IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 15 October 2024, accepted 2 December 2024, date of publication 9 December 2024, date of current version 17 December 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3513713

RESEARCH ARTICLE

An Adversarial Attack on ML-Based IoT Malware Detection Using Binary Diversification Techniques

MAINA BERNARD MWANGI, (Graduate Student Member, IEEE), AND SHIN-MING CHENG, (Member, IEEE)

Department of Computer Science and Information Engineering, National Taiwan University of Science and Technology, Taipei 10607, Taiwan

Corresponding author: Maina Bernard Mwangi (d10915815@mail.ntust.edu.tw)

This work was supported in part by the National Science and Technology Council (NSTC), Taiwan, under Grant 113-2221-E-011-156-MY3 and Grant 113-2221-E-011-157-MY3.

ABSTRACT The integration of machine learning (ML) has revolutionized malware detection, enabling accurate identification of subtle distinctions between malware and benignware. As the threat landscape continually evolves and new malware strains emerge, conventional signature-based detectors are becoming increasingly inadequate, leading to a growing reliance on ML-based detectors. However, ML-based detection systems are particularly vulnerable to adversarial attacks, where subtle alterations to input samples can deceive detectors into misclassifying malware as benignware, highlighting the need for robustness studies, as such misclassifications can lead to significant damage. To this end, we stage a black-box attack on IoT malware detection systems, specifically targeting structure-based detectors, which are predominant due to their ability to detect malware across diverse CPU architectures in IoT environments. Our strategy employs semantic-preserving binary diversification techniques, including function inlining, branch function insertion, control flow graph flattening, and basic block merging and reordering, to modify malware binaries and evade detection. We train a multi-structural substitute detector (based on a combination of control flow graph and function call graph features) on a large-scale dataset of IoT ELF binaries, achieving detection rates of up to 98.24%. Using explainable AI (XAI), we transfer the attack to four structural target detectors, achieving evasion rates of up to 100% on certain detectors, with an average binary size increase of just 8.35%. The modified samples evade detection by a state-of-the-art adversarial detector and several commercial antivirus engines, highlighting the persistent challenge of defending against adversarial threats and emphasizing the need for enhanced and multi-faceted defense mechanisms.

INDEX TERMS Adversarial attack, binary diversification, Internet of Things (IoT) malware detection, machine learning.

I. INTRODUCTION

Machine learning (ML) has become integral to modern cybersecurity, marking a breakthrough in the detection of zero-day malware. The success of ML techniques has led cyber defense researchers and antivirus vendors to increasingly adopt these methods to address the evolving landscape of malware variants [1], [2]. ML-based malware detection essentially involves analyzing benign and malicious files, extracting features through static and dynamic analysis, and utilizing these features to train ML models [3], [4], [5], [6]. These detection systems have demonstrated high success rates in identifying both known malware and novel threats [7], [8]. However, ML systems are particularly vulnerable to adversarial attacks, where slight modifications to input samples can deceive detectors into misclassifying malware as benignware, posing severe cybersecurity risks.

Adversarial attacks pose even greater challenges in resource-constrained IoT systems. Machine learning-based IoT malware detection remains less developed compared to its Windows counterpart. The limitations in IoT environments, such as restricted computational resources and diverse CPU architectures, necessitate lightweight and efficient solutions, making direct adaptation of Windows-based techniques difficult [6]. To address these challenges, researchers in IoT malware detection have predominantly relied on structural

The associate editor coordinating the review of this manuscript and approving it for publication was Ye Liu.

features, such as control flow graphs (CFGs) and function call graphs (FCGs) [9], [10], [11]. These features are particularly effective for detecting malware across the diverse CPU architectures in IoT systems, as noted by Li et al. [12].

Similarly, adversarial attacks on IoT malware detection are still in their infancy compared to those targeting Windows systems. Most studies on adversarial attacks in the IoT domain focus on payload injections into malware samples and involve feature-space manipulations. For instance, [13], [14] embed graphs from benign samples into the malware CFGs to evade detection. Likewise, Esmaeili et al. [15] propose a GNN-based adversarial detector that involves merging CFGs from benign and malware samples to generate adversarial samples, learning the distribution of benign samples to filter out the adversarial ones. Sandor et al. [16] append extra bytes from malware and benign samples into malware binaries to evade detection, followed by adversarial training to harden the detector. Abusnaina et al. [17] demonstrate that most ML IoT malware detection approaches are vulnerable to simple manipulations like packing, stripping, and padding. Khormali et al. [18] introduce the COPYCAT attack, appending adversarial images to malware for IoT and Windows detection evasion. Ngo et al. [19] utilize reinforcement learning to modify PSI-graphs with dummy vertices and edges, followed by adversarial training to improve the detector robustness. While padding and payload injections can trick some malware detectors, these methods can often be mitigated by removing the padded bytes before classification.

This study evaluates the robustness of ML-based IoT malware detection systems against adversarial attacks, focusing on structural detectors due to their prominence in IoT environments. We introduce a novel semantic-preserving black-box adversarial attack on IoT structural detectors. A multi-structural substitute detector is trained on a large IoT dataset using CFG and FCG graphical features, with Explainable AI guiding binary-level manipulations to induce misclassification. Advanced binary diversification methods—function inlining, branch function insertion, control flow graph flattening, basic block merging, and basic block reordering—are used to modify malware binaries at both the basic block and function levels, successfully evading detection. To our knowledge, this is the first use of these techniques in adversarial attacks on ML-based malware detection. The generated adversarial examples demonstrate high transferability, evading detection by four structural detectors, several commercial antivirus engines, and a recent IoT adversarial detector. Our main contributions are summarized below.

1) We introduce a novel black-box functionality-preserving adversarial attack to evaluate the robustness of ML-based structural IoT malware detectors. Our approach employs advanced binary diversification techniques, such as function inlining, branch function insertion, control flow graph flattening, basic block merging, and basic block reordering, to modify malware samples and evade detection. Unlike common methods like payload injection and padding, our strategy does not leave obvious signatures, making it more challenging to defend against.
2) We compile a comprehensive IoT dataset containing over 248,000 Executable and Linkable Format (ELF) binary files from various CPU architectures, including benign and malicious samples from diverse IoT malware families, for our experiments. We then train a multi-structural substitute detector, utilizing both CFG and FCG graphical features, achieving high detection rates of up to 98.27%
3) Leveraging SHAP (SHapley Additive exPlanations) analysis [20], we execute the attack on the substitute detector, generating practical adversarial examples with minimal attack cost. These samples exhibit high transferability, evading four detectors [8], [9], [12], [21] trained on different structural features, with evasion rates up to 100% and an average binary size increase of just 8.35%. Additionally, the adversarial samples evade a recent IoT adversarial detector [15] and several commercial antivirus engines.

The remainder of the paper is organized as follows: Section II covers related work and background information, Section III presents the proposed methodology, Section IV discusses the experimental results and analysis, and Section V concludes the study.

II. BACKGROUND INFORMATION AND RELATED WORK

This section reviews background information and related work, including ML-based malware detection, a literature review of adversarial attacks on malware detection, and binary diversification techniques.

A. MACHINE LEARNING MALWARE DETECTION

Malware detection is critical across various computing platforms, including Windows, Android, and IoT. Considerable efforts have been devoted to effectively detecting malware. Traditional approaches, rooted in signature-based methods, rely on extensive databases of known malware signatures. When a suspicious file is encountered, its signature is compared against those stored in the database. However, this method's reliance on predefined signatures renders it ineffective against novel and unknown malware variants and inadequate for emerging cybersecurity threats. To overcome these limitations, and inspired by the success of machine learning in other domains, ML models have been adapted for malware detection, demonstrating strong generalization capabilities for identifying new and unseen (zero-day) malware variants [22]. ML-based malware detection comprises three main steps: data collection, feature engineering, and model training and evaluation.

1) DATA COLLECTION

This step involves collecting and labeling sufficient malware and benign samples. Labeling is typically done using malware analysis tools like VirusTotal, which detects malware across about 70 modern antivirus engines [23]. However, detection results for the same file may vary across engines. To address this, either the most recognized antivirus engine is chosen, or a voting-based approach is used.

2) FEATURE ENGINEERING

As machine learning models only operate on numeric inputs, feature engineering is a pivotal step in ML malware detection. It involves extracting intrinsic features from the collected files and converting them into corresponding numeric representations, which are then used to train the models to distinguish between benign and malicious files. In malware detection, features fall into three categories based on their extraction method: static, dynamic, and hybrid [3], [24].

Static features, derived directly from samples without the need for execution, are widely employed in malware detection due to their ease of extraction and effectiveness. For instance, printable strings [1], [11], [24], byte sequences [25], [26], PE/ELF headers [5], [27], and grayscale images [3], [22], [28], [29] have proven effective in detecting Windows, Android, and IoT malware.

Dynamic features involve executing binaries in isolated environments like virtual machines or sandboxes and monitoring runtime statuses of system resources, networks, registries, and files. Metrics such as CPU usage, I/O requests, and memory usage are then used to train malware detectors [30], [31]. File status features, obtained through counting and logging of created, deleted, modified, or accessed files, have also proven effective in malware detection [31].

Hybrid features are extracted through a combination of static and dynamic analysis methods. Hybrid features such as opcodes (n-gram sequence, images, frequency, etc.) [11], [28], [29], function call graphs (FCGs) [8], [10], Control flow graphs (CFGs) [9], [11], and API/system calls (sequence, list, graphs, etc.) [1], [4] have been successfully utilized in malware detection.

3) MODEL TRAINING AND EVALUATION

After extracting numeric features, selecting a suitable machine-learning model for malware detection is crucial. Numerous algorithms, including Deep Neural Networks (DNNs), Convolutional Neural Networks (CNNs) [9], [29], Long Short-Term Memory (LSTM) networks, Multi-Layer Perceptrons (MLPs) [8], Graph Neural Networks (GNNs) [12], Support Vector Machines (SVMs) [1], Random Forests (RFs) [8], and Decision Trees (DTs) [27], have been proposed and rigorously evaluated for malware detection. These models exhibit varying success rates depending on different experimental setups and parameter settings.

B. ADVERSARIAL ATTACKS ON MALWARE DETECTION

Despite recent advancements, ML-based malware detection systems remain inherently vulnerable to adversarial attacks that seek to undermine their decision-making processes [13], [40]. These attacks can be categorized based on the attacker's space and knowledge level. The attacker's space categorization includes feature-space attacks, which involve modifications to the input features, and problem-space attacks, which entail modifying real-world inputs like binary executables or source code to deceive the target detector. Based on the attacker's knowledge, adversarial attacks can be categorized as white-box or black-box attacks. In white-box attacks, the attacker has complete knowledge of the target model, while in black-box attacks, adversaries typically have minimal information, usually only the model's prediction output. Gray-box attacks fall between these two extremes, with varying levels of knowledge.

From these categorizations, four fundamental types of adversarial attacks are identified in the existing literature and discussed below. While this paper focuses on adversarial attacks in IoT malware detection, this section will also cover related attacks on Windows and Android platforms to provide a comprehensive overview of the relevant literature.

1) FEATURE-SPACE WHITE-BOX ADVERSARIAL ATTACKS

Esmaeili et al. [15] propose a structural attack on CFG-based IoT malware detectors, similar to the GEA and SGEA frameworks by Abusnaina et al. [13], [14]. Their approach merges control flow graphs (CFGs) from benign samples with target malware CFGs to create adversarial CFGs intended for a graph neural network (GNN)-based detector. They then train an adversarial detector to recognize benign CFG properties and filter out adversarial CFGs before classification.

In another attack on IoT malware detection, Ngo et al. [19] propose a reinforcement learning-based method that performs adversarial attacks on PSI (printable string information) graphs by adding dummy vertices and edges to deceive detectors. They counter these attacks with adversarial retraining.

Kreuk et al. [32] and Suciu et al. [33] successfully execute an adversarial attack against MalConv [7], a prominent raw byte-based Windows malware detector. Kreuk et al. utilize the Fast Gradient Sign Method (FGSM) to append adversarial payloads to the end of the file (append-FGSM) and into the slack regions of the sample (slack-FGSM). Suciu et al. [33] extend this approach by comparing slack-FGSM and append-FGSM, observing that slack-FGSM is more effective than append-FGSM.

Al-Dujaili et al. [35] and Verwer et al. [34] employ FGSM for white-box adversarial attacks on API Call List-based PE malware detectors. These attacks alter the malware's binary feature vector by flipping bits in the feature space. Verwer et al.'s attack dynamically adjusts the flipped bits based

TABLE 1. A summary of the literature review on adversarial attacks. The short forms in the table are as follows: BB for black box, WB for white box, FS for feature space, PS for problem space, and FP stands for functionality preserving.

Attack Name	Platform	Attacker's Knowledge	Attacker's Space	Detector	Modification	Strategy	FP
Esmaeili et al. [15]	IoT	WB	FS	CFG-based	Graph embedding	None	
Ngo et al. [19]	IoT	WB	FS	PSI-graph	Graph manipulations	RL	
Kreuk et al. [32], Suciu et el. [33]	Windows	WB	FS	MalConv [7]	Append/inject payload	FGSM	✓
GRAMS [34], Al-Dujaili et al. [35]	Windows	WB	FS	API Call-based	Add bogus API calls	Gradient-based	
AMAO [36]	Windows	WB	FS	Image-based	NOPs insertion	FGSM, C&W	
ATMPA [37], COPYCAT [18]	IoT, Windows	WB	FS	Image-based	Append noise	Gradient-based	✓
GEA [13], SGEA [14]	IoT	WB	PS	CFG-based	Graph embedding	None	✓
Sandor et al. [16]	IoT	WB	PS	Byte-based	Appending Bytes	Customized	
Abusnaina et al. [17]	IoT	WB	PS	Multiple	Packing, padding, stripping	Binary manipulations	✓
Demetrio [38]	Windows	WB	PS	MalConv [7]	PE header modification	Gradient-based	
AMB [39], Aryal et al. [40]	Windows	WB	PS	MalConv [7]	Padding/Payload injection	Gradient-based	
RAMEn [41]	Windows	WB	PS	MalConv [7]	Content shifting DOS header extension	Gradient-based	
Sharif et al. [42]	Windows	WB	PS	MalConv [7],AvastNet	Binary diversification	Gradient-based	✓
HRAT [43]	Android	WB	PS	MaMaDroid,MalScan	FCG-graph modification	RL	✓
MalGAN [44], Improved-MalGAN [45]	Windows	BB	FS	API call-based	API calls insertion	GAN	
Hu and Tan [46]	Windows	BB	FS	API call sequence-based	API calls insertion	Generative model	
GADGET [47]	Windows	BB	FS	API call sequence-based	API calls insertion	Transferability	✓
BADGER [48]	Windows	BB	FS	API call sequence-based	API calls insertion	Evolutionary algorithm	✓
SRL [49]	Windows	BB	FS	CFG-based	Semantic NOPs insertion	RL	✓
Gym-malware [26], [50], gym-plus [51], gym-malware-min [52]	Windows	BB	PS	General	Functinality-preserving modifications	RL	✓
AIMED [53], MDEA [54], GAMMA [38]	Windows	BB	PS	General, MalConv [7]	Functinality-preserving modifications	Evolutionary algorithms	✓
ARMED [55]	Windows	BB	PS	General	Functinality-preserving modifications	Randomization	✓
GAPGAN [56]	Windows	BB	PS	MalConv [7]	Appending bytes	GAN	✓
MalFox [57]	Windows	BB	PS	General	Obfuscation techniques	GAN	✓
Lucas et al. [58]	Windows	BB	PS	MalConv [7], others	Binary diversification	Hill climbing	✓
AndroidHIV [52]	Android	BB	PS	MaMaDroid [59], Drebin [60]	Code Injection	Gradient-based	✓
EvadeDroid [61]	Android	BB	PS	MaMaDroid [59], Drebin [60]	Payload injection	Randomization	✓
This work	**IoT**	**BB**	**PS**	**Structural [8], [9], [12], [21]**	**Binary diversification**	**Explainability/Greedy**	✓

on solution quality, effectively evading detection by adding irrelevant API calls.

Other attacks, such as ATMPA [37], COPYCAT [18], and AMAO [36] are aimed at image-based detectors. In ATMPA, Liu et al. [37] initially convert malware into a grayscale image and then utilize FGSM and C&W to generate adversarial examples. Similarly, COPYCAT by Khormali et al. [18] employs generic adversarial attacks to generate an adversarial image, which is subsequently appended to the original malware image. Park et al. [36] propose the AMAO adversarial attack, wherein a non-executable adversarial image is first generated using off-the-shelf adversarial attacks. They then attempt to maintain functionality by inserting semantic NOPs into the original malware, making it as similar as possible to the generated non-executable adversarial image.

2) PROBLEM-SPACE WHITE-BOX ATTACKS

Abusnaina et al. [13] introduce Graph Embedding and Augmentation (GEA), a structural adversarial attack on CFG-based IoT malware detectors. GEA induces misclassification by inserting a benign code into the target malware sample, directly modifying its CFG. Subsequently, they propose Sub-GEA (SGEA) [14], which reduces the required embedded graph size for misclassification.

In another study, Abusnaina et al. [17] evaluate the robustness of various machine-learning IoT malware detectors against simple functionality-preserving modifications, such as padding, packing, and stripping. Their findings confirm that these detection systems remain largely vulnerable to such manipulations.

Sandor et al. [16] propose two adversarial strategies for IoT byte-based malware detection: Chunker, which appends chunks of malware to itself, and Disguiser, which embeds malware in benign files. The generated adversarial examples are then used to retrain and harden the target detector.

Kolosnjaji et al. [39] introduce AMB (Adversarial Malware Binary), a gradient-based attack specifically tailored for PE byte-based malware detectors such as MalConv [7]. This method involves appending adversarial bytes, generated via gradient descent, to the end of the original malware binary. Aryal et al. [40] similarly apply gradient-based methods to generate adversarial examples by injecting code into intra-section caves, successfully evading the MalConv detector [7].

Demetrio et al. [38] employ the integrated gradient explainability technique to assess the feature importance of MalConv detector [7]. Realizing MalConv's reliance on PE header features, they then perform a white-box attack by

modifying specific bytes in the PE binary's DOS header, successfully evading detection.

Implementing two functionality-preserving modifications, Shift and Extend, Demetrio et al. [41] develop the RAMEn attack framework against the MalConv detector. By shifting the content of the first section of the PE file and extending the DOS header, the authors inject a carefully crafted adversarial payload, successfully evading detection.

Sharif et al. [42] introduce functionality-preserving binary diversification techniques for adversarial attacks on malware detection to enhance attack effectiveness and stealthiness. They employ code displacement and in-place randomization to conduct a white-box attack using gradient ascent, ultimately achieving high evasion rates.

Zhao et al. [43] introduce the Heuristic Optimization Integrated Reinforcement Learning Attack (HRAT), a code-level structural attack against graph-based Android malware detection. HRAT involves subtle modifications to Function Call Graphs (FCGs), including node deletion, insertion, and edge manipulation.

3) FEATURE-SPACE BLACK-BOX ATTACKS

Hu and Tan [44] propose the MalGAN attack against an API call list-based PE malware detector in a black-box setting by training a substitute model. Adversarial examples are generated by appending irrelevant API calls to the original malware samples. Kawais et al. [45] extend MalGAN to Improved-MalGAN, addressing limitations of the original version by using different API call lists to train MalGAN and the substitute detector.

In another study, Hu and Tan [46] devise a generative model to evade RNN-based PE malware detectors. They generate spurious API call sequences using a generative RNN and insert them into the API call sequence of the original malware. A similar strategy is employed by Rosenberg et al. [47] in an attack named GADGET, which targets detectors trained on API call sequences. Utilizing the transferability property, GADGET first trains a surrogate model, conducts a white-box attack, and then heuristically uses the generated adversarial API call sequences to evade the target detector. Subsequently, Rosenberg et al. [48] propose a similar attack framework named BADGER, which limits the number of queries made to the target detector.

In [49], Zhang et al. introduce SRL, a functionality-preserving reinforcement learning-based attack against graph-based (CFG) PE malware detectors. This attack employs a reinforcement learning agent to iteratively select semantic NOPs for insertion into the CFG blocks of the original malware until the generated adversarial samples successfully evade the target detector.

4) PROBLEM-SPACE BLACK-BOX ATTACKS

This category represents the most realistic and challenging adversarial attacks, as they are completely agnostic to specific malware detectors. Black-box attacks in the problem space often use strategies like heuristic algorithms, evolutionary algorithms, reinforcement learning, and GANs. For example, Anderson et al. [26], [50] employ reinforcement learning in their Gym-malware attack framework to automatically generate functionality-preserving adversarial examples that deceive static malware detectors and antivirus engines. Gym-malware's success inspires further research [51], [62], [63]. Some studies expand the action space [51], while others reduce it and use deterministic sequence selection to improve effectiveness and stealth [62], [63].

Castro et al. [55] introduce ARMED (Automatic Random Malware Modifications), which employs random algorithms to apply nine functionality-preserving modifications from [50] and [26] to malware samples until evasion is achieved. They assess the functionality of the resulting adversarial samples using the Cuckoo sandbox. Similarly, Chen et al. [64] generate adversarial examples by randomly appending blocks of data from benignware to malware, successfully evading the MalConv [7] detector.

Some attacks use evolutionary algorithms, such as AIMED by Castro et al. [53], which applies nine format-preserving modifications using genetic programming. AIMED iteratively modifies the malware binary, achieving a 50% speed increase over randomization. Similarly, the MDEA uses a genetic algorithm to generate adversarial examples with ten functionality-preserving modifications [54], while GAMMA [38] employs the same strategy to modify malware files through section injection and padding.

Yuan et al. [56] propose the GAPGAN framework, which utilizes Generative Adversarial Networks (GANs) to deceive the MalConv [7] detector. The framework trains a generator and discriminator to create adversarial payloads appended to malware samples. The discriminator simulates a black-box attack, achieving up to a 100% evasion rate. Similarly, Zhong et al. [57] develop MalFox using a Convolutional GAN to generate adversarial samples that preserve the original functionality of malware and evade detection by antivirus engines.

Lucas et al. [58] introduce a black-box adversarial attack using binary diversifications, such as in-place randomization and code displacement. Unlike the white-box version [42], this method uses a hill-climbing algorithm and accepts transformations only if the benign probability increases after querying the model.

Chen et al. [52] extract and modify APK source code by injecting non-executable code and repackaging it, altering features like permissions, API calls, and CFG structure to evade detection. Similarly, Bostani et al. [61] use payload injection in malware samples to deceive Android malware detectors.

From the reviewed literature, it is evident that few adversarial attacks specifically target IoT malware detection. Most rely on padding and code injection methods, which can be easily identified and filtered before classification. In many studies, these attacks are conducted in the feature space and assume white-box access, which is less realistic in real-world scenarios. As discussed above, only two papers

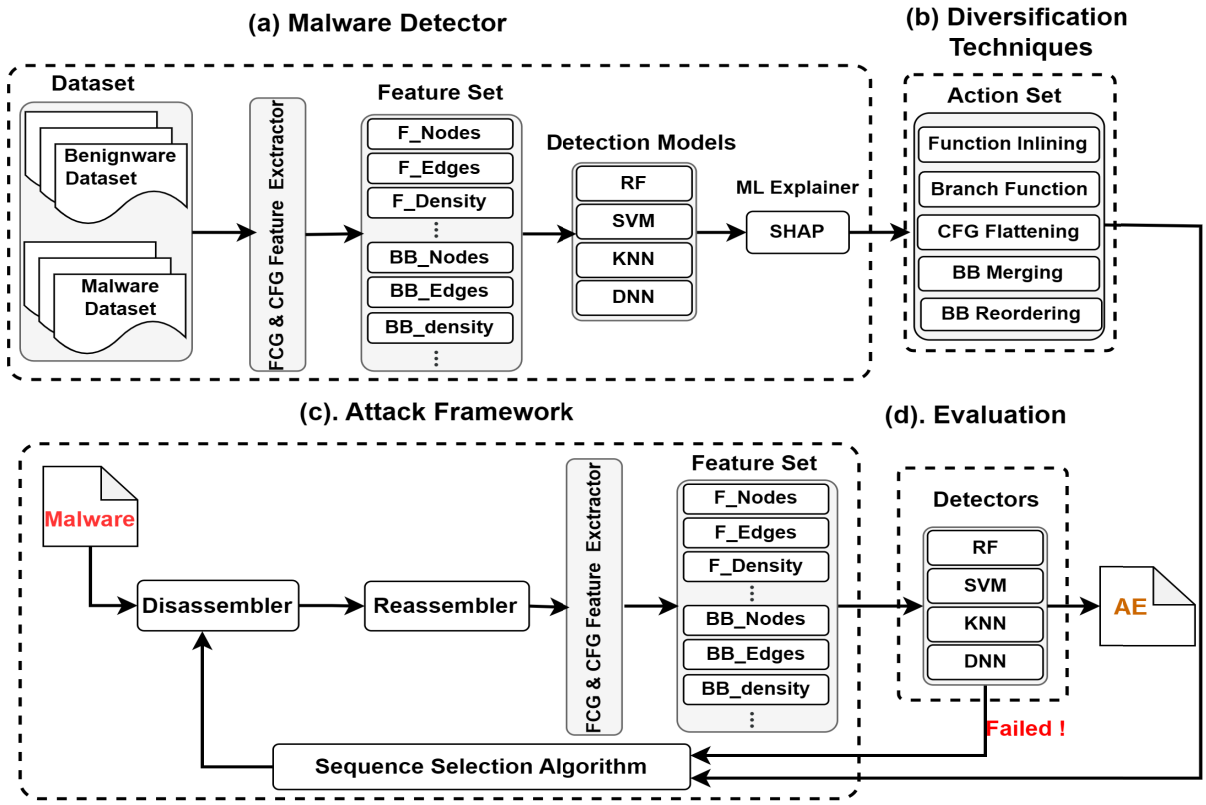

FIGURE 1. The proposed attack framework: AE denotes Adversarial Example, 'BB_' prefixes indicate CFG-based features, and 'F_' prefixes represent FCG-based features.

on PE malware detection have explored binary diversification in this context [42], [58]. When implemented correctly, binary diversification preserves the original functionality of the binary while modifying functional parts, making it stealthier and more challenging to defend against. Therefore, we employ binary diversification to manipulate the structural properties of binaries and evade detection.

C. BINARY DIVERSIFICATION TECHNIQUES

Binary diversification, designed to enhance security against attacks like code reuse, injection, and memory corruption [65], [66], [67], involves creating multiple program versions with identical functionality. Lucas et al. [58] and Sharif et al. [42] pioneered its application in adversarial contexts, using semantic-preserving modifications like in-place randomization and code displacement to bypass raw byte-based PE malware detection. Building on this, we propose an attack framework that uses advanced binary diversification to evade IoT graph-based malware detectors. Unlike Lucas et al.'s method, which relies on instruction-level changes, our approach incorporates structural binary manipulations, such as function inlining, branch function insertion, control flow graph flattening, basic block merging, and basic block reordering [65], [66] (discussed in Section III-C). Additionally, we leverage explainable AI (XAI) and a greedy algorithm, differentiating our strategy from Lucas et al.'s [58] use of reinforcement learning.

III. PROPOSED METHOD

In this section, we present the proposed attack framework, detailing the system model, feature importance analysis, action set, and adversarial example generation algorithm. Figure 1 illustrates the workflow, consisting of four modules that will be discussed in detail later in this section.

A. SYSTEM MODEL

1) THREAT MODEL

Our attack scenario assumes the adversary has black-box access to the target detector, meaning they can only receive the prediction confidence that a file is benign or malicious after querying the model. The goal is to use binary diversification to modify malware samples in the problem space until they are misclassified as benign by the target structural detector while preserving their malicious functionality. With limited black-box access, we build a multi-structural substitute detector trained on control flow graph and function call graph features, execute the attack, and transfer it to the target detector.

2) PROBLEM FORMULATION

In this paper, $\mathcal{X}$ represents the input space (problem space), which includes ELF binary files. Each binary sample $x \in \mathcal{X}$ is associated with a label $y \in \mathcal{Y}$, where $\mathcal{Y} = \{0, 1\}$. Using reverse engineering, we transform each binary sample $x \in \mathcal{X}$

into an n-dimensional feature vector $z \in \mathcal{Z}$, as defined in (1).

$$\tau : \mathcal{X} \mapsto \mathcal{Z} \subseteq \mathbb{R}^n. \tag{1}$$

Specifically, by leveraging the Angr [68] and Radare2 [69] frameworks, we extract control flow graphs and function call graphs from the ELF binary files and utilize NetworkX [70] to extract the graphical features described in Section III-A3. Given the restricted black-box access, the adversary is limited to obtaining prediction probabilities from the target detector $\mathbb{D} : \mathcal{Z} \rightarrow [0, 1]$. The adversary's primary objective is to modify the malware sample $x \in \mathcal{X}$ to mislead the target detector $\mathbb{D}$ into misclassifying the malware as benignware. The proposed attack method is defined as

$$\begin{aligned} &\tilde{z} = \tau(x + \delta) = \tau(\tilde{x}), \\ &\tilde{z} \in \mathcal{Z}, \text{ and } \tilde{x} \in \mathcal{X}, \end{aligned} \tag{2}$$

where δ is the perturbation of the malware samples in the problem space, which entails functionality-preserving modifications achieved through binary diversification techniques discussed in III-C.

To effectively execute the attack, we employ the SHAP [20] algorithm from explainable AI (XAI) to identify the most influential features for the detector $\mathbb{D}$. The explainability analysis, based on the model's prediction results, determines the positive or negative contribution of each feature. The contribution of the i-th sample to the predicted probability can be expressed as:

$$\gamma(\mathbb{D}, z_i) = w_i = \left[w_{i,0}, w_{i,1}, \cdots, w_{i,j}, \cdots, w_{i,m}\right], \tag{3}$$

where $w_{i,j}$ represents the j-th feature's contribution for the i-th sample. To determine the most influential features for the model prediction, we select the target features based on the following constraint:

$$\frac{1}{n}\sum_{i=0}^{n-1}\left|w_{i,j}\right| \geq \frac{1}{m}\sum_{k=0}^{m-1}\left(\frac{1}{n}\sum_{i=0}^{n-1}\left|w_{i,k}\right|\right), \tag{4}$$

for all j in the range $0, 1, \ldots, m$.

Next, we apply binary-level modifications that specifically target these influential features to deceive the detector into classifying a malicious file as benign. This approach focuses on manipulating the most critical features. Our attack strategy is designed to work seamlessly within the problem space, preserving the original functionality of the sample while enhancing the attack's imperceptibility.

3) FEATURE SET

To train the substitute detector, we extract structural features at both the basic block and function levels. Using Radare2 [69], we derive function call graphs (FCGs) from all training binaries and compute various graph properties with NetworkX [70], including nodes, edges, density, connected components, reciprocity coefficient, and the minimum, maximum, and mean values of closeness centrality, betweenness centrality, degree centrality, and shortest path. For basic block-level features, we use the Angr framework [68] to extract control flow graphs (CFGs) and compute the same set of graphical features as for FCGs. In total, we generate 34 features from both CFGs and FCGs (see Table 2) to train the substitute detector, referred to as a multi-structural detector. Preliminary experiments indicate that training on both CFG and FCG features yields a more robust detector compared to training on either feature set alone.

TABLE 2. Feature set for multi-structural detector.

FCG Features (Function level)	CFG Features (BB level)
F_nodes	BB_nodes
F_edges	BB_edges
F_density	BB_density
F_CC	BB_CC
F_reciprocity	BB_reciprocity
F_(Mean, Max, Min)CloCent	BB_(Mean, Max, Min)CloCent
F_(Mean, Max, Min)BtwCent	BB_(Mean, Max, Min)BtwCent
F_(Mean, Max, Min)DegCent	BB_(Mean, Max, Min)DegCent
F_(Mean, Max, Min)ShortPath	BB_(Mean, Max, Min)ShortPath

B. FEATURE IMPORTANCE ANALYSIS

The foundation of our imperceptible adversarial attacks is depicted in part (a) of Figure 1. After training the substitute detector, we use the SHAP [20] technique to analyze feature importance and understand the correlation between each feature and the model's prediction results. Figure 2 shows the distribution of SHAP values for the top 20 features, enabling an intuitive analysis of the predictions. Each row represents the distribution of a feature's SHAP values across all samples, with higher-ranked features having more influence. The color intensity of each point, representing a test dataset sample, indicates its corresponding SHAP value. We select the top 12 influential features to target for modification using binary diversification techniques. These features include six from the FCGs: `F_MaxCloCent`, `F_MaxShortPath`, `F_MaxBtwCent`, `F_MeanShortPath`, `F_reciprocity`, and `F_CC`, as well as six from the CFGs (basic block level): `BB_MaxShortPath`, `BB_MeanBtwCent`, `BB_MaxBtwCent`, `BB_density`, `BB_reciprocity`, and `BB_nodes`.

The SHAP analysis results, illustrated in Fig. 2, provide several key insights. Notably, the model is more likely to classify a sample as malicious when the FCG features `F_MaxCloCent`, `F_MaxBtwCent`, and `F_reciprocity` have higher values. Consequently, our strategy involves reducing the values of these features through structural modifications implemented using binary diversification techniques. Conversely, lower values of the features `F_MaxShortPath`, `F_CC`, and `F_MeanShortPath` increase the likelihood of a sample being classified as malicious. Therefore, we aim to increase the values of these features to cause the malware samples to be misclassified as benign.

At the basic block level, features such as `BB_nodes`, `BB_MaxShortPath`, `BB_MaxBtwCent`, as well as `BB_reciprocity`, exhibit a positive correlation with a

sample's classification as malware. Our strategy, therefore, is to reduce these features' values, thereby deceiving the target detector into misclassifying malicious samples as benign. Conversely, features such as `BB_MeanBtwCent` and `BB_density` show a negative correlation with a sample's classification as malware. As such, our approach involves increasing the values of these features to mislead the detector into misclassifying malware as benign.

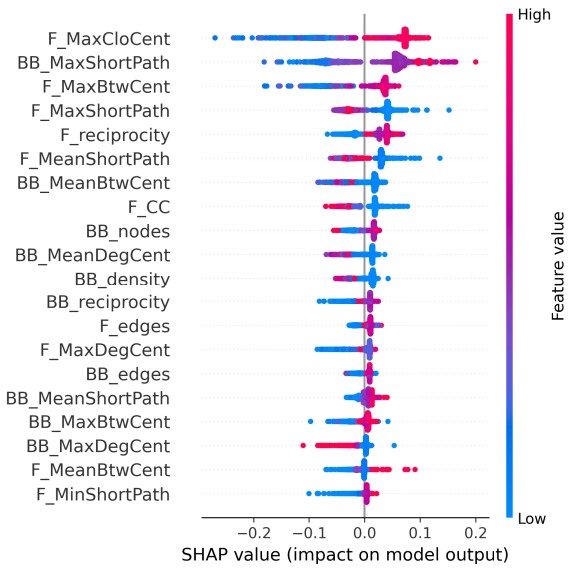

FIGURE 2. The SHAP value distribution of testing dataset top 20 features when label = malicious (RF).

C. ACTION SET

Based on the feature importance analysis discussed above, we develop five functionality-preserving modifications to alter the binary structure at both the basic block and function levels. These modifications utilize binary diversification techniques originally proposed to protect against code-reuse attacks and similar threats [65], [66], [67]. When implemented correctly, these techniques preserve binary semantics, as demonstrated by Wang et al. [67], who generated diverse ELF binary versions using various diversification methods. To mislead structural target detectors, we adopt several techniques employed by Wang et al. [67], including function inlining, branch function insertion, control flow graph flattening, basic block merging, and basic block reordering, which are detailed below.

1) FUNCTION INLINING

Function inlining is an optimization technique used in compilers. It involves replacing function calls with the body of the called function (callee) at the call site. To do this, the call instructions are replaced by *jump* and *push* instructions to maintain the original semantics. In each iteration, we randomly select a function, excluding the main function, and inline it at its direct call sites if its size is less than 300 bytes. For each inlined function, its return instruction is changed to a *jump* instruction, targeting the instruction adjacent to the original call site in the caller function. This transformation significantly alters the structure of the function call graph by reducing the number of edges and nodes, thus reducing the values of `F_MaxCloCent`, `F_MaxBtwCent`, and `F_reciprocity`.

2) BRANCH FUNCTION INSERTION

Branch function insertion (shown in Fig. 3) is a technique that substitutes *jump* instructions with function calls to a predefined "branch routine" function, redirecting the control flow to the original *jump* destination. In each iteration, we randomly select 1% of the *jump* instructions for conversion into function calls. These calls are directed to simple functions that reroute the flow to the original destination addresses of the *jump* instructions. This modification, while minimally impacting the size and performance complexity, significantly alters the binary's structural properties by increasing the number of nodes and edges, thereby achieving the desired effect on features `F_MaxShortPath`, `F_CC`, and `F_MeanShortPath`.

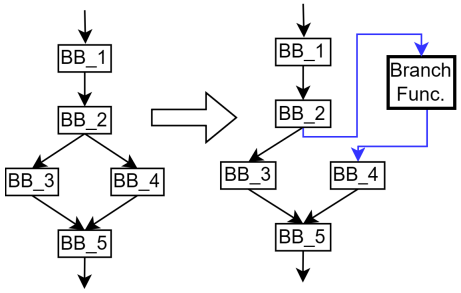

FIGURE 3. Branch function insertion.

3) CONTROL FLOW GRAPH (CFG) FLATTENING

This method, as shown in Fig. 4, transforms a function's control flow graph into a "switch" structure using dispatcher blocks to redirect execution flow while preserving the program's functionality [65], [66]. In this study, we avoid obfuscating functions with indirect jumps due to the complexity of determining control flow destinations. Given the high computational cost of CFG flattening, we adopt a conservative approach by flattening only a small, randomly selected subset of functions. Specifically, in each iteration, we randomly select 1% of functions without indirect jumps for CFG flattening. This modification significantly alters the structure of a function's CFG and achieves the desired effects on features such as `BB_MaxShortPath`, `BB_MaxBtwCent`, `BB_MeanBtwCent`, and `BB_density`.

4) BASIC BLOCK MERGING

Basic block merging consolidates two basic blocks into one, adjusting flow control instructions to preserve semantics.

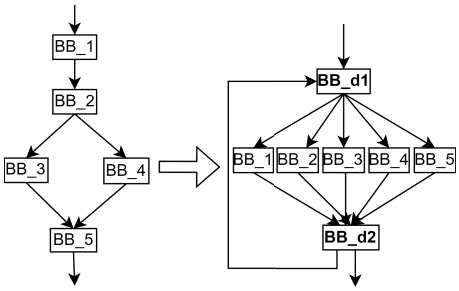

FIGURE 4. Control flow graph flattening.

In each iteration, we randomly select five pairs of basic blocks for merging. Each pair must belong to the same function and be directly connected with exactly one incoming and one outgoing connection. This process significantly alters the binary's structure at the basic block level without introducing significant overhead in size or performance complexity. Block merging achieves the desired outcomes of reducing the values of `BB_MaxShortPath` and `BB_nodes` as well as increasing the value of `BB_density`.

5) BASIC BLOCK REORDERING

Basic block reordering involves changing the relative positions of two or more basic blocks. To maintain functionality, additional control transfers are introduced, which increases the number of edges in the control flow graph. During each iteration, we examine functions with more than three basic blocks and randomly adjust the positions of a selected pair. While this modification increases the number of edges and achieves desired effects on features `BB_MaxShortPath`, `BB_density`, and `BB_MeanBtwCent`, it can also complicate the graph's structure, potentially leading to higher performance costs.

D. ADVERSARIAL EXAMPLE (AE) GENERATION ALGORITHM

In this subsection, we present the details of the algorithm behind the attack framework depicted in Fig. 1, part (c). The target detector $\mathbb{D}$ takes as input the feature vector $\mathbf{z}$ extracted from the binary sample x and outputs a confidence score, $\mathbb{D}(\mathbf{z}) \rightarrow [0, 1]$. If the score exceeds 0.5, the sample is classified as malware; otherwise, it is labeled benign. Our primary goal is to modify the malware sample x using binary diversification techniques discussed in III-C until the target detector classifies it as benign with a confidence score below 0.5, i.e., $\mathbb{D}(\tilde{\mathbf{z}}) < 0.5$. To this end, we design Algorithm 1, a greedy algorithm specially tailored to effectively deceive the target detector while minimizing the attack cost.

In each iteration, we start with an ELF malware binary and transform it from the problem space x to the feature space $\mathbf{z}$ using the transformation function τ. In particular, we employ the Angr [68] and Radare2 [69] frameworks to extract graphical features ($\mathbf{z}$) from the control flow graph and function call graph derived from the ELF binary. We then feed the extracted feature vector $\mathbf{z}$ into the target detector $\mathbb{D}$ to obtain a prediction probability. If the probability is greater than 0.5, indicating that the sample is classified as malware, we disassemble the binary into its assembly code, select an action from the action set $\mathcal{A}$, and apply the action to the disassembled code. We then reassemble the binary, transform it into the feature vector $\tilde{\mathbf{z}}$, and evaluate its prediction probability. This process is repeated for all actions in $\mathcal{A}$, and the action resulting in the lowest prediction probability is selected. If no action reduces the confidence score, the step counter is incremented. To handle stagnation, a maximum stagnation limit $N = 10$ is introduced; if the confidence score does not change in 10 consecutive

Algorithm 1 Diversification-Based Adversarial Attack

Input : Malware sample (x), Target Detector ($\mathbb{D}$), Action set ($\mathcal{A}$), Maximum stagnation (N), Epsilon ($\epsilon \approx 0.1$).
Output : Adversarial example ($\tilde{x}$), Selected sequence (S).

$\tilde{x} \leftarrow x$, $S \leftarrow \phi$, $step \leftarrow 1$, $\tilde{\mathbf{z}} \leftarrow \tau(\tilde{x})$;
while $\mathbb{D}(\tilde{\mathbf{z}}) \geq 0.5$ *and* $step \leq N$ **do**
  $Proba_{min} \leftarrow \mathbb{D}(\tilde{\mathbf{z}})$;
  $best\_a \leftarrow \phi$;
  $r \leftarrow$ UniformRandom(0, 1);
  if $r \leq \epsilon$ **then**
    $best\_a \leftarrow \mathcal{A}$[RandomInteger(0, ($|\mathcal{A}| - 1$))];
  end if
  else
    for $a \in \mathcal{A}$ **do**
      $\tilde{x}^{tmp}_{asm} \leftarrow Disassemble(\tilde{x})$;
      $\tilde{x}^{tmp}_{asm} \leftarrow Diversify(\tilde{x}^{tmp}_{asm}, a, step)$;
      $\tilde{x}^{tmp} \leftarrow Reassemble(\tilde{x}^{tmp}_{asm})$;
      $\tilde{\mathbf{z}}^{tmp} \leftarrow \tau(\tilde{x}^{tmp})$;
      if $\mathbb{D}(\tilde{\mathbf{z}})^{tmp}) < Proba_{min}$ **then**
        $Proba_{min} \leftarrow \mathbb{D}(\tilde{\mathbf{z}})^{tmp})$;
        $best\_a \leftarrow a$;
      end if
    end for
    if $best\_a = \phi$ **then**
      $step \leftarrow step + 1$;
      continue;
    end if
  end
  $step \leftarrow 1$;
  $S \leftarrow S \cup best\_a$;
  $\tilde{x}_{asm} \leftarrow Disassemble(\tilde{x})$;
  $\tilde{x}_{asm} \leftarrow Diversify(\tilde{x}_{asm}, best\_a, step)$;
  $\tilde{x} \leftarrow Reassemble(\tilde{x})$;
  $\tilde{\mathbf{z}} \leftarrow \tau(\tilde{x})$;
end while
return $\tilde{x}$, S

iterations, the algorithm terminates and moves to the next sample. To facilitate exploration and avoid local minima, a randomized parameter is used during action selection; with probability ϵ, a random action is chosen from $\mathcal{A}$ instead of the best-performing action. This process continues until the prediction probability of the transformed binary is below 0.5 or the maximum stagnation limit is reached.

It is noteworthy that the disassembly, modification, and reassembly of binaries require careful handling to mitigate potential errors. In our implementation, we utilize an open-source disassembly-reassembly tool proposed by Wang et al. [71], which is specifically designed for the automatic disassembly of executables in a manner that supports their subsequent reassembly into functional binaries.

IV. EXPERIMENTAL RESULTS AND ANALYSIS

This section presents the experimental results and analysis. It begins with an overview of the dataset used in our experiments, followed by a detailed evaluation of the detection results for the substitute detector and the four structural IoT detectors [8], [9], [12], [21] used to assess the proposed attack. Next, the efficacy of the structural attack is examined, followed by a transferability analysis of the generated adversarial examples against these four IoT detectors, an adversarial detector [15], and commercial antivirus engines.

A. DATASET

To evaluate the effectiveness of the proposed attack framework, a large-scale dataset comprising 248,276 IoT Executable and Linkable Format (ELF) binary files representing diverse CPU architectures, including x86-64, x86, ARM, SPARC, PowerPC, and MIPS, was compiled. Sample labeling was conducted using VirusTotal [23], leveraging its extensive database of over 70 antivirus software vendors. The final classification of samples was determined by a majority voting criterion based on the VirusTotal detection report, establishing both the class label and the specific malware family associated with each malicious sample.

TABLE 3. Substitute detector results in %.

Algorithm	Accuracy	Precision	Recall	F1 Score
RF	98.24	98.30	98.27	98.27
KNN	96.84	96.77	96.90	96.82
SVM	95.61	95.44	95.81	95.58
DNN	97.19	97.73	96.77	97.24

The dataset comprised 115,823 benign and 132,453 malware IoT ELF files spanning different families, including Mirai, Android, Tsunami, Bashlite, Hajime, Dofloo, Xorddos, and Pnscan. Mirai emerged as the predominant family, underscoring its prevalence within the IoT domain. The dataset was split, with 80% designated for the training set and 20% for the test set.

B. IoT MALWARE DETECTION

1) MULTI-STRUCTURAL SUBSTITUTE DETECTOR

Upon data preparation, we built a multi-structural detector to serve as our substitute detector in the proposed black-box attack. This detector was trained on a comprehensive set of 34 features extracted from both the FCGs and CFGs of the IoT ELF binaries, as explained in section III-A3. We trained and selected the best four ML models, including Random Forest (RF), K-Nearest Neighbors (KNN), Deep Neural Networks (DNN), and Support Vector Machines (SVM), achieving accuracy scores ranging from 95.61% to 98.24%. Detailed results are presented in Table 3.

2) ALASMARY ET AL. [9] MALWARE DETECTOR

To implement the malware detector proposed by [9], we utilized r2pipe, a Radare2 Python API, to extract the FCGs from the binaries [69]. Subsequently, we employed NetworkX [70] to compute various graphical properties of the FCGs as proposed by [9]. In total, 23 features were extracted and used to train RF, KNN, DNN, and SVM machine learning models. We obtained detection results ranging from 87.01% to 97.09%, as detailed in Table 4.

3) GRAMAC MALWARE DETECTOR [21]

We also implemented the structural malware detector proposed by [21] to further assess the efficacy of the proposed attack. This detector is based on the caller-callee relationships of sensitive API calls. Specifically, we used radare2 to extract API call graphs and subsequently employed NetworkX to extract various graphical features. Seven features—number of nodes, edges, indegree, outdegree, loops, connected components, and parallel edges—were used to train RF, KNN, DNN, and SVM models. The detection results range from 86.16% to 97.42%, as presented in Table 4.

4) WU et al. [8] MALWARE DETECTOR

We implemented the malware detector proposed by [8] to further evaluate our proposed structural attack. This detector leverages structural features such as nodes, edges, and density, as well as graph embedding features extracted using Graph2Vec. It enhances function-call graphs by unifying user-defined functions (UDFs) through matching opcode sequences and assigning universal identifiers. RF, KNN, MLP, and SVM models were trained, yielding impressive results, as detailed in Table 4.

5) LI et al. [12] MALWARE DETECTOR

We also retrained the Graph Neural Network (GNN)-based malware detector proposed by Li et al. [12]. This method integrates semantic information from Opcodes with structural information from function call graphs through three modules: an instruction-level module for semantic extraction, a structure-level module using GraphSAGE for graph embeddings, and a classification module with a Multi-Layer Perceptron (MLP) for malware detection. This

TABLE 4. Structural IoT malware detection results in percentage (%).

Evaluaton Metrics		Alasmary et al. [9] Detector				Gramac Detector [21]				Wu et al. [8] Detector				Li et al. [12] Detector
		RF	KNN	SVM	DNN	RF	KNN	SVM	DNN	RF	KNN	SVM	MLP	MLP
	Accuracy	**97.09**	96.72	87.01	95.24	**97.42**	95.53	86.16	91.86	98.61	98.60	**98.88**	98.70	**98.98**
	Precision	97.06	96.74	86.94	95.53	97.78	96.04	81.78	90.24	99.12	98.56	99.06	98.65	98.03
	Recall	97.11	96.67	87.07	95.61	94.00	91.80	80.87	87.64	97.24	98.46	98.57	98.46	98.88
	F1-Score	97.08	96.79	86.98	95.57	95.85	93.87	81.82	88.92	98.15	98.55	98.46	98.55	98.77

detector achieved an accuracy of 98.98%, precision of 98.03%, recall of 98.88%, and F1 score of 98.77%.

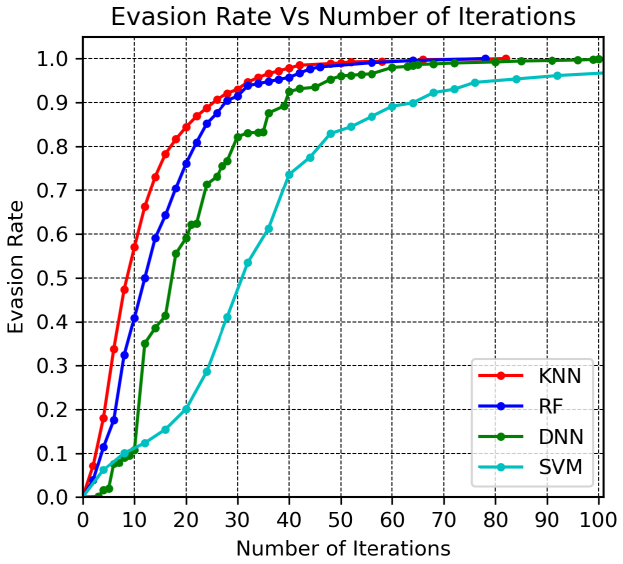

FIGURE 5. Evasion rate Vs Number of iterations.

C. DIVERSIFICATION-BASED ADVERSARIAL ATTACK

To evaluate the effectiveness of the proposed attack, we assembled a test set of 544 IoT ELF malware binaries from the x86 CPU architecture. Using the pre-trained substitute detector discussed previously, we generated adversarial examples with Algorithm 1 across Random Forest (RF), K-Nearest Neighbors (KNN), Deep Neural Networks (DNN), and Support Vector Machines (SVM) models. With a minimal attack cost, defined by the number of iterations and the percentage change in binary size, we produced effective adversarial examples. Specifically, the average percentage changes in the size of the modified binaries for KNN, RF, DNN, and SVM are 8.35%, 12.61%, 15.84%, and 22.51%, respectively. Figures 5 and 6 illustrate the variation in evasion rates with changes in the number of iterations and binary size, respectively.

To assess the robustness of each model, we tested the effectiveness of adversarial examples generated by one model on other models within the substitute detector. Our results

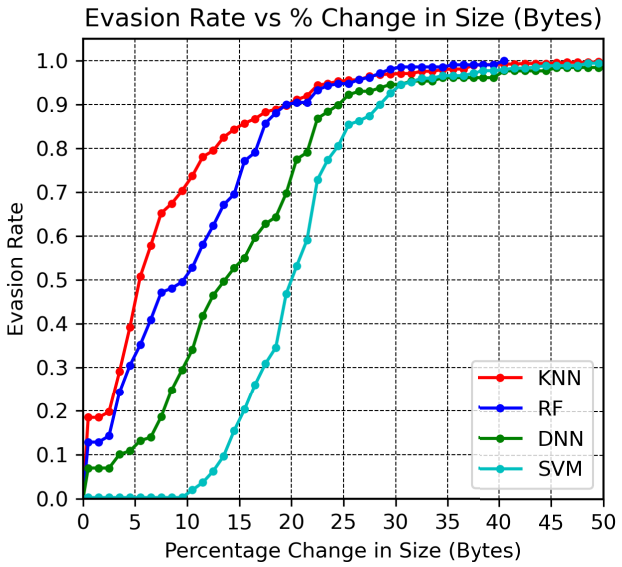

FIGURE 6. Evasion rate Vs % Change in size.

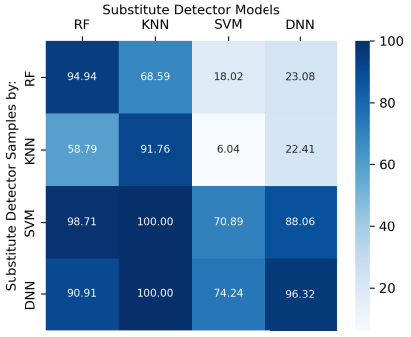

FIGURE 7. Transferability within the substitute detector.

show that the Support Vector Machine (SVM) model, with the lowest detection rate, is the most robust against adversarial examples from other models. In contrast, despite having high detection rates, the K-Nearest Neighbors (KNN) model is the least robust. Figure 7 illustrates the transferability of adversarial examples generated by one model to others within the substitute detector.

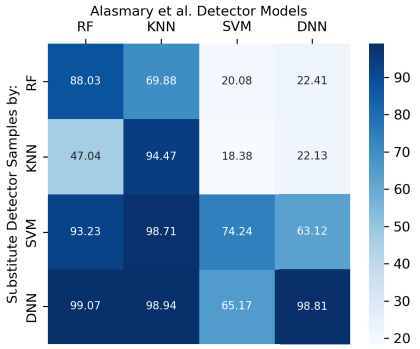

FIGURE 8. Evasion rate on [9] detector.

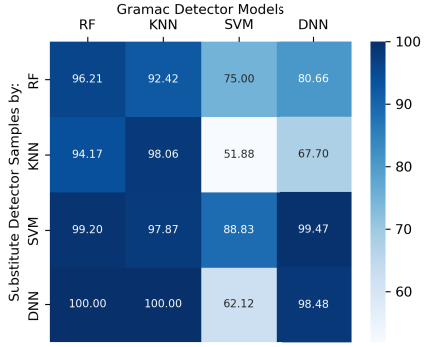

FIGURE 9. Evasion rate on [21] detector.

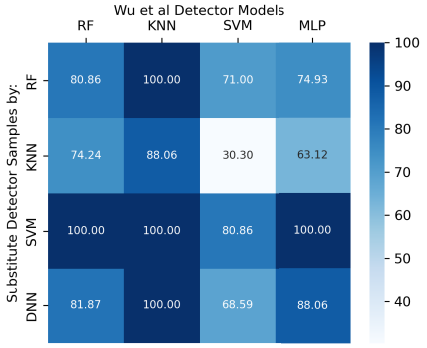

FIGURE 10. Evasion rate on [8] detector.

D. TRANSFERABILITY ANALYSIS

1) EVADING THE STRUCTURAL MALWARE DETECTORS

First, we tested the generated adversarial examples on the detector by Alasmary et al. [9], achieving high evasion rates of up to 99.07%. The SVM and DNN models proved more resilient compared to KNN and RF. Figure 8 shows a heatmap demonstrating how samples generated by the substitute detector models deceive the Alasmary et al. [9] detector.

We also evaluated our generated samples on the Gramac detector [21], achieving evasion rates of up to 100%, with the lowest being 51.88%. The results are detailed in Figure 9.

Similarly, the adversarial examples were successful on the Wu et al. [8] detector, achieving evasion rates of up to 100% with a minimum of 30.30%. Detailed results are shown in Figure 10.

The GNN-based detector by Li et al. [12] proved the most resilient compared to the other detectors. The adversarial examples generated by the SVM model were the most effective, attaining an evasion rate of 74% on the GNN-based detector. Samples generated by the RF, DNN, and KNN models achieved evasion rates of 62.01%, 53.79%, and 30.31%, respectively.

In further experiments, we evaluated how limiting the allowed change in binary size affects the evasion rate. Our results show that generating adversarial examples increases the binary size, potentially impacting performance. Consequently, we restricted the maximum allowable change in binary size to 30% and studied its effect on the evasion rate of the generated samples across the four structural malware detectors. The results, detailed in Table 5, indicate evasion rates exceeding 97% for some detectors.

2) EVADING ESMAEILI et al. [15] ADVERSARIAL DETECTOR

Esmaeili et al. [15] generated adversarial control flow graphs (CFGs) by merging the CFGs of selected benign IoT samples with those of the target malware. They then trained a GNN-based adversarial detector to learn the characteristics of benign CFGs, enabling it to identify and filter out adversarial CFGs before classification. We tested the CFGs of our generated adversarial examples on this detector to determine whether they would be flagged as adversarial. The adversarial detector did not flag our adversarial CFGs and misclassified 95.9% of them as benign.

3) EVADING COMMERCIAL ANTIVIRUS ENGINES

To further evaluate the effectiveness of our attack approach, we submitted the generated adversarial examples to VirusTotal [23] and compared the detection reports with that of the original malware samples. The original malware samples were flagged as malicious by an average of 44.84 antivirus engines. In contrast, the adversarial samples generated by SVM, DNN, RF, and KNN were flagged by 29.00, 28.97, 29.35, and 29.21 engines, respectively. This indicates that more than 15 antivirus engines were deceived by our adversarial examples. Detailed results are shown in Figure 11.

E. COMPARISON WITH EXISTING SIMILAR WORK

Additionally, we compared the adversarial CFGs generated by Esmaeili et al. [15] with those of our generated examples. Esmaeili et al. employed an approach similar to GEA [13], focusing on feature-space manipulations rather than generating executable adversarial examples, and argued theoretically that such an attack could be implemented in the problem space. Our analysis shows that their adversarial CFGs introduced significantly more nodes, edges, and instructions than ours, leading to a substantial increase in binary size, as demonstrated in Figure 12.

TABLE 5. Transferability across detectors with 30% maximum size change.

Substitute Detector		Alasmary et al. [9] Detector				Gramac Detector [21]				Wu et al. [8] Detector				Li et al. [12] Detector
		RF	KNN	SVM	DNN	RF	KNN	SVM	DNN	RF	KNN	SVM	MLP	MLP
	RF	82.24	62.13	11.71	16.32	**90.78**	87.09	72.43	71.27	78.55	**95.89**	60.56	70.02	59.22
	KNN	47.04	**94.01**	18.43	18.61	**91.95**	**95.15**	46.31	61.34	71.93	80.95	28.85	60.21	28.30
	SVM	**92.23**	**94.25**	69.54	59.75	**90.98**	89.37	79.26	87.08	**97.69**	**97.89**	74.42	**97.09**	**70.17**
	DNN	**94.88**	**95.48**	62.47	82.04	**95.99**	**96.92**	57.55	90.10	79.56	**97.82**	63.15	85.16	50.79

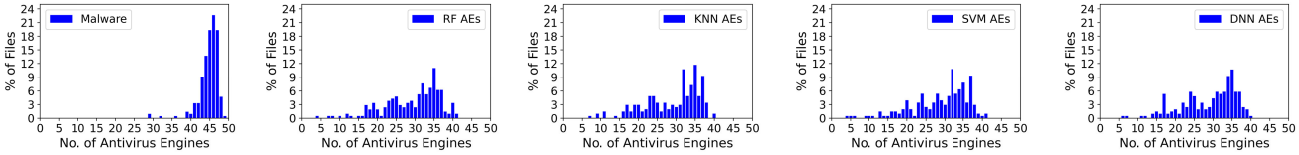

FIGURE 11. Detection rate of original and manipulated IoT malware by antivirus engines.

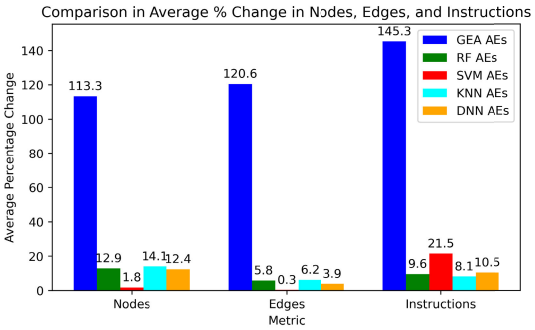

FIGURE 12. Comparison with Esmaeili et al. [15] approach.

V. CONCLUSION

Despite significant advancements, machine learning-based malware detection systems remain highly susceptible to adversarial attacks that disguise malware as benignware. This study evaluated the robustness of structural IoT malware detectors against such attacks through binary-level manipulations. We introduced a novel, functionality-preserving black-box attack that successfully deceived four structural detectors, an adversarial detector, and several commercial antivirus engines, achieving up to 100% evasion with minimal binary size increase. These findings underscore the urgent need for more resilient and adaptive cybersecurity defenses.

However, our study focused on structural IoT malware detectors, excluding other types of detectors that also merit investigation. Additionally, challenges in the disassembly-reassembly process led to failures with some malware samples. Future work will employ a more advanced disassembly-reassembly tool and expand the scope to assess the robustness of a broader range of detection systems. Furthermore, we plan to explore defense strategies against adversarial attacks on malware detection.

REFERENCES

[1] Y.-T. Lee, T. Ban, T.-L. Wan, S.-M. Cheng, R. Isawa, T. Takahashi, and D. Inoue, "Cross platform IoT-malware family classification based on printable strings," in *Proc. IEEE 19th Int. Conf. Trust, Secur. Privacy Comput. Commun. (TrustCom)*, Dec. 2020, pp. 775–784.

[2] M. Al-Fawa'Reh, J. Abu-Khalaf, P. Szewczyk, and J. J. Kang, "MalBoT-DRL: Malware botnet detection using deep reinforcement learning in IoT networks," *IEEE Internet Things J.*, vol. 11, no. 6, pp. 9610–9629, Mar. 2024.

[3] M. Ghahramani, R. Taheri, M. Shojafar, R. Javidan, and S. Wan, "Deep image: A precious image based deep learning method for online malware detection in IoT environment," *Internet Things*, vol. 27, Jul. 2024, Art. no. 101300.

[4] E. Odat and Q. M. Yaseen, "A novel machine learning approach for Android malware detection based on the co-existence of features," *IEEE Access*, vol. 11, pp. 15471–15484, 2023.

[5] H. H. Al-Khshali, M. Ilyas, F. Sohrab, and M. Gabbouj, "Malware detection with subspace learning-based one-class classification," *IEEE Access*, vol. 12, pp. 81017–81029, 2024.

[6] M. Nobakht, R. Javidan, and A. Pourebrahimi, "SIM-FED: Secure IoT malware detection model with federated learning," *Comput. Electr. Eng.*, vol. 116, May 2024, Art. no. 109139.

[7] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. Nicholas, "Malware detection by eating a whole EXE," in *Proc. AAAI*, Oct. 2018, pp. 1–9.

[8] C.-Y. Wu, T. Ban, S.-M. Cheng, T. Takahashi, and D. Inoue, "IoT malware classification based on reinterpreted function-call graphs," *Comput. Secur.*, vol. 125, Feb. 2023, Art. no. 103060.

[9] 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 Things J.*, vol. 6, no. 5, pp. 8977–8988, Oct. 2019.

[10] C.-Y. Wu, T. Ban, S.-M. Cheng, B. Sun, and T. Takahashi, "IoT malware detection using function-call-graph embedding," in *Proc. 18th Int. Conf. Privacy, Secur. Trust (PST)*, Dec. 2021, pp. 1–9.

[11] Q.-D. Ngo, H.-T. Nguyen, V.-H. Le, and D.-H. Nguyen, "A survey of IoT malware and detection methods based on static features," *ICT Exp.*, vol. 6, no. 4, pp. 280–286, Dec. 2020.

[12] C. Li, G. Shen, and W. Sun, "Cross-architecture Intemet-of-Things malware detection based on graph neural network," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, Jul. 2021, pp. 1–7.

[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 *Proc. IEEE 39th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jul. 2019, pp. 1296–1305.

[14] 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 *Proc. Int. Conf. Comput. Data Soc. Netw.*, Nov. 2019, pp. 268–281.

[15] B. Esmaeili, A. Azmoodeh, A. Dehghantanha, G. Srivastava, H. Karimipour, and J. C.-W. Lin, "A GNN-based adversarial Internet of Things malware detection framework for critical infrastructure: Studying gafgyt, mirai, and tsunami campaigns," *IEEE Internet Things J.*, vol. 11, no. 16, pp. 26826–26836, Jul. 2023.

[16] J. Sándor, R. Nagy, and L. Buttyán, "Increasing the robustness of a machine learning-based IoT malware detection method with adversarial training," in *Proc. ACM Workshop Wireless Secur. Mach. Learn.*, Jun. 2023, pp. 3–8.

[17] A. Abusnaina, A. Anwar, S. Alshamrani, A. Alabduljabbar, R. Jang, D. Nyang, and D. Mohaisen, "ML-based IoT malware detection under adversarial settings: A systematic evaluation," 2021, *arXiv:2108.13373*.

[18] A. Khormali, A. Abusnaina, S. Chen, D. Nyang, and A. Mohaisen, "COPYCAT: Practical adversarial attacks on visualization-based malware detection," 2019, *arXiv:1909.09735*.

[19] D. Quoc-Ngo, H. Trung-Nguyen, D. Viet-Nguyen, M. Cong-Dinh, T. Anh-Phung, and T. Quy-Bui, "Adversarial attack and defense on graph-based IoT botnet detection approach," in *Proc. Int. Conf. Electr., Commun., Comput. Eng. (ICECCE)*, Jun. 2021, pp. 1–6.

[20] S. Lundberg and S. Lee, "A unified approach to interpreting model predictions," in *Proc. Adv. Neural Inf. Process. Syst.*, Dec. 2017, pp. 4768–4777.

[21] D. Vij, V. Balachandran, T. Thomas, and R. Surendran, "GRAMAC: A graph based Android malware classification mechanism," in *Proc. 10th ACM Conf. Data Appl. Secur. Privacy*, Mar. 2020, pp. 156–158.

[22] S. Puneeth, S. Lal, M. Pratap Singh, and B. S. Raghavendra, "RMDNet-deep learning paradigms for effective malware detection and classification," *IEEE Access*, vol. 12, pp. 82622–82635, 2024.

[23] Developers. (2024). *VirusTotal*. [Online]. Available: https://www.virustotal.com

[24] A. D. Raju, I. Y. Abualhaol, R. S. Giagone, Y. Zhou, and S. Huang, "A survey on cross-architectural IoT malware threat hunting," *IEEE Access*, vol. 9, pp. 91686–91709, 2021.

[25] T.-L. Wan, T. Ban, S.-M. Cheng, Y.-T. Lee, B. Sun, R. Isawa, T. Takahashi, and D. Inoue, "Efficient detection and classification of Internet-of-Things malware based on byte sequences from executable files," *IEEE Open J. Comput. Soc.*, vol. 1, pp. 262–275, 2020.

[26] H. S. Anderson and P. Roth, "EMBER: An open dataset for training static PE malware machine learning models," 2018, *arXiv:1804.04637*.

[27] F. Shahzad and M. Farooq, "ELF-miner: Using structural knowledge and data mining methods to detect new (Linux) malicious executables," *Knowl. Inf. Syst.*, vol. 30, no. 3, pp. 589–612, Mar. 2012.

[28] H. Lee, S. Kim, D. Baek, D. Kim, and D. Hwang, "Robust IoT malware detection and classification using opcode category features on machine learning," *IEEE Access*, vol. 11, pp. 18855–18867, 2023.

[29] Y. Liu, H. Fan, J. Zhao, J. Zhang, and X. Yin, "Efficient and generalized image-based CNN algorithm for multi-class malware detection," *IEEE Access*, vol. 12, pp. 104317–104332, 2024.

[30] A. Mohaisen, O. Alrawi, and M. Mohaisen, "AMAL: High-fidelity, behavior-based automated malware analysis and classification," *Comput. Secur.*, vol. 52, pp. 251–266, Jul. 2015.

[31] M. Abdelsalam, R. K. Yufei, Huang, and R. Sandhu, "Malware detection in cloud infrastructures using convolutional neural networks," in *Proc. IEEE 11th Int. Conf. Cloud Comput. (CLOUD)*, Jul. 2018, pp. 162–169.

[32] 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," 2018, *arXiv:1802.04528*.

[33] O. Suciu, S. E. Coull, and J. Johns, "Exploring adversarial examples in malware detection," in *Proc. IEEE Secur. Privacy Workshops (SPW)*, May 2019, pp. 8–14.

[34] S. Verwer, A. Nadeem, C. Hammerschmidt, L. Bliek, A. Al-Dujaili, and U.-M. O'Reilly, "The robust malware detection challenge and greedy random accelerated multi-bit search," in *Proc. 13th ACM Workshop Artif. Intell. Secur.*, Nov. 2020, pp. 61–70.

[35] A. Al-Dujaili, A. Huang, E. Hemberg, and U.-M. O'Reilly, "Adversarial deep learning for robust detection of binary encoded malware," in *Proc. IEEE Secur. Privacy Workshops (SPW)*, May 2018, pp. 76–82.

[36] D. Park, H. Khan, and B. Yener, "Generation & evaluation of adversarial examples for malware obfuscation," in *Proc. 18th IEEE Int. Conf. Mach. Learn. Appl. (ICMLA)*, Dec. 2019, pp. 1283–1290.

[37] X. Liu, J. Zhang, Y. Lin, and H. Li, "ATMPA: Attacking machine learning-based malware visualization detection methods via adversarial examples," in *Proc. IEEE/ACM 27th Int. Symp. Quality Service (IWQoS)*, Jun. 2019, pp. 1–10.

[38] L. Demetrio, B. Biggio, G. Lagorio, F. Roli, and A. Armando, "Explaining vulnerabilities of deep learning to adversarial malware binaries," 2019, *arXiv:1901.03583*.

[39] 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 *Proc. 26th Eur. Signal Process. Conf. (EUSIPCO)*, Sep. 2018, pp. 533–537.

[40] K. Aryal, M. Gupta, M. Abdelsalam, and M. Saleh, "Intra-section code cave injection for adversarial evasion attacks on windows PE malware file," 2024, *arXiv:2403.06428*.

[41] 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," *ACM Trans. Privacy Secur.*, vol. 24, no. 4, pp. 1–31, Nov. 2021.

[42] K. Lucas, M. Sharif, L. Bauer, M. K. Reiter, and S. Shintre, "Malware makeover: Breaking ML-based static analysis by modifying executable bytes," 2019, *arXiv:1912.09064*.

[43] K. Zhao, H. Zhou, Y. Zhu, X. Zhan, K. Zhou, J. Li, L. Yu, W. Yuan, and X. Luo, "Structural attack against graph based Android malware detection," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2021, pp. 3218–3235.

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

[45] M. Kawai, K. Ota, and M. Dong, "Improved MalGAN: Avoiding malware detector by leaning cleanware features," in *Proc. Int. Conf. Artif. Intell. Inf. Commun. (ICAIIC)*, Feb. 2019, pp. 040–045.

[46] W. Hu and Y. Tan, "Black-box attacks against RNN based malware detection algorithms," in *Proc. 32nd AAAI Workshops*, Jun. 2018, pp. 1–7.

[47] I. Rosenberg, A. Shabtai, L. Rokach, and Y. Elovici, "Generic black-box end-to-end attack against state-of-the-art API call-based malware classifiers," in *Proc. Int. Symp. Res. Attacks Intrusions Defenses*, Sep. 2018, pp. 490–510.

[48] I. Rosenberg, A. Shabtai, Y. Elovici, and L. Rokach, "Query-efficient black-box attack against sequence-based malware classifiers," in *Proc. 36th Annu. Comput. Secur. Appl. Conf.*, Dec. 2020, pp. 611–626.

[49] L. Zhang, P. Liu, Y.-H. Choi, and P. Chen, "Semantics-preserving reinforcement learning attack against graph neural networks for malware detection," *IEEE Trans. Dependable Secure Comput.*, vol. 20, no. 2, pp. 1390–1402, Feb. 2023.

[50] H. S. Anderson, A. Kharkar, B. Filar, and P. Roth, "Evading machine learning malware detection," in *Proc. Black Hat*, Las Vegas, NV, USA, Jul. 2017, pp. 1–6.

[51] C. Wu, J. Shi, Y. Yang, and W. Li, "Enhancing machine learning based malware detection model by reinforcement learning," in *Proc. 8th Int. Conf. Commun. Netw. Secur. (ICCNS)*, Nov. 2018, pp. 74–78.

[52] 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 Trans. Inf. Forensics Security*, vol. 15, pp. 987–1001, 2020.

[53] R. L. Castro, C. Schmitt, and G. Dreo, "AIMED: Evolving malware with genetic programming to evade detection," in *Proc. 18th IEEE Int. Conf. Trust, Secur. Privacy Comput. Commun. 13th IEEE Int. Conf. Big Data Sci. Eng. (TrustCom/BigDataSE)*, Aug. 2019, pp. 240–247.

[54] X. Wang and R. Miikkulainen, "MDEA: Malware detection with evolutionary adversarial learning," in *Proc. IEEE Congr. Evol. Comput. (CEC)*, Jul. 2020, pp. 1–8.

[55] R. L. Castro, C. Schmitt, and G. D. Rodosek, "ARMED: How automatic malware modifications can evade static detection?" in *Proc. 5th Int. Conf. Inf. Manage. (ICIM)*, Mar. 2019, pp. 20–27.

[56] J. Yuan, S. Zhou, L. Lin, F. Wang, and J. Cui, "Black-box adversarial attacks against deep learning-based malware binaries detection with GAN," in *Proc. ECAI*, 2020, pp. 2536–2542.

[57] F. Zhong, X. Cheng, D. Yu, B. Gong, S. Song, and J. Yu, "MalFox: Camouflaged adversarial malware example generation based on conv-GANs against black-box detectors," *IEEE Trans. Comput.*, vol. 73, no. 4, pp. 980–993, Jan. 2023.

[58] K. Lucas, M. Sharif, L. Bauer, M. K. Reiter, and S. Shintre, "Malware makeover: Breaking ML-based static analysis by modifying executable bytes," in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, May 2021, pp. 744–758.
[59] 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 Trans. Privacy Secur.*, vol. 22, no. 2, pp. 1–34, Apr. 2019.
[60] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck, "Drebin: Effective and explainable detection of Android malware in your pocket," in *Proc. NDSS*, 2014, pp. 23–26.
[61] H. Bostani and V. Moonsamy, "EvadeDroid: A practical evasion attack on machine learning for black-box Android malware detection," *Comput. Secur.*, vol. 139, Jan. 2024, Art. no. 103676.
[62] Z. Fang, J. Wang, B. Li, S. Wu, Y. Zhou, and H. Huang, "Evading anti-malware engines with deep reinforcement learning," *IEEE Access*, vol. 7, pp. 48867–48879, 2019.
[63] J. Chen, J. Jiang, R. Li, and Y. Dou, "Generating adversarial examples for static PE malware detector based on deep reinforcement learning," *J. Phys., Conf. Ser.*, vol. 1575, no. 1, Jun. 2020, Art. no. 012011.
[64] B. Chen, Z. Ren, C. Yu, I. Hussain, and J. Liu, "Adversarial examples for CNN-based malware detectors," *IEEE Access*, vol. 7, pp. 54360–54371, 2019.
[65] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "SoK: Automated software diversity," in *Proc. IEEE Symp. Secur. Privacy*, May 2014, pp. 276–291.
[66] S. Blazy and A. Trieu, "Formal verification of control-flow graph flattening," in *Proc. 5th ACM SIGPLAN Conf. Certified Programs Proofs*, Jan. 2016, pp. 176–187.
[67] H. Wang, S. Wang, D. Xu, X. Zhang, and X. Liu, "Generating effective software obfuscation sequences with reinforcement learning," *IEEE Trans. Dependable Secure Comput.*, vol. 19, no. 3, pp. 1900–1917, Dec. 2022.
[68] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SOK: (State of) the art of war: Offensive techniques in binary analysis," in *Proc. S&P*, May 2016, pp. 138–157.
[69] R. Team. (2024). *Radare2 Github Repository*. [Online]. Available: https://github.com/radare/radare2
[70] A. Hagberg, P. Swart, and D. Chult, "Exploring network structure, dynamics, and function using NetworkX," in *Proc. SciPy*, Jun. 2008, pp. 11–15.
[71] S. Wang, P. Wang, and D. Wu, "Reassembleable disassembling," in *Proc. USENIX Secur.*, Aug. 2015, pp. 627–642.

MAINA BERNARD MWANGI (Graduate Student Member, IEEE) received the M.Sc. degree in computer science and information engineering from the National Taiwan University of Science and Technology, Taipei, Taiwan, in 2021, where he is currently pursuing the Ph.D. degree in computer science and information engineering. His research interests include the IoT and AI security.

SHIN-MING CHENG (Member, IEEE) received the B.S. and Ph.D. degrees in computer science and information engineering from National Taiwan University, Taipei, Taiwan, in 2000 and 2007, respectively. Since 2012, he has been a Faculty Member of the Department of Computer Science and Information Engineering, National Taiwan University of Science and Technology, Taipei, where he is currently a Professor. Since 2022, he has been the Deputy Director-General in Administration for Cyber Security, Ministry of Digital Affairs. His current interests include mobile network security, the IoT system security, malware analysis, and AI robustness. He has received the IEEE Trustcom 2020 Best Paper Awards.