Systematically Evaluating the Robustness of ML-based IoT Malware Detection Systems

Ahmed Abusnaina*
Meta, University of Central Florida
Seattle, WA, United States
ahmed.abusnaina@knights.ucf.edu

Afsah Anwar*
Northeastern University
Boston, MA, United States
a.anwar@northeastern.edu

Sultan Alshamrani
University of Central Florida
Orlando, FL, United States
salshamrani@knights.ucf.edu

Abdulrahman Alabduljabbar
University of Central Florida
Orlando, FL, United States
jabbar@knights.ucf.edu

Rhongho Jang
Wayne State University
Detroit, MI, United States
r.jang@wayne.edu

DaeHun Nyang
Ewha Womans University
Seodaemun, Seoul, South Korea
nyang@ewha.ac.kr

David Mohaisen
University of Central Florida
Orlando, FL, United States
david.mohaisen@ucf.edu

ABSTRACT

The rapid growth of the Internet of Things (IoT) devices is paralleled by them being on the front-line of malicious attacks. This has led to an explosion in the number of IoT malware, with continued mutations, evolution, and sophistication. Malware samples are detected using machine learning (ML) algorithms alongside the traditional signature-based methods. Although ML-based detectors improve the detection performance, they are susceptible to malware evolution and sophistication, making them limited to the patterns that they have been trained upon. This continuous trend motivates large body of literature on malware analysis and detection research, with many systems emerging constantly, outperforming their predecessors. In this paper, we systematically examine the state-of-the-art malware detection approaches, that utilize various representation and learning techniques, under a range of adversarial settings. Our analyses highlight the instability of the proposed detectors in learning patterns that distinguish the benign from the malicious software. The results exhibit that software mutations with functionality-preserving operations, such as stripping and padding, significantly deteriorate the accuracy of such detectors. Additionally, our analysis of the industry-standard malware detectors shows their instability to the malware mutations. Through extensive 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 and calls for the community to streamline their efforts towards testing the robustness of malware detectors to different manipulation techniques.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
RAID 2022, October 26–28, 2022, Limassol, Cyprus
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9704-9/22/10…$15.00
https://doi.org/10.1145/3545948.3545960

CCS CONCEPTS

• **Security and privacy → Malware and its mitigation**;

KEYWORDS

Adversarial Machine Learning; Robust Malware Detection

ACM Reference Format:
Ahmed Abusnaina*, Afsah Anwar*, Sultan Alshamrani, Abdulrahman Alabduljabbar, Rhongho Jang, DaeHun Nyang, and David Mohaisen. 2022. Systematically Evaluating the Robustness of ML-based IoT Malware Detection Systems. In *25th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2022), October 26–28, 2022, Limassol, Cyprus.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3545948.3545960

1 INTRODUCTION

The rising number of IoT devices in many application domains has exposed those devices' susceptibility to attacks and vulnerabilities. This susceptibility is attributed to hardware security flaws, firmware vulnerabilities [52], and the failure to comply with essential security metrics [4]. Even worse, it has been shown recently that IoT devices today are susceptible to software vulnerabilities that were disclosed decades ago, making them an easy target to well-known attacks vectors—malware; *e.g.* Brickerbot [46] and Mirai [15] botnets. The rising number of IoT devices in many application domains has exposed those devices' susceptibility to attacks and vulnerabilities. This susceptibility is attributed to hardware security flaws, firmware vulnerabilities [52], and the failure to comply with essential security metrics [4]. Even worse, it has been shown recently that IoT devices today are susceptible to software vulnerabilities that were disclosed decades ago, making them an easy target to well-known attacks vectors—malware; *e.g.* Brickerbot [46] and Mirai [15] botnets.

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 [41]. Those engines that feed into the likes of VirusTotal are fittingly considered as the up-to-date capability of industry-standard malware detectors.

Considering that these techniques are heavily dependent on the specific data used for their training and testing [38], 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. Moreover, packed software samples are known to be categorized as malicious by the industry-standard malware detectors [11]. While packing is widely used among malicious software, it is not exclusive to malware. This, in turn, limits the usage of packing as a detection modality since that may result in significant 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 [10, 33, 44].

A common practice for inspecting software is using online scan engines, such as VirusTotal [3], which embody the aforementioned malware detection techniques 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 [25]. 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 binary sequence, static disassembly features, and graphs. These representations yield a promising detection performance, with higher than 99% detection accuracy [12, 20, 35, 37, 53]. However, our findings highlight the instability of the learning algorithms in learning useful fundamental patterns that represent the difference between benign and malicious software (more details can be found in section 5).

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. Figure 1 shows the different phases of analysis strategy; feature representation, software manipulation, and evaluation of ML-based malware detectors.

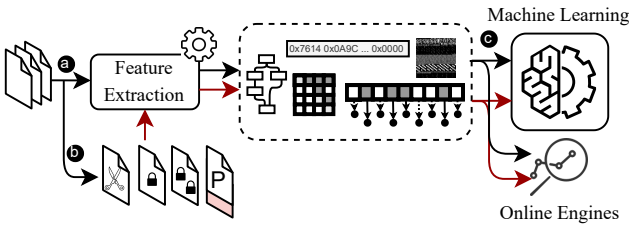


Figure 1: The system pipeline. The software binaries are (a) represented using different state-of-the-art approaches, and (b) manipulated using functionality preserving operations, such as packing, stripping, and padding. The corresponding representations of the original samples and manipulated ones are then (c) tested against pre-trained ML-based malware detectors and industry-standard detection engines.

Contributions. 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 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 detectors against three manipulation techniques: 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 can mislead the state-of-the-art malware detectors.
(5) *Vulnerability of industry-standard malware detectors*: The evaluation of industry-standard malware detection engines shows that most of the engines are rendered useless upon slight modification of the software.

Organization. The rest of this paper is organized as follows. We provide a background on malware detection and evasion techniques in section 2. We discuss the threat model under which we evaluate the robustness of malware detectors in section 3. Overview of the used dataset is provided in section 4. Then, we evaluate the state-of-the-art malware detectors in section 5 and the industry-standard detection engines in section 6. We conclude our work in section 7, providing the main takeaways of this study.

2 BACKGROUND

The increasing security concern for IoT devices has been paralleled by an increasing body of work around 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. In this section, we revisit some of those efforts that propose IoT malware detection systems.

2.1 Malware Detection

Prior works have shown the potential and feasibility of ML to detect malware with more than 99% accuracy [13, 16, 36, 37, 48, 50, 55]. 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 [51].

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. 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 used 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 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 [21].
(5) *Segments* are necessary for program execution. They describe the memory layout of an executable and is interpreted by the kernel during load [40]. Within every segment, there may be code or data divided among *sections*, such as *.text*. Binaries contain symbol tables which are used as references for linking and debugging [40].
(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 [40].
(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 using all the previous representations.

Table 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

For the completeness of the study, we include malware representations proposed by works that are not strictly IoT malware-specific. Table 2 summarizes the malware representations that have been proposed for malware detection, and utilized in this work.

2.2 Representation Evasion

Several software evasion and manipulation techniques were proposed for malware mutation and misclassification. In the following, we briefly discuss the commonly used techniques.

Binary Packing. Packing is used by malware authors to thwart detection or analysis by detectors, analysts. The packer is augmented to compress or encrypt an executable, where 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 [40].

Typical packing software consist of two programs, packer program and the stub program, where the first packs the software while the second deobfuscates the software. While there are 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) [7] is the one most used [21]. UPX utilizes the UCL data compression library algorithm [6] which uses in-place decompression and does not introduce memory overheads.

Binary Stripping. Stripping is utilized to hide information that may leak the functional software strategies. 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 [5].

Adversarial Evasion. With the growth in ML adoption, it is essential to understand and assess the robustness of ML techniques to

Table 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	[31, 37, 48, 54]	✓	✗	✗
CFG	Adjacency	[17, 30, 53]	✗	✓	✓
CFG	Algorithmic	[13, 16, 17]	✗	✓	✓
CODE	String	[12, 16]	✗	✓	✗
CODE	Symbols	[12, 16]	✗	✓	✗
CODE	Sections	[12, 16]	✗	✓	✗
CODE	Segments	[12]	✗	✓	✗
CODE	Hexdumps	[12]	✓	✓	✗
CODE	Combined	[12, 16]	✓	✓	✗

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 [43].

Given a model objective function $f(.)$ and a sample represented by a vector x, the adversary aims 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 model's output.

Several methods have been proposed to generate adversarial examples by directly perturbing the feature space in both black-box and white-box settings [24, 29, 34, 39]. For example, Carlini and Wagner [18] proposed generic adversarial attacks against distilled Neural Networks (NN), which showed its effectiveness against several "robust" deep learning NN.

While initially designed to exploit image-based classifiers, where perturbation can be directly applied to the image pixels [42, 43, 49], adversarial attacks showed high success in malware detection while preserving the software functionality and executability [10, 26]. At the binary-level, several studies [32, 33] 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 [9, 10] 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. Our work investigates the discrepancies between the capabilities of the adversaries and malware detectors, with a focus on the IoT malware detection systems. Our findings, however, are applicable to various machine learning-based malware detectors, irrespective of malware type.

3 THREAT MODEL

This work utilizes both white- and black-box attacks on various machine learning malware detection models. For industrial detection engines (*i.e.,* accessible through VirusTotal API), it is assumed that the adversary does not have access to the malware detection model, nor its configurations. A successful attack under black-box adversarial settings result in a more practical adversarial example. On the other hand, white-box attacks highlights the concerns that should be taken into account when developing a new malware detection engine. These concerns, however, can be exploited without accessing the internal configurations of the implemented malware detection model. This is mainly contributed to the transferability characteristics of adversarial examples, from one model architecture to another, and even one representation to another. This work considers white-box adversarial settings to highlight the drawbacks of machine learning design choices, while black-box settings are utilized to examine the robustness of established malware detection engines. In the following, we discuss the threat models used for systematically evaluating the robustness of the malware detectors.

3.1 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 [27, 28, 47].

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

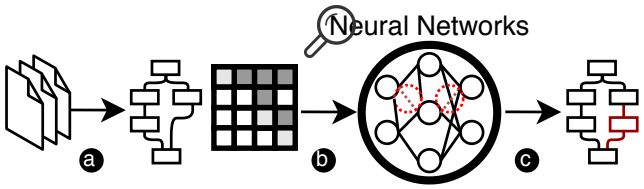


Figure 2: Graph manipulation. The software is reverse-engineered and (a) represented as CFG and adjacency matrix, (b) using the pre-trained neural network, (c) white-box C&W-based perturbation is crafted/applied to the CFG. We limit the allowed actions to adding edge and adding new node to generate a realistic CFG.

is then calculated as follows:

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

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 refrain from using 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, with the continuous increase of the perturbation, the model's accuracy deteriorates over the perturbation space, *e.g.* reaching random guess should would ideally require applying a high perturbation, and not <5% perturbation as shown later. We note that, however, this attack will not generate practical adversarial examples, since it applies the perturbation to the feature space directly. As such, attack scenario is used to measure the detectors' stability.

3.2 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 [19] 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. Figure 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 [9, 10], 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 remaining CFG-based classifiers.

3.3 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.

3.4 Binary Packing

We recall that a binary executable can be packed using a packer software, such as UPX (see subsection 2.2 for more details). The ML-based detectors utilize the features, such as the raw binaries, strings, and segments, from the malware file. These features are, however, suppressed from packing. In this attack, we pack 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 optimized compression methods of UPX. The default level (7) enforces compression ratio over speed using LZMA 7-Zip compression optimization. On the other hand, the optimized packing uses M_LZMA algorithm, prioritizing generating a smaller executable with a time consumption trade-off. The latter approach is strongly recommended before software release to avoid code inference.

3.5 Binary Stripping

Recall that a binary can be stripped of information without affecting its executability (see subsection 2.2). 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.

3.6 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. Figure 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

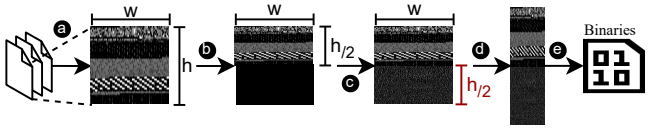


Figure 3: Binary padding attack overview. (a) The software is represented as an $h \times w$ image. (b) The content of the image is then compressed into the size of $\frac{h}{2} \times w$. (c) Using C&W attack, we generate perturbation on the remaining half $\frac{h}{2} \times w$ of the image. (d) The 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.

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.

4 DATASET OVERVIEW

To analyze the robustness of state-of-the-art malware detectors, we start by collecting a dataset of malicious and benign IoT binaries. These IoT malware binaries cover ELF binaries that cover multiple architectures, as has been followed by the prior work [14, 22]. 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 [1], VirusTotal [3], and VirusShare [8], in addition to 2,295 benign samples, compiled from source files on GitHub [23] with different optimization levels.

Ground Truth Class. We used *VirusTotal* [3] 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 in section 2, the dataset samples are transformed to different representations: (1) Represented as images to be fed into an image-based classifier. (2) Using *Radare2* [2], 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) Hexdump 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.

5 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 (see subsection 2.1) against executability- and functionality-preserving software binary manipulations.

5.1 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 [13, 45]. 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.

The CNN-based architecture performs well in extracting patterns in higher dimensionality when the pattern location is irrelevant. Therefore, we use the CNN model with image-, CFG adjacency-, and CFG algorithmic-based feature representations. On the other hand, the DNN-based architecture is used with the static-based vector representations, including Strings-, Symbols-, and Hexdump-based feature representations. 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:

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

where $f(X)$ is the LR's prediction and Y is the ground truth labels.

Random Forest (RF). RF allows for variance reduction in the output of the 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) × (feature map width) × (feature map depth). CNN performs well in extracting patterns in higher dimensionality when the pattern location, in the feature space, is

Table 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

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:

$$h^{(l)} = a(W^{(l)} \times X + b^{(l)}), \quad (1)$$

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.

5.2 Evaluation & Results

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

5.2.1 Baseline Evaluation. We implemented the baseline classifiers on our dataset (see section 4). Table 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.

RQ1. Are the baseline models stable under Gaussian noise? The model's performance decreases with the increase of the perturbation size, to eventually reach random. Stable model's performance deteriorate over the applied perturbation space. However, unstable models performance will rapidly drop after a perturbation threshold. This is mainly contributed to that such models over-fit on exact match, and are more sensitive to adversarial settings. Figure 4 shows the evaluation of the baseline classifiers under the Gaussian noise with 1%-100% applied perturbation rate. 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 the model to over-fit on the *exact match* instead of extracting feasible patterns.

Table 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	Malware Detection (%)
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

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

RQ2. Are the baseline classifiers prone to practical white-box adversarial attacks? 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 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 be 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 attacks are implemented on the NN-based classifier and transferred to 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 Finding: For several representations, practical white-box attacks are possible, and can be transferred to related learning algorithms and representations.

5.2.2 Binary Manipulation Attacks. These settings include evaluating the classifiers under manipulation attacks on the software. Here, we consider binary packing under default and optimized (packing*) conditions, stripping, and padding. Table 5 shows the evaluation results under these manipulation attacks strategies. In the following, we interpret these results posed as research questions.

RQ3. How does 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.* [11], demonstrating that the industry-standard windows malware detection systems identify the

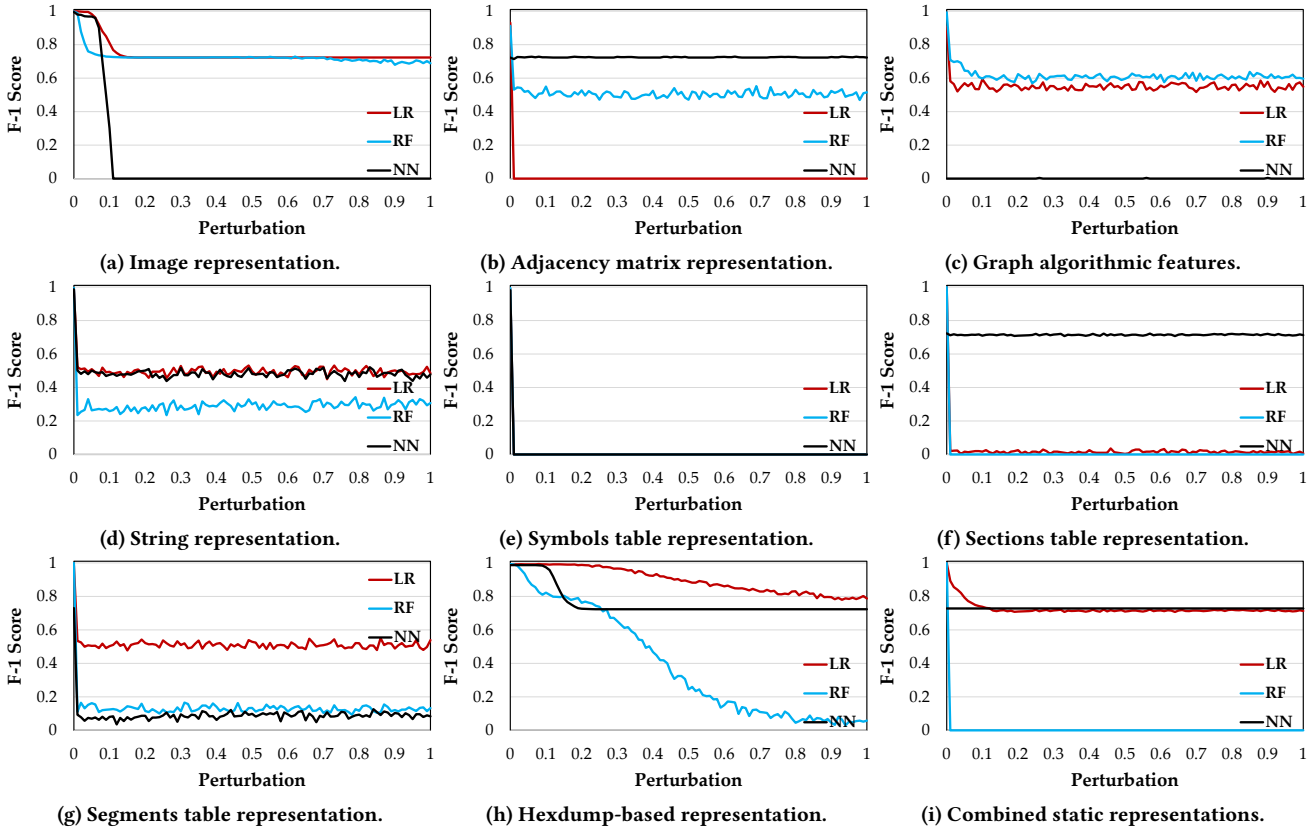


Figure 4: Baseline classifiers evaluation under various Gaussian noise perturbation rates (1%-100%).

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 stripping impact the performance of 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, and Hexdump, are intact.

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

RQ5. Does padding impact the performance 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. What is the suggested robust classifier for IoT malware detection task? 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.

6 INDUSTRY-STANDARD DETECTION ENGINES ROBUSTNESS ANALYSIS

Malware authors check their samples against the industry-standard online detection engines to ensure evading those engines. Given that these engines provide results for a pool of anti-virus scanners, evading them is considered a prototype for malware evolution. These mutations are then used in malware campaigns in the future. We argue that a practical malware detector should also detect such mutations, or at least cover for the low-effort based mutations.

6.1 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

Table 5: Baseline 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

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 discussed in section 5. We do not intend to compare the engines, nor raise concerns against any specific service provider.

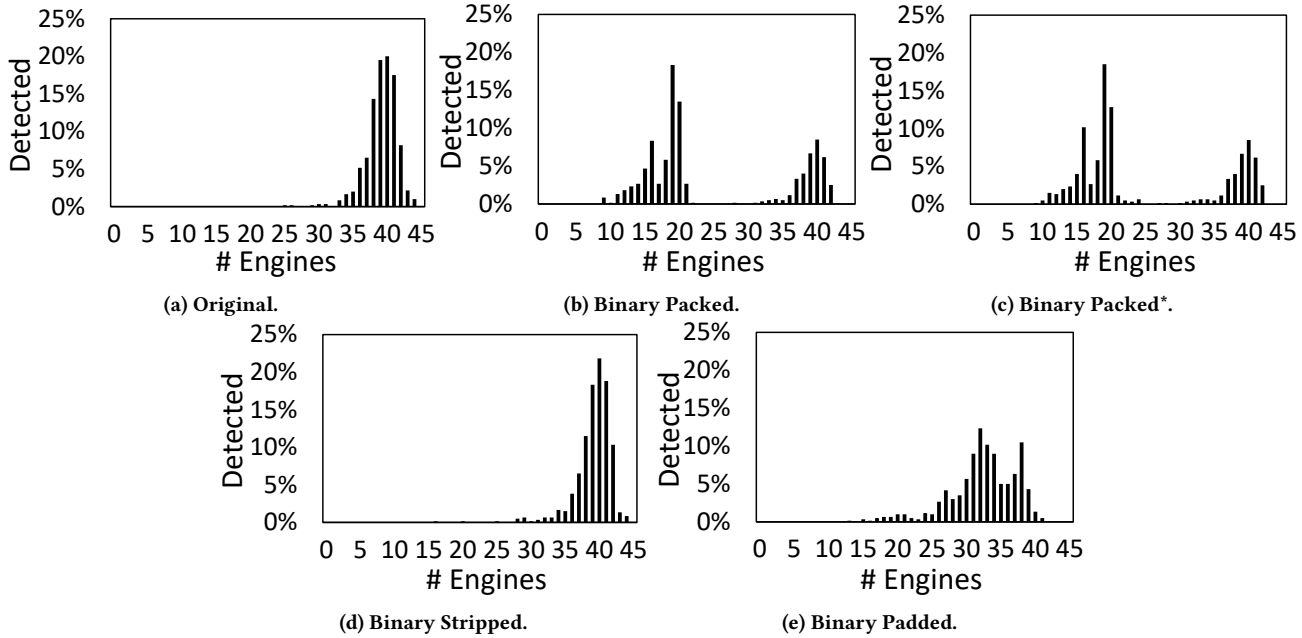
6.2 Evaluation & Results

We interpret the results of the industry-standard malware detectors to understand their behavior, shown in Table 6 and presented as research questions. The major insights are illustrated in Figure 6.

RQ7. What is the affect of manipulations on malware detection? To answer this question, we recorded the number of engines that identify malware as malicious. We begin by probing the original malware samples: Figure 5a shows the distribution of their detection rate by the engines. Notice that malware, on average, is detected by 40 engines, with most of them being detected by 35-45 engines. For the manipulated samples, however, the detection rate varies highly. Figure 5 shows the distribution of malicious samples by the number of engines for each of the manipulation strategies. We notice that stripping (Figure 5d) does not affect the distribution of the samples. However, packing (Figure 5b and Figure 5c) highly affects the detection rate. Moreover, while binary padding had minimal effects on the baseline classifiers' performance (section 5), it

Table 6: The evaluation results (%) of the 64 online IoT malware detection engines. 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 − 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



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

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

Figure 5: The online engines' detection rate of the original and binary manipulated IoT malware samples.

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 manipulations highly decreases the detection confidence.

RQ8. How do industrial engines perform under manipulation? To answer this question, we evaluate each individual detection engine using the original and manipulated benign and malicious software, shown in Table 6. We observe that multiple engines perform poorly, with 36% of the engines (23 out of 64) failing in identifying malware (< 15% malware detection rate), 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. We recall that both packing and stripping results in removal of information. Notice that, in general, industrial engines consider the lack of information as benign behavior, resulting in reduced malware detection rate under packing. This is more evident with packed benign samples, as the reported accuracy remained unchanged (*i.e.,* 100%).

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

RQ9. How do packed software 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 contrasts with previous observations [11] (refer to section 5). However, packing, generally, reduces the accuracy of malware being detected as malware. For instance, "E — 3" performance declined from 99.5% to ≈ 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 malware detection rate in comparison with default packing.

RQ10. How do stripped software affect the 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.

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

RQ11. How do padded software affect the 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.

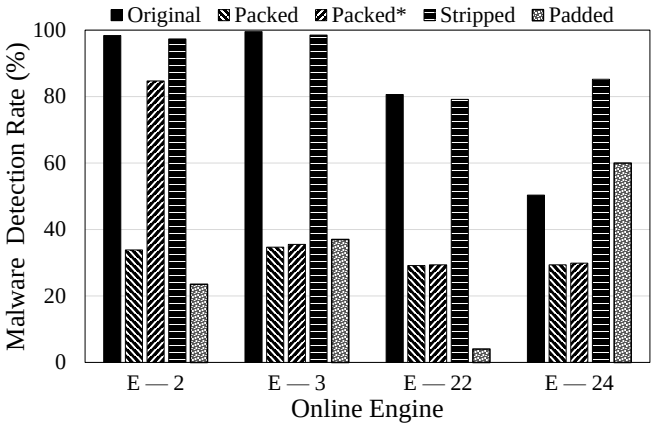


Figure 6: Industry-standard detection engines robustness highlight. Binary packing significantly reduces the detection rate of Malware software ("E — 2"). Binary stripping does not result in noticeable performance degradation, and may increase the malware detection rate ("E — 22"). Simple binary padding to the end of the file may cause significant degradation in the performance ("E — 3" and "E — 22").

7 CONCLUDING REMARKS

Malware analysis and detection have been the focus of the research community, with many advances seen in the AI-backed detection 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 consider the robustness of the proposed systems under adversarial capabilities. Investigating the adversarial settings is crucial to understand the drawbacks of implemented malware detection models. In the literature, it has been discussed that incorporating adversarial examples within the training process may increase the model's robustness. While this is true to some extent, we argue that training on specific adversarial settings and configuration does not guarantee the robustness under different adversarial attacks, nor same attack with different configurations. Due to the large space of adversarial perturbation, it is infeasible to train malware detectors on large set of adversarial attacks. This eventually results in decreased performance, while still vulnerable to various adversarial settings. We note that adversarial attacks exploit poor design choices, obligating for a deep understanding of the underlying learning algorithms and data representations.

Acknowledgement. The authors would like to thank anonymous reviewers of RAID'22 for their valuable suggestions and Erman Ayday for shepherding this work. This work was supported by the Global Research Lab (GRL) Program of the National Research Foundation (NRF) funded by the Ministry of Science, Information, and Communication Technologies (ICT), Future Planning (NRF-2016K1A1A2912757), and a seed grant from CyberFlorida. The work was additionally supported by the NRF grant funded by the Korea government (MSIT) (NRF-2020R1A2C2009372).

REFERENCES

[1] 2019. CyberIOCs. Available at [Online]: https://freeiocs.cyberiocs.pro/.
[2] 2019. Radare2. Available at [Online]: https://https://rada.re/r/.
[3] 2019. VirusTotal. Available at [Online]: https://www.virustotal.com.
[4] 2022. Smart Yet Flawed: IoT Device Vulnerabilities Explained. Available at [Online]: https://bit.ly/2MBykDx.
[5] 2022. Strip: GNU binary Utility. Available at [Online]: https://sourceware.org/binutils/docs/binutils/strip.html.
[6] 2022. UCL Data Compression Library. Available at [Online]: http://www.oberhumer.com/opensource/ucl/.
[7] 2022. UPX: the Ultimate Packer for eXecutables. Available at [Online]: https://upx.github.io/.
[8] 2022. VirusShare. Available at [Online]: https://virusshare.com/.
[9] Ahmed Abusnaina, Hisham Alasmary, Mohammed Abuhamad, Saeed Salem, DaeHun Nyang, and Aziz Mohaisen. 2019. Subgraph-Based Adversarial Examples Against Graph-Based IoT Malware Detection Systems. In *International Conference on Computational Data and Social Networks*. 268–281.
[10] Ahmed Abusnaina, Aminollah Khormali, Hisham Alasmary, Jeman Park, Afsah Anwar, and Aziz Mohaisen. 2019. Adversarial Learning Attacks on Graph-based IoT Malware Detection Systems. In *IEEE International Conference on Distributed Computing Systems, ICDCS*.
[11] Hojjat Aghakhani, Fabio Gritti, Francesco Mecca, Martina Lindorfer, Stefano Ortolani, Davide Balzarotti, Giovanni Vigna, and Christopher Kruegel. 2020. When Malware is Packin'Heat; Limits of Machine Learning Classifiers Based on Static Analysis Features. In *Network and Distributed Systems Security (NDSS) Symposium*.
[12] Mansour Ahmadi, Dmitry Ulyanov, Stanislav Semenov, Mikhail Trofimov, and Giorgio Giacinto. 2016. Novel feature extraction, selection and fusion for effective malware family classification. In *Proceedings of ACM conference on data and application security and privacy*. 183–194.
[13] Hisham Alasmary, Aminollah Khormali, Afsah Anwar, Jeman Park, Jinchun Choi, Ahmed Abusnaina, Amro Awad, DaeHun Nyang, and Aziz Mohaisen. 2019. Analyzing and Detecting Emerging Internet of Things Malware: A Graph-based Approach. *IEEE Internet of Things Journal* (2019).
[14] Omar Alrawi, Charles Lever, Kevin Valakuzhy, Kevin Snow, Fabian Monrose, Manos Antonakakis, et al. 2021. The Circle Of Life: A {Large-Scale} Study of The {IoT} Malware Lifecycle. In *30th USENIX Security Symposium (USENIX Security 21)*. 3505–3522.
[15] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. 2017. Understanding the mirai botnet. In *USENIX security symposium (USENIX Security)*. 1093–1110.
[16] Afsah Anwar, Hisham Alasmary, Jeman Park, An Wang, Songqing Chen, and David Mohaisen. 2020. Statically Dissecting Internet of Things Malware: Analysis, Characterization, and Detection. In *International Conference on Information and Communications Security*. Springer, 443–461.
[17] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. 2006. Detecting self-mutating malware using control-flow graph matching. In *International conference on detection of intrusions and malware, and vulnerability assessment*. Springer, 129–143.
[18] Nicholas Carlini and David A. Wagner. 2017. Towards Evaluating the Robustness of Neural Networks. In *IEEE Symposium on Security and Privacy, SP*. 39–57.
[19] Nicholas Carlini and David A. Wagner. 2017. Towards Evaluating the Robustness of Neural Networks. In *Proceedings of the IEEE Symposium on Security and Privacy*. 39–57.
[20] Zhenxiang Chen, Qiben Yan, Hongbo Han, Shanshan Wang, Lizhi Peng, Lin Wang, and Bo Yang. 2018. Machine learning based mobile malware detection using highly imbalanced network traffic. *Inf. Sci.* 433-434 (2018), 346–364.
[21] Emanuele Cozzi, Mariano Graziano, Yanick Fratantonio, and Davide Balzarotti. 2018. Understanding Linux Malware. In *IEEE Symposium on Security & Privacy*.
[22] Emanuele Cozzi, Pierre-Antoine Vervier, Matteo Dell'Amico, Yun Shen, Leyla Bilge, and Davide Balzarotti. 2020. The tangled genealogy of IoT malware. In *Annual Computer Security Applications Conference*. 1–16.
[23] Developers. 2019. GitHub. Available at [Online]: https://github.com/.
[24] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and Harnessing Adversarial Examples. In *International Conference on Learning Representations, ICLR*.

[25] Mariano Graziano, Davide Canali, Leyla Bilge, Andrea Lanzi, and Davide Balzarotti. 2015. Needles in a Haystack: Mining Information from Public Dynamic Analysis Sandboxes for Malware Intelligence. In *24th USENIX Security Symposium (USENIX Security 15)*.

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

[27] Chuan Guo, Mayank Rana, Moustapha Cisse, and Laurens van der Maaten. 2018. Countering Adversarial Images using Input Transformations. In *International Conference on Learning Representations, ICLR*.

[28] Shengyuan Hu, Tao Yu, Chuan Guo, Wei-Lun Chao, and Kilian Q Weinberger. 2019. A new defense against adversarial images: Turning a weakness into a strength. In *Neural Information Processing Systems, NeurIPS*.

[29] Weiwei Hu and Ying Tan. 2017. Generating Adversarial Malware Examples for Black-Box Attacks Based on GAN. *arXiv preprint arXiv:1702.05983* abs/1702.05983 (2017).

[30] Pankaj Jalote. 2012. *An integrated approach to software engineering*. Springer Science & Business Media.

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

[32] Bojan Kolosnjaji, Ambra Demontis, Battista Biggio, Davide Maiorca, Giorgio Giacinto, Claudia Eckert, and Fabio Roli. 2018. Adversarial Malware Binaries: Evading Deep Learning for Malware Detection in Executables. In *The European Signal Processing Conference, EUSIPCO*. 533–537.

[33] Felix Kreuk, Assi Barak, Shir Aviv-Reuven, Moran Baruch, Benny Pinkas, and Joseph Keshet. 2018. Deceiving end-to-end deep learning malware detectors using adversarial examples. In *Workshop on Security in Machine Learning (NIPS)*.

[34] Alexey Kurakin, Ian J. Goodfellow, and Samy Bengio. 2017. Adversarial examples in the physical world. In *the 5th International Conference on Learning Representations, ICLR*.

[35] Jin Li, Lichao Sun, Qiben Yan, Zhiqiang Li, Witawas Srisa-an, and Heng Ye. 2018. Significant Permission Identification for Machine-Learning-Based Android Malware Detection. *IEEE Trans. Ind. Informatics* 14, 7 (2018), 3216–3225.

[36] Zhiqiang Li, Jun Sun, Qiben Yan, Witawas Srisa-an, and Yutaka Tsutano. 2019. Obfusifier: Obfuscation-Resistant Android Malware Detection System. In *Security and Privacy in Communication Networks - 15th EAI International Conference, SecureComm 2019, Orlando, FL, USA, October 23-25, 2019, Proceedings, Part I (Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, Vol. 304)*, Songqing Chen, Kim-Kwang Raymond Choo, Xinwen Fu, Wenjing Lou, and Aziz Mohaisen (Eds.). Springer, 214–234.

[37] Francesco Mercaldo and Antonella Santone. 2020. Deep learning for image-based mobile malware detection. *Journal of Computer Virology and Hacking Techniques* (2020), 1–15.

[38] Aziz Mohaisen, Omar Alrawi, and Manar Mohaisen. 2015. AMAL: High-fidelity, behavior-based automated malware analysis and classification. *Computers & Security* 52 (2015), 251–266.

[39] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. 2016. DeepFool: A Simple and Accurate Method to Fool Deep Neural Networks. In *IEEE Conference on Computer Vision and Pattern Recognition*. 2574–2582.

[40] Ryan Elfmaster O'Neill. 2016. *Learning Linux Binary Analysis*. Packt Publishing.

[41] Hamed Haddad Pajouh, Ali Dehghantanha, Raouf Khayami, and Kim-Kwang Raymond Choo. 2018. A deep Recurrent Neural Network based approach for Internet of Things malware threat hunting. *Future Gener. Comput. Syst.* 85 (2018), 88–96.

[42] Nicolas Papernot, Patrick D. McDaniel, Ian J. Goodfellow, Somesh Jha, Z. Berkay Celik, and Ananthram Swami. 2017. Practical Black-Box Attacks against Machine Learning. In *Proceedings of the ACM on Asia Conference on Computer and Communications Security, AsiaCCS*. 506–519.

[43] Nicolas Papernot, Patrick D. McDaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, and Ananthram Swami. 2016. The Limitations of Deep Learning in Adversarial Settings. In *IEEE European Symposium on Security and Privacy*. 372–387.

[44] Giorgio Severi, Jim Meyer, Scott Coull, and Alina Oprea. 2021. Exploring Backdoor Poisoning Attacks Against Malware Classifiers. In *USENIX security symposium (USENIX Security)*. 1093–1110.

[45] Shigen Shen, Longjun Huang, Haiping Zhou, Shui Yu, En Fan, and Qiying Cao. 2018. Multistage Signaling Game-Based Optimal Detection Strategies for Suppressing Malware Diffusion in Fog-Cloud-Based IoT Networks. *IEEE Internet of Things Journal* 5, 2 (2018), 1043–1054.

[46] Jiawei Su, Danilo Vasconcellos Vargas, Sanjiva Prasad, Daniele Sgandurra, Yaokai Feng, and Kouichi Sakurai. 2018. Lightweight Classification of IoT Malware Based on Image Recognition. In *IEEE Annual Computer Software and Applications Conference, COMPSAC*. IEEE Computer Society, 664–669.

[47] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. 2014. Intriguing properties of neural networks. In *International Conference on Learning Representations, ICLR*.

[48] Danish Vasan, Mamoun Alazab, Sobia Wassan, Babak Safaei, and Qin Zheng. 2020. Image-based malware classification using ensemble of CNN architectures (IMCEC). *Computers & Security* (2020), 101748.

[49] Bolun Wang, Yuanshun Yao, Bimal Viswanath, Haitao Zheng, and Ben Y. Zhao. 2018. With Great Training Comes Great Vulnerability: Practical Attacks against Transfer Learning. In *Proceedings of the USENIX Security Symposium, USENIX Security*. 1281–1297.

[50] Shanshan Wang, Zhenxiang Chen, Qiben Yan, Ke Ji, Lizhi Peng, Bo Yang, and Mauro Conti. 2020. Deep and broad URL feature mining for android malware detection. *Inf. Sci.* 513 (2020), 600–613.

[51] Carsten Willems, Thorsten Holz, and Felix Freiling. 2007. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security & Privacy* 5, 2 (2007), 32–39.

[52] Teng Xu, James Wendt, and Miodrag Potkonjak. 2014. Security of IoT systems: Design challenges and opportunities. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 417–423.

[53] Zhiwu Xu, Kerong Ren, Shengchao Qin, and Florin Craciun. 2018. CDGDroid: Android malware detection based on deep learning using CFG and DFG. In *International Conference on Formal Engineering Methods*. 177–193.

[54] Sravani Yajamanam, Vikash Raja Samuel Selvin, Fabio Di Troia, and Mark Stamp. 2018. Deep Learning versus Gist Descriptors for Image-based Malware Classification.. In *Icissp*. 553–561.

[55] Anli Yan, Zhenxiang Chen, Haibo Zhang, Lizhi Peng, Qiben Yan, Muhammad Umair Hassan, Chuan Zhao, and Bo Yang. 2021. Effective detection of mobile malware behavior based on explainable deep neural network. *Neurocomputing* 453 (2021), 482–492.