VILNIUS UNIVERSITY

FACULTY OF MATHEMATICS AND INFORMATICS

DATA SCIENCE STUDY PROGRAMME

Master's thesis

Adversarial Machine Learning for Malware Obfuscation

Priešiškas mašininis mokymasis kenkėjiškoms programoms užmaskuoti

Justas Kudirka

Supervisor : Prof., Dr. Olga Kurasova

Vilnius
2026

Contents

List of abbreviations 5

1 Literature Review 7
1.1 Foundations of malware obfuscation 7
1.1.1 Common obfuscation techniques 7
1.1.2 Obfuscation for machine learning 8
1.2 Adversarial machine learning 9
1.2.1 Attack categorization and threat models 9
1.2.2 Constraints of adversarial examples in the security domain 9
1.3 Obfuscation techniques in adversarial machine learning 10
1.3.1 Feature manipulation and perturbation strategies 10
1.3.2 Purpose of exploring black-box evasion approach 10
1.4 Generative models for adversarial malware 11
1.4.1 GAN approach 11
1.4.2 Reinforcement learning approach 12
1.4.3 Selected models and justification 13
1.5 Malware detection dataset 13

2 Investigation & Experiment 14
2.1 Preprocessing 14
2.2 Gradient boosted decision tree model 15
2.3 MalGAN architecture 16
2.4 Reinforcement learning architecture 17
2.5 Results 18

3 Conclusions 23

4 Appendices 27

Abstract

Machine learning-based malware detectors have significantly improved the ability to identify malicious software. However, the attackers have responded by using machine learning to obfuscate malware and evade detection. As such, for each new detection model that is deployed, more advanced adversarial malware generators eventually emerge to defeat it. This continuous cat-and-mouse game requires researchers to study adversarial obfuscation methods. Gaining a better understanding of how different methods improve evasion probability allows researchers to strengthen malware defenses and anticipate the attackers' strategies.

To provide a realistic baseline, this thesis uses the EMBER dataset and a Gradient Boosted Decision Tree classifier as the target black-box detector. The GBDT model from the EMBER framework was chosen for its strong detection performance. Two different adversarial malware obfuscation architectures were evaluated: MalGAN and `malware_rl`. First, MalGAN is the implementation of a generative adversarial network. It is designed to generate obfuscated malware examples. It comprises three parts: a generator, a discriminator, and a black-box detector. The main idea is for the generator and the discriminator to go back and forth, with occasional black-box queries. This allows the generator to learn how to fool the black-box detector. On the other hand, the `malware_rl` framework is a reinforcement learning algorithm, which uses an agent to interact with the black-box model. The agent takes discrete actions to interact with the environment, including the detector, and receives outputs and rewards. These two approaches were chosen because they represent the foundations of their respective architectures. This allows for a faithful comparison of two different architectures.

This thesis shows that MalGAN and the RL-based approach achieved high evasion rates. Recommendations are made to combine these two architectures into a single architecture by leveraging their strengths.

Keywords: malware obfuscation, adversarial machine learning, reinforcement learning, generative adversarial networks.

Santrauka

Mašininio mokymosi pagrindu sukurti kenkėjiškų programų detektoriai gerokai pagerino gebėjimą atpažinti kenkėjišką programinę įrangą. Tačiau užpuolikai į tai reagavo naudodami mašininį mokymąsi, kad paslėptų kenkėjiškas programas ir išvengtų aptikimo. Todėl kiekvienam naujam aptikimo modeliui galiausiai atsiranda pažangesni kenkėjiškų programų generatoriai, kurie ją įveikia. Šis nuolatinis katės ir pelės žaidimas reikalauja, kad tyrėjai tyrinėtų priešiškus maskavimo metodus. Geresnis supratimas apie tai, kaip skirtingi metodai pagerina aptikimo tikimybę, leidžia tyrėjams sustiprinti kenkėjiškų programų apsaugą ir numatyti užpuolikų strategijas.

Siekiant pateikti realų pradinį tašką, šiame darbe kaip tikslinis juodosios dėžės detektorius naudojamas EMBER duomenų rinkinys ir „Gradient Boosted Decision Tree“ (GBDT) klasifikatorius. EMBER platformos GBDT modelis buvo pasirinktas dėl didelio aptikimo našumo. Buvo įvertintos dvi skirtingos kenkėjiškų programų maskavimo architektūros: „MalGAN“ ir „`malware_rl`“. Pirma, „MalGAN“ yra generatyvaus priešiško tinklo įgyvendinimas. Jis skirtas generuoti maskuojamus kenkėjiškų programų pavyzdžius. Jį sudaro trys dalys: generatorius, diskriminatorius ir juodosios dėžės detektorius. Pagrindinė idėja yra ta, kad generatorius ir diskriminatorius veiktų pirmyn ir atgal, retkarčiais atlikdami juodosios dėžės užklausas. Tai leidžia generatoriui išmokti apgauti juodosios dėžės detektorių. Kita vertus, `malware_rl` sistema yra sustiprinto mokymosi algoritmas, kuris naudoja agentą sąveikai su juodosios dėžės modeliu. Agentas atlieka atskirus veiksmus, kad sąveikautų su aplinka, įskaitant detektorių, ir gauna rezultatus bei atlygį. Šie du metodai buvo pasirinkti, nes jie atspindi jų atitinkamų architektūrų pagrindus. Tai leidžia tiksliai palyginti dvi skirtingas architektūras.

Šiame darbe parodoma, kad „MalGAN“ ir RL pagrįstas metodas pasiekė aukštus apėjimo rodiklius. Rekomenduojama sujungti šias dvi architektūras į vieną, išnaudojant jų stipriąsias puses.

Raktiniai žodžiai: kenkėjiškų programų maskavimas, priešiškas mašininis mokymasis, skatinamasis mokymasis, generacinės modeliavimo sistemos.

List of abbreviations

GAN	-	Generative Adversarial Network
RL	-	Reinforcement Learning
PE	-	Portable Executable file
EMBER	-	Elastic Malware Benchmark for Empowering Researchers
JSONL	-	JSON Lines file format
GBDT	-	Gradient-Boosted Decision Tree
FPR	-	False Positive Rate
ACER	-	Actor-Critic with Experience Replay

Introduction

Microsoft Windows has become an imperative part of everyday life. Universities, households, and businesses rely on the operating system to perform critical, routine tasks. As our dependence on this technology grew, so did the interest of malicious actors in abusing the system. One of these methods is to leverage the Portable Executable (PE) file format. This file format has been used as the source of various attacks. As a result, antivirus systems attempt to intercept these attacks and prevent them from executing. In recent years, machine learning models have been incorporated into various malware detection systems. However, adversarial actors also incorporated machine learning models in malware obfuscation tactics. This led to the creation of advanced malware samples capable of evading complex antivirus systems. Each new detector will eventually be broken by more advanced versions of malware generators. To keep up with the pace, creating malware obfuscation systems is imperative, so that we can train our models on newly generated samples and harden these systems. However, there is a lack of standardization when datasets are involved. Malware datasets require at least two sample categories: malicious and benign. However, benign examples are usually proprietary. In addition, malicious samples can be obtained independently by each researcher, leading to a lack of standardization. This raises the question of whether we can trust papers that claim to be improvements on an older malware generation architecture or an iteration of a detector model, since each paper uses its own dataset.

The main goal of this thesis is to compare different machine learning obfuscation architectures to perform evasive attacks on malware classification models. In order to fulfill the purpose of the thesis, it is important to complete these objectives:

1. Choose and train a malware detection model as the target model to evade.
2. Compare generative models on the same malware dataset and evaluate their evasion success.
3. Provide recommendations for improvements.

1 Literature Review

1.1 Foundations of malware obfuscation

Early detection of computer viruses relied heavily on signature-based methods. Detection methods usually compare the byte patterns of known malware against those of incoming files. This method excels against known threats; however, it struggles with novel or obfuscated malware. As the number of malicious software grew, researchers began exploring machine learning techniques to generalize detection. One of the first learning algorithms used was a Naïve Bayes classifier [26]. It used features extracted from programs, such as program headers and strings. These were then used to train the Naïve Bayes classifier to identify unseen malicious files. It detected previously unseen malware more effectively than signature-based methods alone. Due to the growing popularity of learning algorithms in malware detection, the static analysis approach, which inspects executable files without running them, has become an important method in malware detection research [17].

However, static methods have their limitations. Opaque constants, which are primitives that allow the attacker to load a constant into a register in such a way that static analysis cannot determine its value. This allows the attacker to obscure the program control flow and fool a static code analyzer. Due to these and similar limitations, another method emerged: dynamic analysis. During dynamic analysis, suspicious programs are executed in sandboxes to observe runtime behavior and detect malicious software. However, this analysis method is resource-intensive and can be fooled by environment-aware malware. Therefore, modern anti-malware solutions usually combine static and dynamic analysis: in the first part, the bulk of samples is evaluated using static analysis, and the dynamic part is used for more elusive threats. In addition, the efficiency of a detector has been shown to largely depend on the features used and the quality of the training data [15, 22].

1.1.1 Common obfuscation techniques

For malware to successfully affect its targets, it must avoid detection. To achieve this, malware developers use obfuscation techniques. These techniques help the program appear benign to the detector during scanning. These obfuscation methods will be categorized into three groups: code-level, binary-level, and structural-level obfuscations.

Code-level obfuscation primarily focuses on changing the logic and syntax of the program to evade signature-based detection. A common method is instruction substitution, where developers rewrite code to preserve functionality while making it appear different. For instance, a simple increment such as `sum = sum + 1` can be rewritten as `sum = sum - -1` or `sum += 1`. While the execution result remains identical, these changes appear differently in static signatures. This is often paired with dead-code insertion, which introduces unreachable or non-functional instructions to increase file size and entropy, thereby yielding different results when performing statistical analysis. Lastly, code reordering allows developers to rearrange basic blocks and instructions without altering the control flow, successfully changing the code layout and memory addresses [16].

Binary-level obfuscation targets the executable payload itself to hide its true intent from static analysis tools. Packing is a prevalent technique in which executable sections are compressed or encrypted and paired with an unpacking stub that restores the code at runtime. This process dramatically alters static features, such as byte histograms and section characteristics. Next, encryption can be applied more selectively to specific functions or data structures to hinder reverse engineering. One of the more advanced binary-level methods is virtualization, which translates native machine code into a custom bytecode for a virtual machine. This bytecode requires a specialized interpreter for execution, providing strong obfuscation at the cost of increased performance overhead [1, 16].

Structure-level obfuscation involves manipulating the file format and metadata without affecting code execution. Section manipulation allows developers to modify portable executable (PE) headers, such as renaming the `.text` section to `.data` or altering section flags. Similarly, import table manipulation involves adding benign imports, removing unused ones, or reordering entries. Because many machine learning-based detectors rely heavily on import table features, these modifications can be highly effective for evasion. Finally, resource manipulation targets non-executable elements such as icons, version information, and manifests, which contribute to the malware's overall file size and entropy profile [1].

1.1.2 Obfuscation for machine learning

Traditional obfuscation techniques were designed to evade signature-based detection and to make it harder to manually reverse-engineer the code. However, machine learning-based detectors extract multiple features from a program that represents the entire executable file. This makes the detectors more robust to simple obfuscations. The improved detectors motivated attackers to develop adversarial obfuscation techniques specifically targeting machine learning vulnerabilities. Common attacks:

- **Feature-space attacks:** performs malware modifications to alter specific features used by machine learning detectors. For example, if a detector relies on import table features, adding benign imports can improve evasion. Feature-space attacks benefit from knowledge of the feature extraction process [3].

- **Gradient-based attacks:** these attacks are carried out on differentiable detectors and assuming white-box targets. This allows the attacker to target the loss function. The attack is carried out by computing gradients that minimize perturbations to the input features, with the goal of causing a misclassification. This approach is common for images but difficult for malware because of the requirement to preserve functionality [3, 19].

- **Learning-based attacks:** these attacks are carried out by machine learning models and are very powerful because they can discover complex evasion strategies that generalize across malware families. Some common machine learning architectures are **generate adversarial networks**, which train a generator to produce adversarial feature vectors under a substitute detector, and **reinforcement learning**, which trains an agent to select specific sequences of obfuscation actions that maximize evasion [19].

1.2 Adversarial machine learning

Adversarial machine learning studies the behavior of machine learning systems when they encounter adversarial examples. These inputs are perturbations designed to cause misclassification. Deep neural networks are vulnerable to small perturbations that cause confident misclassifications. Intense research into the properties of adversarial examples began once they were discovered in computer vision. Adversarial examples have shown several properties: their adversarial capabilities transfer across different models trained, they can be generated efficiently using gradient-based methods, and they reveal fundamental limitations in how neural networks learn decision boundaries [6, 23].

1.2.1 Attack categorization and threat models

Adversarial attacks can be categorized along multiple dimensions. However, a fundamental distinction exists between white-box and black-box attacks. White-box attacks assume complete knowledge of the target model's architecture, parameters, and training data. This knowledge enables the use of gradient-based optimization methods. In contrast, black-box attacks assume no knowledge of the model's inner workings and must rely solely on querying the model and observing its outputs [3, 21, 24].

Another important dimension is the attack objective. Targeted attacks aim to cause misclassification into a specific target class. Untargeted attacks attempt to cause any misclassification. Considering when the attacks are performed, we can further distinguish attacks into evasion or poisoning attacks. Poisoning attacks target the training data, injecting adversarial examples into the target model's dataset. Meanwhile, evasion attacks target the deployed models themselves by leveraging generated adversarial examples [5]. In the malware domain, untargeted evasion attacks represent the primary threat model [7, 25].

This thesis focuses exclusively on evasive, untargeted black-box attacks, as they represent the most realistic threat scenario in which malware authors attempt to bypass existing detection models.

1.2.2 Constraints of adversarial examples in the security domain

The application of adversarial machine learning to the security domain, specifically malware detection, presents unique challenges. These challenges are distinctly different from computer vision examples. Image classification typically involves pixel perturbations. These perturbations have only a perceptual similarity constraint. This means the image must be difficult for a human to distinguish [4]. However, adversarial malware must satisfy a few strict validity constraints: the modified file must conform to the portable executable file format specification, the malware must retain its original functionality, and modifications should ideally be minimal to avoid detection by other means [7, 25].

Due to the previously mentioned constraints, the adversarial example generation method has to be fundamentally altered. Gradient-based methods developed for continuous input spaces, such as images, cannot be directly applied to discrete and structured inputs like executable files. This requires additional mechanisms to ensure the validity and functionality of the modified example

[12]. This has led to the development of domain-specific adversarial generation techniques that specifically use the previously mentioned structural and functional constraints [1, 14].

1.3 Obfuscation techniques in adversarial machine learning

Black-box evasion attacks represent the most realistic threat model for adversarial malware generation. This is because real-world antivirus systems and malware detectors are treated as black boxes. The attackers do not have access to the model parameters, the architecture, or training data. This constraint forces attackers to adopt strategies that rely solely on model outputs obtained by querying the model. Common model outputs are binary classifications [24].

Several approaches have been developed for generating black-box adversarial malware. In black-box attacks, the attackers do not have the model itself, so it is common to use a substitute target model. In addition, to make the attacks query-efficient, the attackers also introduce several local substitute models that attempt to replicate the decision boundaries of the target detector. Another generative model is used to generate adversarial examples against these local substitute models and exploit the transferability to evade the target substitute model. In addition, different models optimize the perturbations by introducing a modification space [18, 24, 28].

1.3.1 Feature manipulation and perturbation strategies

There are two primary strategies for generating adversarial malware examples in the black-box setting: feature addition and feature removal. These strategies focus on creating perturbations that modify feature representations while maintaining executable validity and functionality [14].

Feature addition involves augmenting malware with benign-looking features or structures. This approach is generally safe for functionality preservation, as adding unused code or data rarely breaks existing functionality. Common feature addition techniques include appending unused imports to the import address table, adding new portable executable sections with benign content, inserting dead code sequences, and appending data to overlay sections [7, 9].

On the other hand, feature removal involves eliminating or modifying existing features. This approach is riskier, as removing critical functionality can render the malware non-functional. An example of critical functionality is application programming interface (API) calls, which are essential for malicious behavior. Because of this, most papers and practical adversarial malware generation systems prioritize adding features over removing them [14].

1.3.2 Purpose of exploring black-box evasion approach

This thesis focuses on black-box evasion attacks for several practical and theoretical reasons. First, real-world deployment scenarios usually involve black-box settings. For example, commercial antivirus vendors do not disclose their detection models, and cloud-based detection services provide only classification outputs. Second, black-box attacks provide a more realistic evaluation of the robustness of a detector. Third, black-box methods offer better generalization and transferability. Adversarial examples generated through black-box methods tend to exploit fundamental weaknesses in feature representations rather than model-specific weaknesses. This leads to higher transferability across different detectors. Finally, black-box approaches better align with the asymmetric nature of the adversarial arms race in malware detection. This means that attackers must operate with limited information while defenders can leverage complete knowledge of their own systems [1, 4, 7, 14, 24, 25].

The restriction evasion-only attacks are also considered more practical. Evasion attacks occur when the model is already deployed. While poisoning attacks require attackers to inject malicious samples into the training pipeline of the detector [5].

1.4 Generative models for adversarial malware

Generative Adversarial Networks (GANs) have emerged as a powerful framework for generating adversarial malware examples. GANs consist of two neural networks: a generator that produces synthetic samples, and a discriminator that distinguishes real from generated samples. They are both trained in an adversarial game. This architecture can easily generate adversarial malware, where the generator learns to produce evasive samples that fool the discriminator, which serves as a drop-in replacement for the target detector [11].

Reinforcement Learning (RL) offers an alternative paradigm for adversarial malware generation that addresses some limitations of GAN-based approaches, particularly the generation of valid, functional executable files [1].

1.4.1 GAN approach

The MalGAN framework is the first application of GANs to adversarial malware generation for black-box attacks [14]. MalGAN introduces a three-component architecture:

- A black-box detector as the target system.
- A substitute detector, which is a neural network or networks trained to mimic the black-box detector.
- A generator network that transforms malware feature vectors into adversarial versions.

The key innovation of MalGAN is its use of a substitute detector to provide gradient information for training the generator, thereby overcoming the black-box constraint. The generator takes as input a malware feature vector m and a random noise vector z, producing a perturbation vector. Then, that vector is combined with the original features via element-wise binary OR operation. This operation ensures that features can only be added, never removed, preserving malware functionality.

MalGAN's training procedure alternates between querying the black-box detector on generated adversarial examples to obtain labels, training the substitute detector on these labeled examples to improve its approximation of the black-box detector, and training the generator to minimize the substitute detector's predicted malicious probabilities. Experimental results demonstrated that MalGAN could reduce detection rates to nearly zero against various machine-learning-based detectors.

The MalFox framework is an extension of the GAN paradigm by combining a convolutional GAN with structured perturbation paths. This means that instead of introducing a single perturbation, MalFox outputs a path composed of up to three transformation methods. These methods are called Obfusmal, Stealmal, and Hollowmal [29].

Obfusmal encrypts a specific code segment of a malware file and attaches a DLL called `shell.dll` at the end of the file. This DLL file then decrypts the code segment and restores the malware's normal execution. Stealmal encrypts the entire malware and appends the program `shell.exe` to the end of the file. It decrypts the malware, creates a suspended process, retrieves the process space, and copies the malware into that space. This allows it to change the process's context to the malware's entry point and run the program. Finally, Hollowmal encrypts the malware itself and attaches it to the end of the file. Then, a DLL called `Hollow.dll` is appended to the modified file and behaves similarly to `Shell.exe`. These obfuscation extensions introduce a novel approach to working with binary files.

The Evolutionary MalGAN framework is an extension of MalGAN [20]. This approach extends the goal of evading a black-box detector by also introducing the goal to survive against auxiliary defenses. Evolutionary MalGAN is described as a bi-objective GAN. This is achieved by implementing a separate screening model that must also fool, in addition to the already present black-box detector. This approach leverages the fact that there is not a single detector attempting to classify incoming software, but rather many.

1.4.2 Reinforcement learning approach

The Malware_rl framework - An RL framework for evading machine-learning-based malware detectors was released as an open-source Gymnasium environment and subsequently improved by one of the co-authors [1, 10]. This framework has an approach that formulates adversarial malware generation as a Markov Decision Process (MDP). This means that an RL agent learns a policy for selecting sequences of functionality-preserving modifications to generate malware samples.

The framework defines a rich action space comprising multiple portable executable file manipulation techniques: adding imports to the import address table, manipulating section names and properties, creating new sections, appending bytes to sections, modifying entry points, removing signature information, manipulating debug information, and packing/unpacking executable files. Each action is designed to preserve executable validity and functionality.

The state representation is a 2,351-dimensional feature vector extracted from portable executable files, including header metadata, section metadata, import/export information, string statistics, byte histograms, and byte-entropy histograms. The reward function is binary: $R = 1$ if the modified sample evades detection, and $R = 0$ if it is detected as malicious.

The MAB-Malware framework is a different approach to the traditional RL based evasion [27]. It aims to balance exploration and exploitation with a limited number of queries. It reuses successful payloads and includes an action-minimization stage to remove ineffective changes, ensuring only efficient actions are included.

The MalInfo framework is a framework that improves upon the basic RL method by selecting an optimal perturbation path [30]. This means not only does the choice of transformations matter, but also the order in which they are applied. This complements the Gymnasium style environments by introducing action ordering.

1.4.3 Selected models and justification

This thesis employs MalGAN and `malware_rl` as evasion models for several reasons:

- Both are foundational approaches for their respective architectures. Other papers attempt to improve these models, but this thesis will compare these architectures directly in their most basic form.
- Comparing these two approaches can provide insight into their strengths and weaknesses.
- Both approaches operate in a black-box, untargeted evasion setting using vectorized features.

1.5 Malware detection dataset

The development of effective machine learning models for malware detection depends on the availability of large-scale, high-quality datasets. Historically, this has been hampered by the lack of publicly available datasets due to privacy concerns, intellectual property restrictions, and the sensitive nature of malware samples [2].

Early malware detection datasets were often small-scale, proprietary, or lacked proper separation between training and testing data. For example, the Microsoft Malware Classification Challenge dataset lacked sufficient benign samples for binary classification.

The EMBER (Elastic Malware Benchmark for Empowering Researchers) dataset addresses many limitations of prior datasets and has become the de facto standard for evaluating machine-learning-based malware detectors. EMBER consists of 1.1 million Windows Portable Executable (PE) files with extracted features: 900,000 training samples (300,000 malicious, 300,000 benign, 300,000 unlabeled) and 200,000 test samples (100,000 malicious, 100,000 benign).

EMBER provides pre-extracted features organized into eight groups: general file information, header information, imported functions, exported functions, section information, byte histogram, byte-entropy histogram, and string information. This feature representation is specifically designed to be manipulated through functionality-preserving modifications on portable executable files.

EMBER uses a strong baseline gradient-boosted decision tree implemented with LightGBM. This model achieved a 92.99% detection rate with a false-positive rate of less than 0.1%. This feature-based approach outperformed the MalConv model, an end-to-end deep learning model that operates directly on raw bytes. This demonstrates the advantage of the feature-based model over featureless deep learning malware detectors. This makes EMBER particularly suitable for adversarial malware research.

2 Investigation & Experiment

The experiments use the EMBER dataset [2]. It is a large benchmark of labeled Windows PE files for static malware detection. EMBER provides a 2,351-dimensional feature vector for each binary, extracted using the LIEF library. These feature vectors encode various static properties of the PE file and are divided into several groups of features:

- **Byte histograms (256 features):** A normalized histogram counting occurrences of each byte value from 0 to 255 in a given file. This captures the overall byte distribution.
- **Byte-entropy histograms (256 features):** This captures the distribution of entropy across byte values.
- **String features (104 features):** These are statistics about strings extracted from the binary of a given file. Some examples are: counts of specific patterns such as file paths, registry keys, and URLs. In addition, there is a histogram of character frequencies in strings.
- **General file information (10 features):** Basic metadata such as file size, number of sections, and symbol table size.
- **Header metadata (62 features):** Values extracted from the COFF and Optional headers. For example, the compile timestamp, subsystem, and DLL characteristics, among others.
- **Section metadata (255 features):** Attributes of each section in the PE file. This includes section sizes, entropies, names, and permissions. These attributes are vectorized to a fixed length, so that all files would have the same number of section-related features, even if the actual number of sections differs across files. This is achieved via the FeatureHasher function from scikit-learn.
- **Imported functions (1280 features):** Information about imported functions. FeatureHasher is also applied here to maintain the same number of import features even if the number of imports differs across files.
- **Exported functions (128 features):** Information about exported functions. FeatureHasher is used with the same purpose as in the imported functions.

Each feature vector contains a label: malicious or benign. For upcoming experiments, only the labeled samples will be used, and all unlabeled data will be removed.

2.1 Preprocessing

Before training the model, the EMBER dataset needs to be processed. Currently, data is stored in JSON Lines (JSONL) format, with each line representing the features of a single file. In addition, multiple JSONL files are used for training data, while a single JSONL file is used for testing data. There will be two ways to prepare the dataset: to accommodate different format requirements for the RL [1, 10] and MalGAN architectures [14].

MalGAN architecture implementation requires only function names from the import table. These functions must be vectorized and saved in a binary format with the `.npy` extension. Binaries should be segregated into benign-only and malware-only features. This is to ensure compatibility with the current MalGAN implementation from [13]. First, the train features are combined into a single large file. Second, read through both the train and test datasets line by line and find features that contain only valid functions. In this context, valid-only means function names must contain only letters, must be 2 to 64 characters long, and exclude function names that contain “ordinal”. This is done because a large number of unique function names containing the word “ordinal” can dominate the function names and create large vectors. If a given file feature contains only valid function names, the set of function names is saved, and the feature is saved to a file. However, due to computational constraints, the number of valid functions a given file feature can have is limited to 14, an arbitrarily selected number. If it is 15 or larger, the function names and features are not saved. Finally, once all valid features and valid unique function names are collected, they are mapped to a numpy array. The column names are the function names. Then, each file containing valid features is iterated line by line. The numpy array is initialized to all zeros, so when a feature finds a function name in the numpy array, it flips the 0 to 1, signaling the presence of the import function. Each row is a unique representation of that feature vector. Finally, when all lines are iterated through, the numpy array is saved in binary format as `.npy`. The temporary files are removed, leaving the large training dataset for the RL architecture. RL architecture implementation requires PE files; however, the implementation has been modified to accommodate the extracted features provided by the EMBER dataset.

2.2 Gradient boosted decision tree model

The baseline malware classifier is a GBDT classifier trained with the Microsoft LightGBM Python package. This model is an ensemble of decision trees trained to maximize detection accuracy. This model, along with the package, is chosen because the aim of this thesis is to be as close as possible to the original paper on which the EMBER dataset is based. The EMBER implementation provides the required information for training and evaluating the classifier [2, 8]. In addition, the GBDT classifier needs to be trained twice: once on the vectorized dataset used in the MalGAN architecture, once on the extracted features dataset used in the RL architecture.

Training is split into two steps: finding the best model parameters and training the model. In the first step, a parameter grid is constructed. The grid contains model parameters with a list of available values for training. The model is trained on a subset of the training data and then tested on test data. The train-test split is a standard 80%-20% split of the original dataset. The subset is further halved, so only 40% of the dataset is used to train the model during parameter exploration, and the model is then tested on the 20% test split. Once the best model parameters are found, they are saved to a file and used to train the model on the standard 80% split of the original dataset. Finally, the trained model is saved to a file for later use.

To evaluate the trained model, a constraint must be imposed on the evaluation metrics. The False Positive Rate (FPR) must not exceed 0.5%. In the real world, a high FPR can cause antivirus software that uses machine learning algorithms to mistakenly treat crucial Microsoft Windows operating system files as malicious and delete them. To avoid this scenario and simulate real-world requirements, the FPR is set to 0.5% or lower. This is implemented in this way. First, the vectorized dataset is loaded and split into training and test sets using the standard 80%-20% split of the original dataset. The model is tested on the test dataset, and it returns probabilities. Then, the scikit-learn function `roc_curve` is used to determine the optimal threshold to use given the FPR constraint. The `roc_curve` function accepts the true labels and the predictions, and outputs FPR, true positive rate, and thresholds. These outputs are all numpy arrays, where FPR contains increasing FPR rate values, and the "thresholds" array contains decreasing threshold values. Then, the index of the largest FPR value that satisfies our FPR requirement is extracted. That index is used to extract the threshold value from the "thresholds" array. The resulting value is the optimal threshold for this model. Applying this threshold to the probabilities yields a set of predicted labels. The evaluation metrics are extracted using scikit-learn functions. These include: accuracy, precision, recall, F1-score, AUC, partial AUC, and confusion matrix. These statistics are logged to the terminal.

After training and evaluation, the model is ready to be used as a black-box detector in both malware generation architectures.

2.3 MalGAN architecture

The MalGAN architecture implementation [13, 14] consists of three models: the detector, discriminator, and generator. The detector is a black-box detector, which is the GBDT model described previously. The discriminator and generator are built from scratch.

Both the generator and the discriminator are feed-forward neural networks. The network layers have two hidden layers. The structure looks like this: input → hidden layer → hidden layer → output. The specific layer sizes change during the experiment, but for illustration purposes, assume the input vector is 128-dimensional, and the layer sizes are 256.

The generator sizes look similar to this: 138 → 256 → 256 → 128. The input size of 138 results from concatenating the original malicious feature vector (128) with a randomly generated noise vector (10). The size of the noise vector is also variable. The noise vector is constrained to have values between 0 and 1. The noise introduces randomness, helping the generator generate variations of the original input. The final 128-dimensional output vector is an adversarial feature vector that resembles a benign file in feature space.

The discriminator sizes look similar to the generator, with key differences in the input and output sizes. The input is a 128-dimensional vector, and the output is a 1-dimensional vector containing the probability that the input is malicious. Then a threshold is applied to transform the probability into a binary result: malicious or benign.

MalGAN is trained in two stages. In the first one, the discriminator is trained on a large set of samples labeled by the black-box model. This is done by querying the black-box model on the training dataset to get either malicious or benign labels. These are used as the source of truth. Then, the discriminator is trained using binary cross-entropy loss with the Adam optimizer.

Next, the generator is trained to fool the discriminator. The generator receives malicious vectors, adds noise, and queries the discriminator. The objective of the generator is to minimize the probability of receiving the malicious label.

An important constraint is placed on the generator. Modifications should not remove malicious characteristics that are essential for the functionality of the malware. This is handled by operating on binary features and using the element-wise max operation to combine the original features with the generated perturbation. This results in the generator only adding new features that appear benign, but not removing any original features. Theoretically, this ensures the malware remains functioning with extra benign imports.

Once both models are trained, the generator can now produce an adversarial version of any malware sample. Then, the generator uses the perturbed sample to query the black-box detector and obtain the label with the expectation of evading detection.

2.4 Reinforcement learning architecture

The RL architecture implementation consists of an agent and the black-box detector [1, 10]. The implementation is based on the Gymnasium environment and uses the Actor-Critic with Experience Replay (ACER) algorithm. In this implementation, the agent performs a set of actions in the environment with the goal of modifying the sample until it evades detection.

Some important modifications had to be performed. The original implementation used an outdated OpenAI Gym package and expected PE files as the dataset. To switch to using extracted features from the EMBER dataset, the implementation had to be modified. One of the largest areas of change was the available actions. The original implementation included 16 actions; however, the agent's actions must preserve functionality, and only 5 were deemed suitable for feature manipulation. These modifications are:

- **Machine type modification:** this action changes the original machine type declaration to one of the following: "AMD64", "IA64", "ARM64", "POWERPC". This is the original list of allowed machine types.
- **Add imports:** this action adds import functions or imports with functions to the extracted features. The allowed imports and functions are provided in the original implementation and remain unchanged.
- **Rename a section:** this action checks for existing section names, chooses one at random, then selects a new section name from the list of allowed section names provided by the original implementation, and changes it.
- **Modify the optional header:** this action modifies the version values of the optional header from an allowed list of modifications provided by the original implementation.
- **Modify timestamp:** this action updates the timestamp to an allowed value from the original implementation's list of available values.

Due to the lowering of the list of allowed actions, a few changes were made to the agent. First, the number of "total_timesteps" used in teaching the agent increased from 2,500 to 100,000. Second, the number of actions the agent is allowed to take before stopping increased from 50 to 5,000. However, to avoid too many steps, a penalty is introduced: each additional action incurs a 0.1 penalty. In addition, a penalty is introduced to reduce the probability of repeating the same action in a row. The value for each consecutive same action is 0.5.

The process looks like this. During each query, the environment provides the agent with the reward and state. The state is the current feature vector of the malware sample. Initially, the feature vector is the malicious feature vector from the EMBER dataset. After each action, the features are modified, and the result is vectorized. Vectorization is implemented the same way it was in the EMBER implementation.

The RL agent receives feedback after each action. This feedback is the reward. The reward is determined by the black-box detector based on the label it produces. If the modified sample is detected and labeled as malware, the reward is set to 0, since no evasion occurred. If the modified sample is not detected and labeled as benign, then the reward is +10 for the successful evasion. The goal of the agent is to maximize reward.

The agent is trained through simulated attack episodes. In each episode, the agent starts with a fresh malware sample that is initially detected by the black-box detector. The agent observes the state and selects an action according to its policy. Then the action is applied to the feature set, and the feature is vectorized. The environment then returns a binary reward (malicious or benign) and the updated state. This loop continues for a number of steps until the malware is misclassified or the maximum number of actions is reached.

The ACER algorithm is then used to update the agent. The policy and value functions of the agent are represented by a deep neural network. Because the state is already a compact feature vector, a simple multi-layer perceptron was used. The network expects a 2381-dimensional state and outputs two results: first, a policy, which is the probability distribution over all available actions; second, a value, which is an estimate of the expected cumulative reward used by the critic part of ACER.

Then, experience replay is used to sample batches of past transitions and apply ACER updates. The agent is trained over a large number of episodes until the performance of the policy plateaus. Once training is complete, a test set of malware is used to evaluate the learned policy. For each new malware sample not seen during training, the agent attempts to evade detection by executing actions according to the learned policy. The level of success, the examples of evasive features, and the number of steps needed to achieve evasion are logged.

2.5 Results

The first experiment is the MalGAN implementation attack on the trained GBDT model. The GBDT model was trained on the vectorized imports dataset. The dataset consists of two files: `vectorized_features_mal.npy` (83,272 rows) and `vectorized_features_ben.npy` (14,329 rows). Each row in both files contains 3,046 features. The statistics of the trained GBDT model are shown in 1 table. The experiment consists of three black-box models: the GBDT model from malgan, the decision tree model from scikit-learn, and the logistic regression model from scikit-learn. Each of these models was trained on the same dataset. The discriminator and generator models are simple feed-forward neural networks. Both models have 4 layers: an input layer, two hidden layers, and an output layer. Each hidden layer consists of a linear transformation followed by a non-linear activation function, with the output layer having the sigmoid activation function. The generator also introduces a noise vector to generate a perturbed adversarial example. For each black-box model, experiments are conducted by varying the size of the noise vector and the activation functions. The batch size is 32, and the epoch size is 100. The experiments have produced results, which are shown in several tables for each black-box detector:

- GBDT - 2 table.
- Decision tree - 3 table.
- Logistic regression - 4 table.

1 table. *Performance summary of the GBDT model trained on the vectorized features dataset.*

Metric	**Value**
Optimal Threshold	0.3767
Accuracy	90.50%
Precision	93.24%
Recall	38.01%
F1-Score	54.00%
Global AUC	93.57%
Partial AUC (0.5%)	64.11%
Total Samples	19,521 (100%)
Total Correct	17,666 (90.50%)
Total Wrong	1,855 (9.50%)
True Negative (TN)	16,577 (84.92%)
False Positive (FP)	79 (0.40%)
False Negative (FN)	1,776 (9.10%)
True Positive (TP)	1,089 (5.58%)

The second experiment is the RL implementation attack on the trained GBDT model. The GBDT model was trained on features stored in JSON Lines (JSONL) files. The dataset consists of two files: `test_features.jsonl` (200,000 rows) and `train_features.jsonl` (600,000 rows). The optimal model parameters are shown in Table 5, and the statistics of the trained GBDT model are shown in 5 table. The experiment was carried out using the Proximal Policy Optimization (PPO) algorithm as the agent with a Multi-Layer Perceptron (MLP) serving as the underlying neural network architecture for the agent's policy. The agent was trained on the same dataset and had 100,000 learning timesteps to compensate for the smaller amount of available actions. In total, the agent was allowed to use up to 5,000 actions. The results of the experiment are shown in 6 table.

2 table. *MalGAN attack evasion results for GBDT detector.*

Activation Function	Noise vector size	Avg. Bits Changed	Orig. Det. Rate	Mod. Det. Rate	Evasion Rate
ReLU	10	2,151	99.73%	0.00%	100.00%
ReLU	50	2,189	99.73%	0.00%	100.00%
ReLU	100	2,103	99.73%	0.00%	100.00%
ELU	10	1,965	99.73%	0.00%	100.00%
ELU	50	1,961	99.73%	0.00%	100.00%
ELU	100	1,888	99.73%	0.00%	100.00%
LeakyReLU	10	2,026	99.73%	0.00%	100.00%
LeakyReLU	50	2,150	99.73%	0.00%	100.00%
LeakyReLU	100	2,101	99.73%	0.00%	100.00%
Tanh	10	1,848	99.73%	0.00%	100.00%
Tanh	50	1,844	99.73%	0.00%	100.00%
Tanh	100	1,961	99.73%	0.00%	100.00%
Sigmoid	10	2,328	99.73%	0.00%	100.00%
Sigmoid	50	1,824	99.73%	0.00%	100.00%
Sigmoid	100	2,253	99.73%	0.00%	100.00%

3 table. *MalGAN attack evasion results for Decision Tree detector.*

Activation Function	Noise vector size	Avg. Bits Changed	Orig. Det. Rate	Mod. Det. Rate	Evasion Rate
ReLU	10	1,763	48.34%	0.00%	100.00%
ReLU	50	1,316	48.27%	0.00%	100.00%
ReLU	100	2,020	48.52%	0.00%	100.00%
ELU	10	1,384	48.13%	0.00%	100.00%
ELU	50	1,813	48.20%	0.00%	100.00%
ELU	100	876	48.13%	0.00%	100.00%
LeakyReLU	10	1,912	48.24%	0.00%	100.00%
LeakyReLU	50	666	48.20%	0.00%	100.00%
LeakyReLU	100	1,850	48.17%	0.00%	100.00%
Tanh	10	1,568	48.24%	0.00%	100.00%
Tanh	50	1,497	48.31%	0.00%	100.00%
Tanh	100	536	48.38%	0.00%	100.00%
Sigmoid	10	2,102	48.27%	0.00%	100.00%
Sigmoid	50	579	48.06%	0.00%	100.00%
Sigmoid	100	2,279	48.10%	0.00%	100.00%

4 table. *MalGAN attack evasion results for Logistic Regression detector.*

Activation Function	Noise vector size	Avg. Bits Changed	Orig. Det. Rate	Mod. Det. Rate	Evasion Rate
ReLU	10	1,317	44.08%	0.00%	100.00%
ReLU	50	1,206	44.08%	0.00%	100.00%
ReLU	100	1,156	44.08%	0.00%	100.00%
ELU	10	488	44.08%	0.00%	100.00%
ELU	50	570	44.08%	0.00%	100.00%
ELU	100	612	44.08%	0.00%	100.00%
LeakyReLU	10	323	44.08%	0.00%	100.00%
LeakyReLU	50	1,462	44.08%	0.00%	100.00%
LeakyReLU	100	1,041	44.08%	0.00%	100.00%
Tanh	10	583	44.08%	0.00%	100.00%
Tanh	50	496	44.08%	0.00%	100.00%
Tanh	100	478	44.08%	0.00%	100.00%
Sigmoid	10	1,449	44.08%	0.00%	100.00%
Sigmoid	50	1,543	44.08%	0.00%	100.00%
Sigmoid	100	1,539	44.08%	0.00%	100.00%

5 table. *Performance summary of the GBDT model trained on the raw features dataset.*

Metric	Value
Optimal Threshold	0.8578
Accuracy	92.88%
Precision	99.42%
Recall	86.27%
F1-Score	92.38%
Global AUC	99.03%
Partial AUC (0.5%)	89.67%
Total Samples	200,000 (100%)
Total Correct	185,770 (92.88%)
Total Wrong	14,230 (7.12%)
True Negative (TN)	99,500 (49.75%)
False Positive (FP)	500 (0.25%)
False Negative (FN)	13,730 (6.87%)
True Positive (TP)	86,270 (43.13%)

6 table. *Reinforcement Learning attack evasion results.*

Metric / Action Type	Value / Count	Percentage
Performance Metrics		
Total Samples Attempted	250	100.00%
Successfully Evaded	186	74.40%
Average Moves to Evade	1,953	–
Modification Actions		
add_imports	216,824	$\approx$ 100.00%
modify_timestamp	1	$<$ 0.01%
Total Actions	**216,825**	**100.00%**

3 Conclusions

In this thesis, the Gradient Boosted Decision Tree (GBDT) classifier was selected as the black-box model for the experiments. The model was trained on the EMBER dataset. The MalGAN and Reinforcement Learning (RL) architecture implementations were used to attack the black-box detector. The goal of the thesis is to compare different machine learning obfuscation architectures. A few insights can be gained from the results:

- All three models (GBDT, Decision Tree, Logistic Regression) in MalGAN attacks have shown high vulnerability, where the MalGAN generator managed to learn to evade all models with a 100% evasion rate.
- In MalGAN attacks, each black-box detector had different initial detection rates and required different amounts of bit changes to be fooled. The most fragile model was the Logistic Regression model, which required only around 500 average bits to change when the Generator used the ELU activation function, and had the lowest initial detection rate of ~44%. The most resistant model was the GBDT model, which initially had a ~99% detection rate and required up to ~2,300 bits to be fooled.
- The Reinforcement Learning (RL) attacks achieved a ~74% evasion rate with an average applied action count of ~1,900. The dominant modification was `add_imports`, suggesting that the model preferred it because it offered the best chance to evade.
- The perfect 100% evasion rate of MalGAN may be due to how the generator works. The generator adds a large number of import functions to the initial sample, which initially has few active import functions. This can be caused by the default threshold of 0.5, which results in large, sharp changes and, therefore, a large impact on the evasion rate.

Both MalGAN and RL achieved high evasion rates. Each has its own strengths and weaknesses. MalGAN excels when the architecture is limited to a fixed number of queries to the black-box detector. This means that if a rate limit were implemented on the black-box detector, MalGAN would outperform RL. However, the current implementation of MalGAN has a very limited pool of available manipulations. If those manipulations are not handled correctly, MalGAN can generate breaking changes and break the functionality preservation constraint. On the other hand, RL has a large number of discrete, functionality-preserving actions it can perform, including working on feature sets and whole files. However, given a detector's rate limit, performance can degrade. A suggestion is to improve performance by combining the two architectures and leveraging their strengths: the GAN's ability to learn with fewer Black-box detector queries, and the RL's ability to take a large number of discrete, functionality-preserving actions.

References and sources

[1] H. S. Anderson, A. Kharkar, B. Filar, D. Evans, P. Roth. "Learning to evade static PE machine learning malware models via reinforcement learning." In: *arXiv preprint arXiv:1801.08917* (2018). https://doi.org/10.48550/arXiv.1801.08917.

[2] H. S. Anderson, P. Roth. "EMBER: An open dataset for training static PE malware machine learning models." In: *arXiv preprint arXiv:1804.04637* (2018). https://doi.org/10.48550/arXiv.1804.04637.

[3] K. Aryal, M. Gupta, M. Abdelsalam. "A survey on adversarial attacks for malware analysis." In: *arXiv preprint arXiv:2111.08223* (2021). https://doi.org/10.1109/ACCESS.2024.3519524.

[4] B. Biggio, G. Fumera, F. Roli. "Security evaluation of pattern classifiers under attack." In: *IEEE Transactions on Knowledge and Data Engineering*. Volume 26. 4. 2013, pages 984–996. https://doi.org/10.1109/TKDE.2013.57.

[5] B. Biggio, B. Nelson, P. Laskov. "Poisoning attacks against support vector machines." In: *Proceedings of the 29th International Conference on Machine Learning*. 2012, pages 1467–1474. https://doi.org/10.48550/arXiv.1206.6389.

[6] N. Carlini, D. Wagner. "Towards evaluating the robustness of neural networks." In: *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2017, pages 39–57. https://doi.org/10.1109/SP.2017.49.

[7] L. Demetrio, S. E. Coull, B. Biggio, G. Lagorio, A. Armando, F. Roli. "Adversarial EXEmples: A survey and experimental evaluation of practical attacks on machine learning for windows malware detection." In: *ACM Transactions on Privacy and Security (TOPS)* 24.4 (2021), pages 1–31. https://doi.org/10.1145/3473039.

[8] elastic. *EMBER implementation in code*. URL: https://github.com/elastic/ember.

[9] Z. Fang. "Utilizing benign files to obfuscate malware via deep reinforcement learning." In: (2022), pages 421–425. https://doi.org/10.1109/IIP57348.2022.00067.

[10] B. Filar. *malware_rl implementation*. URL: https://github.com/bfilar/malware_rl.

[11] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, Y. Bengio. *Generative Adversarial Networks*. 2014. https://doi.org/10.48550/arXiv.1406.2661.

[12] K. Grosse, N. Papernot, P. Manoharan, M. Backes, P. McDaniel. "Adversarial examples for malware detection." In: (2017), pages 62–79. https://doi.org/10.1007/978-3-319-66399-9_4.

[13] Z. Hammoudeh. *MalGAN implementation in code*. URL: https://github.com/ZaydH/MalwareGAN.

[14] W. Hu, Y. Tan. “Generating adversarial malware examples for black-box attacks based on GAN.” In: *arXiv preprint arXiv:1702.05983* (2017). https://doi.org/10.1007/978-981-19-8991-9_29.

[15] Y. Ye, T. Li, D. Adjeroh, S. S. Iyengar. “A Survey on Malware Detection Using Data Mining Techniques.” In: *ACM Comput. Surv.* 50.3 (2017). https://doi.org/10.1145/3073559.

[16] I. You, K. Yim. “Malware Obfuscation Techniques: A Brief Survey.” In: *2010 International Conference on Broadband, Wireless Computing, Communication and Applications*. 2010, pages 297–300. https://doi.org/10.1109/BWCCA.2010.85.

[17] J. Z. Kolter, M. A. Maloof. “Learning to Detect and Classify Malicious Executables in the Wild.” In: *Journal of Machine Learning Research* 7.99 (2006), pages 2721–2744. https://doi.org/10.1145/1014052.1014105.

[18] M. Kozak, M. Jureček, M. Stamp, F. Di Troia. “Creating valid adversarial examples of malware.” In: *arXiv preprint arXiv:2306.13587* (2023). https://doi.org/10.1007/s11416-024-00516-2.

[19] D. Li, Q. Li, Y. (Ye, S. Xu. “Arms Race in Adversarial Malware Detection: A Survey.” In: *ACM Comput. Surv.* 55.1 (2021). https://doi.org/10.1145/3484491.

[20] H. Li, S. Zhou, W. Yuan, J. Li, H. Leung. “Adversarial-Example Attacks Toward Android Malware Detection System.” In: *IEEE Systems Journal* 14.1 (2020), pages 653–656. https://doi.org/10.1109/JSYST.2019.2906120.

[21] M. Macas, W. Fuertes. “Adversarial examples: A survey of attacks and defenses in deep learning-enabled cybersecurity systems.” In: *Expert Systems with Applications* 233 (2023), page 120910. https://doi.org/10.1016/j.eswa.2023.122223.

[22] A. Moser, C. Kruegel, E. Kirda. “Limits of Static Analysis for Malware Detection.” In: *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. 2007, pages 421–430. https://doi.org/10.1109/ACSAC.2007.21.

[23] N. Papernot, P. McDaniel, I. Goodfellow. “Transferability in machine learning: from phenomena to black-box attacks using adversarial samples.” In: *arXiv preprint arXiv:1605.07277*. 2016. https://doi.org/10.48550/arXiv.1605.07277.

[24] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, A. Swami. “Practical black-box attacks against machine learning.” In: *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. 2017, pages 506–519. https://doi.org/10.1145/3052973.3053009.

[25] D. Park, B. Yener. “A survey on practical adversarial examples for malware classifiers.” In: *arXiv preprint arXiv:2011.03705* (2020). https://doi.org/10.1145/3433667.3433670.

[26] M. Schultz, E. Eskin, F. Zadok, S. Stolfo. “Data mining methods for detection of new malicious executables.” In: *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*. 2001, pages 38–49. https://doi.org/10.1109/SECPRI.2001.924286.

[27] W. Song, X. Li, S. Afroz, D. Garg, D. Kuznetsov, H. Yin. "MAB-Malware: A Reinforcement Learning Framework for Blackbox Generation of Adversarial Malware." In: *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*. Association for Computing Machinery, 2022, pages 990–1003. `https://doi.org/10.1145/3488932.3497768`.

[28] D. Zhan, Y. Zhang, L. Zhu, J.-Y. Chen, S. Xia. "Enhancing reinforcement learning based adversarial malware generation to evade static detection." In: *Alexandria Engineering Journal* 99 (2024), pages 308–320. `https://doi.org/10.1016/j.aej.2024.04.024`.

[29] F. Zhong, X. Cheng, D. Yu, B. Gong, S. Song, J. Yu. "MalFox: Camouflaged Adversarial Malware Example Generation Based on Conv-GANs Against Black-Box Detectors." In: *IEEE Transactions on Computers* 73.4 (2024), pages 980–993. `https://doi.org/10.1109/TC.2023.3236901`.

[30] F. Zhong, P. Hu, G. Zhang, H. Li, X. Cheng. "Reinforcement learning based adversarial malware example generation against black-box detectors." In: *Computers & Security* 121 (2022), page 102869. `https://doi.org/10.1016/j.cose.2022.102869`.

4 Appendices

Link to the GitHub repository where the code is located: https://github.com/Jus-tasKud/masters-thesis-code

AI tools used:

- Grammarly - spellcheck
- Overleaf - spellcheck
- ChatGPT - Code suggestions and code analysis