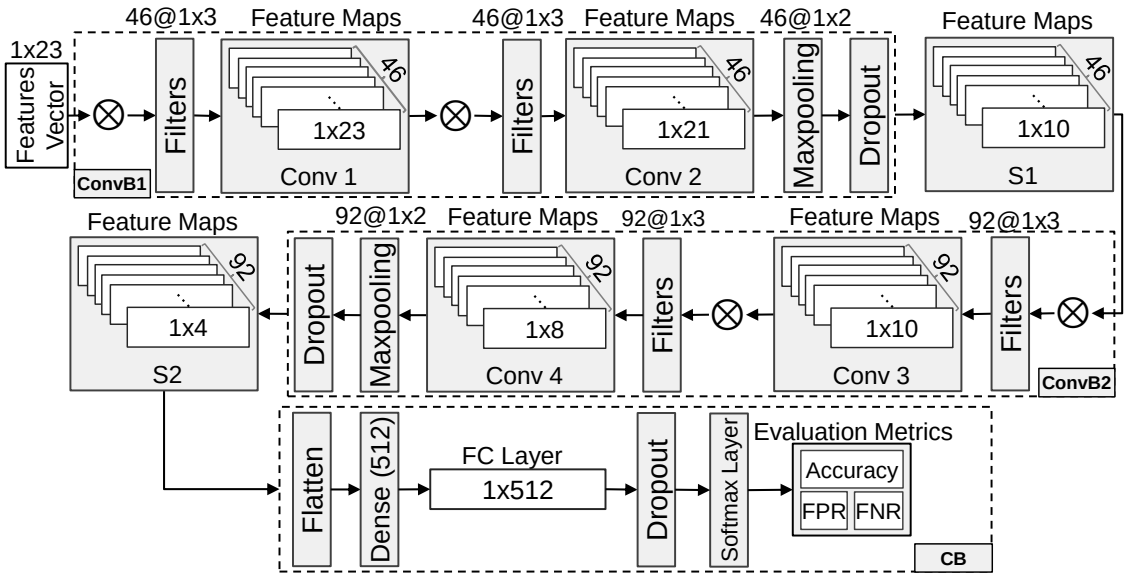


Figure 3.7: Internal design of the CNN architecture used for detection and classification task. Notice that 46@1x3, for example, refers to applying 46 filters each of size 1x3 on the input data. The design consists of four convolutional layers with maxpooling and dropout operations. Then, a dense layer of size 512 is used with a softmax activation function to output the model's prediction.

input and output layers. Given a feature vector X of length q and target y, the DNN-based classifier learns a function $f(.) : R^q \rightarrow R^o$, where q is the input's dimension and o is the output's dimension. With multiple hidden layers, the dimension of the output of every hidden layer decreases with transformation. Each neuron in the hidden layer transforms the values of the preceding layer using linearly weighted summation, $w_1 + w_2 + w_3 + ...w_q$, which passes through a ReLU activation function ($y(x) = max(x, 0)$). The output of the hidden layers is then fed to the output layer, and passed to a softmax activation function h, defined as $h(x) = \frac{1}{1+e^{-x}}$, outputting the prediction of the classifier.

Fig. 3.6 illustrates the DNN-based model utilized for training the IoT malware detector and classification system. The architecture of the DNN-based model consists of two consecutive and fully connected dense layers of size 1×100 connected to the input vector, followed by a dropout operation with a probability of 0.25. The output of the dropout function is fed to another two fully connected dense layers of size 1×100, followed by a dropout operation with a probability of

	B	G	M	T
B	0.995	0.002	0.002	0.002
G	0.007	0.977	0.015	0.002
M	0.015	0.008	0.977	0.000
T	0.009	0.018	0.041	0.931

(a) RF

	B	G	M	T
B	0.960	0.005	0.025	0.010
G	0.007	0.978	0.013	0.002
M	0.017	0.037	0.937	0.010
T	0.023	0.023	0.023	0.931

(b) DNN

	B	G	M	T
B	0.963	0.005	0.023	0.008
G	0.010	0.973	0.015	0.002
M	0.013	0.018	0.967	0.002
T	0.041	0.018	0.041	0.899

(c) CNN

Figure 3.8: Confusion matrices of IoT malware classification systems. Here, each row represents the actual class, whereas, columns represents the predicted labels. Labels are Benign (B), Gafgyt (G), Mirai (M), and Tsunami (T).

0.5. The output is then fed to the softmax layer. This design enables the extraction of deep feature representations and patterns from the feature vectors, and therefore, finding discriminative characteristics for detection and classification processes.

Convolutional Neural Network (CNN). The basic unit of the CNN network is a convolution layer, which consists of several filters convolving over the input to generate feature maps. Once a feature vector is fed into a convolutional layer, it becomes abstracted to a feature map, with the shape of (feature map height) $\times$ (feature map width) $\times$ (feature map depth), with two attributes: 1) convolutional kernels defined by a width and a height (hyper-parameters), 2) the depth of the convolution filter, which is equal to the depth of the input vector representation feature map. The CNN-based model constitutes of three blocks, two feature extraction layers, along with the classification layer. Fig. 3.7 illustrates the CNN-based model architecture used in this study.

Even though, from a machine learning perspective, the RF model seems the most suitable given the data representation and structure, and the limited number of features (*i.e.* 23), we also used DNN and CNN-based approaches as they show their capabilities in multiple applications for automatically producing high-quality encoding of the features towards a highly accurate classification. We note that alongside the aforementioned approaches, we also evaluate the performance of Logistic Regression- and Support Vector Machine-based architectures, however, due to their low performance, the results were not included in the evaluation.

	B	G	M	T
B	0.000	0.000	0.000	0.000
G	0.792	0.183	0.003	0.022
M	0.326	0.002	0.671	0.002
T	1.000	0.000	0.000	0.000

(a) RF-Small

	B	G	M	T
B	0.000	0.000	0.000	0.000
G	1.000	0.000	0.000	0.000
M	0.992	0.000	0.007	0.001
T	1.000	0.000	0.000	0.000

(b) RF-Median

	B	G	M	T
B	0.000	0.000	0.000	0.000
G	1.000	0.000	0.000	0.000
M	1.000	0.000	0.000	0.000
T	1.000	0.000	0.000	0.000

(c) RF-Large

	B	G	M	T
B	0.000	0.000	0.000	0.000
G	0.880	0.100	0.000	0.020
M	0.492	0.028	0.472	0.007
T	0.972	0.000	0.009	0.018

(d) DNN-Small

	B	G	M	T
B	0.000	0.000	0.000	0.000
G	1.000	0.000	0.000	0.000
M	0.876	0.093	0.030	0.000
T	1.000	0.000	0.000	0.000

(e) DNN-Median

	B	G	M	T
B	0.000	0.000	0.000	0.000
G	1.000	0.000	0.000	0.000
M	1.000	0.000	0.000	0.000
T	1.000	0.000	0.000	0.000

(f) DNN-Large

	B	G	M	T
B	0.000	0.000	0.000	0.000
G	0.078	0.608	0.017	0.297
M	0.135	0.000	0.836	0.028
T	0.885	0.009	0.037	0.069

(g) CNN-Small

	B	G	M	T
B	0.000	0.000	0.000	0.000
G	0.560	0.438	0.002	0.000
M	0.873	0.020	0.107	0.010
T	1.000	0.000	0.000	0.000

(h) CNN-Median

	B	G	M	T
B	0.000	0.000	0.000	0.000
G	1.000	0.000	0.000	0.000
M	1.000	0.000	0.000	0.000
T	1.000	0.000	0.000	0.000

(i) CNN-Large

Figure 3.9: GEA: Confusion matrices of IoT malware classification systems. Here, each row represents the actual class, whereas, columns represents the predicted labels. Labels are Benign (B), Gafgyt (G), Mirai (M), and Tsunami (T).

	B	G	M	T
B	0.000	0.000	0.000	0.000
G	1.000	0.000	0.000	0.000
M	0.995	0.000	0.005	0.000
T	1.000	0.000	0.000	0.000

(a) RF

	B	G	M	T
B	0.000	0.000	0.000	0.000
G	0.985	0.000	0.015	0.000
M	0.657	0.139	0.194	0.010
T	0.930	0.000	0.070	0.000

(b) DNN

	B	G	M	T
B	0.000	0.000	0.000	0.000
G	0.602	0.000	0.348	0.005
M	0.736	0.000	0.219	0.045
T	0.915	0.015	0.070	0.000

(c) CNN

Figure 3.10: SGEA: Confusion matrices of IoT malware classification systems. Here, each row represents the actual class, whereas, columns represents the predicted labels. Labels are Benign (B), Gafgyt (G), Mirai (M), and Tsunami (T).

Experimental Setup. We consider both malware detection and classification. Malware classification refers to identifying a malware family a sample, while the detection is simply indicating whether a sample is malicious or benign. Therefore, the detection task can be viewed as a binary classification task. The classification task aims to detect and identify the malicious behavior origin

(*i.e.* family).

Training Process. We trained the model architectures using 100 epochs with a batch size of 32. RF uses 100 decision trees, with no specified maximum length. For DNN and CNN, we used Rectified Linear Units (ReLUs) as the activation function, with softmax activation function at the classification layer. For regularization, we use dropout to prevent over-fitting and allow propagation of robust and distinct features through the model layers. Note that the value 0.25 is widely used within the machine learning community, as 75% of the neurons are considered for feature propagation between layers [126]. This provides better accuracy, and mitigates the noise caused by possible bias within the dataset.

Evaluation Metrics. We report the performance of the trained models using the following metrics: 1) The accuracy of the model, computed as the ratio of the correctly labeled samples (CLS) overall test samples ($|D|$), defined as: $CLS \div |D|$. 2) False Positive Rate (FPR), which is the number of incorrectly labeled benign samples (ILB) over the total number of benign samples ($|D_b|$), computed as $ILB \div |D_b|$. 3) True Positive Rate (FNR), represented as the correctly labeled malicious samples (CLM) divided by the total number of malicious samples ($|D_m|$), $CLM \div |D_m|$. 4) The F-1 score, defined as: F-1 $=$ 2TP$/$(2TP + FP + FN), where TP: the number of malicious samples correctly classified, FP: the number of benign samples incorrectly classified, FN: the number of malware samples incorrectly classified. 5) Misclassification rate, defined as the ratio of the incorrectly labeled samples over all the samples in the test dataset (*i.e.* $1 -$ accuracy).

We also report the confusion matrix when required. The rows represent the actual classes and the columns are the predicted labels. The value at a location (x,y) represents the portion of the samples of class x classified as y.

Table 3.3: Evaluation (%) of the IoT malware detection systems on normal samples (*i.e.* non-adversarial).

Architecture	Accuracy	F-1	FNR	FPR
RF	**98.90**	**99.22**	1.19	**0.83**
DNN	97.42	98.16	1.69	4.67
CNN	98.31	98.80	**1.12**	3.00

Table 3.4: Evaluation (%) of the IoT malware classification systems on normal samples (*i.e.* non-adversarial).

Architecture	Accuracy	F-1
RF	**97.71**	**97.71**
DNN	95.53	95.52
CNN	96.03	96.02

DL-SSMC: Design and Evaluation

This section presents DL-SSMC, **D**eep **L**earning-based **S**ingle **S**hot **M**alware **C**lassification approach. We describe the design and methods for DL-SSMC and present the evaluation.

DL-SSMC: System Design. The implementation of DL-SSMC incorporates machine and deep learning models trained using the extracted CFG-based algorithmic features for malware detection and classification tasks. In this approach, we follow the traditional learning approach. The input (X) to the model is a one-dimensional (1D) vector of size 1×23 representing the extracted features. Using a Softmax activation function, the model outputs whether the software is benign or malicious, alongside the predicted family in the classification task, *i.e.* Gafgyt, Mirai, or Tsunami. To this end, we utilize the aforementioned architectures to train the models for both detection and classification tasks.

DL-SSMC: Evaluation and Results.

DL-SSMC: Baseline Performance.

Table 3.5: GEA: Malware to benign misclassification rate (%) over IoT detection systems.

Architecture	Small	Median	Large
	10	23	1,075
RF	73.88	99.85	100
DNN	37.54	90.04	100
CNN	**18.84**	**64.78**	100

DL-SSMC: Detection Task. We design two-class detection DL-SSMC that distinguish IoT malware from IoT benign applications. The model is trained over 23 CFG-based graph-theoretic features categorized into seven groups. The models achieve an accuracy rate of 98.90%, 97.42%, and 98.31% with an F-1 score of 99.22%, 98.16%, and 98.80% for RF, DNN, and CNN, respectively. Table 3.3 shows the evaluation of each trained model. Notice that the RF-based model holds the highest performance, followed by the CNN model.

DL-SSMC: Classification Task. In addition to detecting the IoT malicious samples, we also design a four-class classification DL-SSMC. The classification task aims to evaluate DL-SSMC for classifying the malicious samples into their corresponding families. We achieved accuracy rates of 97.71%, 95.53%, and 96.03% for RF-, DNN-, and CNN-based models, respectively, as shown in Table 3.4. We also provide the confusion matrices (represented as a percentage of samples) in Fig. 3.8. Here, each row represents samples of an individual class, while the columns represent the predicted family.

DL-SSMC: Robustness Assessment against GEA. We investigate the robustness of DL-SSMC against AEs generated using GEA. In particular, we explore the impact of the size of the graph and discuss the fundamental overhead of using GEA. Note that all generated samples maintain the practicality and functionality of the original code. From the benign software, we selected three graphs as x_{sel}, the selected samples have small, median, and large sizes across the dataset. We then connected each of the graphs with every graph in the malware dataset.

Table 3.6: GEA: IoT classification systems misclassification rates (%). The CNN-based model perform the best, robustness-wise, under the small and median GEA graph embedding attacks.

Architecture	Size	Gafgyt	Mirai	Tsunami
RF	Small	81.66	32.88	100
	Median	100	99.33	100
	Large	100	100	100
DNN	Small	90.00	52.75	98.16
	Median	100	96.99	100
	Large	100	100	100
CNN	Small	**39.16**	**16.36**	**93.11**
	Median	**56.16**	**89.31**	100
	Large	100	100	100

Robustness of Detection Models. The results of DL-SSMC performance against AEs generated by GEA are shown in Table 3.5 and Fig. 3.9. Intuitively, a key finding is the impact of graph size on the misclassification rate, since the increase in the graph size, *i.e.* the included number of nodes, results in a higher misclassification rate. The main reason for the misclassification is the injection of benign-like patterns that introduce noise, which in turn, distorts the existing malicious patterns observed by the learning algorithm. Embedding larger graphs, for instance, introduces higher distortion to the feature space, considering the extracted 23 algorithmic features. This distortion affects the existing patterns learned by the machine/deep learning model, and therefore causes higher misclassification. In general, GEA achieves a misclassification rate of 100% by embedding a large graph, while achieving a low as 18.84% misclassification rate by embedding a small graph. Notice that while the RF-based model provides the best clean baseline performance, the CNN-based architecture shows the best robustness against embedding small and medium graphs.

Robustness of Classification Models. Table 3.6 shows the misclassification rates over the IoT malware classification task. Here, GEA achieves a misclassification rate of 100% from all malicious families using large graph embedding. Similarly, Tsunami is more likely to be misclassified to benign, as using median and large graph embedding misclassify all Tsunami malware to benign as shown in Fig. 3.9. Similar to the IoT malware detection system, the misclassification rates in-

Table 3.7: SGEA: Malware to benign IoT malware detection system evaluation. Here, MR: misclassification rate, AVG. Size: the average subgraph size required to achieve misclassification.

Architecture	MR (%)	AVG. Size
RF	99.17	6.28
DNN	90.21	6.83
CNN	**41.79**	**7.15**

Table 3.8: SGEA: Misclassification rate (%) over IoT classification systems. MR: misclassification rates.

Architecture	Gafgyt	Mirai	Tsunami
RF	100	99.50	100
DNN	100	80.59	100
CNN	100	**78.10**	100

crease with the increase in the number of nodes for the injected graph, and the CNN-based model shows better robustness (*i.e.* lower misclassification rate) against graph embedding, in comparison to its counterparts.

DL-SSMC: Robustness Assessment against SGEA. While GEA achieves a high misclassification rate, it comes with a computational cost and increased binary size that accommodates the combination of two samples into one. These costs are reduced by using SGEA, in which, the size of injection is reduced by carefully selecting a subgraph that achieves the adversarial objective.

Robustness of Detection Models. Table 3.7 show the results of SGEA against DL-SSMC detection models. SGEA achieves above 90% malware to benign misclassification rate against RF- and DNN-based models with less than 7 nodes subgraph embedding, outperforming the GEA approach with an average subgraph size of 6.28 and 6.83, respectively. However, for CNN, the misclassification rate is noticeably lower, 41.79%, as the CNN-based model shows better robustness against GEA and SGEA.

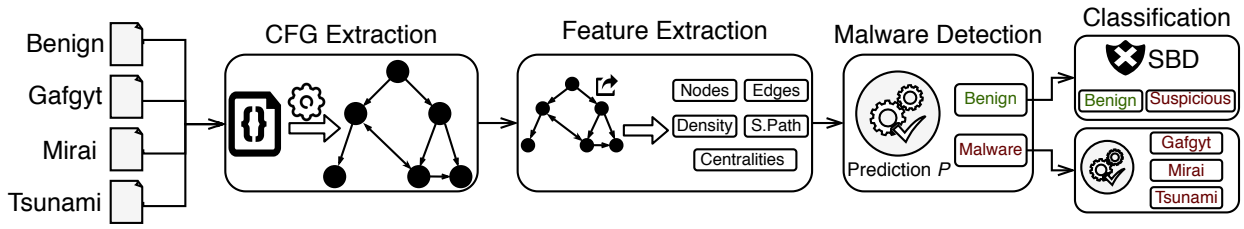


Figure 3.11: DL-FHMC system flow. First, corresponding CFGs of the IoT software are extracted, then, 23 algorithmic features are extracted from the CFGs. Afterward, an IoT malware detection system classifies samples into benign and malware, all malware samples are directed to IoT malware classification system, while benign samples are directed into suspicious behavior detection system (SBD) for further investigation.

Robustness of Classification Models. Table 3.8 and Fig. 3.10 show the performance of the DL-SSMC classification models against the SGEA approach. For instance, the SGEA approach successfully misclassifies all Gafgyt and Tsunami malware in all models, while having lower misclassification rates for Mirai malware. Further, the RF-model is considered the least robust model, as shown in Fig. 3.10a, as malicious samples were classified as benign.

Even though RF-based models provide the best classification performance on the clean dataset, this does not hold true under adversarial settings. Our evaluation shows that using a CNN-based model is noticeably better, considering a loss of $< 2\%$ performance on clean samples for both malware detection and classification tasks, while delivering higher robustness against GEA and SGEA.

DL-FHMC: Coping with AEs

Machine learning methods for malware detection and classification, *e.g.* DL-SSMC, are susceptible to AEs and fall short of delivering a robust system against adversarial settings. This motivates us to explore methods and alternative designs to cope with such vulnerabilities to adversarial attacks. In this section, we propose DL-FHMC, **F**ine-grained **H**ierarchical Learning for **M**alware **C**lassification, a robust system for malware detection and classification that leverages deep learning in a fine-grained and hierarchical manner to detect malicious behaviors.

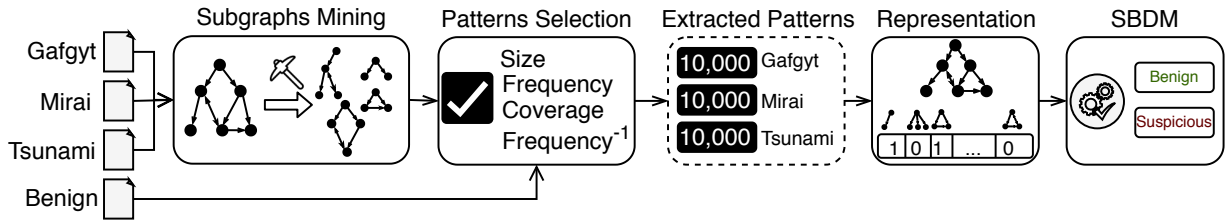


Figure 3.12: Suspicious behavior detection system design. The design consists of four modules, a subgraphs mining module to extract frequent subgraphs from three IoT malicious families. Afterward, the subgraphs are ranked by the pattern selection module, where the top 10,000 patterns of each malicious family are selected. Further, the CFG of each sample is redirected to the Suspicious Behavior Detector, and represented as a vector of size $30,000$. The vector representation is fed to Suspicious Behavior Detector Model (SBDM) to be classified into benign and suspicious.

DL-FHMC: System Design. The design of DL-FHMC consists of five components as illustrated in Fig. 3.11. Description of each component is in the following.

- **CFG Extraction.** This component is responsible for extracting the CFGs of the software samples using Radare2, and presenting them as labeled CFGs for further analysis.

- **Feature Extraction.** The feature extraction component calculates 23 algorithmic features from the samples' CFGs. Details of the extracted features are in Table 3.2.

- **Malware Detection.** The malware detection component utilizes the IoT malware detection models as in Table 3.3. The purpose of this model is to classify the samples into malware and benign. Samples classified as benign are directed to the suspicious behavior detection process, while samples classified as malware are directed to the classification model.

- **Malware Classification.** This component is fed by the samples classified as malware by the malware detection component. The goal of this component is to classify the sample into three IoT malicious families, *i.e.* Gafgyt, Mirai, and Tsunami. The design and architecture of this component are similar to the ones used for the classification task in DL-SSMC, except that it only contains the malicious classes (*i.e.* no benign label).

- **Suspicious Behavior Detector.** This component detects a potential suspicious behavior within the samples. Since the adversary may generate AEs with the purpose of fooling the system in assigning them to benign class, this component further investigates the potential of suspicious behavior within the benign-classified sample using the extracted CFG.

Suspicious Behavior Detector . Suspicious Behavior Detector is a graph mining-based technique to investigate suspicious malicious patterns within software samples classified as benign. Fig. 3.12 highlights the design of the Suspicious Behavior Detector, which consists of four modules, (1) subgraphs mining, (2) pattern selection, (3) data representation, and (4) suspicious behavior detection model. In the following, we describe each module.

(1) **Subgraphs Mining:** This module extracts common subgraphs within each IoT malware family. Using gSpan, we extracted and collected frequent subgraphs of a size range between 5 to 20 nodes from each malicious family. In particular, we used the gSpan algorithm to extract subgraphs from the training samples of each malicious family. This process took more than 160 hours to finish and resulted in over 2,150,170 patterns distributed as: 22,953 for Gafgyt, 127,217 for Mirai, and over 2,000,000 for Tsunami families. The extracted frequent subgraphs (patterns) for each malicious family are then subjected to further analysis.

(2) **Pattern Selection:** This module ranks the extracted patterns based on four factors: pattern size, frequency, coverage, and inverse frequency. For example, large patterns are assigned higher value since they are distinctive and more likely to be unique to their family. Moreover, large patterns can be further decomposed into smaller patterns. Further, the number of occurrences of a pattern within a malicious family is considered as an indicator of its maliciousness. On the other hand, less frequent patterns are more likely to be function-oriented and solely contribute to the functionality of the code rather than the general behavior of the malware family. Therefore, we excluded all patterns that occurred in less than 5% of the targeted family samples. The coverage of the pattern is defined as $\sum_{i=1}^{n} 1/\text{occurrence}_i - 1$, where n is a set of samples in which the pattern occurred, and $occurrence_i$ is the number of patterns contained within the sample i. For example, if a sample

contains only one pattern, that pattern will have the highest rank.

In addition, we compute the number of occurrences for each pattern in the benign training samples. Note that benign samples may have patterns similar to the ones in the malware due to the abstract nature of the CFG and the considered size and functionality of patterns. To ensure that all patterns hold some behavioral characteristics of the malicious family, we excluded all malicious patterns that appeared in more than ten benign samples. We filtered the patterns and selected the top 10,000 ranked patterns from each family to be its representative patterns. This results in a total of 30,000 malware patterns for the three malware families. We denote this set of patterns as P.

(3) **Data Representation:** To investigate an IoT software, we find whether each of the selected 30,000 patterns is a subgraph in the CFG of the software using the VF2 subgraph isomorphism algorithm [45]. Each sample is represented as a binary vector in the space of the patterns extracted in the previous module, i.e., $v \in \{0,1\}^{|P|}$. Specifically, we represent each sample by a hot-encoding vector v of size 30,000, where $v_i = 1$ if the i^{th} pattern is a subgraph of the sample's CFG, i.e., $v_i = 1$ if $p_i \subseteq G$, $p_i \in P$. Time-wise, representing a software's CFG as a hot-encoding vector may require several minutes and up to several hours, depending on its size (number of nodes) and structure.

(4) **Suspicious Behavior Detection Model:** The suspicious behavior detector model is a machine learning model trained on the feature representations extracted from the training dataset shown in Table 3.1. The goal of this module is to investigate suspicious behavior within the sample. If the sample is classified as suspicious, further analysis is required by an analyst or dynamic analysis approach.

Experimental Setup. We trained the Suspicious Behavior Detector model on the feature representation of the training dataset, where all malicious samples are labeled as suspicious. We did not incorporate AEs in the training process, as doing so may bias the evaluation of the system toward samples generated using the same approach (*e.g.* SGEA). Further, we generated AEs using both GEA and SGEA, to force the detection model to misclassify the IoT malware samples as benign,

Table 3.9: DL-FHMC classifier evaluation (%) on clean dataset for IoT malware classification task.

Architecture	Acc.	F-1	Benign	Gafgyt	Mirai	Tsunami
RF	**97.67**	**97.66**	**99.16**	**97.66**	**98.00**	92.69
DNN	95.74	95.73	95.33	97.33	95.33	93.60
CNN	96.38	96.38	97.00	97.16	95.66	**94.52**

thereby, directing the samples to the suspicious behavior detector component.

DL-FHMC: Evaluation and Results. The DL-FHMC system aims to establish a robust malware classification approach through hierarchical levels of abstractions. In the following, we evaluation the baseline classification performance, alongside the robustness of DL-FHMC.

DL-FHMC: Baseline Performance. The first task of the system is malware detection and classification. For the malware detection module, we used the same approach as in DL-SSMC, achieving the same baseline results (Table 3.3). All software classified as malware are then forwarded to the classification module, where they are labeled as Gafgyt, Mirai, or Tsunami. Table 3.9 shows the overall performance of DL-FHMC on the clean dataset (*i.e.* non-adversarial). We report the overall accuracy and F-1 score, along with the individual classes true positive rates. We note that as we use the malware detector as of DL-SSMC, the model is susceptible to adversarial attacks (*i.e.* GEA and SGEA). All benign samples, alongside all successful AEs, are forwarded to the suspicious behavior detection module.

DL-FHMC: Suspicious Behavior Detection Task. This component aims to further investigate the benign-classified samples based on patterns extracted from their structural components. The task of the suspicious behavior detector is to determine whether a given sample is signaling a suspicion of malicious behavior, and therefore it is operating as an AEs detection technique. We evaluate the Suspicious Behavior Detector using the original benign samples and malicious AEs. As shown in Table 3.10, the RF-based detector achieves an overall performance of 89.23% with a benign accuracy of 95% (*i.e.* false positive rate of 5%), while achieving a performance of 92.71% with a false positive rate of 10%. We note that the DNN- and CNN-based detectors are not effective as

Table 3.10: DL-FHMC Suspicious Behavior Detector evaluation (%) on benign and adversarial samples. DO refers to Data origin.

DO	FPR	GEA			SGEA	Overall
		Small	Median	Large		
RF	1	74.84	70.04	53.06	74.26	74.24
	3	84.86	79.10	100	79.31	88.05
	5	86.79	82.48	100	81.90	89.23
	10	**89.24**	**97.40**	**100**	**86.90**	**92.71**
DNN	1	63.28	59.50	49.76	64.90	67.28
	3	63.81	59.66	49.28	65.72	67.09
	5	63.72	59.40	48.79	65.78	66.54
	10	63.21	58.56	47.52	65.54	64.96
CNN	1	62.70	55.50	49.76	56.77	64.74
	3	62.87	55.40	49.28	57.30	64.37
	5	62.69	55.04	48.79	57.16	63.73
	10	62.06	54.02	47.53	56.49	62.02

modalities for suspicious behavior detection.

While GEA large graph-based AEs achieve 100% misclassification rate, DL-FHMC can detect them with 100% accuracy. Further, while the detection performance for small GEA embeddings is relatively lower than the other configurations, the smaller the x_{sel} graph size is, the lower the success rate of the attack. Fig. 3.13 shows the performance of the detector with different false positive rates ($1 -$ benign detection accuracy).

As shown in the results, using DL-FHMC enables systematic methods of coping with adversarial manipulation to malware. When suspicious behavior is detected for a given sample, other methods can be adopted to further analyze the sample in order to provide a secure and robust evaluation of malicious activities. In general, using CNN-based architecture for baseline malware detection and classification tasks, while using the RF-based model for suspicious behavior detection provides the best trade-off between the accuracy and robustness, as it minimizes the number of samples misclassified by the malware detector, and forward fewer samples toward the suspicious behavior detection system.

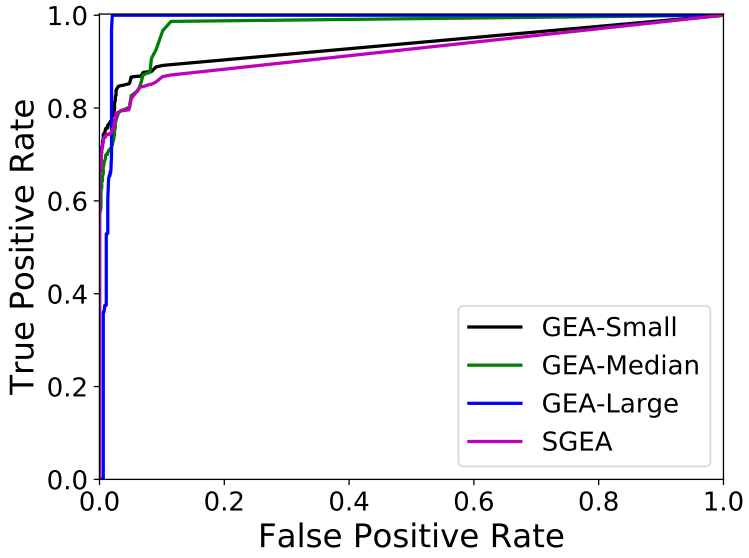


Figure 3.13: DL-FHMC: RF-based suspicious behavior detector ROC performance over GEA and SGEA attacks.

Discussion

DL-FHMC: Cost of Security. The security of machine learning algorithms is important for adoption in many applications. In the malware classification field, AEs pose critical security implications as emerging studies have shown that AEs can fool the machine learning-based malware detection system [13, 12, 129]. However, limited studies have investigated potential defenses. In this work, we show that launching adversarial attacks against malware detection systems can lead to a misclassification rate of as high as 100%. To cope with such adversarial settings and capabilities, we introduced DL-FHMC that operates on multiple levels of behavioral analysis of the software to ensure its security. Since AEs are derived from a combination of benign and malware components, detecting them is a challenging task and often comes at the cost of misclassifying a portion of benign samples as malicious, hence producing false alarms. For example, in our experiments, we show the trade-off (*i.e.* the cost) of performance of the suspicious behavior detector and the benign samples misclassified as adversarial, know as the sensitivity of the detector, illustrated in Fig. 3.13.

DL-FHMC Robustness. The suspicious behavior detector in DL-FHMC consists of a machine/deep learning model trained on the vector representations obtained by checking the existence of the predefined list of malicious subgraphs extracted and filtered using a large dataset of IoT malware. Using the VF2 subgraph isomorphism algorithm [45] on a list of 30,000 malicious subgraphs, the vector representation of the CFG is generated indicating whether malicious patterns exist in the CFG. Ideally, the suspicious behavior detector represents the benign samples into a vector of all zeros. In our dataset, $\approx 85\%$ of the benign samples are represented as a vector of all zeros, indicating that none of the malicious patterns were captured within them, *i.e.* none of 30,000 malicious subgraph structure were found within their CFGs.

Considering the highly-accurate current state-of-the-art methods for detecting malware samples, our approach can be viewed as a robust layer on top of such methods. We argue that in the context of adversarial attacks, malware detection systems should be robust against perturbation to the original samples. Otherwise, they will suffer from the effects of AEs (i.e., misclassification). Since running the VF2 subgraph isomorphism algorithm against the 30,000 subgraphs introduces a computational overhead, a malware detection system benefits from our approach by testing samples that are classified as benign (with low classification confidences) indicating the possibility of adversarial scenario for evading the malware detection process.

Since the suspicious behavior detector in DL-FHMC uses a comprehensive list of malicious patterns, generating successful AEs against DL-FHMC requires modifying the functionality of the malware to conceal malicious patterns (*i.e.* subgraphs) that exist in the CFG. This requires applying direct modifications to the control flow of the program, which in turn, contrasts the practicality and functionality requirements of a practical AE.

We note that detecting suspicious AEs is not a trivial process even with using a subgraph isomorphism matching algorithm, and it requires careful considerations in designing the system. For example, due to the abstract graph modality, 15% of the benign samples have suspicious subgraph structures within their CFGs. This does not, in most cases, indicate an embedded malicious

functionality, but due to the diversity of the benign dataset, control flow structures similar to the ones occurring in the malicious samples may be found. This motivates using machine/deep learning methods to determine whether a sample is malicious considering the generated binary vector representation of the 30,000 subgraphs. Moreover, considering the evolution of malware and the emergence of new variants, the list of malicious subgraphs should be continuously updated to incorporates emerging patterns.

Suspicious Behavior Detector as an Individual Modality. The suspicious behavior detector is a machine learning-based CFG subgraph malicious patterns detector. It has its own feature space and representation, and independent model. While in this study we consider the detection and classification modules similar to the ones considered in DL-SSMC, the suspicious behavior detector is not exclusive to them only. In reality, following the same process, the detection and classification modules can be replaced with models that operate under different data representations. Then, forwarding all samples classified as benign to the suspicious behavior detector enables the AEs detection.

CFG for Malware Classification. Using CFG-based representations for malware detection and classifications address different challenges that may be raised by other representation techniques [112]. For instance, binary representations are susceptible to binary padding and injection. However, the added binaries are not typically executed, and therefore will not appear in the extracted CFG. Similarly, modifications on the header of the file, and stripping the binaries will not affect the final CFG. In general, changes that produce decision branches, such as conditions and loops, are the only modifications that alter the extracted CFG. We argue that adding API and system calls will simulate the GEA and SGEA attacks, as they only result in introducing new nodes, while the original structure is maintained, and thus can be accurately detected by DL-FHMC. Generating CFG-based AEs assumes a more powerful adversary, and even though we do not address other attacks that generate AEs using the alteration of the malware binaries or code, our work extends the robustness against such methods as changing such methods do not conceal existing malicious

patterns from the CFG.

Summary & Concluding Remarks

This work introduces DL-FHMC, a novel hierarchical approach for robust malware detection and classification with AEs detection. To set out, first, an in-depth analysis of malware binaries is conducted through constructing abstract structures using CFG, which are analyzed from multiple aspects, such as the number of nodes and edges, as well as graph algorithmic constructs, such as average shortest path, betweenness, closeness, density, etc. Then, we evaluate the robustness of the traditional CFG-based IoT malware classification approaches against GEA and SGEA, achieving a misclassification rate of up to 100%. To address this, we use different graph mining techniques, CORK and gSpan, to extract malicious discriminative graphs from the malicious software, and use it as a modality to detect malicious behavior. Through our evaluation, DL-FHMC achieves a high malware detection and classification accuracy, as well as AEs detection performance under different GEA and SGEA configurations.

CHAPTER 4: SYSTEMATICALLY EVALUATING THE ROBUSTNESS OF ML-BASED IOT MALWARE DETECTION SYSTEMS

IoT malware have been the focus of the security research community and the industry alike. These efforts have resulted in various malware detection approaches, intended for safeguarding the IoT infrastructure against increasing targeted attacks. These proposed detectors leverage the traditional signature-based approach or the capabilities of the learning algorithms to build Artificial Intelligent (AI)-based detectors. These detection systems leverage modalities generated through static and dynamic software analysis techniques, along with deep learning and natural language processing, for generalizing detection to previously unseen IoT malware [109].

Considering that these techniques are heavily dependent on the specific data used for their training and testing, it is plausible that they would have a reduced performance when tested in an uncontrolled environment due to various practical settings. For example, the constant evolution of malware that employ obfuscation may impact the performance of these detectors over time, especially the static-based techniques. While packing is widely used among malicious software, it is not exclusive to malware. This limit the usage of packing as a detection modality, since that may result in a large number of false positives. Even in the absence of packing, malware detection systems have been shown to be susceptible to adversarial attacks. An adversary can manipulate the features of any software, directly or indirectly, to force the detector to output the adversary's desired decision [122, 14, 86].

A common practice for inspecting software is using online scan engines, such as VirusTotal [8], which embody the aforementioned techniques for malware detection and provide reports that contain the detection results of a pool of anti-virus engines. Additionally, these online scanners are utilized by the malware developers to check if their malicious payloads can evade detection from the anti-virus engines before starting a malware campaign [68]. Altogether, before deploying such malware detection systems in practice, it is essential to understand the shortcomings of state-of-the-

art IoT malware detection systems under adversarial settings that can be abused by the adversaries towards future malware campaigns.

In this work, we examine state-of-the-art malware detection approaches, including those that rely on different representation and learning algorithms. We consider techniques that represent the software as a sequence of binary, static features extracted from the disassembly, and graph. These representations yield a promising detection performance, with higher than 99% detection accuracy [97, 17, 147]. However, our findings highlight the instability of the learning algorithms in learning useful fundamental patterns that represent the difference between benign and malicious software.

By systematically evaluating the robustness of various malware detectors, we demonstrate that manipulating the malicious software with functionality-preserving operations, such as stripping and binary padding, significantly reduces the detectors' performance. Towards this, we generate four equivalent binaries for each software using means of packing (with different compression levels), stripping, and padding. We evaluate each of the resultant software against various IoT malware detection approaches, along with the industry-standard malware detection engines. The results show a concerning behavior, where one or more detectors fail to hold a reasonable performance (lower than 50% detection rate) in detecting malware mutations. Fig. 4.1 shows the different phases of analysis strategy; feature representation, software manipulation, and evaluation of several ML-based malware detectors.

Summary of Completed Work

This work highlights the discrepancies between the capabilities of the adversary and the assumed adversarial capabilities by the research community. Particularly, we make the following contributions:

1. ***Validity of the baseline***: We examine nine state-of-the-art malware detection representations and three learning algorithms and evaluate their performance using a total of 5,295 IoT

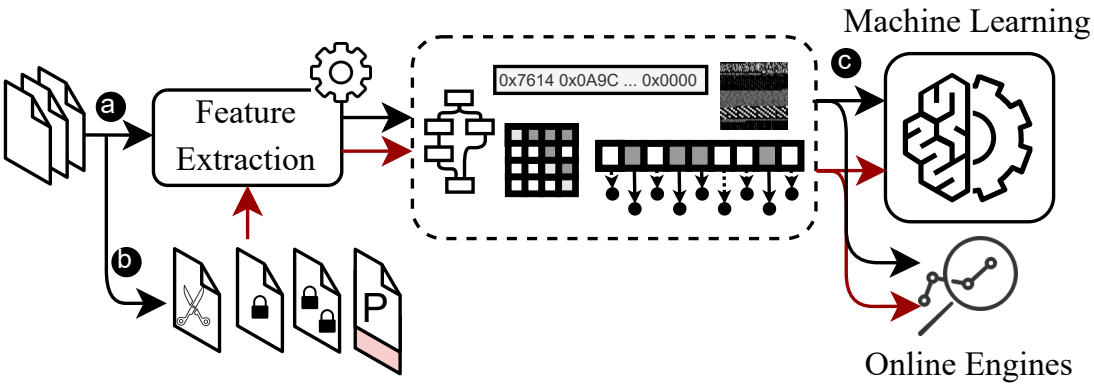


Figure 4.1: The system pipeline. The software binaries are (a) represented using different state-of-the-art approaches, and (b) manipulated using functionality preserving operations, such as packing, stripping, and padding. The corresponding representations of the original samples and manipulated ones are then (c) tested against pre-trained ML-based malware detectors and industry-standard detection engines.

software binaries. The evaluation shows the effectiveness of each representation in detecting malicious IoT software with high accuracy in a level playing field.

2. ***Model instability***: We investigate the stability of the baseline malware detectors. Our results demonstrate the inconsistency of the learning process, *i.e.* with the introduction of a small random perturbation to the input space, the detector is rendered useless (outputs random label).

3. ***Vulnerability to adversarial settings***: We examine the robustness of the IoT malware detectors under white-box and black-box adversarial settings, resulting in an accuracy reduction of up to 70%.

4. ***Vulnerability to binary manipulation***: We evaluate the malware detectors against three binary manipulation techniques, including packing, stripping, and padding. These techniques are practicality and functionality preserving, where the generated software is identical in functionality to the original software. Our evaluation shows that such software is capable of misleading the state-of-the-art malware detectors.

5. ***Vulnerability of industry-standard malware detectors***: We evaluate the behavior of AI-based industry-standard malware detection engines against binary manipulation. The evaluations show that most of the engines are rendered useless upon slight modification of the software.

6. ***Adversarial Surface Reduction***: We investigate the binary-level adversarial attacks surface, identifying volatile feature space and representations. We then propose three software preprocessing techniques to limit and mitigate the effect of these attacks. Our evaluation uncover that volatile features are used by online engines, and can be exploited to reduce the detection rate by 30%.

Background

The increasing security concern for IoT devices has been paralleled by an increasing body of work in the area of IoT security, particularly addressing malware analysis and detection. Building towards our work, it is important to outline the efforts that propose IoT malware detection systems and the methods of evasion that will elucidate the susceptibility of the malware detection systems to various adversaries.

Malware Detection. Prior works have shown the potential and feasibility of ML to detect malware with more than 99% accuracy [22, 29, 137, 97]. The performance of these detection systems depends on the choice of software representations, which are a result of two common analysis techniques. In *dynamic analysis*, a malware is executed in a monitored sandbox environment. The behavioral patterns are then used as feature representation. However, dynamic analysis is time and space-consuming, thereby limiting its scalability [140].

The *static analysis* involves analyzing the binary executable without executing it. The fast and scalable extraction of representations makes static analysis the primary analysis technique for malware detection. Malware binaries have multiple features that can be statically extracted and used

as modalities for malware representation.

Selected Representations. Altogether, we focus on representations that are (1) extensively used in the prior works, (2) fast to generate, and (3) can be extracted for malware detection on the fly. We summarize the utilized representations in the following.

1. A common strategy is to transform the malware into a *grayscale image*. Particularly, the byte-code is visualized as a grayscale image of a fixed size of $(h \times w)$ where every byte is a pixel in the image.

2. *CFG adjacency*. Another strategy is to extract the assembly instructions by disassembling malware and further transforming them into a Control Flow Graph (CFG) by dissecting them into basic blocks depending on the instruction branching or jumps. The CFG is then represented as a square matrix representing edges between nodes.

3. *CFG algorithm*. Graph algorithms have been augmented to extract graph attributes that represent the connectivity patterns in the CFG. These features are exhibited in Table 4.1.

4. *Strings* are a sequence of printable characters in the binary codebase. The strings of a program are analyzed to understand the possible behavioral patterns of the malware and can also be used to prepare a sandbox environment for the dynamic analysis [46].

5. *Segments* are necessary for program execution. They describe the memory layout of an executable and is interpreted by the kernel during load [106]. Within every segment, there may be code or data divided among *sections*, such as *.text* and *.rodata*. Binaries contain symbol tables which are used as references for linking and debugging [106].

6. *Symbols* are symbolic references to code or data and include global variables or functions. Every executable generally has two symbol tables: the symbol table that contains all symbol references and the dynamic symbol table which only contains references for dynamic symbols [106].

Table 4.1: The CFG extracted algorithmic features, categorized into seven groups. When possible, the minimum, maximum, median, mean, and standard deviation are calculated.

Feature category	# of features
Betweenness centrality	5
Closeness centrality	5
Degree centrality	5
Shortest path	5
Density	1
# of Edges	1
# of Nodes	1
Total	23

7. *Hexdump* represents a malware as a sequence of hexadecimal values, where each value represents two bytes (in 0-255 range), the frequency of which is then recorded as a vector of size 1×256.

8. *Feature fusion* represents a unified (combined) representation of all of the aforementioned representations.

For the completeness of the study, we include malware representations proposed by works that are not strictly IoT malware-specific. Table 4.2 summarizes the malware representations that have been proposed for malware detection, and utilized in this work.

Representation Evasion. In the literature, several software evasion and manipulation techniques were proposed for malware mutation and misclassification. In the following, we briefly discuss the commonly-used software evasion techniques.

Binary Packing. Packing is mainly used by malware authors to thwart detection or analysis by detectors, analysts, or reverse engineers. A packer software is augmented to compress or encrypt an executable, such that the code and data are hidden from the analysts. Considering that portions of the executable are compressed, it needs to be decompressed before it is executed in memory [106].

In general, the packer software has two programs, the packer program, and the stub program. The

Table 4.2: The state-of-the-art static analysis representations used in this work. Most of the representations require reverse-engineering (R.E.), while image-based representation directly used the raw binaries (Bin.). CODE: features extracted from the disassemble binaries.

Type	Feature	Work	Bin.	R.E.	Graph
Binary	Image	[80, 137, 148, 97]	✓	✗	✗
CFG	Adjacency	[147, 77, 37]	✗	✓	✓
CFG	Algorithmic	[22, 37, 29]	✗	✓	✓
CODE	String	[17, 29]	✗	✓	✗
CODE	Symbols	[17, 29]	✗	✓	✗
CODE	Sections	[17, 29]	✗	✓	✗
CODE	Segments	[17]	✗	✓	✗
CODE	Hexdumps	[17]	✓	✓	✗
CODE	Combined	[17, 29]	✓	✓	✗

packer program packs the software, while the stub is responsible for deobfuscating the software. While there have been many packing programs, such as *DacryFile* by Grugq, *Burneye* by Scut, *Shiva* by Neil and Shawn, and *Maya's Veil* by Ryan, the *Ultimate Packer for eXecutables* (UPX) [3] is the one most commonly used [46]. UPX utilizes the UCL data compression library algorithm [2] which uses in-place decompression, and does not introduce memory overheads.

Binary Stripping. Stripping is utilized to hide the information that may leak the functional strategies of a software. A codebase can be compiled with no standard library linking (*gcc-nostlib*). Alternatively, parts of the ELF file can be hidden such that the different constituents of the binary format can be obfuscated such that the interpretation can be halted. The resultant binaries would be void of information such as debug and relocation information, section headers, and symbols [1].

Adversarial Evasion. With the rapid growth in ML adoption in critical fields, it is essential to understand and assess the robustness of such techniques to several adversarial settings. These settings include adversarial examples, in which an adversary crafts perturbation to misguide the model output to its desired label by applying a minimal perturbation to the original sample [111].

Given a model objective function $f(.)$ and a sample represented by the vector x, the aim of the adversary is to introduce perturbation (δ) in the feature space $x' = x + \delta$ such as $f(x) \neq f(x')$.

Crafting the perturbation can be derived from two perspectives: targeted and non-targeted attacks.

Targeted attacks. The adversary in this attack generates an adversarial example x' that forces the classifier to misclassify into a specific target class t. For instance, the adversary generates a set of malicious IoT software samples, which are classified as benign. That is: $x' : [f(x') = t]$.

Untargeted attacks. The adversary's goal is to misclassify the output of the model to any class other than the original label. That is $x' : [f(x') \neq f(x)]$. In this work, we only consider the two-class classification task, where targeted and untargeted attacks behave the same.

Adversarial attacks can be launched under different adversarial capabilities that allow for either black-box or white-box attacks. In a white-box attack, the adversary has full knowledge of the inner networking paradigm of the model. In a black-box attack, the adversary has only access to the model via an oracle and can only observe the output of the model.

Several methods have been proposed to generate adversarial examples by directly perturbing the feature space in both black-box and white-box settings [67, 75, 103, 88]. For example, Carlini and Wagner [40] proposed generic adversarial attacks against distilled Neural Networks (NN), which showed its effectiveness against several state-of-the-art "robust" deep learning NN.

While initially designed to exploit image-based classifiers, where perturbation can be directly applied to the image pixels [111, 110, 138], adversarial attacks showed high success in malware detection while preserving the software functionality and executability [69, 14]. At the binary-level, several studies [85, 86] generated practical adversarial examples by appending binaries to the original file. While it is effective against signature- and binary-based classifiers, it can be countered by reverse-engineering the software to extract the corresponding representations.

Other studies [12, 14] introduced adversarial attacks on the execution flow of the code, by injecting benign functionalities within the malware and vice versa. However, such a perturbation should be applied to the source code, and is only possible by the malware author, unlike the binary padding approach.

To investigate the effectiveness of different malware representation and learning approaches, we

examine a wide set of adversarial settings, including direct generic and modified adversarial attacks, as well as the black-box adversarial settings.

Threat Model

Learning algorithms are widely used to obtain state-of-the-art performance in several fields, including malware detection. However, the usage of ML in critical domains is subject to adversarial attacks. In the following, we discuss the threat models used for systematically evaluating the robustness of the malware detectors.

Gaussian Noise. A stable learning model is argued to be immune to misclassification under the introduction of Gaussian noise in the feature space, as unguided perturbation is unlikely to disrupt the existing patterns to some extent [74, 71, 131].

A correctly trained model that can distinguish benign and malicious samples with high confidence, is constraint by three factors. 1. *Data representation*: A robust software representation should contain meaningful patterns that can distinguish the malicious from the benign software, 2. *Learning algorithm*: The learning algorithm should be able to capture such patterns even at a higher dimensionality without over-fitting or under-fitting, and 3. *Training data*: The trained model should be generalizable to unseen new samples, and samples that are not fundamentally different from the ones in the training dataset. This requires the training data to be cohesive and the samples of each class to be an accurate representation of that class. While the first two factors are considered, the third is an open challenge, and we consider it out-of-scope of this work.

In this work, we use the Gaussian noise as a metric to measure the stability of the representations. Given the model objective function $f(.)$, data points (samples) $x \in X$ with feature space of n features, the output of the model is defined as $y = f(x)$. The Gaussian noise is then calculated as follows:

$$x'_i = x_i + max(X_i) \times \delta, \quad \forall i \in n,$$

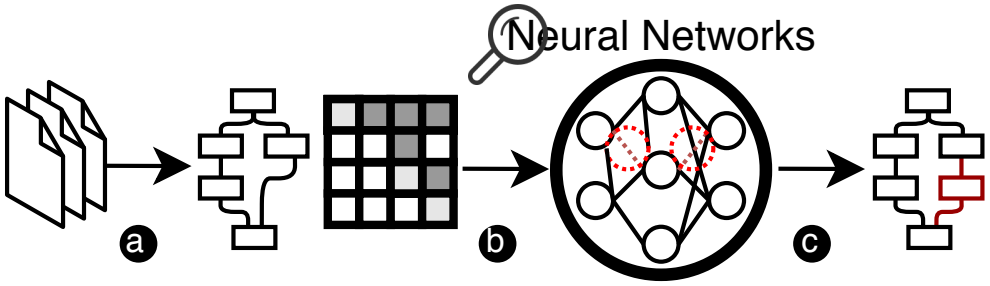


Figure 4.2: Graph manipulated attack overview. The software is reverse-engineered and (a) represented as CFG and corresponding adjacency matrix, (b) using the pre-trained neural network, (c) white-box C&W-based perturbation is crafted and applied to the CFG.

where X_i is a list of the i^{th} features of all $x \in X$. A stable model is then defined as:

$$f(x) = f(x'), \text{ if } \delta < threshold.$$

In this work, we do not introduce a cut-off threshold for a stable model. However, we observe the model's behavior when a perturbation in the range of [1%, 100%] is introduced. Ideally, the relationship between the accuracy and perturbation should be linear: with an increasing perturbation, the accuracy should linearly decrease, *e.g.* to reach random (50%) at 100% perturbation given the two-class classification task. We note that this attack will not generate practical adversarial examples, as it applies the perturbation to the feature space directly. Rather, it is used to measure the detectors' stability.

Graph Manipulation. This configuration targets the graph-based representations, including the adjacency- and algorithmic-based representations extracted from the software's corresponding CFGs. Given a CFG $G = \{V, E\}$, where V is the set of nodes in the graph, and E is the set of edges, the adversary's goal is to introduce a carefully crafted perturbation that misclassifies the system to the desired output. To introduce such a perturbation, we used the adjacency matrix representation as a baseline to craft the perturbation. Then, the Carlini & Wagner L_∞ (C&W) attack [39] is used to craft the perturbation under the white-box settings. The C&W is a gradient-based attack

that optimizes the penalty and distance metrics on L_∞ norms in the process of generating adversarial examples. This method ensures that the added perturbation will be minimal while causing misclassification.

Using the adjacency matrix representation, the adversary aims to craft a perturbation $\delta \in \mathbb{R}^{d \times d}$ as a domain-specific range of possible features that can be observed in ordinary samples. This perturbation achieves the adversarial goal if $y = f(x) \neq f(x + \delta)$, where y' is the classifier's prediction after applying the perturbation δ to the original feature space x. Fig. 4.2 shows the outline of the attack. To keep the generated CFG realistic, we limit the actions done by C&W attack to only adding nodes and edges. This is done by modifying the original attack to prevent deleting existing edges, and only limiting the process to adding edges.

While CFG manipulation preserves the original functionality [12, 14], we do not have access to the source code of the samples. Therefore, we cannot generate practical adversarial binaries using CFG manipulation. Given that, we used this attack to evade the graph-based detectors using direct white-box attacks on NN-based adjacency matrix-based classifier, while transferring the attack to the remaining CFG-based classifiers.

Static String Manipulation. Another white-box attack is the string manipulation attack. In this representation, the software is represented as a vector V of bag of words W of size $1 \times |W|$, where $|W|$ is the number of words considered in the representation. Similar to the graph manipulation attack, we used C&W L_∞ attack to craft a minimal perturbation to misclassify the model. Given that the crafted perturbation cannot be applied directly to the binaries, we consider it as a practical attack under the assumption of the availability of the source code. We evaluate this attack by crafting the perturbation using the NN baseline and transferring the attack to the remaining baseline models.

Binary Packing. Recall that a binary executable can be packed using packer software, such as UPX. The ML-based detectors utilize the features, such as raw binaries, strings, and segments, from the malware. These features are, however, suppressed from packing. In this attack, we pack

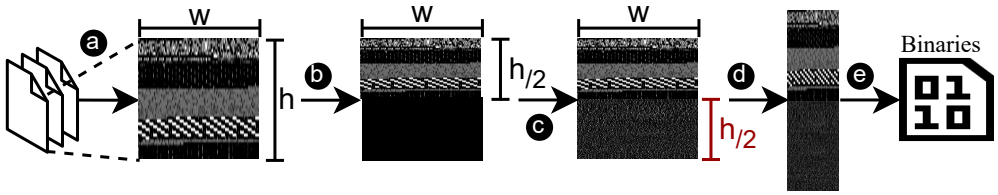


Figure 4.3: Binary padding attack overview. (a) The software is represented as an $h \times w$ image. (b) The content of the image is then compressed into the size of $\frac{h}{2} \times w$. (c) Using C&W attack, we generate perturbation on the remaining half $\frac{h}{2} \times w$ of the image. (d) The generated image perturbation is then rescaled to the original size of the software, and then (e) reshaped to a 1-D vector represented the binaries to be appended.

the malware and probe the performance of the representation used in the literature. Moreover, UPX supports different degrees of packing. For this study, we utilized the default settings and the best compression method of UPX.

Binary Stripping. Recall that a binary can be stripped of information without affecting its executability. In this attack, we probe the impact of a stripped binary on an ML-based detector's performance. Particularly, we strip the binaries of their debug information and the symbol information that are not needed for relocation.

Binary Padding. In this attack, the adversary aims to craft a white-box practical (executable) adversarial example by appending binaries to the end of the software binaries. Fig. 4.3 shows the process of generating perturbation in the white-box settings for image-based representation baselines. For a software s of size z_s represented as an image img of size $h \times w$, we first compress the content of the image into the space $\frac{h}{2} \times w$. Afterward, we craft a minimal perturbation using C&W attack. To prevent the attack from applying a perturbation to the upper half of the image, the attack is modified allowing changes in the lower half of the image. After the evasion, we convert the generated lower half of the image of size $\frac{h}{2} \times w$ back to the actual size z_s of the software s, and then converting it to 1-D vector by concatenating the rows. We note that this attack will introduce a perturbation size of 100%, as the perturbation has the same size as that of the original file, and the generated software s' will be of size $z_{s'} = 2 \times z_s$. This attack generates an adversarial software that

is executable. We evaluate the generated software on the image-based baseline models, in addition to the other representations by re-extracting the features from the manipulated software.

Dataset Overview

To systematically analyze the robustness of state-of-the-art malware detectors, we start by collecting a dataset of malicious and benign IoT software in binary forms. The dataset was collected between November 2018 and December 2020, where 3,000 malware samples of three families—Gafgyt, Mirai, and Tsunami—were retrieved from CyberIOCs [6], VirusTotal [8], and VirusShare [4], in addition to 2,295 benign samples, compiled from source files on GitHub [52] with different optimization levels.

Ground Truth Class. We used *VirusTotal* [8] to validate the malicious and benign samples in our dataset. The samples were first uploaded to VirusTotal. After 24 hours, the scan results corresponding to each sample were retrieved.

Data Augmentation. As aforementioned, the dataset samples are transformed to different representations: (1) Represented as images to be fed into an image-based classifier. (2) Using *Radare2* [7], a reverse-engineering open-source framework for analyzing binaries, the samples were reverse-engineered to obtain various features, such as strings, symbols, sections, and segments. (3) Hex-dump representation is used to represent the "*.text*" section of the binaries. (4) The software CFG is extracted using *Radare2*, which then used to generate the software adjacency matrix and different graph-theoretic features, shown in Table 4.1.

Robustness Analysis

In the arm race between malware detectors and malware authors, malware detection and identification require an accurate understanding of the capabilities of malware authors. In this section, we evaluate the existing on-the-fly static-based malware detection techniques against executability-

and functionality-preserving software binary manipulations.

Experimental Setup. Towards evaluating the robustness of the state-of-the-art IoT malware detectors, the dataset is transformed using the nine representations. Then, four learning algorithms are used to establish the baseline classifiers.

Learning Algorithms. Several classification algorithms have been adopted and used in various domains in IoT malware detection and classification [22, 124].

In this study, we evaluate the robustness of four ML algorithms, namely, Logistic Regression (LR), Random Forest (RF), Convolutional Neural Networks (CNN), and Deep Neural Networks (DNN). The selection of learning algorithms is for multiple reasons. They are (1) commonly used in this domain, (2) fundamentally different in the learning process, (3) highly sophisticated approaches, such as DNN and CNN, and simpler ML algorithms, such as LR and RF. For instance, the LR-based classifier is selected to extract the relationships between variables in the feature space, with no deep representations. CNN, on the other hand, was selected to extract deep patterns in higher dimensionality. The nature of the selected models will help in investigating the robustness and stability of the feature representations and the learning algorithms more accurately and on a larger scale. In the following, a brief description of each learning algorithm is provided.

Logistic Regression (LR). LR models a binary dependent variable, known as binary classification ("0" or "1"), using a logistic function. Given (X, Y) as an input training set, LR trains to classify segments as positive ("1") and negative ("0") by estimating and optimizing the boundary between the two classes ("0", and "1") and minimizing the following function:

$$\mathsf{Loss}(f(X), Y) = \begin{cases} -\log(f(X)), \ Y = 1 \\ -\log(1 - f(X)), \ Y \neq 1 \end{cases},$$

where $f(X)$ is the LR model current prediction, and Y is the ground truth labels.

Random Forest (RF). RF learning algorithm allows for variance reduction in the output of the

Table 4.3: Accuracy (%) of the baseline models. Each representation is evaluated using LR, RF, and NN-based classifiers. Note that almost all representations hold high performance (up to 99%) in detecting IoT malware.

Type	Feature	LR	RF	NN
Binary	Image	99.90	99.81	100
CFG	Adjacency	91.67	89.90	92.25
CFG	Algorithmic	90.20	99.22	92.09
CODE	String	98.48	99.43	98.48
CODE	Symbols	98.77	99.43	97.82
CODE	Sections	100	100	58.16
CODE	Segments	98.39	100	58.16
CODE	Hexdumps	98.96	99.24	98.48
CODE	Combined	100	99.90	57.79

individual trees and mitigates the effect of noise on the training process. RF consists of N decision trees and is used with non-linear classification tasks. Each tree is trained on random features to allow for variance reduction in the individual trees' output and decreases the effect of noise on the training process. The final prediction is calculated by a majority prediction vote of the decision trees or by the average prediction of all the trees.

Convolutional Neural Network (CNN). CNN is a powerful deep learning model used in image classification and pattern recognition. A convolution layer, which generates feature maps, is the basic unit of the CNN network. Once a feature vector is fed into a convolutional layer, it becomes abstracted to a feature map, with the shape of (feature map height) $\times$ (feature map width) $\times$ (feature map depth). CNN performs well in extracting patterns in higher dimensionality when the pattern location, in the feature space, is irrelevant. Therefore, we use the CNN model with image-, CFG adjacency-, and CFG algorithmic-based feature representations.

Deep Neural Networks (DNN). DNN model is used to extract deep encoded patterns and contains multiple consecutive fully connected layers. In the learning stage, the model configures the

parameters of each single layer l, denoted by:

$$h^{(l)} = a(W^{(l)} \times X + b^{(l)}), \tag{4.1}$$

where, for a layer l, $a(.)$ is the activation function, $W^{(l)}$ is the weights of the features, and $b^{(l)}$ is the bias. We use the DNN model with the static-based representations, including Strings-, Symbols-, and Hexdumps-based representations.

Training Stage. The dataset is split into 80% training and 20% testing. The Neural Network (NN) classifiers were trained with ten epochs, and a learning rate of 0.01.

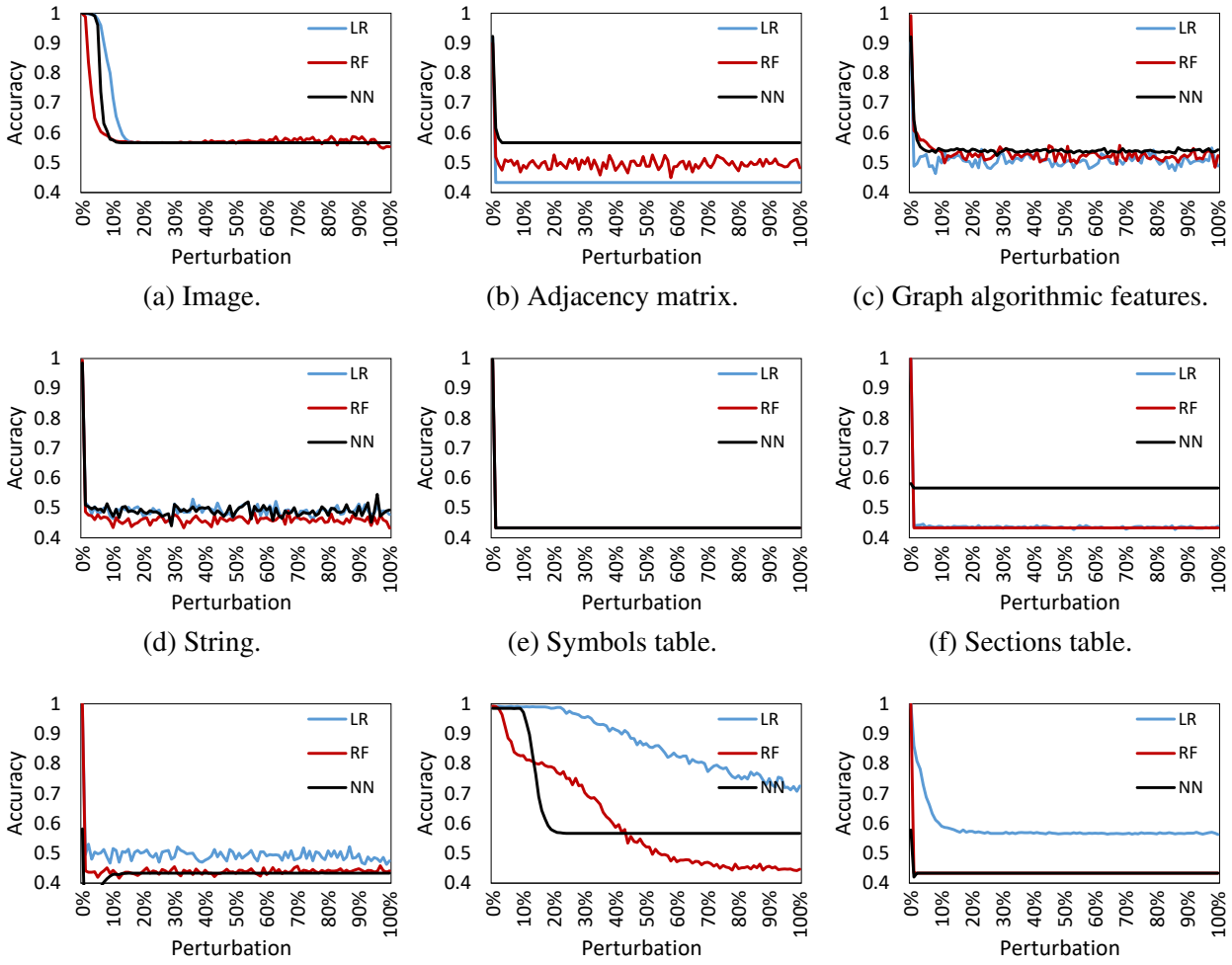
Evaluation & Results. To better understand the robustness of the IoT malware detection systems, we evaluate each of the settings separately.

Baseline Evaluation. We implemented the baseline classifiers on our dataset. Table 4.3 shows the performance of the classifiers. Eight out of the nine representations achieve a high detection accuracy of 99% with at least one learning algorithm. The only exception is the CFG-based adjacency matrix representation, with an accuracy of 92.25%. We recall that high accuracy does not reflect accurate learning, nor the quality of the learned patterns.

Model Stability.

RQ1:
Are the baseline models correctly trained with no over-fitting and under-fitting?

A stable model's performance should ideally decrease linearly with the increase of the perturbation size, to eventually reach random (50% given the two-class classification). Fig. 4.4 shows the evaluation of the baseline classifiers under the Gaussian noise with 1%-100% perturbation. Except for the Hexdump representation, with the introduction of a perturbation size of $1\% \leq \delta \leq 5\%$, the classifiers fail to deliver beyond the random guess. This highlights that the used representations are not stable and may fail due to the temporal changes in the data over time. A likely reason for this is the frequent appearance of different versions of the same or identical malware, thereby forcing



(a) Image. (b) Adjacency matrix. (c) Graph algorithmic features.

(d) String. (e) Symbols table. (f) Sections table.

(g) Segments table. (h) Hexdump-based. (i) Combined Features.

Figure 4.4: Baseline classifiers evaluation under various Gaussian noise perturbation rates (1%-100%).

the model to be over-fitted to look for the exact match instead of extracting feasible patterns.

Key Finding:
Except for Hexdump-based representation, the baseline classifiers demonstrate high instability in their performance under small perturbation (1% Gaussian noise).

White-box Attacks.

RQ2:
Are the classifiers prone to practical white-box adversarial attacks?

Table 4.4: Baseline classifiers evaluation under white-box settings. Only realistic and practical adversarial attacks are considered. All attacks are done on the NN and transferred to the LR- and RF-based classifiers.

Type	Feature	Attack Type	Model	Accuracy (%)
Binary	Image	Transferred	LR	63.73
		Transferred	RF	72.71
		Direct	CNN	63.73
CFG	Adjacency	Transferred	LR	81.77
		Transferred	RF	79.60
		Direct	CNN	81.30
CFG	Algorithmic	Transferred	LR	59.95
		Transferred	RF	60.70
		Transferred	CNN	59.95
CODE	String	Transferred	LR	29.08
		Transferred	RF	30.02
		Direct	DNN	30.59

Evaluating the classifiers against white-box settings is essential to understand their point-of-failure. In this context, we evaluate the white-box attacks that can be implemented directly on the binaries, or on the source code by the malware author. Table 4.4 shows the evaluation of the baseline models under white-box attacks, including binary padding and graph and string manipulation. While the binary padding can be also applied to the remaining representations (as shown later), it is considered as a white-box attack on the image-based representation only, and therefore reported here. We note that all considered white-box attacks are implemented on the NN-based classifier, and transferred to the other learning algorithms. The CFG-based algorithmic representation was evaluated using the perturbation generated on the adjacency-based representation (*i.e.* transferred) due to their feature dependencies.

Key Findings:
For several representations, practical white-box attacks are possible, and can be transferred to related learning algorithms and representations.

Binary Manipulation Attacks. These settings include evaluating the classifiers under manipulation attacks on the software. Here, we consider binary packing under default and optimized

Table 4.5: Baseline classifiers evaluation under binary manipulation (%). Packed*: optimized packing, L.A.: learning algorithm.

Type	Feature	L.A.	Benign					Malware				
			Original	Packed	Packed*	Stripped	Padded	Original	Packed	Packed*	Stripped	Padded
Binary	Image	LR	100	3.92	4.35	6.31	63.73	99.83	98.00	98.00	98.00	98.33
		RF	99.56	2.39	2.17	2.39	72.71	100	96.66	96.66	92.00	85.00
		NN	100	6.31	6.31	2.17	63.73	100	100	100	100	100
CFG	Adjacency	LR	87.36	33.11	33.55	87.36	87.36	95.50	77.33	77.50	95.50	95.50
		RF	88.01	98.91	99.12	88.01	88.01	91.50	73.16	73.16	91.50	91.50
		NN	86.92	1.74	1.74	86.92	86.92	96.33	79.16	79.16	96.33	96.33
CFG	Algorithmic	LR	91.54	1.96	1.96	91.54	91.54	89.04	89.86	89.64	89.04	89.04
		RF	99.51	99.56	99.78	99.51	99.51	98.96	88.76	88.76	98.96	98.96
		NN	93.23	2.17	2.17	93.23	93.23	91.11	91.85	91.62	91.11	91.11
CODE	String	LR	96.51	3.48	3.48	96.51	96.51	100	100	100	100	100
		RF	98.69	2.39	2.39	98.69	98.69	100	100	100	100	100
		NN	96.51	0.00	0.00	96.51	96.51	100	100	100	100	100
CODE	Symbols	LR	97.16	1.08	1.08	97.16	97.16	100	100	100	100	100
		RF	98.69	2.17	2.17	98.69	98.69	100	100	100	100	100
		NN	94.98	3.26	3.26	94.98	94.98	100	100	100	100	100
CODE	Sections	LR	100	100	100	3.48	100	100	34.66	34.66	100	100
		RF	100	3.48	3.48	100	100	100	100	100	100	100
		NN	0.00	0.00	0.00	0.00	0.00	100	100	100	100	100
CODE	Segments	LR	96.51	0.00	0.00	96.51	96.51	99.83	99.83	99.83	99.83	99.83
		RF	100	3.48	3.48	100	100	100	100	100	100	100
		NN	3.48	3.48	3.48	3.48	3.48	100	100	100	100	100
CODE	Hexdumps	LR	98.03	97.60	97.60	98.03	98.03	99.66	86.16	86.16	99.66	99.66
		RF	98.25	1.74	1.74	98.25	98.25	100	92.83	92.83	100	100
		NN	96.51	0.00	0.00	96.51	96.51	100	100	100	100	100
CODE	Combined	LR	100	3.48	3.48	3.48	100	100	100	100	100	100
		RF	99.78	3.26	3.26	99.56	99.78	100	100	100	100	100
		NN	0.00	0.00	0.00	0.00	0.00	100	100	100	100	100

(packing*) conditions, stripping, and padding. Table 4.5 shows the evaluation results under these manipulation attacks strategies. In the following, we interpret these results posed as research questions.

RQ3:
Does binary packing affect the performance of the baseline classifiers?

The evaluation results show that most of the classifiers identify packed software as malicious. This indicates that they identify packing as a malicious pattern. This observation is in line with Aghakhani *et al.* [16], demonstrating that the industry-standard windows malware detection systems identify the packed software as malicious. However, our results bring forward an exception, where Hexdump-based LR classifier maintains its performance under the two levels of packing.

Key Finding:
Baseline classifiers, in general, identify packing as malicious behavior.

RQ4:
Does binary stripping affect the behavior of the baseline classifiers?

Recall that stripping removes information, such as the debug information, from the software binaries. However, the results exhibit that the performance of most of the representations, such as the CFG, strings, symbols, segments, and Hexdump, are intact.

Key Finding:
Generally, existing approaches maintain high accuracy under binary stripping.

RQ5:
Does binary padding affect the behavior of the baseline classifiers?

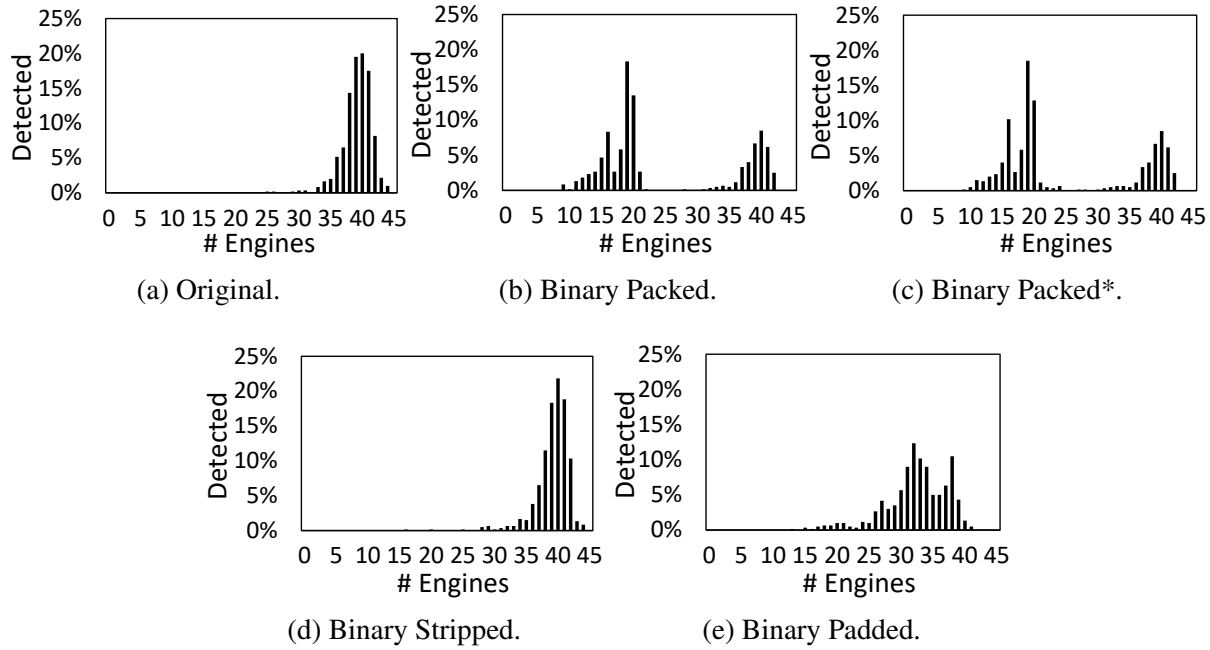
Given that with binary padding we do not remove any existing functional codebase, it does not affect the analyses of the software. Therefore, it only affects the binary/image-based representation.

Key Finding:
Binary padding only reduces the performance of binary/image-based classifiers and can be countered by reverse-engineering the software samples.

RQ6:
Which of the representations and learning algorithms are best suited for malicious IoT software detection?

To answer this question, we considered the following metrics: (1) Baseline accuracy. A detector should have a minimal detection error (*i.e.* false positive and negative rates). (2) Performance consistency. The performance of the classifiers should be robust to various binary manipulation techniques. (3) Model stability. The robustness of the classifier should encompass Gaussian noise, to some extent. Altogether, the classifier that performed best is the Hexdump-based LR classifier, followed by the CFG algorithmic-based RF classifier.

Key Finding:
Hexdump-based LR classifier is the most robust classifier, providing a stable 98.96% baseline accuracy.



(a) Original. (b) Binary Packed. (c) Binary Packed*.

(d) Binary Stripped. (e) Binary Padded.

Figure 4.5: The online engines' detection rate of the original and binary manipulated IoT malware samples.

Industry-Standard Detection Engines Robustness

Malware authors check their software on the online detection engines to ensure that their resultant product evades the scanning engines. Given that these scan engines provide results for a pool of anti-virus engines, evading the detection from these engines is considered as a prototype for malware evolution. These mutations are then used in malware campaigns in the future. We argue that a practical malware detector should detect such mutations, or at least cover for the low-effort based mutations.

Experimental Setup. Online scan engines, such as VirusTotal, are commonly used by researchers to inspect software. VirusTotal reports contain the detection results of a pool of state-of-the-art anti-virus engines that can be considered as the up-to-date capability of industry-standard malware detectors. Overall, it contains reports from 66 IoT malware detection engines. Therefore, to have a comprehensive evaluation of the existing IoT malware detectors, we also evaluate the industry-

standard malware detection systems.

VirusTotal Reporting. The original and manipulated software were uploaded to VirusTotal using their Large File Scan API. To account for the time the AI engines take to properly scan the uploaded files, we wait for 24-hours before gathering the reports. Each of the reports contains details about the uploaded file, including the date, size, header information, and the scan results of each available detection engine. Each report contains results of multiple engines (45-66), each highlighting if it detects the file as malicious or otherwise. Additionally, we found two engines that report for less than ten samples, which we removed from our list. Ultimately, we scan the malicious and benign software through 64 detection engines.

AI-based Engines. The next step is to separate the AI-based engines from other engines. This step is challenging as the detection engines are unlikely to share their detection approaches with the public. We manually inspect each detection engine website, searching for the used approaches. Engines that explicitly mention AI or ML are labeled as AI (✓), while others are labeled as uncertain (✗).

Ethical Considerations. As stated by VirusTotal, the API is not meant to be used to compare between the engines, nor be used to draw conclusions of whether engine X is better than engine Y. Toward this, we take the following considerations: (1) All engines are renamed as "E — i", where i is a given index for the engine. (2) The usage of the API is to assert that state-of-the-art scan engines are vulnerable and behave similar to the research-based detection approaches. We do not intend to compare the engines, nor raise concerns against any specific service provider.

Evaluation & Results. In the following, we interpret the results of the industry-standard malware detectors to understand their behavior, shown in Table 4.6 and presented as research questions. The detailed evaluation of the 64 engines is shown in Table 4.7, and the major insights are illustrated in Fig. 4.6.

RQ7:
Does binary manipulation affect the malware detection rate?

Table 4.6: The online IoT malware detection engines evaluation (%). Packed*: optimized packing.

Engine	AI	Benign					Malware				
		Original	Packed	Packed*	Stripped	Padded	Original	Packed	Packed*	Stripped	Padded
E — 1	✓	100	**86.41**	**89.68**	100	100	100	82.79	82.94	100	100
E — 2	✓	100	100	100	100	100	98.33	33.83	34.67	97.33	**23.5**
E — 3	✓	100	100	100	100	100	99.5	**34.67**	**35.5**	98.5	**37.0**
E — 4	✓	100	100	100	100	100	99.33	94.5	96.33	99.33	95.29
E — 5	✓	100	—	—	100	100	100	100	100	100	100
E — 6	✓	100	100	100	100	100	99.67	99.67	99.67	99.66	99.67
E — 7	✓	100	100	100	100	100	**0.0**	0.0	0.0	0.0	0.0
E — 22	✗	100	100	100	100	100	80.61	29.15	29.34	79.16	**4.04**
E — 23	✗	100	100	100	100	100	99.67	99.67	99.5	99.5	97.33
E — 24	✗	100	100	100	100	100	50.34	29.36	29.88	**85.21**	59.97
E — 25	✗	100	100	100	100	100	84.8	28.42	28.52	81.27	4.65
E — 26	✗	100	100	100	100	100	100	58.29	58.66	98.99	40.37
E — 27	✗	100	**85.84**	**90.07**	100	100	100	82.78	82.8	100	100
E — 28	✗	100	100	100	100	100	99.83	99.83	99.83	99.66	95.41
E — 29	✗	100	100	100	100	100	**0.0**	0.0	0.0	0.0	0.0

To answer this question, we recorded the number of engines that identify malware as malicious. We begin by probing the original malware samples: Fig. 4.5a shows the distribution of their detection rate by the engines. Notice that malware, on average, is detected by 40 engines, with a majority of them being detected by 35-45 engines. For the manipulated samples, however, the detection rate varies highly. Fig. 4.5 shows the distribution of malicious samples by the number of engines for each of the manipulation strategies. We notice that stripping (Fig. 4.5d) does not affect the distribution of the samples. However, packing (Fig. 4.5b and Fig. 4.5c) highly affects the detection rate. Moreover, while binary padding had minimal effects on the baseline classifiers' performance, it highly affects their detection among the online engines. This indicates that several engines use binary-based representations (*e.g.* binary sequence and image) to detect malicious software.

Key Finding:
Except for binary stripping, binary manipulation highly decreases the detection confidence.

RQ8:
How individual engines generally perform?

To answer this question, we evaluate each individual detection engine using the original and manipulated benign and malicious software, shown in Table 4.6 and Table 4.7. We observe that multiple

Table 4.7: The evaluation results (%) of the online IoT malware detection engines.

Engine	AI	Benign					Malware				
		Original	Packed	Packed*	Stripped	Padded	Original	Packed	Packed*	Stripped	Padded
E — 1	✓	100	86.41	89.68	100	100	100	82.79	82.94	100	100
E — 2	✓	100	100	100	100	100	98.33	33.83	34.67	97.33	23.5
E — 3	✓	100	100	100	100	100	99.5	34.67	35.5	98.5	37.0
E — 4	✓	100	100	100	100	100	99.33	94.5	96.33	99.33	95.29
E — 5	✓	100	—	—	100	100	100	100	100	100	100
E — 6	✓	100	100	100	100	100	99.67	99.67	99.67	99.66	99.67
E — 7	✓	100	100	100	100	100	0.0	0.0	0.0	0.0	0.0
E — 8	✓	100	100	100	100	100	53.17	24.0	24.0	53.17	51.83
E — 9	✓	100	100	100	100	100	87.0	86.5	86.83	86.81	95.33
E — 10	✓	100	100	100	100	100	91.33	31.83	31.83	91.33	91.33
E — 11	✓	100	100	100	100	100	99.67	47.58	47.58	99.67	97.17
E — 12	✓	100	100	100	100	100	97.83	33.5	33.67	97.33	97.33
E — 13	✓	100	100	100	100	100	0.0	0.0	0.0	0.0	0.0
E — 14	✓	100	100	100	100	100	0.0	0.0	0.0	0.0	0.0
E — 15	✓	100	100	100	100	100	32.06	12.36	11.74	31.69	30.57
E — 16	✓	100	100	100	100	100	100	34.67	34.67	100	100
E — 17	✓	100	100	100	100	100	82.47	27.67	27.67	82.15	81.47
E — 18	✓	100	100	100	100	100	99.45	96.69	96.52	99.27	95.0
E — 19	✓	100	100	100	100	100	19.69	0.51	0.51	19.49	19.39
E — 20	✓	100	100	100	100	100	0.0	0.0	0.0	0.0	0.0
E — 21	✓	100	100	—	100	100	—	0.0	0.0	0.0	0.0
E — 22	✗	100	100	100	100	100	80.61	29.15	29.34	79.16	4.04
E — 23	✗	100	100	100	100	100	99.67	99.67	99.5	99.5	97.33
E — 24	✗	100	100	100	100	100	50.34	29.36	29.88	85.21	59.97
E — 25	✗	100	100	100	100	100	84.8	28.42	28.52	81.27	4.65
E — 26	✗	100	100	100	100	100	100	58.29	58.66	98.99	40.37
E — 27	✗	100	85.84	90.07	100	100	100	82.78	82.8	100	100
E — 28	✗	100	100	100	100	100	99.83	99.83	99.83	99.66	95.41
E — 29	✗	100	100	100	100	100	0.0	0.0	0.0	0.0	0.0
E — 30	✗	100	100	100	100	100	0.33	0.0	0.0	0.33	0.0
E — 31	✗	100	100	100	100	100	0.0	0.0	0.0	0.0	0.0
E — 32	✗	100	100	100	100	100	100	90.82	92.67	99.67	99.67
E — 33	✗	100	100	100	100	100	96.82	33.9	35.9	98.3	36.81
E — 34	✗	100	100	100	100	100	99.5	34.67	35.5	98.5	37.0
E — 35	✗	100	100	100	100	100	99.83	99.83	99.83	99.5	96.31
E — 36	✗	100	100	100	100	100	99.33	34.34	36.06	98.99	75.79
E — 37	✗	100	100	100	100	100	0.0	0.0	0.0	0.0	0.0
E — 38	✗	100	100	100	100	100	0.0	0.0	0.0	0.0	0.0
E — 39	✗	100	100	100	100	100	99.83	34.72	36.5	99.5	75.17
E — 40	✗	100	100	100	100	100	99.83	85.98	85.83	99.0	95.0
E — 41	✗	100	100	100	100	100	1.34	0.5	0.5	1.34	0.0
E — 42	✗	100	100	100	100	100	0.0	0.0	0.0	0.0	0.0
E — 43	✗	100	100	100	100	100	0.0	0.0	0.0	0.0	0.0
E — 44	✗	100	100	100	100	100	99.0	34.33	35.17	98.83	98.5
E — 45	✗	100	100	100	100	100	0.0	0.0	0.0	0.0	0.0
E — 46	✗	100	100	100	100	100	99.5	34.5	34.5	99.17	97.83
E — 47	✗	100	100	100	100	100	99.67	85.67	85.33	97.0	94.83
E — 48	✗	100	100	100	100	100	0.0	0.0	0.0	0.0	0.0
E — 49	✗	100	100	100	100	100	99.33	99.5	99.83	99.5	95.33
E — 50	✗	100	100	100	100	100	99.64	88.27	89.54	99.47	95.07
E — 51	✗	100	100	100	100	100	98.17	39.0	39.0	94.17	90.67
E — 52	✗	100	100	100	100	100	100	75.3	75.09	100	97.64
E — 53	✗	100	100	100	100	100	99.83	99.83	99.83	99.33	95.17
E — 54	✗	100	100	100	100	100	0.0	0.0	0.0	0.0	0.0
E — 55	✗	100	100	100	100	100	97.98	33.28	33.56	96.96	0.51
E — 56	✗	100	100	100	100	100	0.0	0.0	0.0	0.0	0.0
E — 57	✗	100	100	100	100	100	100	34.45	34.51	98.83	97.82
E — 58	✗	100	100	100	100	100	2.5	1.17	1.17	2.33	0.0
E — 59	✗	100	100	100	100	100	97.65	96.66	96.64	96.46	96.3
E — 60	✗	100	100	100	100	100	78.86	26.63	26.63	78.96	74.92
E — 61	✗	100	100	100	100	100	0.0	0.0	0.0	0.0	0.0
E — 62	✗	100	100	100	100	100	99.17	93.33	95.33	99.33	95.33
E — 63	✗	100	100	100	100	100	99.67	99.67	99.67	99.67	99.67
E — 64	✗	100	100	100	100	100	0.0	0.0	0.0	0.0	0.0

engines perform poorly, with 36% of the engines (23 out of 64) failing in identifying malware ($\approx$ 0% accuracy), such as "E — 7" and "E — 29". Additionally, except for "E — 1" and "E — 27", the benign detection accuracy is 100%, similar trends were observed for packed, stripped, and padded benign software.

Key Finding:
Several engines (36%) exhibit reduced performance for detecting original and binary manipulated malicious software.

RQ9:
Does packing affect the engines' performance?.

The evaluations exhibit that packing does not affect the performance of the engines in accurately detecting benign software (except for "E — 1" and "E — 27"). This observation is in contrast to previous observations [16]. However, packing, generally, reduces the accuracy of malware being detected as malware. For instance, "E — 3" performance declined from 99.5% to $\approx$ 35% when tested with packed malware. We also observed that optimized packing does not decrease the detection rate, in fact, it slightly increases the chance of malicious software being detected, as compared to the standard packing. Additionally, for engines, such as "E — 5", we observe that no results were reported for benign packed binaries, while achieving 100% in other categories. This can be attributed to the low confidence of the engine in labeling benign packed samples.

Key Finding:
Although packing reduces the detection rate of malicious software, it has no effect on the benign software detection rate. Optimized packing has a higher detection rate in comparison with default packing.

RQ10:
Does binary stripping affect the online engines' performance?

There is no noticeable decrease ($<$1%) in the detection accuracy of stripped software in the case of online engines. In fact, for some engines (*i.e.* "E — 24"), the malware detection performance increased from 50.34% to 85.21% after stripping.

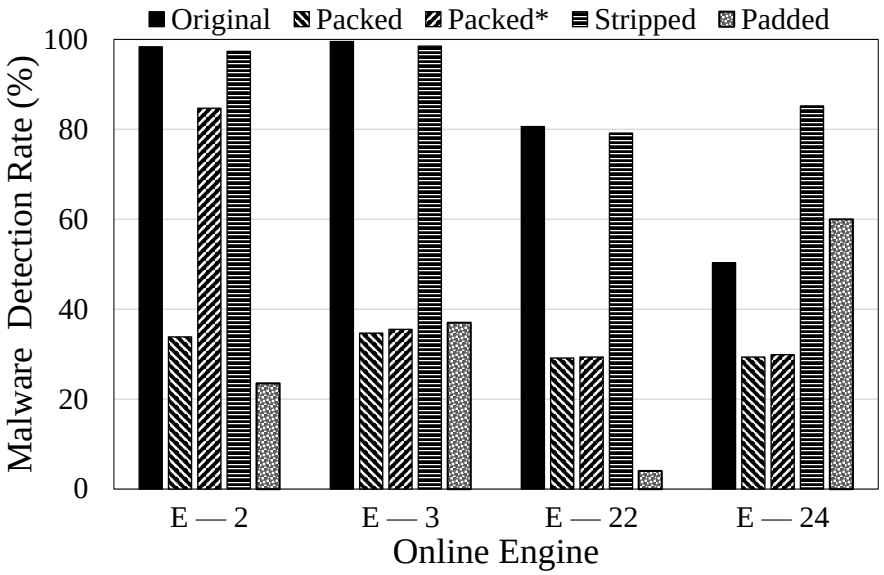


Figure 4.6: Industry-standard detection engines robustness highlight. Binary packing significantly reduces the detection rate of Malware software ("E — 2"). Binary stripping does not result in noticeable performance degradation, and may increase the malware detection rate ("E — 22"). Simple binary padding to the end of the file may cause significant degradation in the performance ("E — 3" and "E — 22").

Key Finding:
Stripping has no negative effect on the performance of the engines, albeit increasing the accuracy in some instances.

RQ11:
Does binary padding affect the online engines' performance?

Binary padding significantly decreases the performance of several online engines, such as "E — 2", "E — 3", and "E — 22". This is maybe attributed to the fact that appending binaries disrupt the existing signatures. The online engines' reports show that $> 53\%$ of them are affected negatively, with $> 14\%$ of them exhibiting a drastic decrease in performance ($> 70\%$ decrease). Although padding does not affect the reverse-engineered features, the decrease in performance, regardless, indicates that the engines use the raw binary representations (*e.g.* binary sequence- and image-based) for classification, which apparently can be easily disrupted.

Key Finding:
Binary padding highly reduces the performance of several engines, while leaving others intact.

Threat Surface Reduction

In this work, we highlight the vulnerability of machine learning-based malware detection frameworks to adversarial examples. This vulnerability is contributed to two components: (i) inconsistencies within the learning algorithms, and (ii) volatile feature space. In this section, we identify potential volatile feature space, and propose pre-processing mechanisms to counter exploiting such features. This process enable trustful malware detection, limiting the adversarial capabilities.

Restricted Learning Algorithm. Benign software are diverse, and not bounded by existing patterns. On the other hand, malware families are bounded by shared behavioral functionalities. A common malpractice is to train the machine/deep learning model on the benign patterns, causing over-fitting and potential pattern injection exploitation. Toward robust learning process, we investigate limiting the learning process to only learn the malicious patterns (*i.e.* monotonic learning), and not over-fit on the benign patterns. In this process, the learning algorithms decision making is associated with the existence of information, rather than absence of information.

Remove Volatile Features. On the feature space-level, the usage of volatile, easy to modify, features is a major concern toward exploiting the malware detection process. For instance, binary padding is only effective due to malware detectors considering the binaries that exist beyond the file execution boundaries within the feature space. Altering or removing arbitrary information, such as the file name and padded binaries, should not cause misclassification.

Adversarial Capabilities. Before diving into the reduction of the attack surface in context of binary-level malware detection, we need to understand the adversarial capabilities under such settings. For instance, the adversary may apply perturbation directly to the binary sequences of the malicious file. Such perturbation can be either (1) modifying the values of existing bytes, or (2)

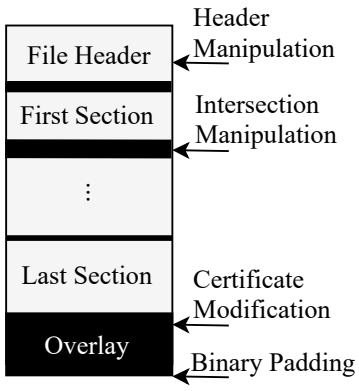


Figure 4.7: The generic file format. Different attacks utilizes different attack channels to cause misclassification

add new byte sequences to the file. This perturbation should be carefully applied to ensure that the (i) functionality, (ii) executability, and (iii) malicious behavior of the software are intact. In the following, we investigate the binary-level malware mutations that cause effective malware misclassification, shown in Fig. 4.7

Header Information Manipulation [49]. In this attack, the adversary modifies the arbitrary values within the file header to cause misclassification. Information includes, but is not limited to, size of image, program signature, characteristics flags, time-date stamp, signature, and the number of sections. Such information are easy to modify, as they do not contribute to the functionality of the program.

Insights on Effectiveness: This attack is effective when the malware detection frameworks utilize easy-to-modify header information for malware detection [27]. While such information may momentarily increase the malware detection performance, as malware within the same category and time period may have shared header information, it might be exploited to reduce the detection performance or confidence.

Binary Padding [85]. In this attack, the adversary appends binaries at the end of the program binaries. This process is functionality-preserving, and widely used for misclassifying binary-based

classifiers. The padded binaries can be generated using different approaches, including benign injection, random injection, or gradient padding.

Insights on Effectiveness: While padded bytes are not mapped to any functionality, nor scanned by the operating system on the run-time, this approach take advantage of the on-the-fly fast raw binary-based malware detection approaches, including bytes sequences [115], software visualization [128, 96], and bytes n-grams and histograms [152, 33]. Such techniques, while being light-weighted and utilized at the end-point devices, can be exploited to cause misclassification using binary padding, as shown earlier in this dissertation.

Intersection Injection [49, 129]. In contrast to the binary padding, this approach does not increase the size of the program, but rather manipulates the "unused" (*i.e.* unmapped) bytes of the program to cause misclassification. This, however, limit the space of perturbation, and may result in a reduced attack surface. Unused bytes are defined as bytes resulting from the memory page allocation process, where bytes are padded between sections due to the difference between the virtual size and allocated memory size.

Insights on Effectiveness: Similar to binary padding, this attack exploits the lack of software structural analysis in raw binary-based detection models. However, it is harder to detect, as it does not increase the size of the file, in contrast to the binary padding.

Omitting Volatile Information. In this section, we discuss simple yet effective software preprocessing techniques to omit volatile features that do not contribute to the malicious functionality, and, in turn, significantly mitigate the effect of binary-level mutations.

Software Stripping. The process of removing information from executable binaries that is not essential or required for normal and correct execution is known as software stripping. While such information may increase the performance of the detection framework, it is considered a fertile ground for adversary exploitation, particularly the header and debugging information.

Excessive Length Removal. We refer to the process of removing the padded binaries occur after

Table 4.8: IoT malware detection performance evaluation using gradient boosting model with traditional and monotonic patterns learning under different software cleaning and processing techniques. Notice that three feature representations are rendered unusable after software processing, indicating that the extracted patterns were associated with volatile features (*i.e.* non-robust). Unpadded binaries include the intersection byte resetting. S&U: Both binary stripping and unpadding were applied.

Feature Type	Processing	Traditional			Monotonic Learning		
		F-1 score	AUC-ROC	Malware Detection (0.1% FPR)	F-1 score	AUC-ROC	Malware Detection (0.1% FPR)
Image	Original	99.87	99.99	99.73	99.8	99.99	99.47
	Stripped	99.64	99.99	98.26	99.64	99.54	91.19
	Unpadded	99.64	99.99	98.33	99.28	99.93	83.5
	S&U	99.59	99.99	99.96	96.07	99.66	86.34
Hexdump	Original	99.80	99.99	99.91	93.35	98.8	0.0
	Stripped	99.94	99.99	99.96	99.09	99.94	94.24
	Unpadded	99.84	99.99	100.0	97.35	99.42	58.89
	S&U	99.89	99.99	99.92	99.09	99.94	95.46
Bytes N-gram	Original	100	100	100	100.0	100.0	100.0
	Stripped	99.89	99.99	99.85	99.94	99.99	99.96
	Unpadded	100.0	100.0	100.0	100.0	100.0	100.0
	S&U	99.94	99.99	99.96	99.94	99.99	99.85
Strings	Original	96.56	99.06	94.09	94.23	97.68	85.63
	Stripped	96.97	99.55	96.45	94.6	98.71	95.11
	Unpadded	44.98	66.71	33.43	44.98	66.3	32.61
	S&U	44.98	66.71	33.43	44.98	66.3	32.61
Sections	Original	99.93	100.0	100.0	100.0	100.0	100.0
	Stripped	100.0	100.0	100.0	99.44	99.94	99.67
	Unpadded	100.0	100.0	100.0	84.96	93.35	86.71
	S&U	100.0	100.0	100.0	44.98	81.25	62.5
Imports	Original	98.46	98.36	5.81	97.15	97.73	5.72
	Stripped	53.38	55.95	3.29	51.88	55.18	3.33
	Unpadded	44.98	51.42	2.84	44.98	51.42	2.84
	S&U	44.98	51.42	2.84	44.98	51.42	2.84
Relocations	Original	95.09	94.34	5.19	91.33	91.5	5.11
	Stripped	45.07	51.43	2.86	45.07	51.43	2.86
	Unpadded	44.98	50.95	1.91	44.98	50.95	1.91
	S&U	44.98	51.05	2.1	44.98	51.05	2.1
Combined	Original	100.0	100.0	100.0	99.93	100.0	100.0
	Stripped	100.0	100.0	100.0	100.0	100.0	100.0
	Unpadded	100.0	100.0	100.0	100.0	100.0	100.0
	S&U	100.0	100.0	100.0	100.0	100.0	100.0

the end of the file, and do not appear in the header or sections information (*i.e.* not mapped to the software) as binary unpadding. Obtaining the information regarding the sections boundaries and virtual sizes can be effectively utilized to omit the padded binaries from the end of the software.

Bytes Resetting. In the software, there exists an area between the mapped sections, mainly caused by the memory paging system and the difference between virtual size and raw size of the section.

This area is typically exploited by modifying the byte sequences to generate code caves and different malware mutations. Resetting these bytes to a pre-defined op-code, such as $0x00$, removes the possibility of conducting such adversarial attacks. In this work, we include this pre-processing technique within the binary unpadding process.

Experimental Evaluation. Table 4.8 shows the experimental evaluation of IoT malware detection performance, reported in F-1 score, AUC-ROC score, and the malware detection rate at 0.1% false positive rate (*i.e.* 99.9% benign detection rate) for both traditional learning and monotonic learning. Note that traditional learning allows for learning benign patterns, whereas monotonic learning limit the learnt patterns to be within malware samples. Our results unveil that three representations rendered useless under binary stripping and unpadding, indicating that the extracted features from these representations are volatile, and can be exploited. While omitting such features will reduce the attack surface, it will also render the model unusable in case of software strings representation.

While traditional learning, accompanied with software processing, ensures robustness against binary padding and header manipulation attacks, the model is still vulnerable to pattern injection attacks. This is when monotonic learning becomes handful, as allowing only malicious patterns meaning that injecting patterns will only increase the maliciousness of the software, instead of being classified as benign. Overall, several representations achieve a consistent performance across different settings, with minimal performance to robustness trade-off cost.

Online Engines: Volatile Features. To better understand whether online engines utilize volatile information for malware detection, we uploaded the malware samples to VirusTotal API after appending one byte (0xFF) to the end of each file to prevent signature recognition and blacklisting. We noticed a significant decrease of the online detection engines performance on the unpadded binaries, with a 28% decrease, and stripped and unpadded binaries, with 30% decrease. However, we only observe a detection performance decrease of only 3% after binary stripping. This indicates that most engines does not utilize header information within their models, but utilize the overlay information, including certificate and software signature information, to detect malicious software.

We recall that such features, while initially increase the detection performance, can be exploited toward misclassification.

Summary & Concluding Remarks

Malware analysis and detection have been the focus of the research community and the industry alike, with many advances in defenses with the use of AI-backed systems. Despite those advances, these systems have been shown to be vulnerable to several simple-yet-effective adversarial attacks, such as binary stripping and packing. With this work, we systematically evaluate the state of a range of malware detectors, proposed by the research community and industry-standard.

Our efforts show that malware detectors proposed in the literature are vulnerable to adversarial perturbation and binary manipulation attacks. Similarly, industry-standard malware detectors are prone to such attacks. Our efforts also unveil the status-quo of the existing detectors, and bring forward various insights to consider when proposing detection systems. Particularly, in addition to optimizing baseline malware detection accuracy, researchers should take into account the robustness of the proposed systems under adversarial capabilities. This obligates for a deep understanding of the underlying learning algorithms and data representations, alongside the learned patterns and their characteristics.

CHAPTER 5: EXPOSING THE LIMITATIONS OF MODEL RETRAINING IN MACHINE LEARNING MALWARE DETECTION

Malware detection systems have conveniently employed signature-based and heuristics-based approaches that are largely effective across malware families with similar characteristics. The performance of both the signature- and heuristics-based approaches, however, decreases significantly when they encounter new and previously unseen malware families. For a broad spectrum of malware detection across multiple families, machine learning techniques are now widely adopted, and they are capable of detecting a wide range of malware properties as they evolve over time [26, 116, 120, 90, 25].

Although machine learning-based techniques have been shown to be very promising in detecting malware and generalizing to new unseen patterns of malware families, these techniques' performance consistency is constrained by the notion of *"concept drift"*. The concept drift is caused by a rapid change in the malware features being examined by the detection technique [79]. The effect of concept drift on malware evolution is very clearly demonstrated in the recent reports by *VirusTotal* [10], where it has been shown unequivocally that the numbers of malware samples and families seen in the wild are increasing exponentially, at the rate of 1.5 million samples per day. The fundamental idea behind concept drift, which is exploited by adversaries to evade detection, is that it is possible to invoke a rapid malware mutation, making the malware a moving target for the machine learning-based detection system. These new developments have initiated an arms race between malware families and machine learning-based detection systems, with defenses appearing to be mostly ineffective in light of recent reports [10].

To combat this critical issue in malware classification, Zhang *et al.* [155] proposed time- and space-robust features to mitigate the effects of concept drift. While these features work initially, they do not solve the concept drift issue fundamentally, but rather slow down their immediate effects and eventually result in a reduction of the performance of the classifier over time. Commonly used

approaches to combat the concept drift problem are model retraining and active learning. Model retraining includes retraining the machine learning model [82, 134] by incorporating new malware samples in the training process. Active learning, on the other hand, is considered as a light-weighted solution for reducing the computational cost of model retraining [145]. In particular, active learning reduces the retraining dataset by only retraining on carefully selected new malware samples (*i.e.* samples with low detection confidence), to optimize the computational overhead for a performance gain trade-off.

While several studies have incorporated active learning to reduce the computational overhead in malware detection [154, 143, 113], in this work, we focus on the limitations and drawbacks of model retraining, as (i) model retraining provides, in most cases, the best achievable model's performance [64, 130, 95], and (ii) except for the high computational overhead of model retraining, our findings can be generalized for active learning as well.

Through model retraining, the learning algorithm uncovers and incorporates the behavior of the new malware samples, thereby improving the detection accuracy. However, model retraining suffers from two major shortcomings depending on when retraining is done.

On the one hand, the frequent model retraining has sometimes a diminishing return: the model accuracy may not improve despite surmounting the computation overhead. This case typically occurs when the new samples incorporated into the retraining process are not significantly different from the original training set (*i.e.* samples with minimal malware mutations). As a result, a minor shift in malware characteristics over a short period of time (*e.g.* a week) does not necessarily result in the required misclassification. As such, with the pattern shift being below a misclassification threshold, the model retraining only incurs a computational penalty in the detection process while yielding no performance improvements.

On the other hand, an optimistic adjustment to the retraining process by retraining at a lower frequency to capture significant pattern changes, which are a required condition for misclassification, may lead to a degraded performance. Retraining infrequently is problematic since malware evo-

lution is unpredictable and does not follow any fixed pattern. Therefore, not retraining for a long period of time during malware mutation means that the patterns encountered by the model may no longer be recognized as malicious.

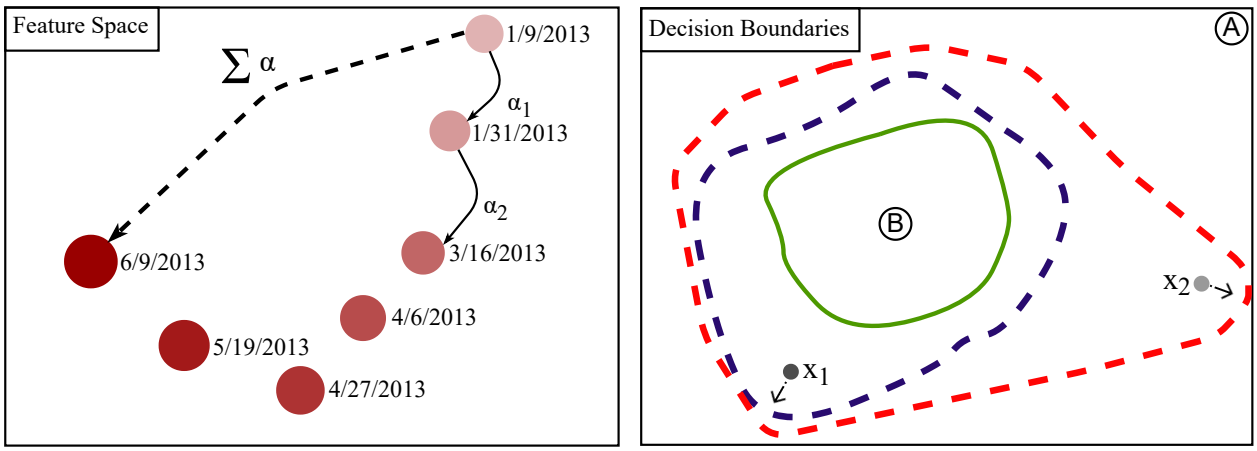
The limitations of model retraining is not limited to finding the optimal retraining timeline. Our preliminary analysis reveals that model retraining also suffers from another major caveat: the continuous model retraining over an extended period of time will eventually cause a reduction in model's performance in detecting benign samples. This caveat uncovers a hidden benign-to-malware mutations detection trade-off, which significantly limits the capabilities of the detection frameworks in the ongoing arms race.

Summary of Completed Work

1. *Systematic Exposition of Model Retraining.* First, we conduct a systematic analysis of malware detection and retraining to expose their limitations. Our data-driven analysis on Windows malware detection reveals an under-reported trade-off between the misclassification of malicious and benign samples, due to the retraining approach. Our evaluation also reveals that while retraining leads to a temporary increase in the detection of malicious samples, it degrades the detection of benign samples by $\approx$15%.

2. *Refining Model Retraining.* Guided by the findings in our preliminary analysis, we refine the model retraining process for accurate detection by leveraging confidence- and out-of-distribution-based techniques. We use both techniques to detect unseen mutations in malware samples, and further leverage our detection model for a timely invocation of the retraining process. Through a timely retraining, we achieve a high malware detection rate (**$>$97%**). A byproduct of our refined retraining is (i) tracking the temporal evolution of malware families through the captured mutations, and (ii) enabling effective malware family emergence detection.

3. *Discovering Malware Respreading.* We uncover that the retraining process shortcomings



(a) Malware Mutation

(b) Decision Boundary Change

Figure 5.1: Fig. 5.1a is the t-SNE visualization of zbot mutated malicious software in the period January 2013 - June 2013. Fig. 5.1b is the illustration of the decision boundary changing of two classes A (undetermined) and B (determined).

provide new attack opportunities. For instance, retraining optimizations to improve the performance for benign samples by discarding old malware samples leads to reviving and respreading old malware samples, thereby enabling malware authors to defeat the detection systems. We find evidence that older malware reappear as new malware samples in the future, suggesting that malware authors are reusing old malware samples to escape detection. These findings raise concerns on the efficacy of malware detection engines and their used strategies, and the possibility of using malware respreading to bypass the state-of-the-art detection approaches at a minimal cost. To the best of our knowledge, this is the first study that exposes the problem of malware revival and respreading by exploiting the shortcomings of the existing malware detection systems.

Problem Statement

In this section, we discuss the shortcomings of using machine learning techniques for malware detection. We start by describing the *modus operandi* of malware mutation, followed by a review

of its implications on machine learning models.

Malware Mutation. Malware mutation is the process in which malware properties change at the feature level due to a change in the actual malware code, resulting in *concept drift* in the machine learning model used for the malware detection. For instance, Fig. 5.1a shows the t-SNE visualization of the zbot mutated malware family. In this figure, each mutation results in a shift in the feature space (α_i), which then results in the distortion of the existing patterns learned by the baseline machine learning model. As a result of several mutations over an extended period of time, the accumulated shift (α) results in a high distortion of the learned patterns of malware, thereby necessitating a retraining step to learn the resulting mutations.

Limitations of Model Retraining. While model retraining may work temporarily, we argue that it is not effective over several mutations. For instance, assume a detection model D that classifies an input x into two classes $\{A, B\}$ ($D(x) \in \{A, B\}$). Also assume that the class A is diverse with no existing patterns within its samples (*i.e.* benign), while class B is bounded by the existing patterns (*i.e.* malware). If certain patterns exist within x, D classifies x in class B, and in class A otherwise. In Fig. 5.1b, we provide an illustration of this process by showing a visualization of the decision boundaries of both class A, defined as F_A, and class B, defined as F_B. With the ongoing mutations within samples of the class B, inputs such as x_1 will be observed outside of the original decision boundary. In order to accurately detect x_1, the machine learning model pushes the boundaries of class B towards x_1, which is represented in the blue color. Moreover, assume an input x_2 from class B that has no shared patterns with the existing samples from the same class. As such, the boundaries of class B will be pushed even further from its center to accurately detect x_2, once the model is retrained on x_2—this case is represented in the red color.

While this process ensures high performance in terms of accurately detecting samples in class B, the same process will deteriorate the decision boundary of class A over time, since the decision boundary of class A (F_A) is defined as $F_{(A \cup B)} - F_B$. Overall, this retraining process will result in a reduced performance over class A, where multiple samples will be falsely classified as class B.

Considering this issue, we specify the possible actions that could be taken by the detection model to mitigate the performance loss as follows:

1. **A1: No action.** With no action, the model is not retrained on the new mutations of class B samples, eventually degrading the detection performance of new mutations.

2. **A2: Continuous model retraining.** The model is continuously retrained on mutations of class B samples, resulting in degraded performance for existing and new class A samples.

3. **A3: Retraining on new samples only.** The model is retrained on the mutations of class B samples, but the old B samples are discarded and considered as "obsolete." This process results in degraded detection performance for class B's old samples, where the old samples are classified as class A.

While the commonly-held belief is that *model retraining is a highly useful solution to detect mutated malware samples*, the evaluation of the aforementioned actions (**A1–A3**) clearly demonstrates that model retraining has obvious limitations that have not been concretely considered nor evaluated in the past. To empirically expose those shortcomings, this work investigates the retraining performance of windows malware detection models in detecting old and new unseen malware samples. We use the aforementioned actions as our baseline evaluation criteria to understand the implications of each action on the detection performance of the baseline detection model. Additionally, inspired by the insight obtained in examining **A3**, we investigate possible vulnerabilities in the existing research and industrial windows malware detection systems due to "phasing out" old malware samples from the model training.

Data Representation & Learning

Dataset Overview. A significant part of this work is an empirical examination of various retraining strategies and associated impact. As such, we obtained 56,115 malicious Windows software

Table 5.1: The distribution of the collected Windows binaries. After data filtration, 20,042 unique malicious binaries were considered for our evaluation.

Family	Total Samples	Unique samples
zbot	1,552	1,511
playtech	748	589
bladabindi	1,105	401
gamarue	744	244
webdialer	228	228
fareit	1,389	206
razy	397	175
ursu	238	163
darkkomet	159	129
emotet	10,788	108
Others	30,194	9,638
Singleton	8,573	6,650
Benign		8,118
Overall	64,233	28,160

binaries from VirusShare [4]. The dataset includes malicious samples with first seen dates from 2008 to 2020. Moreover, we collected 8,118 benign binaries from default configurations of operating system instances of Windows XP, 7, 8, 8.1, and 10.[1]

Ground Truth Class. Next, we used *VirusTotal* [8] to examine the malicious and benign samples in our dataset. The samples were first uploaded to *VirusTotal*. After 24 hours, the scan results corresponding to each sample were retrieved. Among the malicious samples, we found 1,421 unique malware families for Windows OS, alongside 8,573 singleton malware samples. Additionally, we obtained the *first seen* date of the malicious samples, which we consider as the ground-truth appearance date in this work. All benign samples were not detected by *VirusTotal*, supporting our default assumption.

Data Representation & Augmentation. After collecting the samples and obtaining the ground truth data, we transform each sample into seven different representations to train and evaluate

[1]Because of the default configuration of each operating system only contains software provisioned by Microsoft, we assume that all those software binaries are benign, by default, which is confirmed through our sanity check.

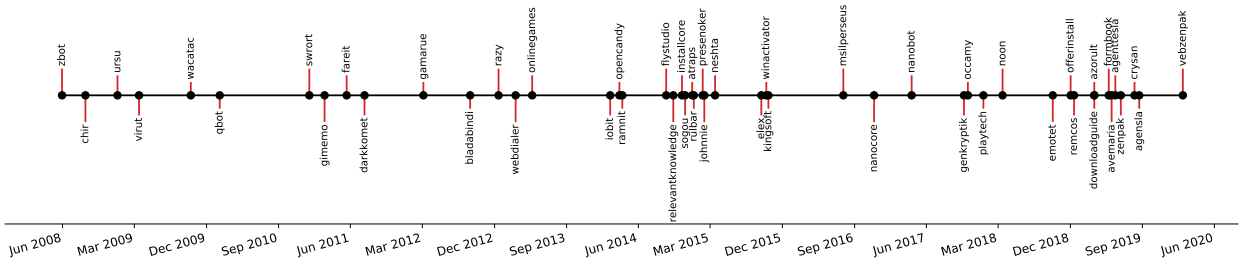


Figure 5.2: The first appearance date of the top-50 Windows malicious families in the collected dataset.

the machine learning models. For sample transformation, we used a reverse-engineering open-source framework called *Radare2* [7]. The selected representations selected because they are: (1) extensively used in the prior works, (2) can be generated efficiently from binaries, and (3) can be employed for building malware detectors relatively easily. In this work, we utilized *Visualization*, *Hexdump*, *Function Calls* and *Entropies*, *Program Sections Information*, *Relocations*, and *Strings* software representations. We note that the extracted representations include different variations of Ember representation [27], as our main goal is to investigate the temporal robustness of different software representation-based models. In the following, we provide a brief description of the utilized data representations.

Data Representation. In this work, we collected Windows malware that was captured in the period of 2008–2020. To better understand the family distribution of the collected malware, Fig. 5.2 shows the appearance date of the top-50 malware families. The top families are reported by the number of captured unique samples. For software representation, we utilized seven different static software representations for malware detection. The extracted representations are fast, utilized extensively in the literature for the malware detection task, and can be extracted on-the-fly. In Table 5.2, we show the utilized data representations, described as follows:

- **Grayscale Image.** The grayscale image representation is a common technique for transforming malware samples into a 2D matrix of values, and each of those values is between 0 and 255. Particularly, the byte-code is visualized as a grayscale image of a fixed size of

$(h \times w)$, where every byte is a pixel in the image.

- **Hexdump.** The hexdump represents the malware as a sequence of hexadecimal values. Each value represents a single byte (in the 0-255 range), the frequency of which is then recorded as a vector of size 1×256.

- **Function Entropy.** The entropy calculates the level of randomness for a given code; *i.e.* the randomness of the opcode sequences in different functions of the software.

- **Function Calls.** A function name describes the intent of the procedure it represents. Therefore, we utilize function calls in the disassembly of a program as a behavioral representation of the malware.

- **Program Sections.** Windows binaries are divided into sections to conveniently load the file in memory during execution by providing a logical and physical separation between different program parts.

- **Program Relocations.** In the object file, the linker keeps relocation records for symbols and codes such as functions. This is done to ensure that records for symbols and codes are properly referenced from an executable program [106]. The relocation records can then be used to represent the software.

- **Program Strings.** Program Strings are a sequence of printable characters in the binary codebase, where their analysis reveals the behavioral patterns [46].

Malware Filtering. Upon transforming the dataset using *Radare2* and representing samples using the above features, we conducted malware filtering to remove duplicates. For instance, two samples may have the same feature representation but using different command and control servers resulting in different hashes for two identical samples. We considered malware of the same family that appeared on the same day, with identical feature representations (except for Hexdump, since it is sensitive to minor modifications), as duplicates.

Table 5.2: The state-of-the-art representations used in this work. The image-based representation uses the raw binaries, while other representations require reverse engineering. Bin.: binary-based, R.E.: reverse-engineered.

Representation	Work	Bin.	R.E.
Binary – Image	[80, 137, 148, 97]	✓	✗
Static – Hexdumps	[17]	✓	✓
Static – Entropy	[17, 29]	✗	✓
Static – Functions	[17, 29]	✗	✓
Static – Sections	[17, 29]	✗	✓
Static – Relocations	[29]	✗	✓
Static – String	[17, 29]	✗	✓

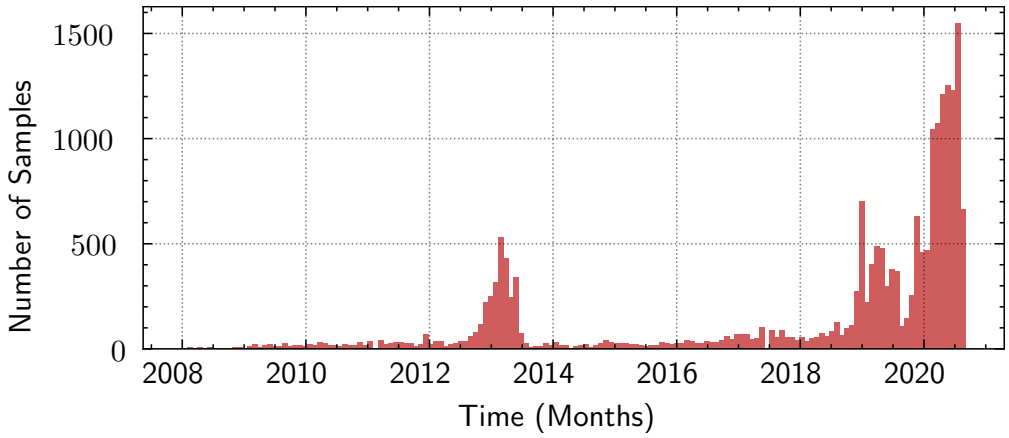
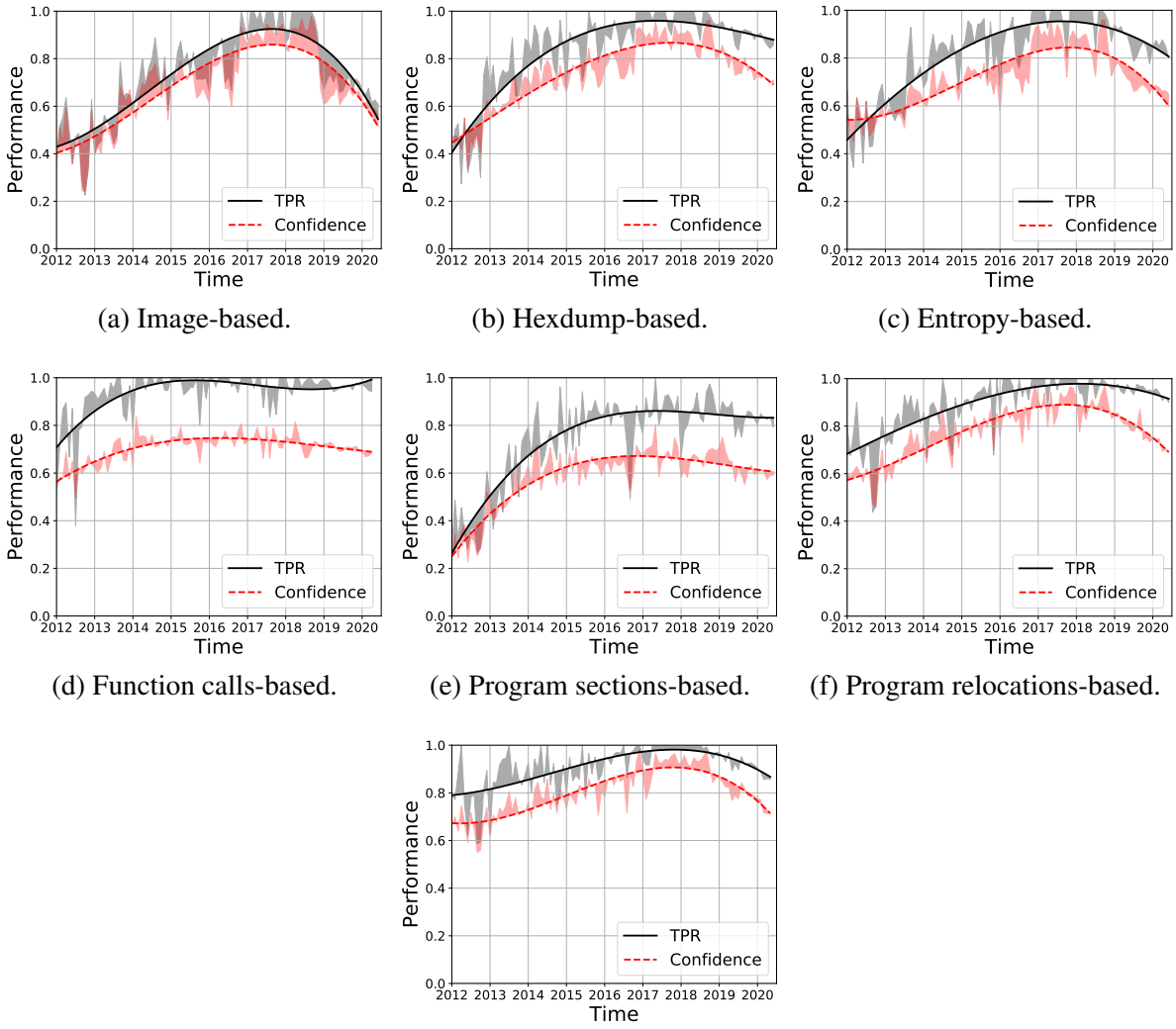


Figure 5.3: The time distribution of the collected (filtered) malicious samples. Notice that most of the samples were collected in 2013, and within the 2018-2020 duration.

Table 5.1 shows the distribution of the collected Windows software binaries in our dataset before and after filtering out the duplicates. We note that among the 64,233 Windows software with unique hashes, only 28,160 Windows software have unique feature representations.

The distribution of the first seen date of the samples in our dataset is shown in Fig. 5.3. Notice that most of the malware samples appeared between 2018 and 2020, highlighting the exponential growth of malware activity in recent years [9]. A similar trend was also observed in 2013, when the malware activity grew significantly [5]. Even though the "zbot" malware family was first seen in 2008, its samples continue to be active as of 2021.



(a) Image-based. (b) Hexdump-based. (c) Entropy-based.

(d) Function calls-based. (e) Program sections-based. (f) Program relocations-based.

(g) Strings-based.

Figure 5.4: The overtime malware detection performance evaluation using seven different data representations. The baseline models are trained on 80% of the malware samples captured in the period 2017-2018, and evaluated on the remaining samples on weekly-basis. The highlighted areas are the actual weekly performances, while the lines represent the performance trends. The reported confidence is the detection confidence of the malware detection model in detecting malware.

Experiment Setup and Metrics. In this work, we examine the benefits of retraining on the temporal robustness, which is defined as the consistent performance of a model in detecting new unseen malware mutations over time. As a starting point, we used the Random Forest (RF) and Convolutional Neural Networks (CNN) as baseline learning algorithms to implement machine

learning-based malware detectors.

Utilized Learning Algorithms. In this section, we review the learning algorithms used in our experiments and evaluation. We use two learning algorithms, namely a Random Forest (RF) model and a Convolutional Neural Networks (CNN) model. We use both to model malware, and in the following, we review both algorithms alongside arguments for their suitability for our problem space.

Random Forest (RF). RF is a machine learning model that consists of N decision trees. Each decision tree is trained on a collection of random features from the feature space, and it is used to calculate the non-linear relationship between the input features and the output decision. The final prediction of the RF classifier with N decision trees is determined by either (1) the majority vote over the predictions, or (2) the average prediction obtained from all N trees.

Convolutional Neural Network (CNN). CNN is a deep learning model used in image classification and pattern recognition. The basic unit of CNN is a convolutional layer, which generates feature maps capable of extracting deep feature representations of the input. Once a feature vector is fed into a convolutional layer, it is transformed to a feature map whose shape can be determined by (feature map height) $\times$ (feature map width) $\times$ (feature map depth). The CNN performs well in extracting patterns in higher dimensionality when the pattern location in the feature space is irrelevant.

In our work, we use CNN for image-based representations, and RF for all other representations. Moreover, while the usage of the aforementioned learning algorithms may not be optimal for some representations, we believe such an optimality is irrelevant to the main focus of this work: our goal is to investigate the shortcomings of model retraining for malware detection, along with observations and insights generalizable to other learning algorithms. As such, the performance of both algorithms is only a baseline to measure the impact.

We divide our dataset into training and testing sets, following the standard 80%-20% data splitting. We then trained a temporal model on 80% of the benign software, and 80% of malware samples

that appeared between 2017 and 2018, followed by a weekly evaluation of the trained model to detect the malicious samples that appeared between 2012 and 2020. Considering the insufficient weekly data of samples before 2012, we do not consider them for our weekly experiments.

Evaluation Metric. The trained models were evaluated using the true positive rate and confidence of the model. The *true positive rate* is the ratio between the correctly classified malware and the total number of malware samples. The model's confidence is the probability of the model ascribing the "malware" label to a malicious sample. From our experiments on the trained model, we achieved a benign detection performance of over 90% for all representations.

Malware Detection Temporal Robustness

After the training phase, we evaluated the temporal robustness of seven different malware detection systems, each of which used a unique and discrete set of features. In this evaluation, we only report the true positive rate (correctly classified malware samples). The performance of our trained models on the benign samples was >90% across all configurations, which is slightly low in comparison with state-of-the-art performance. The reduced performance is due to (1) dataset filtering through which we ensure the uniqueness of each sample by removing the bias towards previously observed samples, and (2) data splitting, which is based on the *first seen* time reported by the *VirusTotal* API. In prior works, the authors did not conduct temporal data splitting, which is necessary to ensure the effectiveness of the used data representation and learning techniques, especially in examining retraining. Therefore, despite a marginally lower accuracy, our data splitting approach provisions correctness across all experiments. This approach is in essence similar to the experimental settings proposed by Pendlebury *et al.* [113].

Overtime Detection Performance. Fig. 5.4 shows the performance of the baseline models trained on samples captured between 2017 and 2018 on a weekly basis. The shaded region represents the actual performance, while the line represents the performance trend. For each result, we report the true positive rate alongside the model's classification confidence.

Fig. 5.4 shows that the model's performance equally degrades over time for the new malware samples (after 2018), and the old malware samples (before 2017). This is largely due to the malware mutations over time and the appearance of new malware families.

For some representations (*i.e.* relocation representation, refer to Fig. 5.4f), the detection performance does not significantly deteriorate over time. However, for those representations, the confidence of the model decreased significantly. This means that while the malware sample is detected by the model, the probability that it is assigned to the malware class is quite low.[2] The decrease in the model confidence can be attributed to (1) the appearance of new malicious families that did not exist during 2017-2018, and (2) the mutations of the existing malware families (within that duration), causing shifting in the behavioral patterns. To ascertain these attributions, we investigate the models' performance on new malware families, and mutations of existing malware families. We split the malware occurring between 2019 and 2020 into three subsets:

(1) Seen families: This subset includes all malware samples from the families that were observed during 2017 and 2018.

(2) Unseen families: This subset includes (a) all malware samples from families that emerged during 2019 and 2020, or (b) the families that have been seen in the past (*i.e.* before 2017), but did not appear during 2017 and 2018.

(3) Singleton samples: This subset includes samples that could not be labeled as a known family. The problem with labeling occurs due to a conflict in assigning families by the antivirus engines (*e.g.* equal number of *VirusTotal* reports identify the sample as different families).

Table 5.3 shows the performance of the baseline models trained using the traditional dataset splitting approach (dataset is split randomly to 80%-20% training and testing subsets), alongside the 2017-2018 period trained models. Notice that the highest performance was recorded during the same training period, with a significant performance degradation before 2017 and after 2018. Dur-

[2]This behavior indicates that the sample is within the decision boundary of the malware class, but is drifting towards the boundary edge (Figure 5.1b).

Table 5.3: The performance evaluation (%) of different malware detection systems. Random: the data is randomly split into 80% training and 20% testing sets. Meanwhile, other performances are reported for baseline models trained on 80% of the malware samples captured in the period 2017-2018. During: refers to the performance (malware detection rate) on the remaining 20% at the same period.

Representation	Random	During	Before 2017	After 2018			
				Overall	Seen	Unseen	Singleton
Image-based	88.11	85.00	66.60	65.41	71.88	63.06	72.90
Hexdump	90.72	96.32	69.16	89.86	92.79	87.28	89.67
Entropy	87.85	92.20	62.14	86.39	89.26	81.28	89.51
Functions	91.95	98.99	79.50	94.93	95.86	94.19	94.84
Sections	88.82	87.12	54.09	84.43	86.55	80.81	86.56
Relocations	93.44	98.05	78.92	94.61	95.70	93.07	95.31
Strings	93.86	98.63	85.76	91.28	92.45	88.25	93.77

Table 5.4: The malware detection performance evaluation using periodic retraining approach. BL: the baseline model's malware detection rate without retraining (%), T: model's retraining frequency.

Represent.	BL	Weekly		Bi-weekly		Monthly		2-Months		3-Months		6-Months		Yearly	
		Per.	T	Per.	T	Per.	T	Per.	T	Per.	T	Per.	T	Per.	T
Image	65.41	86.74	91	86.42	45	84.46	21	88.21	10	86.23	6	85.43	3	77.21	1
Hexdump	89.86	98.23	91	98.32	45	98.08	21	97.89	10	97.59	6	97.14	3	94.78	1
Entropy	86.39	97.96	91	97.70	45	97.76	21	97.59	10	97.02	6	96.24	3	92.86	1
Functions	94.93	98.59	91	98.56	45	98.53	21	98.49	10	98.36	6	98.24	3	96.78	1
Sections	84.43	94.60	91	94.53	45	94.29	21	94.14	10	94.26	6	93.30	3	89.86	1
Relocations	94.61	98.98	91	99.00	45	98.94	21	98.78	10	98.57	6	98.21	3	97.33	1
Strings	91.28	98.20	91	98.10	45	97.94	21	97.86	10	97.81	6	97.27	3	95.78	1

ing 2019 and 2020, the seen and singleton subsets performance is nearly identical. However, their performance is lower than the performance reported during 2017 and 2018. The detection rate of the unseen families' samples is considerably lower than the other subsets. Clearly, the decrease in the detection rate of *unseen families* subset highlights the limitation of the malware detection approaches in detecting emerging malware families.

Key Takeaways. From the above analysis, we make the following key observations. (1) The commonly used data splitting approach does not reflect the real-world performance of the malware

Table 5.5: The malware detection performance evaluation using several retraining approaches. BL: the baseline model's malware detection rate without retraining (%), T: model's retraining frequency.

Represent.	BL	Confidence		OOD (L_2)		OOD (MD)	
		Per.	T	Per.	T	Per.	T
Image	65.41	87.75	18	86.57	81	86.37	87
Hexdump	89.86	97.18	7	97.89	47	98.21	84
Entropy	86.39	97.41	15	97.79	68	97.98	85
Functions	94.93	98.15	9	98.52	38	98.61	74
Sections	84.43	93.66	7	91.84	24	94.45	57
Relocations	94.61	98.47	7	98.90	69	98.93	80
Strings	91.28	97.58	9	97.99	57	98.04	76

detection models. (2) The state-of-the-art Windows malware detection models may not generalize to unseen malicious behaviors and patterns. (3) Training on mutated variances of malware does not ensure accurate detection of the original old malware. (4) In contrast to the assumptions made in the prior works that singleton samples are "familyless" with unique patterns [43, 91], our results show that they have a similar detection rate to seen families samples. Therefore, singleton samples are likely the unrecognized mutations of existing malware families with shared behavioral patterns.

Malware Detection Model Retraining

Model retraining is an approach to keep up with the emerging threats since the performance of outdated models deteriorates over time. However, model retraining involves various actions that need to be systematically undertaken in order to ensure accurate detection. First, the malware mutation timeline follows a random pattern [143, 79, 154], which makes it hard to infer the timeline of model retraining. Second, while the existing approaches may be able to retrain the model within the optimal time frame [51, 143], however, they are incapable of characterizing the mutation patterns of malware families.

By making progress along those two frontiers we can significantly boost the performance of the de-

tection frameworks in the ongoing arms race. Therefore, we formulate the following two questions as our objectives to strengthen the model retraining process: (1) *"What is the suitable timeline to retrain a model?"*, and (2) *"What approaches to use to detect emerging and mutated families?"*.

Model Retraining. In this section, we analyze three different retraining approaches. Our evaluations show the trade-off between the model retraining frequency and detection performance loss.

Retraining Approaches. There are different approaches to invoke retraining, including periodic retraining, confidence-based retraining, and out-of-distribution-based retraining.

Periodic Retraining. It is the most widely adopted approach in the existing machine learning-based malware detection literature [94, 83, 135]. Therefore, replicating periodic retraining enables us to understand the operation of a wide range of deployed detection systems. In this approach, the retraining process is periodically invoked, *e.g.* after every week. In our experiments, we implemented weekly, bi-weekly, monthly, 2-months, 3-months, bi-yearly, and yearly periodic retraining. To exemplify how retraining is implemented, the case of weekly retraining is considered. At the i^{th} week, the model is retrained on malware captured at weeks $0, 1, \ldots, i-2, i-1$, and evaluated on malware captured at the i^{th} week.

Confidence-based Retraining. In a two-class classification model, the confidence is defined as the probability of the model in classifying input x into "malware" label. In our implementation of confidence-based retraining, we invoked the retraining process when (1) the average model's confidence for the $(i-1)^{th}$ week was below a certain threshold T_1, or (2) the confidence of N samples captured in the $(i-1)^{th}$ week is below a threshold, T_2. The selection of the confidence thresholds T_1 and T_2 is configurable, and we selected $T_1 = 80\%, T_2 = 60\%$ in this study.[3]

Out of Distribution (OOD)-based Retraining. OOD-based techniques capture the anomaly samples that do not follow a specific distribution (*i.e.* patterns). Therefore, OOD-based techniques can

[3]Selecting different thresholds may change the reported performance. We noticed that the model's confidence starts decreasing below 80% upon introducing new and mutated samples. Therefore, we used T_1=80% as a threshold. Moreover, the models' confidence on the new emerging families is typically low (around 50-60%), and therefore T_2 is selected to be 60%.

Table 5.6: The malware detection performance evaluation (%) on seen, unseen, and singleton samples after using several retraining approaches. Seen samples evaluation refers to the detection rate of the model on samples of malware families that it was trained on. Sing.: Singleton Windows malicious samples.

Represent.	BL			3-Months			Confidence			OOD (L_2)			OOD (MD)		
	Seen	Unseen	Sing.	Seen	Unseen	Sing.	Seen	Unseen	Sing.	Seen	Unseen	Sing.	Seen	Unseen	Sing.
Image	71.88	63.06	72.90	87.07	84.58	85.13	88.26	88.1	85.27	87.44	86.93	84.77	86.92	87.01	84.4
Hexdump	92.79	87.28	89.67	98.33	95.72	96.49	97.97	94.95	96.27	98.54	95.85	96.96	98.87	96.24	97.49
Entropy	89.26	81.28	89.51	97.09	95.99	97.05	97.64	96.05	97.33	97.59	96.57	97.83	98.28	96.45	97.83
Functions	95.86	94.19	94.84	98.91	98.05	97.65	99.05	97.7	97.27	98.98	98.34	97.78	99.05	98.43	98.03
Sections	86.55	80.81	86.56	94.86	93.17	92.60	94.34	92.5	91.9	93.15	90.51	89.28	95.07	93.41	92.72
Relocations	95.70	93.07	95.31	98.81	97.01	98.24	98.85	96.77	98.02	99.11	97.32	98.57	99.05	97.32	98.73
Strings	92.45	88.25	93.77	97.7	98.0	97.55	97.44	97.97	97.25	98.06	98.35	97.27	97.91	98.08	97.65

Table 5.7: The evaluation (%) of using the hybrid approaches of 3-months-based retraining and confidence-, OOD-, and MD-based retraining approaches. The model is retrained if 1) last retraining occurred three months ago, or 2) the approaches invoke the retraining process. T: model's retraining frequency. Sing.: performance on Singleton samples.

Represent.	Confidence					OOD (L_2)					OOD (MD)				
	Overall	Seen	Unseen	Sing.	T	Overall	Seen	Unseen	Sing.	T	Overall	Seen	Unseen	Sing.	T
Image	88.29	88.41	88.54	86.76	20	86.27	87.0	87.04	84.4	82	86.58	87.05	87.37	84.62	88
Hexdump	97.74	98.56	95.73	96.66	10	97.84	98.52	95.87	96.83	48	98.23	98.87	96.32	97.49	85
Entropy	97.58	97.65	96.23	97.63	17	97.8	97.58	96.57	97.83	69	97.99	98.28	96.45	97.86	86
Functions	98.51	99.05	98.23	97.75	12	98.61	99.06	98.49	97.76	39	98.62	99.05	98.43	97.97	75
Sections	94.18	94.88	93.07	92.39	10	94.35	94.99	93.26	92.64	25	94.48	95.07	93.45	92.78	58
Relocations	98.73	98.93	96.95	98.47	9	98.94	99.11	97.32	98.69	70	98.93	99.05	97.32	98.73	81
Strings	97.73	97.67	98.01	97.44	11	98.03	98.12	98.32	97.36	58	98.08	97.97	98.04	97.74	77

identify anomalies when *concept drift* causes a high distortion in the existing patterns. In our work, we apply OOD-based techniques to model the retraining process in order to capture the distortions created by the *concept drift*. This method enabled us to capture new and mutated malware samples, which can be used to (1) invoke the retraining process, or (2) estimate the malware family activity.

For a sample x to be considered as OOD it must contain new patterns that do not exist within the known samples. The new patterns cause a shift in the feature space by a factor α. If the shift is beyond a certain threshold, T_3, the sample x is considered OOD. Instead of manually selecting T_3, we utilized the Local Outlier Factor [36] to find the anomalous samples by measuring their local deviation from their neighbors.

We trained the OOD-based model on all captured samples from 2017 to the time when the model

Table 5.8: The retraining effects on the malware detection performance (%) using five distance metrics for OOD approach, T: model's retraining frequency. Per: detection rate (%).

Represent.	Euclidean		Manhattan		Cosine		Hamming		Mahalanobis	
	Per.	T	Per.	T	Per.	T	Per.	T	Per.	T
Image	86.56	81	87.11	86	86.74	91	89.97	50	86.37	87
Hexdump	97.89	47	98.07	64	97.92	54	98.26	85	98.21	84
Entropy	97.80	68	97.82	68	98.07	86	97.56	53	97.98	85
Functions	98.55	38	98.67	49	98.58	78	98.67	49	98.61	74
Sections	91.84	24	93.48	25	91.87	27	93.48	25	94.45	57
Relocations	98.90	69	98.97	79	98.88	65	98.97	79	98.93	80
Strings	98.00	57	97.89	58	98.04	74	91.28	0	98.04	76

is last retrained. The output of this retraining method helped us in detecting the novel out-of-distribution samples every week. When we captured ten novel out-of-distribution samples, we retrained the baseline model along with the local outlier model. To identify the OOD samples, the local outlier factor model utilizes different distance metrics.

Out-of-Distribution Distance Metrics. To identify the OOD samples, the local outlier factor model utilizes different distance metrics. In this work, we focus on the five most commonly-used distance metrics: Euclidean, Manhattan, Cosine, Hamming, and Mahalanobis distances. We provide a description of each metric in the following:

1. **Euclidean (L_2) Distance:** For two samples, x_1 and x_2, the euclidean distance measures the shortest distance among pairs of points from the two samples as: $\sqrt{\sum_{i=1}^{n}(x_1^i - x_2^i)^2}$, where x_1^i is the i^{th} feature in the feature representation of sample x_1.

2. **Manhattan (L_1) Distance:** For x_1 and x_2, Manhattan distance is measured with respect to each axis (*i.e.* feature) in the feature space as: $\sum_{i=1}^{n} |x_1^i - x_2^i|$, where the distance between two samples is the sum of the absolute differences of their feature representations.

3. **Cosine Distance:** For two vector representations of x_1 and x_2, the cosine distance is defined as 1 minus the cosine of the angle between the two vectors in the feature space, represented

as follows: $1 - \frac{\sum_{i=1}^{n} x_1^i \times x_2^i}{\sqrt{\sum_{i=1}^{n} (x_1^i)^2} \times \sqrt{\sum_{i=1}^{n} (x_2^i)^2}}$.

4. **Hamming Distance:** In contrast to the aforementioned distance metrics, the hamming distance records the count of features that are not identical in the feature representation of two samples, and can be considered similar to the xor ($\oplus$) operation. In our application domain, the input sample is feature-wise compared to its neighborhood in order to compute the average hamming distance.

5. **Mahalanobis Distance (MD):** The Mahalanobis distance measures the distance between a sample x_1 and a samples' distribution d. In particular, it measures the distance between x_1 and d, calculated using the standard deviation as a distance unit. The Mahalanobis distance can be considered similar to the Euclidean distance, but operating in a transformed feature space.

In Table 5.8, we report the OOD-based retraining results using the five distance metrics. Overall, the Mahalanobis-based detector outperforms its counterparts. Therefore, it was used as the baseline for the malware emerging and mutation experiments and analysis. The Mahalanobis-based detector was used as the baseline for malware emerging and mutation experiments and analysis, as it outperforms its counterparts.

Model Retraining Evaluation. Table 5.4 and Table 5.5 show the overall performance using the periodic, confidence-based, and OOD-based retraining approaches. Overall, the average performance of the baseline model increased from 89.86% to 98.23% for the Hexdump representation after weekly retraining. However, achieving the performance increase required retraining the model **91 times** between January 2019 to September 2020. While retraining the model every three months (a total of **six times**), a detection rate of 97.59% was achieved.

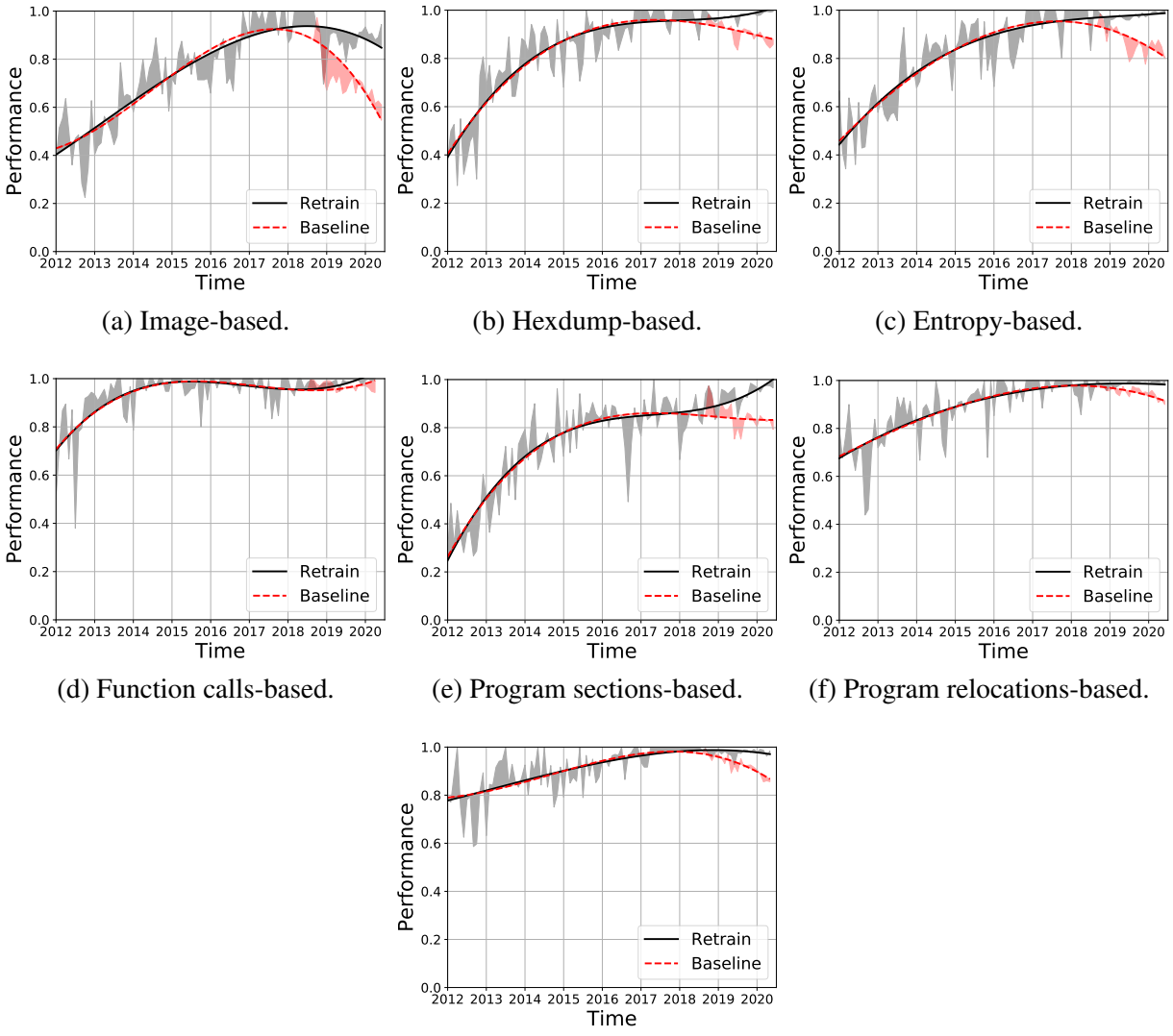
For confidence-based retraining, the retraining frequency depends on the sample representation and the model structure. In contrast, the OOD-based approaches depend only on the sample representation. We found that the OOD-based approaches require frequent retraining with no

significant performance increase (*i.e.* **84** retraining times compared to **seven** retraining times in the confidence-based approach for the Hexdump representation). In Table 5.8 in the appendix, we report the OOD-based retraining results using the five distance metrics. We found that the Mahalanobis-based detector outperforms its counterparts, therefore, it was used for the subsequent experiments and analysis.

Seen vs. Unseen Malicious Families. After a high-level analysis of model retraining, we now evaluate the model's performance using the different retraining approaches on seen families, unseen families, and singleton samples. This step is crucial to understand when and why the model's performance deteriorates. Table 5.6 shows that while all approaches have a similar performance on the seen malicious families, the Mahalanobis-based OOD-based retraining provides the best performance over unseen and singleton samples. This could be due to Mahalanobis-based OOD's capability of effectively detecting OOD samples and timely invoking the retraining process upon encountering those samples.

Hybrid Retraining Approach. Retraining the baseline model every three months provides the best overall performance, considering the reduced retraining overhead. However, for the *unseen families*, this approach does not perform well compared to the other approaches. To address this shortcoming, we adopt a hybrid retraining approach which invokes retraining either (1) when the confidence-based or OOD-based approaches invoke the retraining process, or (2) when the model has not been retrained in the previous three months. Table 5.7 shows the performance of the hybrid confidence-based, L_2 and MD OOD-based retraining approaches. Overall, the hybrid confidence-based retraining approach provides the best trade-off between the performance and retraining frequency.

Using the hybrid confidence-based approach as a baseline, Fig. 5.5 shows the performance of each of the representation models before and after the retraining. Note that retraining significantly improves the performance, particularly for the image-based and function entropy-based representations.



(a) Image-based. (b) Hexdump-based. (c) Entropy-based.

(d) Function calls-based. (e) Program sections-based. (f) Program relocations-based.

(g) Strings-based.

Figure 5.5: The overtime detection performance evaluation after model's retraining using the hybrid confidence based approach.

Shortcomings of Model Retraining. Having established the benefits of retraining on the detection rate of malicious samples, we now proceed by analyzing the impact of retraining on the performance of the benign samples. Recall that retraining on a diverse sample set reduces the decision boundaries of the benign class, and therefore introduces false positives.

Recall that the benign dataset was split into 80%-20% training-testing sets. To study the effect of

Table 5.9: The model's retraining effects on the benign detection rate (%). BL: the baseline benign detection accuracy.

Represent.	BL	Total Months Retrained			
		3	6	12	21
Image	95.54	95.34	94.88	92.21	91.60
Hexdump	96.93	95.63	94.63	93.63	90.14
Entropy	97.52	94.83	91.94	89.54	86.65
Functions	94.18	93.33	92.16	91.21	89.29
Sections	91.54	83.25	82.27	81.07	76.16
Relocations	96.44	94.11	93.67	93.00	82.00
Strings	93.63	91.88	91.22	89.68	86.61

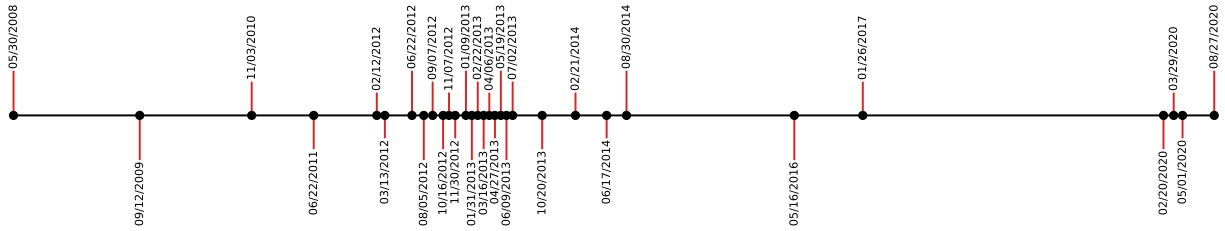


Figure 5.6: The estimated mutations timeline for zbot malware family. Over the period 2008-2020, 31 mutations are detected.

the retraining process on benign performance, we evaluated the test set (20%) using the baseline model. The baseline model was trained on malware samples captured between 2017–2018, and the models were retrained after 3, 6, 12, and 21 months. Due to the limited number of benign samples, we balance the weights accordingly in the retraining process.

We report the results in Table 5.9, with the following key takeaways. (1) Generally, the benign detection rate decreases over time, and (2) The benign detection rate was not higher than the baseline in any of the representations across all retraining frequencies. Interestingly, we found that, after 21 months, the detection performance decreases by up to 15% for program section and relocation detection models. The results clearly support our hypothesis that retraining only guarantees a temporary improvement in malicious detection, while continuously losing ground for benign detection. Taking into account the computational complexity associated with retraining,

and the reduced benign detection performance, it is logical to question the benefits of retraining over its caveats.

Emerging Malware Detection. In our dataset, 908 Windows malware families appeared in 2019–2020. Using OOD for detecting new malware families, 896 families were detected as OOD using the image-based Mahalanobis distance OOD detector. Similarly, 884 families were detected using Hexdump representation, with the function calls representation having the lowest detection rate of 86.01% (781 families). Alongside the new emerging families, other samples of existing families were detected as OOD. This behavior might be a natural result of malware mutations, where the mutated malware is detected as an anomaly.

Leveraging this behavior, we found that OOD techniques are useful for malware mutations detection and temporal characterization. In the following, we showcase the OOD-based detector's capability by providing two case studies.

Case Study I: zbot Mutations. We used the Mahalanobis-based OOD detector to estimate the zbot malware family activity (*i.e.* evolution and mutation). Our dataset contained 1,511 zbot samples captured over the entire evaluation timeline (2008–2020). A majority of those samples were captured between June 2012 and July 2013, indicating an elevated activity of the malware family during that time. We trained the OOD-based detector on the first $i-1$ zbot samples and predicted whether the i^{th} sample is OOD. Samples detected as OOD were then categorized as zbot mutations.

Since this process is data representation-dependent, we only considered the samples that were classified as a mutation by the OOD-based detectors trained using the seven representations. This selective approach allowed us to remove the bias in our dataset. After training our models, we estimated the timeline of zbot family mutation, and plot them in Fig. 5.6. Overall, we found 31 mutations in the zbot family, with 18 mutations observed during 2012–2013. Since we considered a subset of samples, our results provide a conservative estimate of malware mutations. In practice, however, there could be more mutations in the wild beyond our conservative estimate. Never-

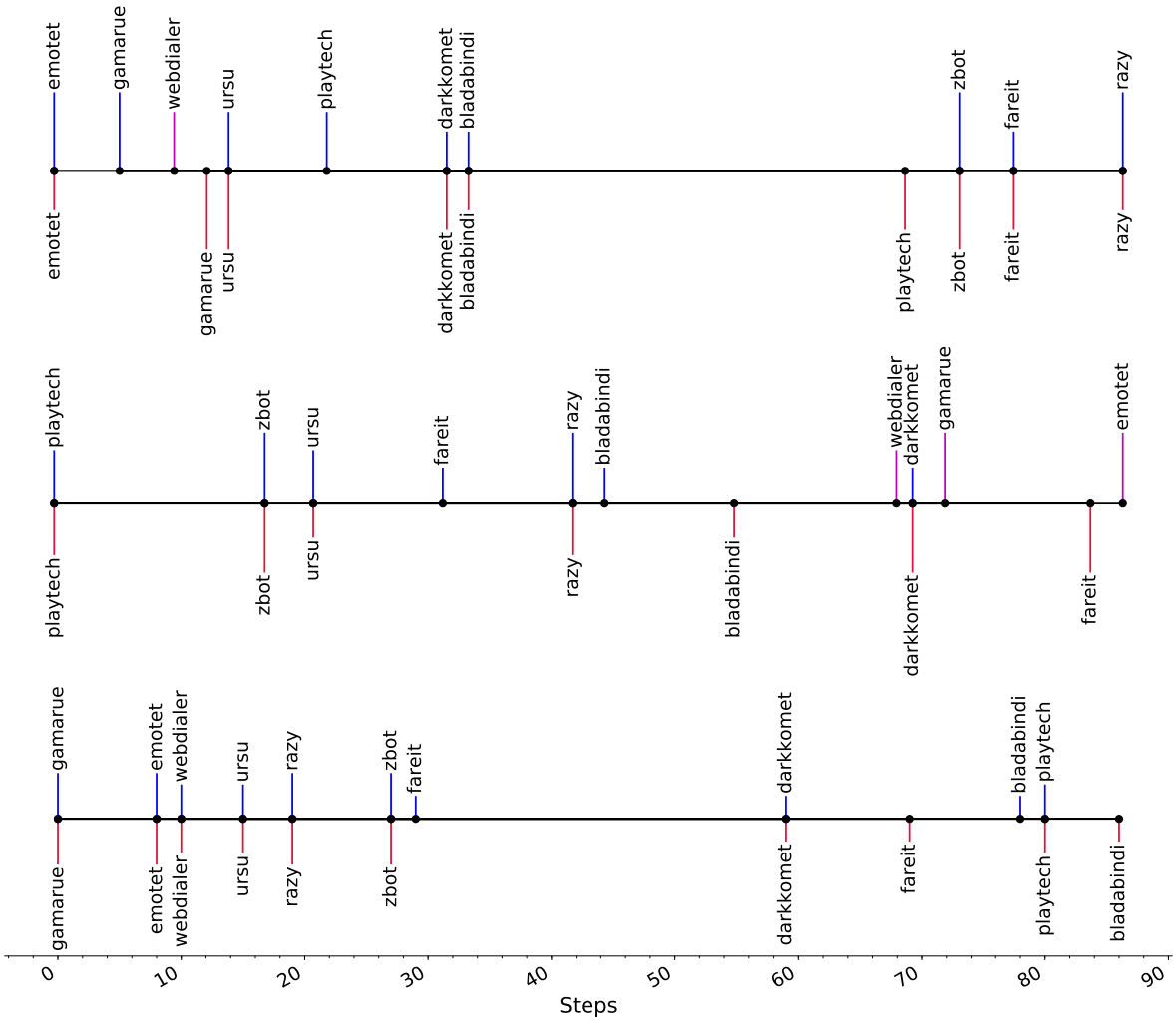


Figure 5.7: The visualization of emerging family detection for three trails. For each trial, the upper part (above the horizontal line) represent the steps at which each family emerged, and the lower part represents when the OOD detector raise the alarm regarding the family emergence. — The family emergence was detected, — the step at which the family was detected, and — the detector failed to detect the family emergence.

theless, by applying Mahalanobis-based OOD detector, we were able to monitor the behavior of malware families, which can be further leveraged to enhance our understanding of the evolving threat landscape.

Case Study II: Emerging Malware Families Detection. In this case study, we investigated the effectiveness of the Mahalanobis-based OOD detector in estimating the malware family mutations.

In the following, we systemically evaluate its capabilities in detecting the emergence of new malicious families under virtual malware timeline distribution.

Virtual Timeline Settings: We consider a timeline consisting of 100 steps (*e.g.* weeks). At the i^{th} step, the OOD-based detector is trained on all the samples that appeared at steps $1, 2, 3, \ldots, i^{th-1}$, using combined representation, and new malware samples captured at the i^{th} step are forwarded to the detector for emerging malicious family detection. Once a sample is deemed out-of-distribution, the detector raises an alarm indicating the emergence of a new malicious family.

Malware Distribution Pool: For this experiment, we only consider the malware samples that belong to the top ten malicious families in our dataset (refer to Table 5.1). Initially, a randomly selected family's malware samples (1—10 samples) appear at step 1. For the other nine families, the steps at which they will emerge are randomly selected in the range of $[1, 100]$. At the i^{th} step, one to ten samples of each already-emerged family (*i.e.* families emerged at the steps $[0, i]$) will be observed and forwarded to the detector. We note that, for each family, we follow the same time distribution of the captured samples (*i.e.* zbot malware that appeared in May 2017 will always appear after a zbot malware that appeared in March 2013 in our experiments).

Training & Evaluation Metrics: We run the aforementioned experimental settings 1,000 times (*i.e.* trials). At each trial, the samples distribution and the order in which the malicious families emerged are randomized. For each trial, an OOD-based detector is trained at each step, a total of 99 times, to detect the emergence of new malicious family samples. To evaluate the detector, three metrics are considered:

- False Alarm Rate (FAR) (lower is better): This metric represents the percentage of seen families samples that caused false alarms, where the detector deemed malware as a new emerging unseen family wrongfully.
- Average Detection Rate (ADR) (higher is better): This metric represents the percentage of emerging malicious families that the detector correctly detected their samples as out-of-distribution.

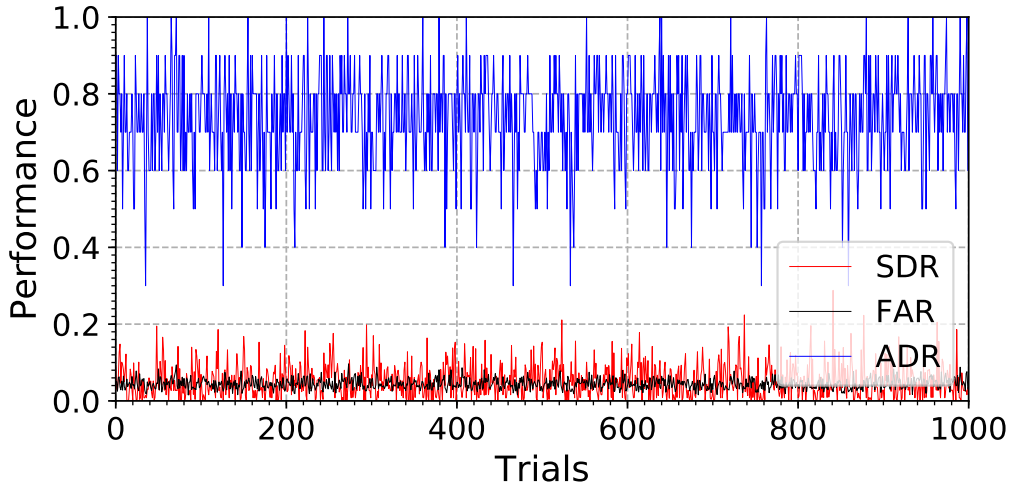


Figure 5.8: The emerging malware families detection metrics evaluation over the 1,000 trials. Each trail consist of different randomized configurations of malware and family distributions over 100 steps. The average values of the evaluation metrics accross the 1,000 trails are as follow: ADR is 74.55%, SDR is 5.20%, and FAR is 4.17%.

- Step Deviation Rate (SDR) (lower is better): In addition to detecting the emergence of a new malware family, we also measure the delay of the detector in detecting the emergence. For instance, a family that emerged at the i^{th} step, and was detected at j^{th} step will have a step deviation rate of $(j - i)/100$.

Across the 1,000 trials, we observe an average of 74.55% ADR, 5.20% SDR, and 4.17% FAR. We also show the visualization of OOD detector operation over three trials in Fig. 5.7. Note that the detector is effective in detecting the emergence of malicious families samples, with reduced false alarm rates.

Fig. 5.8 shows the performance of the OOD detector's FAR, ADR, and SDR evaluation metrics for emerging family detection over 1,000 trials. We observe average values of 74.55% ADR, 5.20% SDR, and 4.17% FAR. We note that these results are highly correlated to the configuration of the OOD detector, where a sensitivity of 2.5% was selected in the training process (see Fig. 5.9 for different configurations of the OOD detector's sensitivity parameter).

Key Takeaways. In this section, we show the effectiveness of out-of-distribution detection ap-

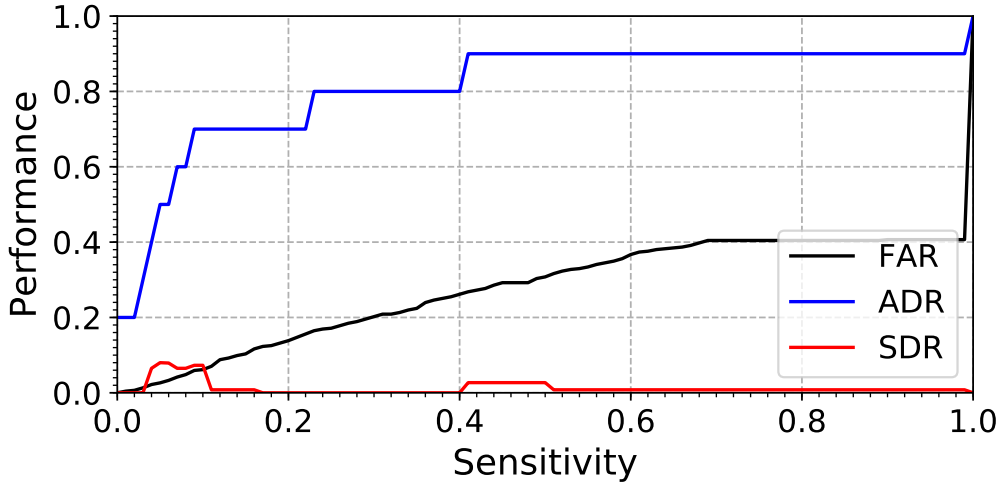


Figure 5.9: The effects of varying the OOD detector's sensitivity on the emerging malware detection. Increasing the sensitivity will result in higher detection rate (ADR), but also cause higher false alarms (FAR).

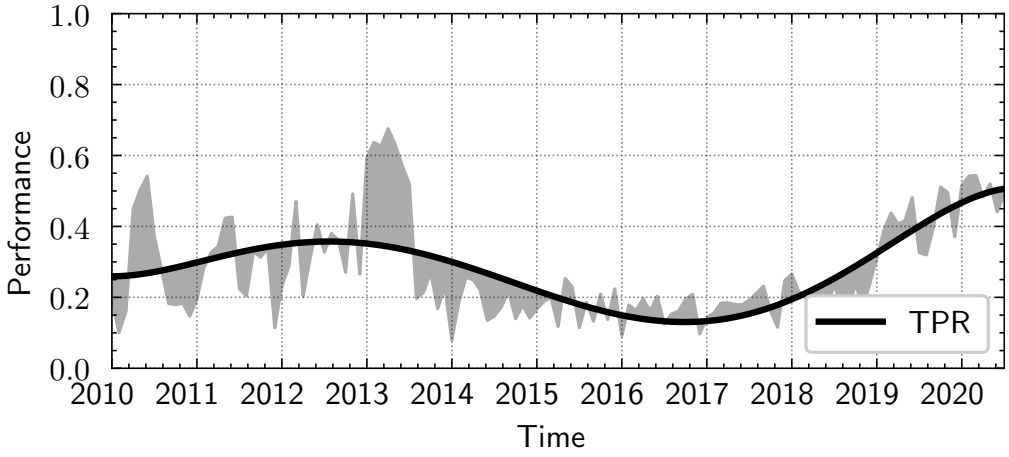
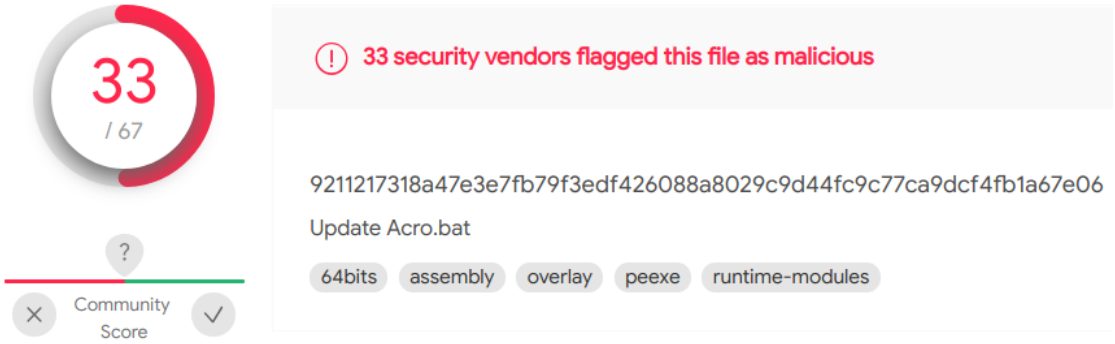
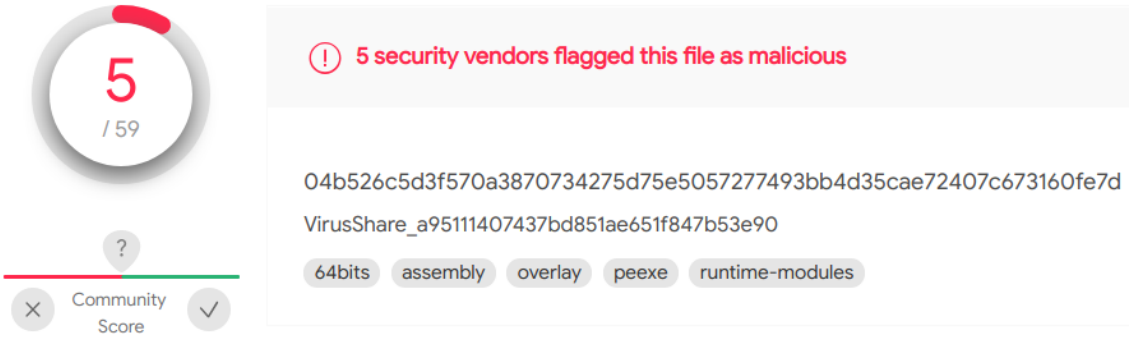


Figure 5.10: The *VirusTotal* online malware detection engines average detection rate (TPR). Unlike the common perception that new malware is hard to detect, malware captured in the period 2015 – 2018 has the lowest detection rate ($\approx$12%).

proaches in detecting and predicting the emergence of new and mutated malicious families. Our experiments showed that while new malware families samples can be detected, the detection confidence is reduced over time (*i.e.* as malware mutates), causing eventual misclassification. Early detection of the emergence of a new malicious family leads to analyzing its unique behavior for



(a) Before (MD5: a95111407437bd851ae651f847b53e90).



(b) After (MD5: 9d3284472a78344ec67fdd45afcfa67f).

Figure 5.11: The detection rate of Windows malicious sample using *VirusTotal* before and after padding one byte (0xFF) at the of the binaries. The new hash is not recognized by *VirusTotal*, and therefore is re-analyzed.

accurate malware detection. In this study, we inch towards enabling proactive emerging-malware detection and mitigation.

Online Detection Engines Are Vulnerable

In addition to evaluating the temporal robustness of state-of-the-art malware detection approaches, we also investigate the performance of online malware detection engines. We used the *VirusTotal* API to obtain a report for each malware sample. The reports contained information, including the engines that analyzed the malware sample, the first time the malware sample was captured, and the

last scan time.

For each engine, the report returns whether the sample was detected as malware or not. *VirusTotal* does not re-analyze malware with previously seen hash, and instead provides the latest report saved in a database. Therefore, in order to examine the temporal behavior of online engines towards detecting old and new malware, the malware sample needs to be altered and re-uploaded at the detection engine. Ideally, to preserve the malware functionality and prevent an increase in the file size, the file alteration should be minimal. An optimal strategy for that purpose is to add one byte towards the end of the file (without affecting its practicality nor executability [85]). We followed that approach by appending a single byte (0xFF) at the end of the binary file and re-uploaded the file to *VirusTotal*.

Fig. 5.10 shows the average detection rate of the malware samples during 2010 and 2020. The average detection rate refers to the ratio between the number of engines correctly detecting malicious samples, and the total number of detection engines. The results in Fig. 5.10 show that the detection engines are able to detect the new malware more accurately than the old malware. For instance, the malware samples that appeared in 2020 had an average detection rate of up to 50%. In contrast, the malware samples that appeared during 2015 and 2017 had a low average detection rate of $\approx$12%. Surprisingly, this imbalance in the average detection rate contradicts the common perception that new malware is harder to detect [82, 79, 155].

Our results paint a different picture, where we see an improved average detection rate for the recent malware samples compared to the old samples. This improvement is likely due to the "new malware retraining only" policy adopted by the malware engines. As discussed earlier, and comprehensively established later, the "retraining on new samples only," despite being effective in detecting new malware, has several limitations which reduce its efficacy.

Experiment and Case Study. In the following, we present an experiment and a case study to demonstrate that the state-of-the-art malware engines fail to accurately detect old malware samples. For our experiment, we modified the 5,783 malware samples appearing before 2019 by appending

one byte to the end of the software binaries, and re-uploaded them on *VirusTotal*. Among those samples, the detection rate of 2,872 (49.66%) malware samples decreased. In contrast, the detection rate of 137 (2.36%) samples increased, while the detection rate of 2,774 (47.98%) samples remained the same.

In Fig. 5.11, we present a case study of the "redcap" malware by showing the *VirusTotal* report before and after altering the malware sample. The report stored in the *VirusTotal* database indicates that 33 engines accurately detected the software as malicious in 2013. However, in comparison, only 5 engines detected the malware as malicious in 2021, showing a decrease in the detection rate. Among the 33 engines that previously detected the malware sample in 2013, 30 engines failed to detect the same sample in 2021.

Malware Respreading. The decrease in the detection rate can be exploited by malicious actors to re-use the old samples to reasonably easily bypass the detection engines by launching new attacks without mutating the malware. Toward investigating malware reuse and respreading, we examine the dataset for malware samples with identical representations. As malware respreading requires modifying the command and control server IP address, we exclude Hexdump representation due to its high sensitivity.

In our dataset, we found 514 cases (with 4,632 malware hashes) of malware re-use over one or more years. We refer to these discovered cases as ***revival chains***, as they reflect a series in which they were revived and reused by the malicious actors. Fig. 5.12a[4] shows visualization of malware revival chain, two more examples are available in Fig. 5.12 in the appendix. We notice that in 220 chains, the malware is labeled by VirusTotal engines to different malware families as the malware is progressing. Toward understanding this phenomenon, we construct a graph of the families appeared in the revival chains, shown in Fig. 5.13. Here, a direct link between two malicious families indicates that a malware was assigned to both families (as its label) during the revival chain. We

[4]The malware first appeared in 03/12/2020 as *torrentlocker* (MD5: $2dbd72500eb87da88c89bd38d9f6f8d0$), reappeared in 04/02/2020 as *zbot* (MD5: $e3fa4d022203e117384f38b46064b634$), and last appeared in 07/20/2020 as *torrentlocker* (MD5: $3d1eebbf3eb392151e096ad600a0b344$).

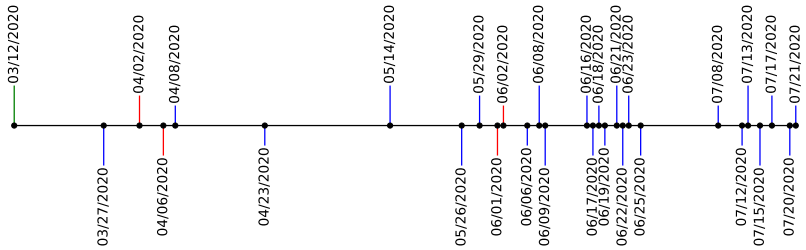
notice the existence of closed loops, such as *torrentlocker—cutwail—zbot*, where a malware is labeled as one of these families on its re-appearance in the revival chain. This indicates that the extracted features, and inherently the associated behaviors, of these three families, are highly similar. To validate this hypothesis, we visualized the feature space of the samples of the malicious families that appeared in the revival chains (shown in Fig. 5.14). Notice that while some families' malware samples create their own clusters, the samples of different families overlap within the feature space.

We note that the 514 chains identified in our dataset are limited to the malware samples that were correctly detected by the online engines across the timeline (*i.e.* 2008–2020). This phenomenon is concerning given that, as shown previously, 49.66% of the old malware are currently not being detected by most of the malware detection engines. Considering that some of those samples are being revived, this leads to negative takeaways: (1) malware revival is being exploited, and there are no safeguards against that, and (2) the existing techniques employed by the machine learning-based malware detectors (*i.e.* retraining) are insufficient to combat this phenomenon.

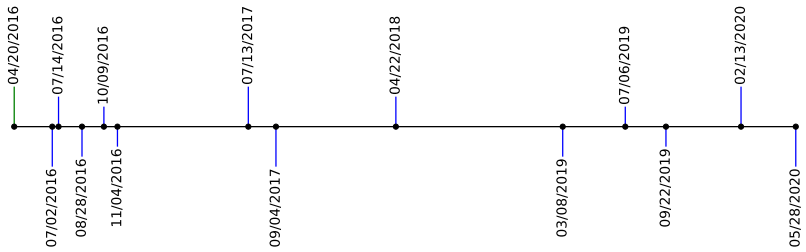
Key Takeaways. From the analysis of online malware detection engines, we make the following key observations. First, there are discrepancies in the detection rate of online detection engines. Some engines may detect old and new malware, while most of them lose detection accuracy for the old samples. Second, we found evidence to suggest that the discrepancies in the detection rate can be feasibly exploited to respread old malware, thereby bypassing the detection systems without mutation. Our analysis clearly reveals that the detection engines are not up-to-speed with malware in the ongoing arms race. Therefore, those engines need to be refined with improved detection accuracy with temporal persistence.

Overtime Family Labeling Inconsistency

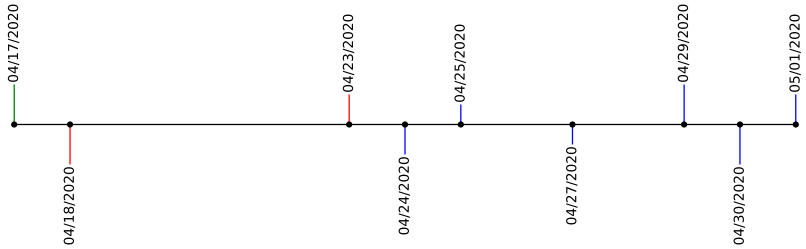
In this work, we highlight the existence of an ongoing malware revival and re-spreading chains, where malware with exact feature representation re-appear months or years after its initial seen



(a) The malware first appeared in 03/12/2020 as *torrentlocker* (MD5: $2dbd72500eb87da88c89bd38d9f6f8d0$), re-appeared in 04/02/2020 as *zbot* (MD5: $e3fa4d022203e117384f38b46064b634$), and last appeared in 07/20/2020 as *torrentlocker* (MD5: $3d1eebbf3eb392151e096ad600a0b344$).



(b) The malware first appeared in 04/20/2016 as *sogou* (MD5: $2dbd72500eb87da88c89bd38d9f6f8d0$), and last appeared in 05/28/2020 as *sogou* (MD5: $3279c2bd76f8891769b561ff5b9d2b1a$).



(c) The malware first appeared in 04/17/2020 as *regotet* (MD5: $8df49fd30a7193206bdef9e99c35e6bf$), re-appeared in 04/23/2020 as *gozi* (MD5: $bb17736081cbd7380ea03cc4a2cf2e01$), and last appeared in 07/20/2020 as *gozi* (MD5: $20c72cc97e12ee46cc85b4ea4c49535c$).

Figure 5.12: The visualization of malware revival and re-spreading chains. — The malware first seen date, — the malware family (label), retrieved from VirusTotal, did not change (*i.e.* similar to the one previously observed), — the malware family (label) changed from the last time the malware was seen.

date. Our observations include that after the revival of malware, *VirusTotal* engines might assign it to a different family. This assignments are common between certain families (i.e., *graz* and *netwalker*), reducing the probability of being false labeling, and indicating a potential inter-family shared patterns and associations. To better understand this phenomenon, and how malware assigned families can vary overtime, we conduct a comprehensive overtime malware family anal-

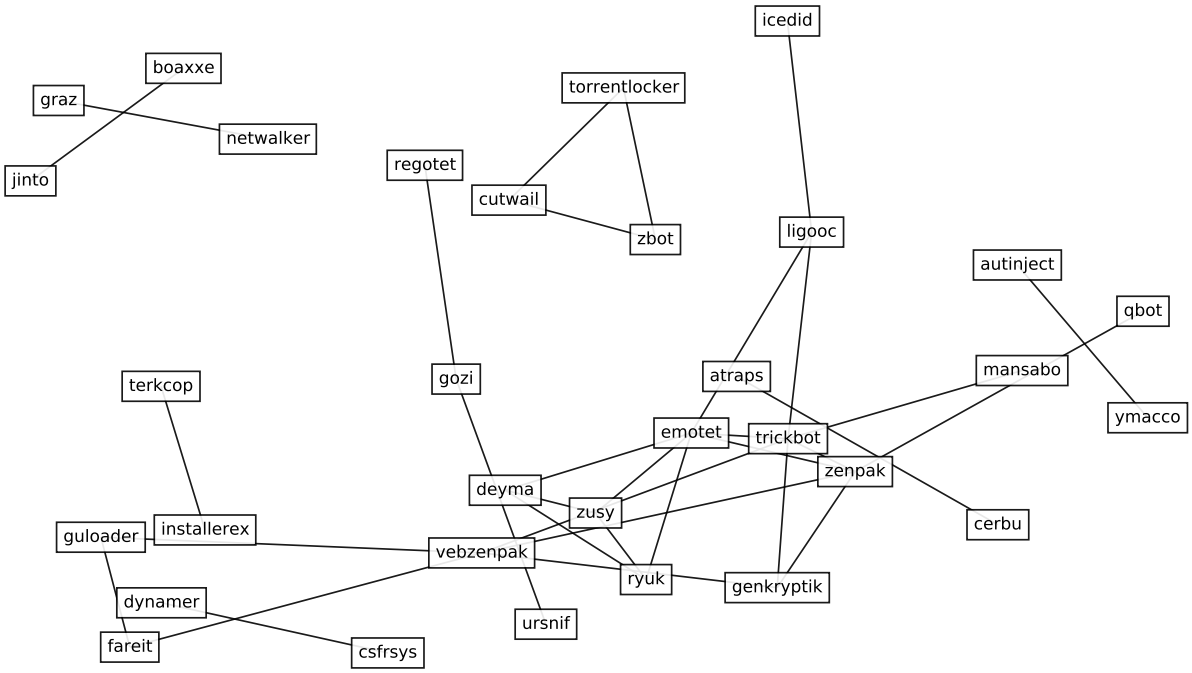


Figure 5.13: The malicious families that appeared in the malware re-spreading and revival chains. A direct connection between two families indicate that a malware appeared at different dates, and was labeled as both families.

ysis.

Family Labeling Inconsistency. To highlight the existence of such inconsistency, we used *VirusTotal* API original and re-analyzed reports to obtain the assigned families. Among our dataset, we observed 2,524 cases, in which a malware assigned family changed, shown in Table 5.10. Notice that some families was renamed, such as *rrat* and *revetrat*, *rescoms* and *remcos*, and *agensla* and *agenttesla*. For other cases, the assigned family is completely different, such as 42 *offerinstall* malware being reassigned to *appster*.

Inter-Family Shared Patterns Extraction. We further investigate the labeling inconsistencies, by exploring the existence of inter-family shared patterns within malware samples. In particular, we utilize YARA [11] rules toward examining the shared patterns within the samples of different families. Doing so will provide better understanding of the malicious family behavior and origin, taking a step forward toward robust malware classification.

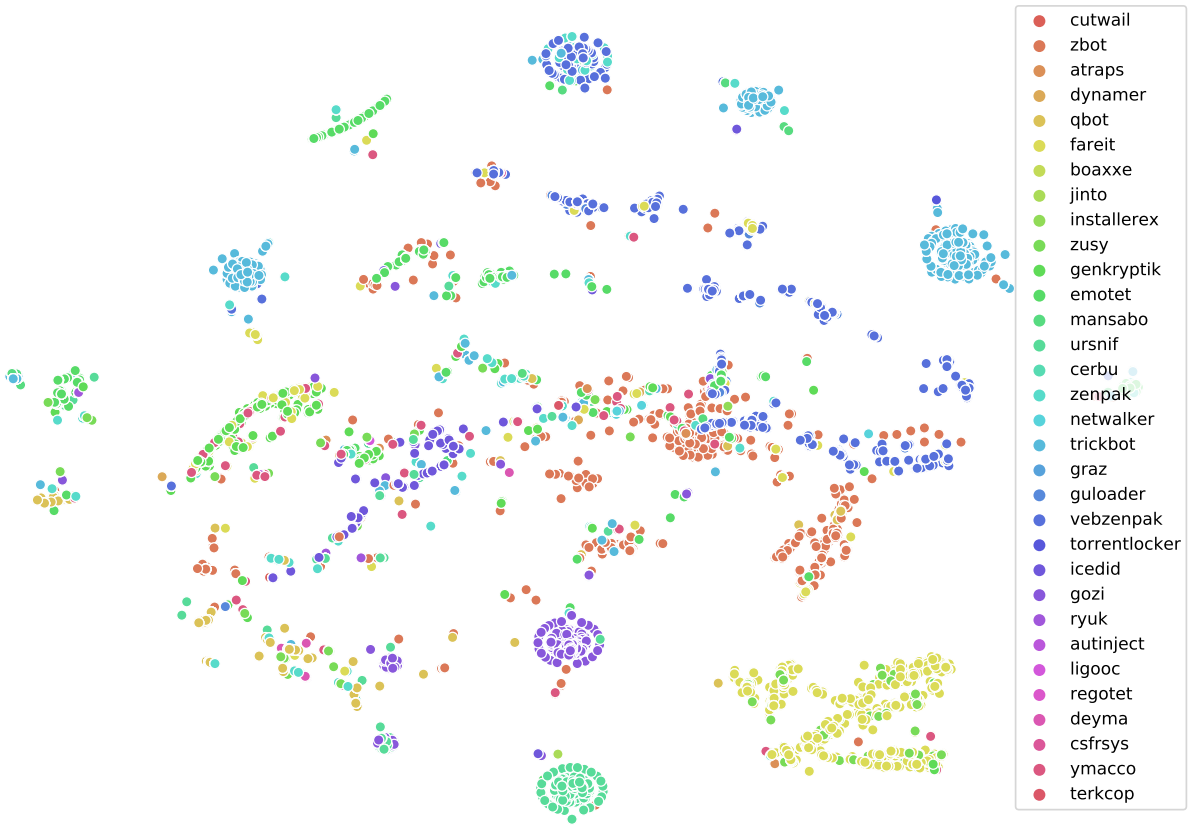


Figure 5.14: The t-SNE distribution of the malware belonging to the families appeared in the malware revival chains. Notice that while some samples create their own clusters, samples of different families overlap within the feature space.

To accurately extract malicious behavior-related rules, we first used yarGen [118] to white-list Windows benign op-codes and strings. This includes updating the existing Microsoft OS white-list database. Using the collected benign samples in our dataset, we updated the white-list database to include the common shared strings and op-codes within our benign dataset. This step is essential to ensure that the extracted malicious Yara rules represent malicious behavior, instead of capturing generic benign behavior within the extracted rules. Using yarGen, we extracted 3,651 Yara rules associated with 881 malicious families.

Fig. 5.15 shows an extracted Yara rule shared between *gozi*, *zenpak*, *fareit*, and *trickbot* malicious families. Notice that the rule consists of 20 strings and 3 op-code sequences. The strings combinarions can be further used to understand the shared common characteristics within malware

Table 5.10: A comparison between the original assigned malware families and the new assigned malware families as of January 2022. Notice that some families were renamed, such as revetrat and rrat, while other samples' assigned labels are completely different (*i.e.* offerinstall to appster).

Family		# Samples
Before	After	
offerinstall	appster	42
ursu	syncopate	35
gamarue	fareit	33
rrat	revetrat	32
econnect	dial	30
agensla	agenttesla	17
crysan	razy	17
vebzenpak	guloader	15
rescoms	remcos	13
razy	discord	10
Other Connections		2,280

Table 5.11: The most common strings among top five inter-family extracted Yara rules.

#	Associated Families	Strings		
		#1	#2	#3
1	deyma - zusy - ryuk - emotet	***CIrdaPort***	***.?AVCIrdaPort@@***	*AT+CPBR=%d*
2	vebzenpak - guloader - fareit	***TEMPORALNESSCRIT***	***DENTALIZEKERAT***	***RUMFANGET***
3	ligooc - trickbot - emotet	*2q6wcm0g1om3lEM...*	*jAjah@ D*	*J8mCzMpMfWj...*
4	fareit - genkryptik - ymacco	***bookshell***	***get_BookShell***	*W2JMFDRAQSR...*
5	atraps - zusy - cerbu	*2.3¡3C3M3[3i3n3y3*	*1a2f2l2r2{2*	*5)585¿5N5Z5e5t5*

families. In particular, we utilized yarAnalyzer [117] to analyze the generated rules and extract the most common strings and op-code sequences, shown in Table 5.11. In this table, the most common strings among top five inter-family extracted Yara rules are shown. Notice that while some strings are not human-readable, there exist several strings that are, and may indicate shared common functions and API calls, such as *"CIrdaPort"* and *"bookshell"*.

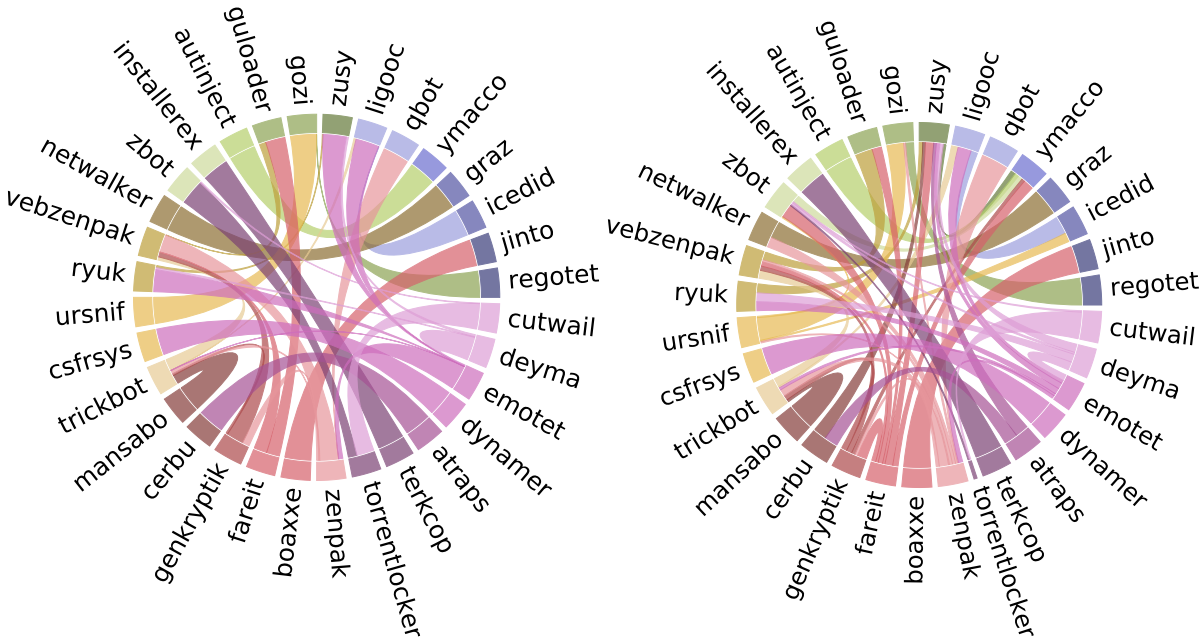
Extracting the inter-family shared patterns helps in understanding the evolution of malware capabilities over time. We recall that the extracted revival chains unveil hidden connections within malicious families (shown in Fig. 5.16a). In this figure, a connection between *jinto* and *boaxxe*

```
rule gozi_zenpak_fareit_trickbot{
   strings:
      $s1 = "<8tPaRcePNHebK" fullword ascii
      $s2 = "Ribh}ry" fullword ascii
      $s3 = "Oalv?mo" fullword ascii
      $s4 = "leafyFi" fullword ascii
      $s5 = "fUBz?oM(QN" fullword ascii
      $s6 = "OtrWEnA" fullword ascii
      $s7 = "odu~yHatzle" fullword ascii
      $s8 = "ndnUeHRGdlN" fullword ascii
      $s9 = "KEap2Nlo`\"" fullword ascii
      $s10 = "pRtg!*#" fullword ascii
      $s11 = "Vsltu{rAl~qc" fullword ascii
      $s12 = "W?FzbGF:v" fullword ascii
      $s13 = "Imdg}iN" fullword ascii
      $s14 = "GThQ8ad" fullword ascii
      $s15 = "xOEFjaWqiP" fullword wide
      $s16 = "\\h6|&+" fullword ascii
      $s17 = "@ewc=ur" fullword ascii
      $s18 = "{@d`0m" fullword ascii
      $s19 = "Inm;oVi?hOf" fullword ascii
      $s20 = "Qyf-btQ_o" fullword ascii
      $op0 = { 85 c9 74 45 8d 55 f0 52 8d 85 28 ff ff ff 50 8d }
      $op1 = { a1 48 db 4d 00 03 05 34 db 4d 00 89 45 fc 8b 0d }
      $op2 = { a1 14 db 4d 00 89 45 e8 c7 45 f8 ff ff ff ff 8b }
   condition:
      uint16(0) == 0x5a4d and ( 8 of them and all of ($op*) )
}
```

Figure 5.15: A sample Yara rule shared between *gozi*, *zenpak*, *fareit*, and *trickbot* malicious families.

malicious families indicate that they appeared within the same revival chain. To further investigate the shared patterns, we illustrate the families connections based on the extracted shared Yara rules, shown in Fig. 5.16b. Notice that we limit the results to the 32 families that appeared within the revival chains for readability purposes. While *zusy* malicious family only appears with *emotet* in the revival chains, we uncover its nine shared behavioral patterns with different malicious families.

Our observations leads to the question of *"whether malicious families share the same origin?"* Answering this question allows for better understanding their capabilities, and the family labeling



(a) Revival chains-based connections.

(b) Yara rules shared patterns-based connections.

Figure 5.16: A breakdown of the malicious families connections within the revival chains and extracted Yara shared patterns.

inconsistencies. In an attempt to answer this question, we combined the extracted information regarding shared Yara rules and *VirusTotal* relabeling inconsistency. In particular, we deem two malware families to share the same origin if (i) at least five malicious samples were labeled as both families across the studied period (*i.e.* by observing family assignment altering through time), and (ii) there is at least ten shared Yara rules among the samples of these two families. If both conditions applied, we create a connection between both malicious families. We note that this is a conservative approach that depends on both observation and malware analysis. Further, we obtained the malicious family first seen date from Microsoft Security Intelligence [53] to better understand the malicious families evolvement timeline. Fig. 5.17 shows the malicious families that fulfill the aforementioned conditions. Notice that similar to Table 5.10, multiple family renaming appear within the connections. Further, our analysis uncover a direction connection between *"swrort"*, *"rozena"*, and *"shelma"* malicious families, appearing in 2010, 2013, and 2018, respectively.

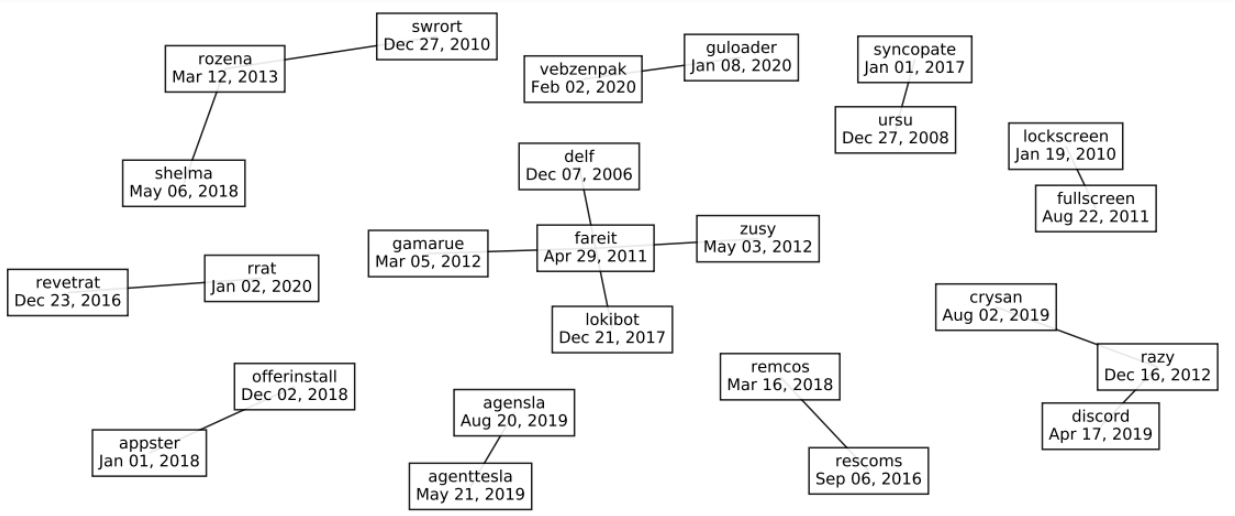


Figure 5.17: Shared malicious family origin analysis. Two malicious families are connected if (i) at least five malicious samples were labeled as both families across the studied period, and (ii) there is at least ten shared Yara rules among the samples of these two families.

The extracted connections highlight that either (i) one malicious family is an evolution of another malicious family, with common shared capabilities, or (ii) both families share the same origin (*i.e.* libraries, functions, and source codes). While this behavior is not surprising, as malicious code is built on top of existing malicious capabilities, it causes family labeling inconsistencies, that in turn hinders the machine learning capabilities in accurately extracting meaningful patterns that distinguish malicious families from one another.

Lessons Learnt in Malware Detection

Malware Detection Generalization. The evolving nature of the malware creates a need for generalizable machine learning-based malware detection models to unseen variants of malware. Although this problem is not limited to the malware detection domain, however, keeping in mind the ongoing arms race and its likely consequences, there is a significant need to address this challenge in malware detection.

Throughout our experiments, the machine learning-based detectors showed a lower malware de-

tection performance for unseen malware, even when the model is exposed to other samples of the same malicious family. Moreover, we also noticed that while both samples from *seen* and *unseen families* have reduced performance over time, the performance of *unseen families*'s samples is more significantly impacted. A major reason for the performance degradation in the *unseen families* is that their samples may not have shared patterns with malware in the training set.

Model Retraining. While model retraining is considered a popular solution to improve the accuracy of malware detection and to cope with mutations, however, our work shows that model retraining provisions minimal benefits. In fact, model retraining over an extended period of time reduces the performance of the model in terms of detecting benign samples. We ascribe the performance degradation on benign detection to the diverse nature of malware samples, whereby the continuous mutation affects the model training.

Emerging Malware Detection. Given the criticality of this threat, we realized that exposing the limitations of retraining process may not be sufficient, and efforts must be made towards refining the retraining process. Towards that larger goal, we showed that using OOD-based detection techniques can effectively detect new emerging families and estimating malware mutations. In particular, by using image-based representation, malware detectors can detect the new emerging families as out-of-distribution with 98.67% accuracy. We strengthen our proposition by providing two case studies, where we provide a conservative estimate of malware mutations by exposing 31 mutations of the "zbot" malware family in the period of 2008–2020, and showcase the capabilities of OOD-based approaches in detecting emerging families.

Old Malware Paradigm. To overcome the shortcomings of model retraining, malware detectors tend to discard old malware samples from the training phase, thereby resulting in a high malware detection performance, without impacting the benign accuracy. However, in the long term, the model gets trained on the malware samples that are fundamentally different from the primordial "old" malware. As a result, old malware is classified as benign, thereby opening up a new attack vector whereby old malware can be revived and respread to defeat the detection frameworks.

Throughout our evaluation, we show that, in contrast to the common belief that newer malware is harder to detect, malware that appeared in the period 2015–2017 has a lower detection rate by the online detection engines. This phenomenon appeared for almost half the malware captured before 2019, potentially causing a respreading of malware with minimal efforts (*e.g.* modifying the command and control server).

Summary & Concluding Remarks

In this work, we first exposed the limitations of using model retraining for malware detection. We replicated the behavior of detection engines by following their retraining approaches. Our results reveal that model retraining provides only marginal improvements in the detection of malicious samples. Moreover, a natural caveat of model retraining is the performance degradation of benign detection, which further leads to the possibility of malware respreading (without mutations). The existing approaches of model retraining do not provide robust protection in the ongoing arms race with malware authors. To address some of the limitations of existing retraining approaches, we leveraged a hybrid scheme that maximizes the utility of model retraining while also providing clear insights into the behavior of malware family mutations. We further unveil the existing family labeling inconsistency, with malware being assigned to different families over time. This is mainly due to the shared capabilities and functionalities within malicious samples, with multiple malware embracing the same open sources capabilities, or building on top of other existing malicious families. While our work addresses the model retraining overhead-to-performance trade-off, it opens several research directions that can be pursued to increase the efficacy of the existing malware detection frameworks.

CHAPTER 6: CONCLUSION & FUTURE DIRECTION

This dissertation analyzes machine learning-based malware detection systems, including the detection and mitigation of adversarial malware examples. Machine learning for malware detection unleashed the capabilities of malware detection toward zero-day attack detection, achieving state-of-the-art performance. This allows for better scalability, in comparison with heuristic approaches. However, these advancements were at the cost of discovering new attack channels and vulnerabilities that may render the detection engine useless. In this dissertation, we highlight these challenges, pinpointing the points of failures, and propose defenses toward designing robust malware detection. In particular, we investigated the attack surface of malware detection systems, shedding light on the vulnerability of the underlying learning algorithms and industry-standard machine learning malware detection systems against adversaries in both IoT and Windows environments. Toward robust malware detection, we investigated software pre-processing and monotonic machine learning, achieving robust malware detection at a low performance cost. In addition, we examined potential exploitation caused by actively retraining malware detection models. We uncovered an unreported benign to malware performance trade-off, causing the malware to revive and be classified as a benign or different malicious family. This behavior leads to family labeling inconsistencies, hindering the efforts toward the understanding of malicious families.

This dissertation builds toward designing robust malware detection, by analyzing and detecting adversarial examples. This work highlights the vulnerability of industry-standard applications to black-box adversarial settings, including the continuous evolution of malware over time. Our findings highlight common practices that can be exploited toward misclassification, including the usage of volatile features for detection, or over-fitting on benign patterns to boost the validation accuracy. We further highlight that the commonly assumed uniqueness of each malicious family only hinders the detector capabilities. Instead, malware detectors should adapt addition attacks resilient learning process for malicious patterns recognition. In the future, we would like to explore and study multiple directions regarding the origin and shared behaviors of malware. Understanding

malware behavior temporal shifting helps in identifying mutations and new threats, and estimating the future adversarial capabilities. This, however, can be only achieved by examining the code base of the malware, understanding the trends and shifts in their functionalities over time.

APPENDIX: PUBLICATIONS COPYRIGHT

IEEE COPYRIGHT FORM

To ensure uniformity of treatment among all contributors, other forms may not be substituted for this form, nor may any wording of the form be changed. This form is intended for original material submitted to the IEEE and must accompany any such material in order to be published by the IEEE. Please read the form carefully and keep a copy for your files.

DL-FHMC: Deep Learning-based Fine-grained Hierarchical Learning Approach for Robust Malware Classification
Abusnaina, Ahmed ; Abuhamad, Mohammed; Alasmary, Hisham ; Anwar, Afsah; Jang, Rhongho; Salem, Saeed; Nyang, DaeHun; Mohaisen, David
Transactions on Dependable and Secure Computing

COPYRIGHT TRANSFER

The undersigned hereby assigns to The Institute of Electrical and Electronics Engineers, Incorporated (the "IEEE") all rights under copyright that may exist in and to: (a) the Work, including any revised or expanded derivative works submitted to the IEEE by the undersigned based on the Work; and (b) any associated written or multimedia components or other enhancements accompanying the Work.

GENERAL TERMS

1. The undersigned represents that he/she has the power and authority to make and execute this form.
2. The undersigned agrees to indemnify and hold harmless the IEEE from any damage or expense that may arise in the event of a breach of any of the warranties set forth above.
3. The undersigned agrees that publication with IEEE is subject to the policies and procedures of the IEEE PSPB Operations Manual.
4. In the event the above work is not accepted and published by the IEEE or is withdrawn by the author(s) before acceptance by the IEEE, the foregoing copyright transfer shall be null and void. In this case, IEEE will retain a copy of the manuscript for internal administrative/record-keeping purposes.
5. For jointly authored Works, all joint authors should sign, or one of the authors should sign as authorized agent for the others.
6. The author hereby warrants that the Work and Presentation (collectively, the "Materials") are original and that he/she is the author of the Materials. To the extent the Materials incorporate text passages, figures, data or other material from the works of others, the author has obtained any necessary permissions. Where necessary, the author has obtained all third party permissions and consents to grant the license above and has provided copies of such permissions and consents to IEEE

BY TYPING IN YOUR FULL NAME BELOW AND CLICKING THE SUBMIT BUTTON, YOU CERTIFY THAT SUCH ACTION CONSTITUTES YOUR ELECTRONIC SIGNATURE TO THIS FORM IN ACCORDANCE WITH UNITED STATES LAW, WHICH AUTHORIZES ELECTRONIC SIGNATURE BY AUTHENTICATED REQUEST FROM A USER OVER THE INTERNET AS A VALID SUBSTITUTE FOR A WRITTEN SIGNATURE.

David Mohaisen	12-07-2021
Signature	**Date (dd-mm-yyyy)**

Information for Authors

AUTHOR RESPONSIBILITIES

The IEEE distributes its technical publications throughout the world and wants to ensure that the material submitted to its publications is properly available to the readership of those publications. Authors must ensure that their Work meets the

requirements as stated in section 8.2.1 of the IEEE PSPB Operations Manual, including provisions covering originality, authorship, author responsibilities and author misconduct. More information on IEEE's publishing policies may be found at http://www.ieee.org/publications_standards/publications/rights/authorrightsresponsibilities.html Authors are advised especially of IEEE PSPB Operations Manual section 8.2.1.B12: "It is the responsibility of the authors, not the IEEE, to determine whether disclosure of their material requires the prior consent of other parties and, if so, to obtain it." Authors are also advised of IEEE PSPB Operations Manual section 8.1.1B: "Statements and opinions given in work published by the IEEE are the expression of the authors."

RETAINED RIGHTS/TERMS AND CONDITIONS

- Authors/employers retain all proprietary rights in any process, procedure, or article of manufacture described in the Work.
- Authors/employers may reproduce or authorize others to reproduce the Work, material extracted verbatim from the Work, or derivative works for the author's personal use or for company use, provided that the source and the IEEE copyright notice are indicated, the copies are not used in any way that implies IEEE endorsement of a product or service of any employer, and the copies themselves are not offered for sale.
- Although authors are permitted to re-use all or portions of the Work in other works, this does not include granting third-party requests for reprinting, republishing, or other types of re-use.The IEEE Intellectual Property Rights office must handle all such third-party requests.
- Authors whose work was performed under a grant from a government funding agency are free to fulfill any deposit mandates from that funding agency.

AUTHOR ONLINE USE

- **Personal Servers**. Authors and/or their employers shall have the right to post the accepted version of IEEE-copyrighted articles on their own personal servers or the servers of their institutions or employers without permission from IEEE, provided that the posted version includes a prominently displayed IEEE copyright notice and, when published, a full citation to the original IEEE publication, including a link to the article abstract in IEEE Xplore. Authors shall not post the final, published versions of their papers.
- **Classroom or Internal Training Use.** An author is expressly permitted to post any portion of the accepted version of his/her own IEEE-copyrighted articles on the author's personal web site or the servers of the author's institution or company in connection with the author's teaching, training, or work responsibilities, provided that the appropriate copyright, credit, and reuse notices appear prominently with the posted material. Examples of permitted uses are lecture materials, course packs, e-reserves, conference presentations, or in-house training courses.
- **Electronic Preprints.** Before submitting an article to an IEEE publication, authors frequently post their manuscripts to their own web site, their employer's site, or to another server that invites constructive comment from colleagues. Upon submission of an article to IEEE, an author is required to transfer copyright in the article to IEEE, and the author must update any previously posted version of the article with a prominently displayed IEEE copyright notice. Upon publication of an article by the IEEE, the author must replace any previously posted electronic versions of the article with either (1) the full citation to the IEEE work with a Digital Object Identifier (DOI) or link to the article abstract in IEEE Xplore, or (2) the accepted version only (not the IEEE-published version), including the IEEE copyright notice and full citation, with a link to the final, published article in IEEE Xplore.

Questions about the submission of the form or manuscript must be sent to the publication's editor.
Please direct all questions about IEEE copyright policy to:
IEEE Intellectual Property Rights Office, copyrights@ieee.org, +1-732-562-3966

IEEE COPYRIGHT AND CONSENT FORM

To ensure uniformity of treatment among all contributors, other forms may not be substituted for this form, nor may any wording of the form be changed. This form is intended for original material submitted to the IEEE and must accompany any such material in order to be published by the IEEE. Please read the form carefully and keep a copy for your files.

Adversarial Learning Attacks on Graph-based IoT Malware Detection Systems
Ahmed Abusnaina, Aminollah Khormali, Hisham Alasmary, Jeman Park, Afsah Anwar, Aziz Mohaisen
2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)

COPYRIGHT TRANSFER

The undersigned hereby assigns to The Institute of Electrical and Electronics Engineers, Incorporated (the "IEEE") all rights under copyright that may exist in and to: (a) the Work, including any revised or expanded derivative works submitted to the IEEE by the undersigned based on the Work; and (b) any associated written or multimedia components or other enhancements accompanying the Work.

GENERAL TERMS

1. The undersigned represents that he/she has the power and authority to make and execute this form.
2. The undersigned agrees to indemnify and hold harmless the IEEE from any damage or expense that may arise in the event of a breach of any of the warranties set forth above.
3. The undersigned agrees that publication with IEEE is subject to the policies and procedures of the IEEE PSPB Operations Manual.
4. In the event the above work is not accepted and published by the IEEE or is withdrawn by the author(s) before acceptance by the IEEE, the foregoing copyright transfer shall be null and void. In this case, IEEE will retain a copy of the manuscript for internal administrative/record-keeping purposes.
5. For jointly authored Works, all joint authors should sign, or one of the authors should sign as authorized agent for the others.
6. The author hereby warrants that the Work and Presentation (collectively, the "Materials") are original and that he/she is the author of the Materials. To the extent the Materials incorporate text passages, figures, data or other material from the works of others, the author has obtained any necessary permissions. Where necessary, the author has obtained all third party permissions and consents to grant the license above and has provided copies of such permissions and consents to IEEE

You have indicated that you DO wish to have video/audio recordings made of your conference presentation under terms and conditions set forth in "Consent and Release."

CONSENT AND RELEASE

1. In the event the author makes a presentation based upon the Work at a conference hosted or sponsored in whole or in part by the IEEE, the author, in consideration for his/her participation in the conference, hereby grants the IEEE the unlimited, worldwide, irrevocable permission to use, distribute, publish, license, exhibit, record, digitize, broadcast, reproduce and archive, in any format or medium, whether now known or hereafter developed: (a) his/her presentation and comments at the conference; (b) any written materials or multimedia files used in connection with his/her presentation; and (c) any recorded interviews of him/her (collectively, the "Presentation"). The permission granted includes the transcription and reproduction of the Presentation for inclusion in products sold or distributed by IEEE and live or recorded broadcast of the Presentation during or after the conference.
2. In connection with the permission granted in Section 1, the author hereby grants IEEE the unlimited, worldwide, irrevocable right to use his/her name, picture, likeness, voice and biographical information as part of the advertisement, distribution and sale of products incorporating the Work or Presentation, and releases IEEE from any claim based on right of privacy or publicity.

BY TYPING IN YOUR FULL NAME BELOW AND CLICKING THE SUBMIT BUTTON, YOU CERTIFY THAT SUCH ACTION CONSTITUTES YOUR ELECTRONIC SIGNATURE TO THIS FORM IN ACCORDANCE WITH UNITED STATES LAW, WHICH AUTHORIZES ELECTRONIC SIGNATURE BY AUTHENTICATED REQUEST FROM A USER OVER THE INTERNET AS A VALID SUBSTITUTE FOR A WRITTEN SIGNATURE.

Ahmed Abusnaina	15-04-2019
Signature	**Date (dd-mm-yyyy)**

Information for Authors

AUTHOR RESPONSIBILITIES

The IEEE distributes its technical publications throughout the world and wants to ensure that the material submitted to its publications is properly available to the readership of those publications. Authors must ensure that their Work meets the requirements as stated in section 8.2.1 of the IEEE PSPB Operations Manual, including provisions covering originality, authorship, author responsibilities and author misconduct. More information on IEEE's publishing policies may be found at http://www.ieee.org/publications_standards/publications/rights/authorrightsresponsibilities.html Authors are advised especially of IEEE PSPB Operations Manual section 8.2.1.B12: "It is the responsibility of the authors, not the IEEE, to determine whether disclosure of their material requires the prior consent of other parties and, if so, to obtain it." Authors are also advised of IEEE PSPB Operations Manual section 8.1.1B: "Statements and opinions given in work published by the IEEE are the expression of the authors."

RETAINED RIGHTS/TERMS AND CONDITIONS

- Authors/employers retain all proprietary rights in any process, procedure, or article of manufacture described in the Work.
- Authors/employers may reproduce or authorize others to reproduce the Work, material extracted verbatim from the Work, or derivative works for the author's personal use or for company use, provided that the source and the IEEE copyright notice are indicated, the copies are not used in any way that implies IEEE endorsement of a product or service of any employer, and the copies themselves are not offered for sale.
- Although authors are permitted to re-use all or portions of the Work in other works, this does not include granting third-party requests for reprinting, republishing, or other types of re-use.The IEEE Intellectual Property Rights office must handle all such third-party requests.
- Authors whose work was performed under a grant from a government funding agency are free to fulfill any deposit mandates from that funding agency.

AUTHOR ONLINE USE

- **Personal Servers**. Authors and/or their employers shall have the right to post the accepted version of IEEE-copyrighted articles on their own personal servers or the servers of their institutions or employers without permission from IEEE, provided that the posted version includes a prominently displayed IEEE copyright notice and, when published, a full citation to the original IEEE publication, including a link to the article abstract in IEEE Xplore. Authors shall not post the final, published versions of their papers.
- **Classroom or Internal Training Use.** An author is expressly permitted to post any portion of the accepted version of his/her own IEEE-copyrighted articles on the author's personal web site or the servers of the author's institution or company in connection with the author's teaching, training, or work responsibilities, provided that the appropriate copyright, credit, and reuse notices appear prominently with the posted material. Examples of permitted uses are lecture materials, course packs, e-reserves, conference presentations, or in-house training courses.
- **Electronic Preprints.** Before submitting an article to an IEEE publication, authors frequently post their manuscripts to their own web site, their employer's site, or to another server that invites constructive comment from colleagues. Upon submission of an article to IEEE, an author is required to transfer copyright in the article to IEEE, and the author must update any previously posted version of the article with a prominently displayed IEEE copyright notice. Upon publication of an article by the IEEE, the author must replace any previously posted electronic versions of the article with either (1) the full citation to the

IEEE work with a Digital Object Identifier (DOI) or link to the article abstract in IEEE Xplore, or (2) the accepted version only (not the IEEE-published version), including the IEEE copyright notice and full citation, with a link to the final, published article in IEEE Xplore.

Questions about the submission of the form or manuscript must be sent to the publication's editor.
Please direct all questions about IEEE copyright policy to:
IEEE Intellectual Property Rights Office, copyrights@ieee.org, +1-732-562-3966

LIST OF REFERENCES

[1] Strip: GNU binary utility.

[2] Ucl data compression library.

[3] UPX: the Ultimate Packer for eXecutables.

[4] VirusShare.

[5] Panda security: 20% of all malware ever created appeared in 2013. Available at [Online]: `https://bit.ly/3aEVxOt`, 2014.

[6] Cyberiocs. Available at [Online]: `https://freeiocs.cyberiocs.pro/`, 2019.

[7] Radare2. Available at [Online]: `https://https://rada.re/r/`, 2019.

[8] VirusTotal. Available at [Online]: `https://www.virustotal.com`, 2019.

[9] Av-test: Malware statistics & trends report. Available at [Online]: `https://www.av-test.org/en/statistics/malware/`, 2021.

[10] VirusTotal Statistics. Available at [Online]: `https://www.virustotal.com/en/statistics/`, 2021.

[11] Yara rules, October 2021.

[12] A. Abusnaina, H. Alasmary, M. Abuhamad, S. Salem, D. Nyang, and A. Mohaisen. Subgraph-based adversarial examples against graph-based iot malware detection systems. In *International Conference on Computational Data and Social Networks*, pages 268–281, 2019.

[13] A. Abusnaina, A. Khormali, H. Alasmary, J. Park, A. Anwar, and A. Mohaisen. Adversarial learning attacks on graph-based IoT malware detection systems. In *Proceedings of the 39th IEEE International Conference on Distributed Computing Systems, ICDCS*, 2019.

[14] A. Abusnaina, A. Khormali, H. Alasmary, J. Park, A. Anwar, and A. Mohaisen. Adversarial learning attacks on graph-based IoT malware detection systems. In *IEEE International Conference on Distributed Computing Systems, ICDCS*, 2019.

[15] A. Abusnaina, D. Nyang, M. Yuksel, and A. Mohaisen. Examining the security of ddos detection systems in software defined networks. In *Proceedings of the 15th International Conference on emerging Networking EXperiments and Technologies*, pages 49–50, 2019.

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

[17] M. Ahmadi, D. Ulyanov, S. Semenov, M. Trofimov, and G. Giacinto. Novel feature extraction, selection and fusion for effective malware family classification. In *Proceedings of ACM conference on data and application security and privacy*, pages 183–194, 2016.

[18] A. Al-Dujaili, A. Huang, E. Hemberg, and U. O'Reilly. Adversarial deep learning for robust detection of binary encoded malware. In *Proceedings of the IEEE Security and Privacy Workshops, SP Workshops*, pages 76–82, 2018.

[19] S. Alam, R. N. Horspool, I. Traoré, and I. Sogukpinar. A framework for metamorphic malware analysis and real-time detection. *Computers & Security*, 48:212–233, 2015.

[20] H. Alasmary, A. Abusnaina, R. Jang, M. Abuhamad, A. Anwar, D. Nyang, and D. Mohaisen. Soteria: Detecting adversarial examples in control flow graph-based malware classifier. In *40th IEEE International Conference on Distributed Computing Systems, ICDCS*, pages 1296–1305, 2020.

[21] H. Alasmary, A. Anwar, J. Park, J. Choi, D. Nyang, and A. Mohaisen. Graph-based comparison of IoT and android malware. In *Proceeding of the 7th International Conference on Computational Data and Social Networks, CSoNet*, pages 259–272, 2018.

[22] H. Alasmary, A. Khormali, A. Anwar, J. Park, J. Choi, A. Abusnaina, A. Awad, D. Nyang, and A. Mohaisen. Analyzing and Detecting Emerging Internet of Things Malware: A Graph-based Approach. *IEEE Internet of Things Journal*, 2019.

[23] H. Alasmary, A. Khormali, A. Anwar, J. Park, J. Choi, A. Abusnaina, A. Awad, D. Nyang, and A. Mohaisen. Analyzing and detecting emerging internet of things malware: a graph-based approach. *IEEE Internet of Things Journal*, 6(5):8977–8988, 2019.

[24] K. Allix, T. F. Bissyandé, Q. Jérome, J. Klein, R. State, and Y. L. Traon. Empirical assessment of machine learning-based malware detectors for android - measuring the gap between in-the-lab and in-the-wild validation scenarios. *Empirical Software Engineering*, 21(1):183–211, 2016.

[25] O. Alrawi, C. Lever, K. Valakuzhy, R. Court, K. Snow, F. Monrose, and M. Antonakakis. The circle of life: A large-scale study of the iot malware lifecycle. In *USENIX Security Symposium (USENIX Security 21)*, 2021.

[26] O. Alrawi, C. Lever, K. Valakuzhy, K. Snow, F. Monrose, M. Antonakakis, et al. The circle of life: A large-scale study of the iot malware lifecycle. In *USENIX Security Symposium (USENIX Security 21)*, 2021.

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

[28] M. Antonakakis, R. Perdisci, Y. Nadji, N. Vasiloglou, S. Abu-Nimeh, W. Lee, and D. Dagon. From throw-away traffic to bots: Detecting the rise of DGA-based malware. In *USENIX Security*, pages 491–506, 2012.

[29] A. Anwar, H. Alasmary, J. Park, A. Wang, S. Chen, and D. Mohaisen. Statically dissecting internet of things malware: Analysis, characterization, and detection. In *International Conference on Information and Communications Security*, pages 443–461. Springer, 2020.

[30] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck. DREBIN: effective and explainable detection of android malware in your pocket. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium, NDSS*, 2014.

[31] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *Ndss*, volume 14, pages 23–26, 2014.

[32] A. Azmoodeh, A. Dehghantanha, and K.-K. R. Choo. Robust malware detection for Internet Of (Battlefield) Things devices using deep eigenspace learning. *IEEE Transactions on Sustainable Computing*, 4(1):88–95, 2019.

[33] F. Barr-Smith, X. Ugarte-Pedrero, M. Graziano, R. Spolaor, and I. Martinovic. Survivalism: Systematic analysis of windows malware living-off-the-land. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2021.

[34] U. Bayer, I. Habibi, D. Balzarotti, E. Kirda, and C. Kruegel. A view on current malware behaviors. In *LEET*, 2009.

[35] L. Bilge and T. Dumitraş. Before we knew it: an empirical study of zero-day attacks in the real world. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 833–844, 2012.

[36] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander. Lof: identifying density-based local outliers. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 93–104, 2000.

[37] D. Bruschi, L. Martignoni, and M. Monga. Detecting self-mutating malware using control-flow graph matching. In *International conference on detection of intrusions and malware, and vulnerability assessment*, pages 129–143. Springer, 2006.

[38] N. Carlini and D. A. Wagner. Adversarial examples are not easily detected: Bypassing ten detection methods. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security, AISec@CCS*, pages 3–14, 2017.

[39] N. Carlini and D. A. Wagner. Towards evaluating the robustness of neural networks. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 39–57, 2017.

[40] N. Carlini and D. A. Wagner. Towards evaluating the robustness of neural networks. In *IEEE Symposium on Security and Privacy, SP*, pages 39–57, 2017.

[41] D. Caselden, A. Bazhanyuk, M. Payer, S. McCamant, and D. Song. HI-CFG: construction by binary analysis and application to attack polymorphism. In *18th European Symposium on Research in Computer Security*, pages 164–181, 2013.

[42] L. Cen, C. S. Gates, L. Si, and N. Li. A probabilistic discriminative model for android malware detection with decompiled source code. *IEEE Transaction on Dependable and Secure Computing*, 12(4):400–412, 2015.

[43] T. Chakraborty, F. Pierazzi, and V. Subrahmanian. Ec2: Ensemble clustering and classification for predicting android malware families. *IEEE Transactions on Dependable and Secure Computing*, 17(2):262–277, 2017.

[44] X. Chen, C. Li, D. Wang, S. Wen, J. Zhang, S. Nepal, Y. Xiang, and K. Ren. Android hiv: A study of repackaging malware for evading machine-learning detection. *IEEE Transactions on Information Forensics and Security*, 15:987–1001, 2019.

[45] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (Sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10):1367–1372, 2004.

[46] E. Cozzi, M. Graziano, Y. Fratantonio, and D. Balzarotti. Understanding Linux malware. In *IEEE Symposium on Security & Privacy*, 2018.

[47] J. R. Crandall, G. Wassermann, D. A. S. de Oliveira, Z. Su, S. F. Wu, and F. T. Chong. Temporal search: detecting hidden malware timebombs with virtual machines. In J. P. Shen and M. Martonosi, editors, *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, pages 25–36. ACM, 2006.

[48] Z. Cui, F. Xue, X. Cai, Y. Cao, G. Wang, and J. Chen. Detection of malicious code variants based on deep learning. *Trans. Industrial Informatics*, 14(7):3187–3196, 2018.

[49] L. Demetrio, S. E. Coull, B. Biggio, G. Lagorio, A. Armando, and F. Roli. Adversarial EXEmples: A survey and experimental evaluation of practical attacks on machine learning for windows malware detection. *CoRR*, abs/2008.07125, 2020.

[50] A. Demontis, M. Melis, B. Biggio, D. Maiorca, D. Arp, K. Rieck, I. Corona, G. Giacinto, and F. Roli. Yes, machine learning can be more secure! A case study on android malware detection. *IEEE Transaction on Dependable and Secure Computing*, 16(4):711–724, 2019.

[51] A. Deo, S. K. Dash, G. Suarez-Tangil, V. Vovk, and L. Cavallaro. Prescience: Probabilistic guidance on the retraining conundrum for malware detection. In *Proceedings of the 2016 ACM Workshop on Artificial Intelligence and Security*, pages 71–82, 2016.

[52] Developers. Github. Available at [Online]: `https://github.com/`, 2019.

[53] Developers. Microsoft security intelligence, 2022.

[54] A. K. *et al.*. Adversarial machine learning at scale. In *5th International Conference on Learning Representations, ICLR 2017*, 2017.

[55] A. M. N. *et al.*. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, 2015.

[56] A. V. *et al.*. Deep learning for computer vision: A brief review. *Comput. Intell. Neurosci.*, 2018:7068349:1–7068349:13, 2018.

[57] D. B. *et al.*. Neural machine translation by jointly learning to align and translate. In *3rd International Conference on Learning Representations, ICLR*, 2015.

[58] F. S. *et al.*. Facenet: A unified embedding for face recognition and clustering. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, pages 815–823. IEEE Computer Society, 2015.

[59] G. H. *et al.*. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal processing magazine*, 29(6):82–97, 2012.

[60] I. J. G. *et al.*. Explaining and harnessing adversarial examples. In *3rd International Conference on Learning Representations, ICLR*, 2015.

[61] K. S. *et al.*. Very deep convolutional networks for large-scale image recognition. In *3rd International Conference on Learning Representations, ICLR*, 2015.

[62] L. E. *et al.*. A rotation and a translation suffice: Fooling cnns with simple transformations. *CoRR*, 2017.

[63] W. Z. *et al.*. Towards end-to-end speech recognition with deep multipath convolutional neural networks. In *12th International Conference on Intelligent Robotics and Applications ICIRA*, 2019.

[64] L. P. G. Evans, N. M. Adams, and C. Anagnostopoulos. Estimating optimal active learning via model retraining improvement. *CoRR*, abs/1502.01664, 2015.

[65] J. Fu, J. Xue, Y. Wang, Z. Liu, and C. Shan. Malware visualization for fine-grained classification. *IEEE Access*, 6:14510–14523, 2018.

[66] A. Gerber. Connecting all the things in the Internet of Things. Available at [Online]: `https://ibm.co/2qMx97a`, 2018.

[67] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. In *International Conference on Learning Representations, ICLR*, 2015.

[68] M. Graziano, D. Canali, L. Bilge, A. Lanzi, and D. Balzarotti. Needles in a haystack: Mining information from public dynamic analysis sandboxes for malware intelligence. In *24th USENIX Security Symposium (USENIX Security 15)*, 2015.

[69] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel. Adversarial examples for malware detection. In *European Symposium on Research in Computer Security*, pages 62–79. Springer, 2017.

[70] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. D. McDaniel. Adversarial examples for malware detection. In *Proceedings of the 22nd European Symposium on Research Computer Security - ESORICS, Part II*, pages 62–79, 2017.

[71] C. Guo, M. Rana, M. Cisse, and L. van der Maaten. Countering adversarial images using input transformations. In *International Conference on Learning Representations, ICLR*, 2018.

[72] H. HaddadPajouh, A. Dehghantanha, R. Khayami, and K.-K. R. Choo. A deep recurrent neural network based approach for internet of things malware threat hunting. *Future Generation Computer Systems*, 85:88–96, 2018.

[73] D. Hendrycks and T. G. Dietterich. Benchmarking neural network robustness to common corruptions and surface variations. *arXiv preprint arXiv:1807.01697*, 2018.

[74] S. Hu, T. Yu, C. Guo, W.-L. Chao, and K. Q. Weinberger. A new defense against adversarial images: Turning a weakness into a strength. In *Neural Information Processing Systems, NeurIPS*, 2019.

[75] W. Hu and Y. Tan. Generating adversarial malware examples for black-box attacks based on GAN. *arXiv preprint arXiv:1702.05983*, abs/1702.05983, 2017.

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

[77] P. Jalote. *An integrated approach to software engineering*. Springer Science & Business Media, 2012.

[78] Y. J. Jia, Q. A. Chen, S. Wang, A. Rahmati, E. Fernandes, Z. M. Mao, and A. Prakash. ContexIoT: Towards providing contextual integrity to appified IoT platforms. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium, NDSS*, pages 1–15, 2017.

[79] R. Jordaney, K. Sharad, S. K. Dash, Z. Wang, D. Papini, I. Nouretdinov, and L. Cavallaro. Transcend: Detecting concept drift in malware classification models. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 625–642, 2017.

[80] K. Kancherla and S. Mukkamala. Image visualization based malware detection. In *IEEE Symposium on Computational Intelligence in Cyber Security (CICS)*, pages 40–44. IEEE, 2013.

[81] H. Kang, J.-w. Jang, A. Mohaisen, and H. K. Kim. Detecting and classifying android malware using static analysis along with creator information. *International Journal of Distributed Sensor Networks*, 11(6):479174, 2015.

[82] A. Kantchelian, S. Afroz, L. Huang, A. C. Islam, B. Miller, M. C. Tschantz, R. Greenstadt, A. D. Joseph, and J. Tygar. Approaches to adversarial drift. In *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, pages 99–110, 2013.

[83] A. Kantchelian, S. Afroz, L. Huang, A. C. Islam, B. Miller, M. C. Tschantz, R. Greenstadt, A. D. Joseph, and J. D. Tygar. Approaches to adversarial drift. In A. Sadeghi, B. Nelson, C. Dimitrakakis, and E. Shi, editors, *AISec'13, Proceedings of the 2013 ACM Workshop on*

Artificial Intelligence and Security, Co-located with CCS 2013, Berlin, Germany, November 4, 2013, pages 99–110. ACM, 2013.

[84] Y. Kim. Convolutional neural networks for sentence classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP*, pages 1746–1751. ACL, 2014.

[85] B. Kolosnjaji, A. Demontis, B. Biggio, D. Maiorca, G. Giacinto, C. Eckert, and F. Roli. Adversarial malware binaries: Evading deep learning for malware detection in executables. In *The European Signal Processing Conference, EUSIPCO*, pages 533–537, 2018.

[86] F. Kreuk, A. Barak, S. Aviv-Reuven, M. Baruch, B. Pinkas, and J. Keshet. Deceiving end-to-end deep learning malware detectors using adversarial examples. In *Workshop on Security in Machine Learning (NIPS)*, 2018.

[87] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of 26th Annual Conference on Neural Information Processing Systems NIPS*, pages 1106–1114, 2012.

[88] A. Kurakin, I. J. Goodfellow, and S. Bengio. Adversarial examples in the physical world. In *the 5th International Conference on Learning Representations, ICLR*, 2017.

[89] D. Lee, I. S. Song, K. J. Kim, and J.-h. Jeong. A study on malicious codes pattern analysis using visualization. In *Proceedings of the International Conference on Information Science and Applications (ICISA)*, pages 1–5, 2011.

[90] B. Li, K. A. Roundy, C. S. Gates, and Y. Vorobeychik. Large-scale identification of malicious singleton files. In G. Ahn, A. Pretschner, and G. Ghinita, editors, *Proceedings of the Seventh ACM Conference on Data and Application Security and Privacy, CODASPY*, pages 227–238. ACM, 2017.

[91] F. Li and V. Paxson. A large-scale empirical study of security patches. In *the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2201–2215, 2017.

[92] X. Li and F. Li. Adversarial examples detection in deep networks with convolutional filter statistics. In *IEEE International Conference on Computer Vision, ICCV*, pages 5775–5783, 2017.

[93] M. Lindorfer, A. Di Federico, F. Maggi, P. M. Comparetti, and S. Zanero. Lines of malicious code: Insights into the malicious software industry. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 349–358, 2012.

[94] F. Maggi, W. K. Robertson, C. Krügel, and G. Vigna. Protecting a moving target: Addressing web application concept drift. In E. Kirda, S. Jha, and D. Balzarotti, editors, *Recent Advances in Intrusion Detection, 12th International Symposium, RAID 2009, Saint-Malo, France, September 23-25, 2009. Proceedings*, volume 5758 of *Lecture Notes in Computer Science*, pages 21–40. Springer, 2009.

[95] D. Mahajan, R. B. Girshick, V. Ramanathan, K. He, M. Paluri, Y. Li, A. Bharambe, and L. van der Maaten. Exploring the limits of weakly supervised pretraining. In V. Ferrari, M. Hebert, C. Sminchisescu, and Y. Weiss, editors, *Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part II*, volume 11206 of *Lecture Notes in Computer Science*, pages 185–201. Springer, 2018.

[96] A. Makandar and A. Patrot. Malware class recognition using image processing techniques. In *Proceedings of the 2017 International Conference on Data Management, Analytics and Innovation (ICDMAI)*, pages 76–80, 2017.

[97] F. Mercaldo and A. Santone. Deep learning for image-based mobile malware detection. *Journal of Computer Virology and Hacking Techniques*, pages 1–15, 2020.

[98] J. H. Metzen, T. Genewein, V. Fischer, and B. Bischoff. On detecting adversarial perturbations. In *the 5th International Conference on Learning Representations, ICLR*, 2017.

[99] T. Miyato, S.-i. Maeda, M. Koyama, K. Nakae, and S. Ishii. Distributional smoothing with virtual adversarial training. In *International Conference on Learning Representations.*, pages 1–12, 2016.

[100] A. Mohaisen and O. Alrawi. Unveiling Zeus: automated classification of malware samples. In *the 22nd International World Wide Web Conference, WWW*, pages 829–832, 2013.

[101] A. Mohaisen and O. Alrawi. AV-Meter: An evaluation of antivirus scans and labels. In *Detection of Intrusions and Malware, and Vulnerability Assessment, DIMVA*, pages 112–131, 2014.

[102] A. Mohaisen, O. Alrawi, and M. Mohaisen. AMAL: high-fidelity, behavior-based automated malware analysis and classification. *Computers & Security*, 52:251–266, 2015.

[103] S. Moosavi-Dezfooli, A. Fawzi, and P. Frossard. DeepFool: A simple and accurate method to fool deep neural networks. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 2574–2582, 2016.

[104] L. Nataraj, S. Karthikeyan, G. Jacob, and B. Manjunath. Malware images: visualization and automatic classification. In *Proceedings of the 8th international symposium on visualization for cyber security*, page 4, 2011.

[105] S. Ni, Q. Qian, and R. Zhang. Malware identification using visualization images and deep learning. *Computers & Security*, 77:871–885, 2018.

[106] R. E. O'Neill. *Learning Linux Binary Analysis*. Packt Publishing, 2016.

[107] L. Onwuzurike, E. Mariconti, P. Andriotis, E. D. Cristofaro, G. Ross, and G. Stringhini. Mamadroid: Detecting android malware by building markov chains of behavioral models (extended version). *ACM Transactions on Privacy and Security (TOPS)*, 22(2):1–34, 2019.

[108] Y. M. P. Pa, S. Suzuki, K. Yoshioka, T. Matsumoto, T. Kasama, and C. Rossow. IoTPOT: A novel honeypot for revealing current IoT threats. *Journal of Information Processing*, 24:522–533, 2016.

[109] H. H. Pajouh, A. Dehghantanha, R. Khayami, and K. R. Choo. A deep recurrent neural network based approach for internet of things malware threat hunting. *Future Gener. Comput. Syst.*, 85:88–96, 2018.

[110] N. Papernot, P. D. McDaniel, I. J. Goodfellow, S. Jha, Z. B. Celik, and A. Swami. Practical black-box attacks against machine learning. In *Proceedings of the ACM on Asia Conference on Computer and Communications Security, AsiaCCS*, pages 506–519, 2017.

[111] N. Papernot, P. D. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami. The limitations of deep learning in adversarial settings. In *IEEE European Symposium on Security and Privacy*, pages 372–387, 2016.

[112] D. Park and B. Yener. A survey on practical adversarial examples for malware classifiers. *arXiv preprint arXiv:2011.05973*, 2020.

[113] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro. TESSERACT: eliminating experimental bias in malware classification across space and time. In N. Heninger and P. Traynor, editors, *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 729–746. USENIX Association, 2019.

[114] J. Qiu, J. Zhang, W. Luo, L. Pan, S. Nepal, and Y. Xiang. A survey of android malware detection with deep neural models. *ACM Computing Surveys (CSUR)*, 53(6):1–36, 2020.

[115] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. K. Nicholas. Malware detection by eating a whole EXE. In *The Workshops of the The Thirty-Second AAAI Conference on Artificial Intelligence, New Orleans, Louisiana, USA, February 2-7, 2018*, volume WS-18 of *AAAI Workshops*, pages 268–276. AAAI Press, 2018.

[116] E. Raff, J. Sylvester, and C. Nicholas. Learning the PE header, malware detection with minimal domain knowledge. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security, AISec@CCS 2017*, pages 121–132. ACM, 2017.

[117] F. Roth. yaranalyzer, 2015.

[118] F. Roth. yargen, 2015.

[119] A. Saracino, D. Sgandurra, G. Dini, and F. Martinelli. MADAM: effective and efficient behavior-based android malware detection and prevention. *IEEE Transaction on Dependable and Secure Computing*, 15(1):83–97, 2018.

[120] J. Saxe and K. Berlin. Deep neural network based malware detection using two dimensional binary program features. In *10th International Conference on Malicious and Unwanted Software, MALWARE 2015, Fajardo, PR, USA, October 20-22, 2015*, pages 11–20. IEEE Computer Society, 2015.

[121] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero. AVclass: A tool for massive malware labeling. In *Processing of the International Symposium on Research in Attacks, Intrusions, and Defenses, RAID*, pages 230–253, 2016.

[122] G. Severi, J. Meyer, S. Coull, and A. Oprea. Exploring backdoor poisoning attacks against malware classifiers. In *USENIX security symposium (USENIX Security)*, pages 1093–1110, 2021.

[123] M. Z. Shafiq, S. M. Tabish, F. Mirza, and M. Farooq. Pe-miner: Mining structural information to detect malicious executables in realtime. In *Recent Advances in Intrusion Detection, 12th International Symposium, RAID 2009, Saint-Malo, France, September 23-25, 2009. Proceedings*, volume 5758 of *Lecture Notes in Computer Science*, pages 121–141. Springer, 2009.

[124] S. Shen, L. Huang, H. Zhou, S. Yu, E. Fan, and Q. Cao. Multistage signaling game-based optimal detection strategies for suppressing malware diffusion in fog-cloud-based IoT networks. *IEEE Internet of Things Journal*, 5(2):1043–1054, 2018.

[125] S. Siby, R. R. Maiti, and N. O. Tippenhauer. IoTScanner: Detecting privacy threats in IoT neighborhoods. In *Proceedings of the 3rd ACM International Workshop on IoT Privacy, Trust, and Security*, pages 23–30, 2017.

[126] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

[127] J. Su, D. V. Vargas, S. Prasad, D. Sgandurra, Y. Feng, and K. Sakurai. Lightweight classification of IoT malware based on image recognition. *arXiv preprint arXiv:1802.03714*, 2018.

[128] J. Su, D. V. Vargas, S. Prasad, D. Sgandurra, Y. Feng, and K. Sakurai. Lightweight classification of iot malware based on image recognition. In *IEEE Annual Computer Software and Applications Conference, COMPSAC*, pages 664–669. IEEE Computer Society, 2018.

[129] O. Suciu, S. E. Coull, and J. Johns. Exploring adversarial examples in malware detection. In *2019 IEEE Security and Privacy Workshops, SP Workshops*, pages 8–14, 2019.

[130] C. Sun, A. Shrivastava, S. Singh, and A. Gupta. Revisiting unreasonable effectiveness of data in deep learning era. In *IEEE International Conference on Computer Vision, ICCV 2017, Venice, Italy, October 22-29, 2017*, pages 843–852. IEEE Computer Society, 2017.

[131] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. J. Goodfellow, and R. Fergus. Intriguing properties of neural networks. In *International Conference on Learning Representations, ICLR*, 2014.

[132] A. Tamersoy, K. A. Roundy, and D. H. Chau. Guilt by association: large scale malware detection by mining file-relation graphs. In *the 20th ACM International Conference on Knowledge Discovery and Data Mining, KDD*, pages 1524–1533, 2014.

[133] M. Thoma, H. Cheng, A. Gretton, J. Han, H. Kriegel, A. J. Smola, L. Song, P. S. Yu, X. Yan, and K. M. Borgwardt. Discriminative frequent subgraph mining with optimality guarantees. *Statistical Analysis and Data Mining*, 3(5):302–318, 2010.

[134] K. Thomas, C. Grier, J. Ma, V. Paxson, and D. Song. Design and evaluation of a real-time url spam filtering service. In *2011 IEEE symposium on security and privacy*, pages 447–462, 2011.

[135] K. Thomas, C. Grier, J. Ma, V. Paxson, and D. Song. Design and evaluation of a real-time URL spam filtering service. In *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA*, pages 447–462. IEEE Computer Society, 2011.

[136] F. Tramèr, A. Kurakin, N. Papernot, D. Boneh, and P. D. McDaniel. Ensemble adversarial training: Attacks and defenses. In *Proceedings of the 2018 International Conference on Learning Representations.*, 2018.

[137] D. Vasan, M. Alazab, S. Wassan, B. Safaei, and Q. Zheng. Image-based malware classification using ensemble of cnn architectures (imcec). *Computers & Security*, page 101748, 2020.

[138] B. Wang, Y. Yao, B. Viswanath, H. Zheng, and B. Y. Zhao. With great training comes great vulnerability: Practical attacks against transfer learning. In *Proceedings of the USENIX Security Symposium, USENIX Security*, pages 1281–1297, 2018.

[139] Q. Wang, W. Guo, K. Zhang, A. G. O. II, X. Xing, X. Liu, and C. L. Giles. Adversary resistant deep neural networks with an application to malware detection. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1145–1153, 2017.

[140] C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security & Privacy*, 5(2):32–39, 2007.

[141] T. Wüchner, A. Cislak, M. Ochoa, and A. Pretschner. Leveraging compression-based graph mining for behavior-based malware detection. *IEEE Transaction on Dependable and Secure Computing*, 16(1):99–112, 2019.

[142] T. Wüchner, M. Ochoa, and A. Pretschner. Robust and effective malware detection through quantitative data flow graph metrics. In *Detection of Intrusions and Malware, and Vulnerability Assessment Conference, DIMVA*, pages 98–118, 2015.

[143] K. Xu, Y. Li, R. H. Deng, K. Chen, and J. Xu. Droidevolver: Self-evolving android malware detection system. In *IEEE European Symposium on Security and Privacy, EuroS&P 2019, Stockholm, Sweden, June 17-19, 2019*, pages 47–62. IEEE, 2019.

[144] W. Xu, D. Evans, and Y. Qi. Feature squeezing: Detecting adversarial examples in deep neural networks. In *the Network and Distributed System Security Symposium, NDSS*, 2018.

[145] Y. Xu, F. Sun, and X. Zhang. Literature survey of active learning in multimedia annotation and retrieval. In K. Lu, T. Mei, and X. Wu, editors, *International Conference on Internet Multimedia Computing and Service, ICIMCS '13, Huangshan, China - August 17 - 19, 2013*, pages 237–242. ACM, 2013.

[146] Z. Xu, K. Ren, S. Qin, and F. Craciun. CDGDroid: Android malware detection based on deep learning using CFG and DFG. In *Proceedings of the 20th International Conference on Formal Engineering Methods, ICFEM*, pages 177–193, 2018.

[147] Z. Xu, K. Ren, S. Qin, and F. Craciun. Cdgdroid: Android malware detection based on deep learning using cfg and dfg. In *International Conference on Formal Engineering Methods*, pages 177–193, 2018.

[148] S. Yajamanam, V. R. S. Selvin, F. Di Troia, and M. Stamp. Deep learning versus gist descriptors for image-based malware classification. In *Icissp*, pages 553–561, 2018.

[149] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining, 2002. Proceedings.*, pages 721–724, 2002.

[150] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. A. Porras. DroidMiner: Automated mining and characterization of fine-grained malicious behaviors in android applications. In *Proceedings of the 19th European Symposium on Research in Computer Security*, pages 163–182, 2014.

[151] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 116–127, 2007.

[152] M. Yousefi-Azar, L. G. Hamey, V. Varadharajan, and S. Chen. Malytics: a malware detection scheme. *IEEE Access*, 6:49418–49431, 2018.

[153] X. Yuan*et al.*. Adversarial examples: Attacks and defenses for deep learning. *IEEE transactions on neural networks and learning systems*, 30(9):2805–2824, 2019.

[154] J. Zhang, Z. Qin, H. Yin, L. Ou, and Y. Hu. IRMD: malware variant detection using opcode image recognition. In *Proceedings of the 22nd IEEE International Conference on Parallel and Distributed Systems, ICPADS*, pages 1175–1180, 2016.

[155] X. Zhang, Y. Zhang, M. Zhong, D. Ding, Y. Cao, Y. Zhang, M. Zhang, and M. Yang. Enhancing state-of-the-art classifiers with api semantics to detect evolved android malware. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 757–770, 2020.