Department of Informatics
King's College London
University of London
United Kingdom

7CCSMPRJ MSc Dissertation

Using Generative Adversarial Networks to Create Evasive Feature Vectors for Malware Classification

Supervisor
Prof. Lorenzo Cavallaro

Candidate
Mohamed Abouhashem

This dissertation is submitted in partial fulfilment of the requirements for the degree of **MSc in Advanced Computing.**

2019/2020

Acknowledgement

I would like to express my gratitude to my supervisor Professor Lorenzo Cavallaro for his invaluable support and supervision during the course of this project, and for inspiring my interest in the subject of adversarial machine learning. My gratitude extends to the Centre for Doctoral Studies Support (CDS) and the Faculty of Natural & Mathematical Sciences for awarding me King's International Scholarship to undertake my master's studies at the Department of Informatics. I am also grateful to Al-Alfi Foundation for sponsoring and supporting me during my studies.

Abstract

Machine learning (ML) models have been found to be vulnerable to a variety of adversarial attacks. Most notably, **adversarial examples (AEs)** are a category of attacks that mislead a target model to produce an outcome that is influenced by an adversary. The most common type of AE-based attacks is **evasion**, where a valid sample is *perturbed* to cause misclassification by the target model.

Most of the current methods to generate AEs are **iterative algorithms** that transform a given sample into an AE to be misclassified by the target model. More recently, **generative models** have been proposed as a more efficient and generalisable way to generate AEs for any given input. In this project, we propose a novel framework for AE generation based on **generative adversarial networks (GANs)**. The proposed framework, called **EvadeGAN**, can operate in three different modes to produce either **sample-dependent** or **sample-independent (universal) perturbations**. Also, **EvadeGAN** works in **grey-box settings**, thus requiring no access to the internals of the target model.

EvadeGAN has been evaluated against **DREBIN**, a state-of-the-art classifier for Android malware. The results show that **EvadeGAN** achieved **perfect evasion** with **minimal perturbations** against the **DREBIN** classifier in all different modes.

Keywords: adversarial machine learning, adversarial examples, evasion attacks, generative adversarial networks, DREBIN

List of Tables

List of Figures

Table of Contents

1. Introduction

Machine learning (ML) has been used in a variety of security applications such as malware classification, intrusion detection, and spam filtering. Interestingly, it has also introduced a new attack surface that could be exploited in adversarial environments. This has inspired a new field of research known as **adversarial machine learning**, which investigates adversarial attacks on ML models. The most common attack is **evasion**, which causes the target model to misclassify adversarial examples (AEs) without influencing its parameters.

Most of the current methods to generate AEs follow an *iterative* approach to transform a given sample into an AE that is misclassified by the target model. More recently, **generative models** have been proposed as a more efficient and generalisable method to generate AEs for any input. This project investigates the use of **generative adversarial networks** (GANs) in evading ML-based security models. In particular, the primary objective of this project is to investigate the effectiveness of GANS in generating **evasive adversarial examples** against malware classifiers in **grey-box settings**. The proposed framework is called **EvadeGAN.** The target model we test EvadeGAN against is **DREBIN** [1], a state-of-the-art SVM classifier for Android malware.

This report is organised as follows. **Chapter 2** (**Background**) reviews a variety of topics that are relevant to this project. It starts with the general topic of **adversarial attacks in machine learning**, then briefly covers **adversarial examples** and methods of their generation. It also covers **support vector machines**, on which the target classifier is based, then describes the original **DREBIN** classifier and its dataset. The chapter concludes with a review of **GANs**. **Chapter 3** (**Related Work**) reviews previous work on evasion attacks against malware classifiers as well as related research on using generative models to generate AEs. **Chapter 4** (**Methodology & Implementation**) describes the used dataset, the training and evaluation of the target classifier, the threat model, and the implementation of EvadeGAN. **Chapter 5** presents the **results** and their discussion, and **Chapter 6** includes **conclusions and future** work.

2. Background

2.1 Adversarial Attacks in Machine Learning

Adversarial machine learning is concerned with the vulnerabilities of ML models in adversarial environments, covering both attacks and defences [2]. Adversarial attacks can be classified based on various criteria, including influence, timing, specificity, security violation, and knowledge of the model [3, 4]. The taxonomy of adversarial attacks is summarised in **Figure 1**.

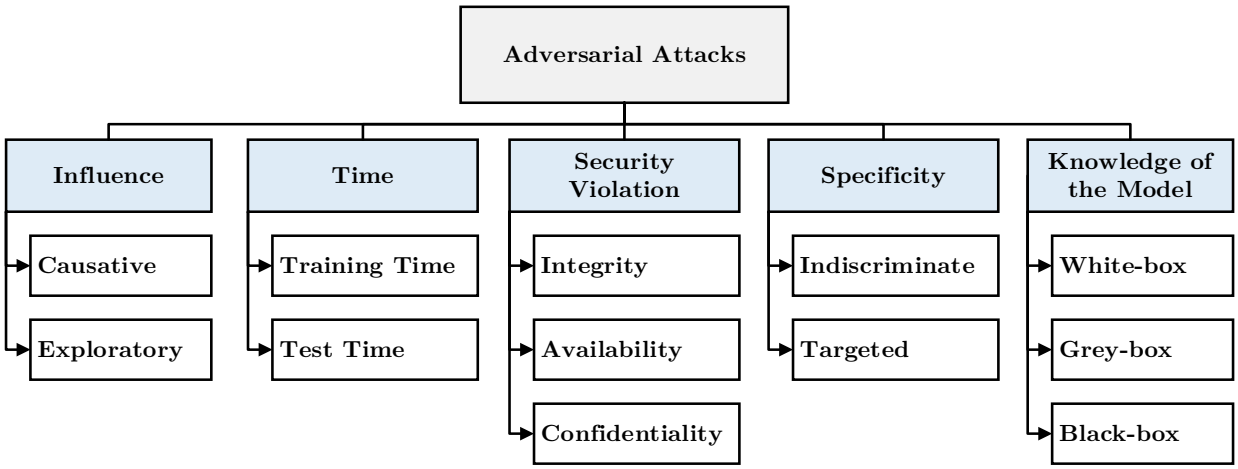


Figure 1. Taxonomy of adversarial attacks

Influence and **timing** are related in the context of adversarial attacks. Based on **influence**, attacks could be either causative or exploratory [3]. **Causative attacks** attempt to influence model parameters by tampering with training data. This type of attacks is called **poisoning** and is carried out at **training time**. **Exploratory attacks**, on the other hand, do not influence learning, but rather exploit the predictions of a trained model at **test time**. One such attack is **evasion**, where the attacker exploits the **adversarial space** of a model to cause it to misclassify an **adversarial example**. The **adversarial space** of a model refers to regions in the feature space that are not well represented in the distribution of the training data or not well captured by the model [5]. Evasion attacks draw adversarial examples from this space to cause misclassification and evade detection. **Figure 2** gives a conceptual representation of the adversarial space.

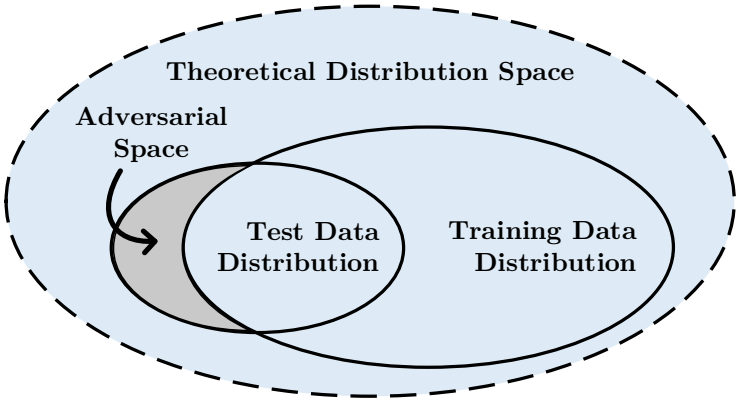


Figure 2. Adversarial space comprises regions not captured in the training data [5]

Adversarial attacks can violate any of the **security properties** of the target system: integrity, availability, and confidentiality/privacy [3, 4]. **Integrity attacks** aim to circumvent a system through *false negatives*, either actively (causative) or passively (exploratory). For example, an evasion attack that passively allows a malicious sample to bypass a classifier without detection is an exploratory integrity attack. **Availability attacks** compromise the functionality of a target system, usually via *false positives*. For example, a poisoning attack that causes a system to misclassify benign samples as malicious is a causative availability attack. **Confidentiality and privacy attacks** obtain confidential/private information about the system, its users or training data. Examples include extraction, inversion, and membership inference attacks [6].

Specificity describes the goal of the attack in terms of either the *target inputs* (**attack specificity**) or the *target output* (**error specificity**) [4].[1] **Attack specificity** refers to whether the attack targets a *specific input sample* (**targeted**) or a *wider set of samples or any sample* (**indiscriminate**). For example, an evasion attack aiming to slip a specific spam message through a filter is *targeted*, while a poisoning attack degrading the overall performance of the filter is *indiscriminate*. **Error specificity**, on the other hand, refers to whether the attack aims to have samples misclassified as a *specific class* (**targeted error**) or just *any wrong class* (**indiscriminate error**). For example, an attack that causes a malware classifier to misclassify some samples specifically as class X is an attack with a *targeted error*.

[1] This distinction was proposed in [4] because of the conflicting use in literature. Barreno et al. [3] originally used *specificity* to refer to the *target input(s)*, while much of the literature use specificity to refer to the *desired output*.

Based on the attacker's **knowledge of the model**, adversarial attacks could be classified as **white-box**, **grey-box**, or **black-box** [4]. Knowledge of the model includes its architecture, parameters, feature set, learning algorithm, and training data. In **white-box attacks**, the attacker has *full access to such knowledge* which allows maximum exploitation capability. **Grey-box attacks** use *limited knowledge* of the target model such as the feature set and model type. A typical attack will also have access to *a surrogate dataset,* ideally from the same distribution as training data. This allows the attacker to train a surrogate classifier to approximate the target model and attack it offline. Due to the *transferability of adversarial examples* [7], attacks against the surrogate classifier could transfer to the target model. **Black-box attacks** are carried out with almost no knowledge of the target model and with only *oracle access* to it. That is, the attacker presents inputs to the target model and observes its outputs. These interactions could be used to infer usable information about the target model or data as in extraction, inversion, and membership inference attacks [6]. Alternatively, the attacker could use the input-output pairs to train a surrogate classifier and use it to craft adversarial examples against the target model [7].

2.2 Adversarial Examples

Current definitions of adversarial examples (AEs) are based on perturbing a given input sample to cause misclassification.[1] Formally, given a target classifier f, a data point $\boldsymbol{x} \in \mathbb{R}^m$ with true class $y = f(\boldsymbol{x})$, and an operator $\mathcal{P} : \mathbb{R}^m \to \mathbb{R}^m$, an adversarial example $\boldsymbol{x}' = \mathcal{P}(\boldsymbol{x})$ is a perturbed sample with predicted class $f(\boldsymbol{x}') \neq y$. **Additive perturbations,** $\mathcal{P}(\boldsymbol{x}) = \boldsymbol{x} + \boldsymbol{\delta}$, are the most commonly used transformation in generating AEs, although other approaches have also been used.[2]

The following sections will briefly cover methods for generating AEs, considerations for malware AEs, as well as proposed methods of defence.

[1] Definitions based on input perturbation fail to include *naturally occurring* AEs, as shown in [8].

[2] Examples of *non-additive perturbations* include spatial, colour, and geometric transformations [9]. A more complex transformation could map inputs into a dense representation where perturbations can be applied, as in [10].

2.2.1 Generating Adversarial Examples

Generating AEs is typically formulated as a constrained objective function that the attacker wishes to optimise. A common objective is to minimise the amount of perturbation that leads to misclassification, as measured by the distance from the original sample. This could serve various purposes such as minimising the detectability of perturbations, maximising similarity to the original data, minimising the cost of the attack, and propagating class labels [11]. Distance is typically measured in terms of ℓ_p-norms, where $0 \leq p \leq \infty$. For $p \geq 1$, the ℓ_p-norm of a vector $\boldsymbol{x} \in \mathbb{R}^d$ is denoted by $\|\boldsymbol{x}\|_p = \left(\sum_{i=1}^{d} |x_i|^p\right)^{1/p}$. Two special cases are the zero norm ℓ_0 which returns the number of non-zero elements in a vector, and the maximum norm ℓ_∞ which returns the maximum of the absolute element values. Assuming perturbations are additive ($\boldsymbol{x}' = \boldsymbol{x} + \boldsymbol{\delta}$), an attacker could generate AEs by solving the following optimisation problem [12], which could be extended to include other objectives or constraints:[1]

$$\boldsymbol{x}^* = \underset{\boldsymbol{x}'}{\operatorname{argmin}} \|\boldsymbol{x}' - \boldsymbol{x}\|_p = \boldsymbol{x} + \underset{\boldsymbol{\delta}}{\operatorname{argmin}} \| \boldsymbol{\delta} \|_p$$
$$\text{subject to: } f(\boldsymbol{x}') \neq f(\boldsymbol{x})$$

Several techniques have been used to generate AEs through solving such optimisation problems, either in white-box or black-box scenarios. White-box methods are typically based on the gradient of an objective function. Notable examples include limited-memory BFGS [12], Fast Gradient Sign Method (FGSM) [15] with an iterative variation (I-FGSM) [16], Jacobian Saliency Map Attack (JSMA) [17], DeepFool [18], the Carlini-Wagner attack [13], and universal perturbation attacks [19]. Black-box methods, on the other hand, are gradient-free. A few examples include zeroth-order optimisation (ZOO) [20], rejection sampling (Boundary Attack) [21], and simultaneous perturbation stochastic approximation (SPSA) [22]. Evolutionary algorithms have also been used in black-box scenarios. These include differential evolution [23], evolution strategies [24], genetic programming [25], genetic algorithms [26], and particle swarm optimisation [27].

[1] Another common objective is to maximise the confidence of the attack. To achieve this, Carlini & Wagner [13] used the predicted scores, while Biggio et al. [14] estimated the density of the target class.

2.2.2 Malware Adversarial Examples

Most of the methods mentioned in the previous section have been developed to work with images and similar continuous domains. When dealing with malware, however, some distinctions need to be made. The two main differences are:

1. Discrete data. Malware samples are typically represented by feature vectors of binary/discrete values. This requires constraining the optimisation problem to deal with such data as well as choosing a suitable similarity/distance metric.

2. Semantics. For the attack to be meaningful, perturbations need to preserve the functionality and maliciousness of the generated AEs. A few approaches have been proposed to enforce this challenging constraint. Examples include limiting perturbations to only add/increment features, restricting the set of features that can be altered, and limiting the amount of change. These constraints, however, give no guarantee that the perturbed vector in the feature space corresponds to a valid artefact-free sample in the input/problem space. This is referred to as the *inverse feature mapping problem*. In [28], Pierazzi et al. proposed a formalisation that provides necessary and sufficient conditions for *problem-space* attacks, which is further discussed in the **Related Work** chapter.

2.2.3 Defences against Adversarial Examples

Although there is no single solution to defend against AEs, several countermeasures have been proposed to detect adversarial inputs, mitigate their impact, or harden the target model. These measures can be broadly categorised as either *reactive* or *proactive* [4].

Reactive methods are aimed at *detecting AEs* to discard them altogether, or at *transforming inputs* to alleviate the impact of perturbations [29]. Examples of the first approach include detection via statistical analysis [30-32], feature squeezing [33], or using a dedicated classifier [34, 35]. Examples of the second approach include randomised transformations [36], image-specific transformations (e.g., compression, bit-depth reduction) [37], thermometer encoding [38], and Defense-GAN [39].

Proactive methods, on the other hand, build or change a model in one of two approaches: *security-by-design* or *security-by-obscurity* [4]. Security-by-design methods are aimed against white-box attacks. Notable examples include adversarial training [15] and defensive distillation [40]. Security-by-obscurity methods, such as gradient masking [7], have been proposed against black-box attacks. They, however, have shown little effectiveness against such attacks [7, 41].

2.3 Support Vector Machines

A **support vector machine** (**SVM**) is a supervised learning model that defines a *maximum-margin hyperplane* to separate two classes. Maximising the margin aims to *minimise* the bound on the *generalisation error* of the model, thus reducing the risk of overfitting. In addition to classification, SVMs are also used in regression, outlier detection, and clustering.

Formally [42], given a dataset $\mathcal{D} = \{(\boldsymbol{x}_i, y_i) \mid \boldsymbol{x}_i \in \mathbb{R}^d,\ y_i \in \{-1, +1\}\}_{i=1}^n$ that is *linearly separable*, we need to find a linear discriminant function $f(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x} + b$ with an *optimal* hyperplane $f(\boldsymbol{x}) = 0$. A hyperplane is optimal under two conditions: **1)** all data are correctly classified: $y_i(\boldsymbol{w^T x_i} + b) \geq 1\ \forall i$, and **2)** the margin $2/\|\boldsymbol{w}\|$ is maximised. This could be formulated as a constrained minimisation problem:

$$\min_{\boldsymbol{w},b} J(\boldsymbol{w}) = \frac{1}{2}\|\boldsymbol{w}\|^2 \quad \text{subject to } y_i(\boldsymbol{w^T x_i} + b) \geq 1\ \forall i$$

Using Lagrange multipliers $\alpha_i \geq 0$, we obtain an unconstrained **Lagrangian** (**Eq. 1**), from which we can derive the **dual form** (**Eq. 2**). For optimal $\boldsymbol{w}$ and b, the dual form can be reduced to **Eq. 3**, where $\langle\cdot,\cdot\rangle$ is the inner product. Solving the dual problem, with a quadratic solver, leads to the maximum-margin classifier f in **Eq. 4**.

$$\textbf{Lagrangian:} \quad \mathcal{L}(\boldsymbol{w}, b, \alpha) = \frac{1}{2}\|\boldsymbol{w}\|^2 - \sum_{i=1}^{n} \alpha_i (y_i(\boldsymbol{w}^T\boldsymbol{x}_i - 1) \tag{1}$$

$$\textbf{Dual:} \quad \max_{\alpha \geq 0} \min_{\boldsymbol{w},b} \left(\frac{1}{2}\|\boldsymbol{w}\|^2 - \sum_{i=1}^{n} \alpha_i (y_i(\boldsymbol{w}^T\boldsymbol{x}_i - 1) \right) \tag{2}$$

$$\rightarrow \quad \max_{\alpha \geq 0} \left(\sum_{i=1}^{n} \alpha_i - \sum_{i,j}^{n} \alpha_i \alpha_j y_i y_j \langle \boldsymbol{x}_i, \boldsymbol{x}_j \rangle \right), \qquad \text{s. t.} \sum_{i=1}^{n} \alpha_i y_i = 0 \tag{3}$$

$$\textbf{Classifier}^1\textbf{:} \quad f(\boldsymbol{x}) = \ b + \boldsymbol{w}^T\boldsymbol{x} = b + \sum_{i=1}^{n} \alpha_i y_i \langle \boldsymbol{x}_i, \boldsymbol{x} \rangle \tag{4}$$

Of all the points in the dataset, only a few determine the parameters of the classifier. These are called **support vectors**, and they lie on two hyperplanes parallel to the separating hyperplane, where $f(\boldsymbol{x}) = \pm 1$. **Figure 3.a** shows an illustration of a linear SVM for 2D data, where the separating hyperplane is a 1D line. In general, a feature space of d dimensions is partitioned by a $(d-1)$-dimensional hyperplane.

[1] The notation of f is slightly abused here, as it is used to refer to the *decision function* as well as the *classifier*, which simply thresholds the decision function to output class labels. It is generally understood from the context, but distinction will be made when necessary.

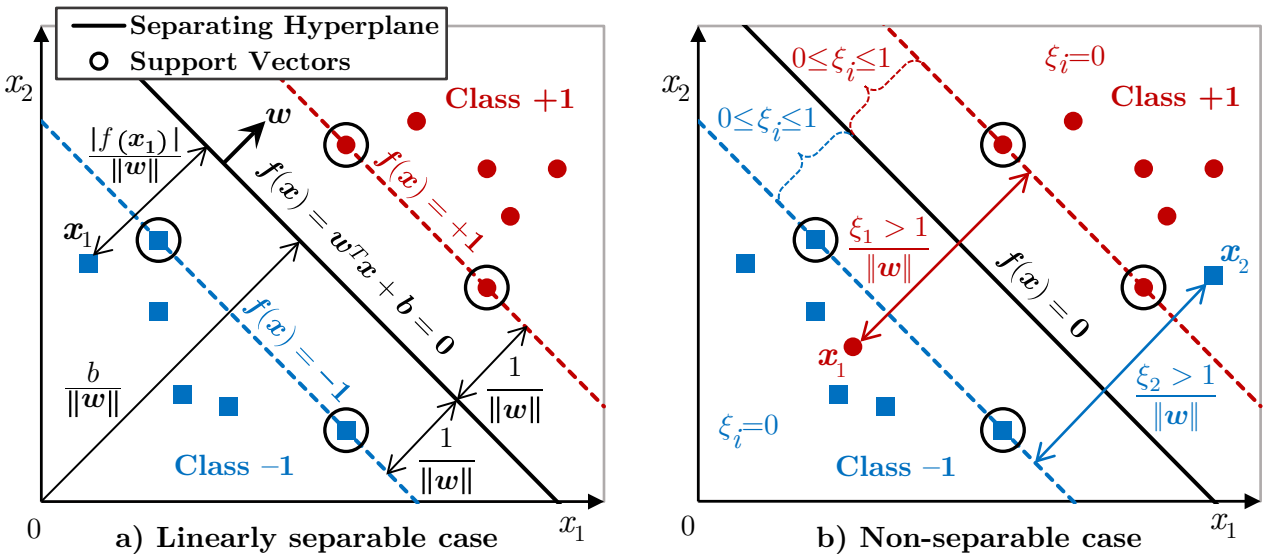


Figure 3. Illustration of a linear SVM.

SVMs can allow misclassification in a controlled way to deal with non-separable data. In this case, slack variables $\xi_i \geq 0$ are used to soften the classification constraint to become $y_i(\boldsymbol{w^T x_i} + b) \geq 1 - \xi_i$, with $\xi_i > 1$ indicating misclassification as illustrated in **Figure 3.b**. These slack variables can be represented as a *hinge loss* function where $\xi_i = \max\big(0, 1 - y_i(\boldsymbol{w^T x_i} + b)\big)$. The total error $\sum \xi_i$ is added to the objective function with a penalty $C \geq 0$ that represents a trade-off between the two competing goals: *maximising the margin* and *minimising the error*. This results in a **soft margin SVM** which solves the following optimisation problem:

$$\min_{\boldsymbol{w},b,\xi} J(\boldsymbol{w},\xi) = \tfrac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{i=1}^{n}\xi_i \qquad \text{s.t.:} \quad y_i(\boldsymbol{w^T x_i} + b) \geq 1 - \xi_i, \quad \xi_i \geq 0$$

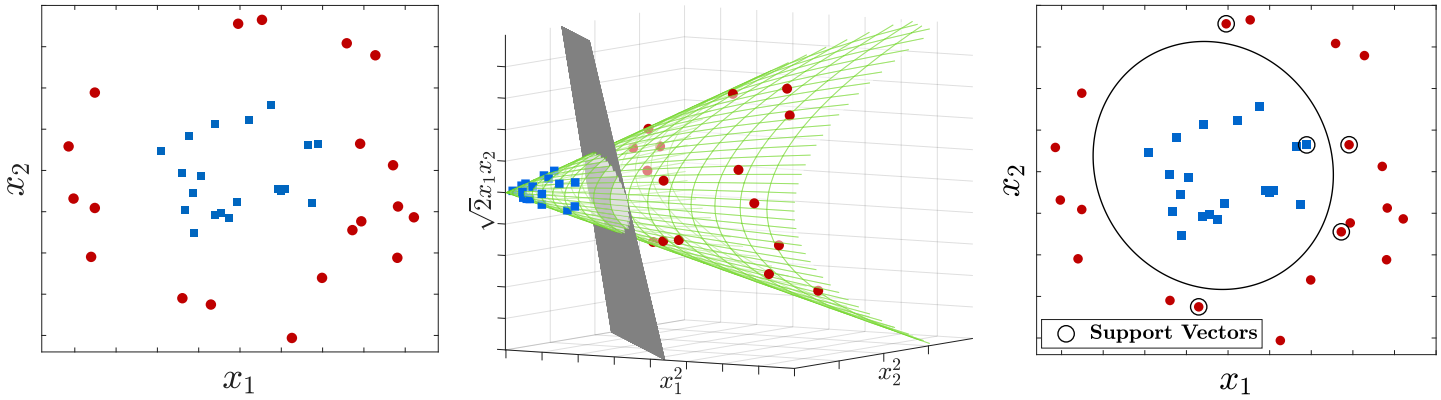
This gives the same **dual problem** as the hard margin case in **Eq. 3**, but with C as the upper bound for the Lagrangian weights α_i:

$$\max_{\alpha \geq 0} \left(\sum_{i=1}^{n}\alpha_i - \sum_{i,j}^{n}\alpha_i\alpha_j y_i y_j \langle \boldsymbol{x}_i, \boldsymbol{x}_j\rangle\right), \quad \text{s.t.} \sum_{i=1}^{n}\alpha_i y_i = 0, \quad \boldsymbol{0 \leq \alpha_i \leq C} \qquad \textbf{(5)}$$

Although SVMs are linear classifiers, they can be extended to perform **non-linear classification**. This can be done by mapping the non-linearly separable data into another space where classes become linearly separable. In principle, linear separability can always be achieved by mapping into a space with sufficiently high dimensionality. **Figure 4** shows an example of this scenario, where a polynomial function is used for mapping. Formally, given a mapping Φ, we can generalise the classifier f in **Eq. 4** by replacing the inner product of the data $\langle \boldsymbol{x_i}, \boldsymbol{x}\rangle$ with the inner

product of their images under Φ. With SV being the set of support vectors, the new definition of f becomes:

$$f(\boldsymbol{x}) = b + \sum_{i|\boldsymbol{x}_i \in SV} \alpha_i y_i \langle \Phi(\boldsymbol{x_i}), \Phi(\boldsymbol{x}) \rangle \tag{6}$$



(a) Data in their 2D space. **(b)** Data mapped into a 3D space. (c) The resulting decision boundary.

Figure 4. Mapping non-linearly separable data into another space to achieve linear separability. A quadratic polynomial is used to map the non-linearly separable 2D data shown in **(a)** into a 3D space where they are linearly separable. The intersection between the separating hyperplane and the data surface in **(b)** results in a non-linear decision boundary when mapped back into the 2D space as shown in **(c)**.

A more efficient approach is applying the '**kernel trick**', where an equivalent kernel function $K(\boldsymbol{x}, \boldsymbol{x}') = \langle \Phi(\boldsymbol{x}), \Phi(\boldsymbol{x}') \rangle$ is used instead of mapping data explicitly into another space. This gives us the following general form of **kernel SVMs**:

$$f(\boldsymbol{x}) = b + \sum_{i|\boldsymbol{x}_i \in SV} \alpha_i y_i K(\boldsymbol{x_i}, \boldsymbol{x}) \tag{7}$$

The basic case is the **linear kernel** $K(\boldsymbol{x}, \boldsymbol{x}') = \langle \boldsymbol{x}, \boldsymbol{x}' \rangle$ which simply returns the inner product in the original space, giving the same classifier as in **Eq. 4**. The most commonly used non-linear kernels are the polynomial and radial basis function (RBF) kernels. A **polynomial** kernel of degree q is defined as $K(\boldsymbol{x}, \boldsymbol{x}') = (a\langle \boldsymbol{x}, \boldsymbol{x}' \rangle + c)^q$. The **RBF** (Gaussian) kernel has a general formula of $K(\boldsymbol{x}, \boldsymbol{x}') = \exp\left(-\gamma \|\boldsymbol{x} - \boldsymbol{x}'\|^2\right)$ where γ determines the radius of influence from the centre of the kernel.

As for the output of SVMs, they typically perform *hard classification* based on the sign of f, with a predicted class label $\hat{y} = \text{sgn}\big(f(\boldsymbol{x})\big) \in \{-1, +1\}$. However, some methods use *soft classifiers* that return *real values* based on the distance from the decision boundary [43]. Also, since SVMs are *binary* classifiers, several approaches have been developed to combine multiple SVMs to solve multi-class problems. Examples

include one-vs-one, one-vs-all, and directed acyclic graph SVMs. Another approach is to include all classes in a single optimisation problem rather than decomposing the task into several binary sub-problems [44].

2.4 The DREBIN Dataset and Classifier

The original DREBIN dataset contains 129,013 android applications, of which 123,453 are benign, and 5,560 are malicious samples from 179 malware families [1]. The feature set S consists of eight subsets of static features that are extracted from either the manifest file (`AndroidManifest.xml`) or the bytecode (classes.dex). Features are binary, with their values determined by an indicator function $I(\boldsymbol{x}, s)$ that returns 1 if application $\boldsymbol{x}$ has feature s and 0 otherwise. The original dataset produced a very large feature set of size $|S| \approx 545{,}000$. However, samples are represented with highly sparse vectors, each exhibiting only a few features. **Table 1** lists the eight subsets of features, their source, and their cardinalities as per the original dataset.

Table 1. The feature subsets of the original DREBIN dataset [45].

ID	Subset	Cardinality	Source
S_1	Hardware components	4513	Manifest
S_2	Requested permissions	3812	
S_3	Application components	218951	
S_4	Filtered intents	6379	
S_5	Restricted API calls	733	Code
S_6	Used permissions	70	
S_7	Suspicious API calls	315	
S_8	Network addresses	310447	

The DREBIN classifier is a linear SVM model, with a decision function $f(\boldsymbol{x}) = \sum_{s \in S} I(\boldsymbol{x}, s) \cdot w_{\mathrm{s}} - t$, where t is the classification threshold. Since each application exhibits only a few features, the decision function is simplified as $f(\boldsymbol{x}) = \sum_{s \in \boldsymbol{x}} w_{\mathrm{s}} - t$. The predicted class label is $\hat{y} = \mathrm{sgn}(f(\boldsymbol{x})) \in \{+1\ (\mathit{malware}), -1\ (\mathit{benign})\}$. The trained model has a true positive rate (recall) of 94% and a false positive rate (fallout) of 1% as reported in [1].

2.5 Generative Adversarial Networks

Generative adversarial networks (**GANs**) were introduced in 2014 by Ian Goodfellow et al. [46]. A **GAN** is a *generative model* consisting of a **generator** and a **discriminator** that are simultaneously trained by competing against each other. The goal of a GAN is to *implicitly* model a target data distribution and generate new samples that simulate the real data. The **generator** $\boldsymbol{G}$ transforms a vector of random noise $\boldsymbol{z}$ (from a latent space) into a fake sample $\boldsymbol{x}' = G(\boldsymbol{z})$. The **discriminator** $\boldsymbol{D}$ receives both fake and real samples and aims to distinguish between the two. Both models are trained in an adversarial *minimax* game. The generator attempts to minimise the discriminator's ability to distinguish fake samples from real ones, while the discriminator aims to maximise such ability. As the generator improves, the performance of the discriminator becomes worse until it can no longer discriminate fake samples from real data. Equilibrium or convergence is reached when the accuracy of the discriminator is 50%. The basic GAN structure is illustrated in **Figure 3.**

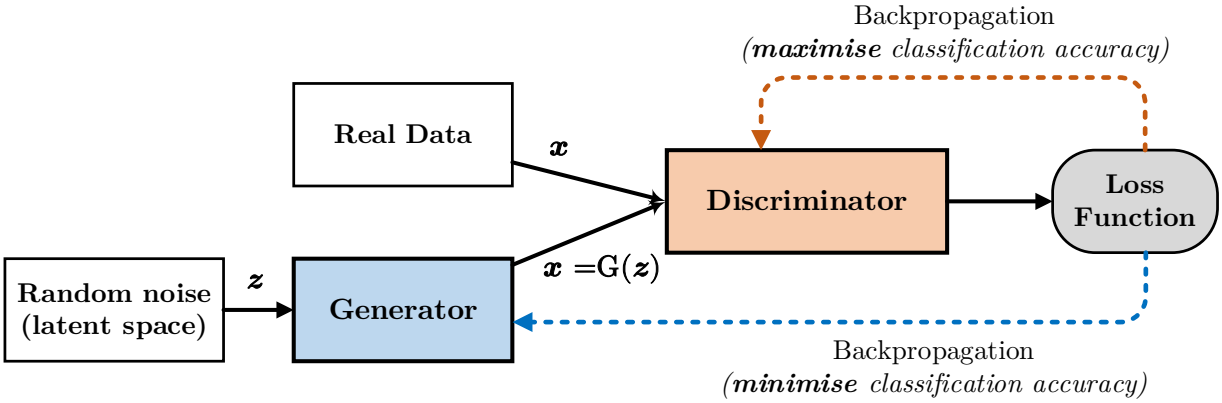


Figure 5. Basic GAN structure

Formally, the minimax game is formulated as a value function based on the cross-entropy between the distribution of the real data and the distribution of the generated samples [46]:

$$\min_{\mathrm{G}} \max_{D} V(D,G) = \mathbb{E}_{\boldsymbol{x}}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}}\big[\log\big(1 - D\big(G(\boldsymbol{z})\big)\big)\big] \tag{8}$$

The goal of the generator is to minimise this function, while the discriminator aims to maximise it. In particular, the discriminator is trained to *maximise* the two loss terms by maximising its estimates in detecting real data $D(\boldsymbol{x})$ while minimising its estimates for the generated samples $D(G(\boldsymbol{z}))$. On the other hand, the generator can only

influence the second term, so it is trained to *minimise* $\log\big(1 - D\big(G(\boldsymbol{z})\big)\big)$. To balance the game between the two players, the adversarial network alternates between training the discriminator and the generator. The loss functions are optimised using stochastic gradient descent. The steps for GAN training are shown in **Algorithm 1**.

Algorithm 1. GAN training using minibatch stochastic gradient descent [46].

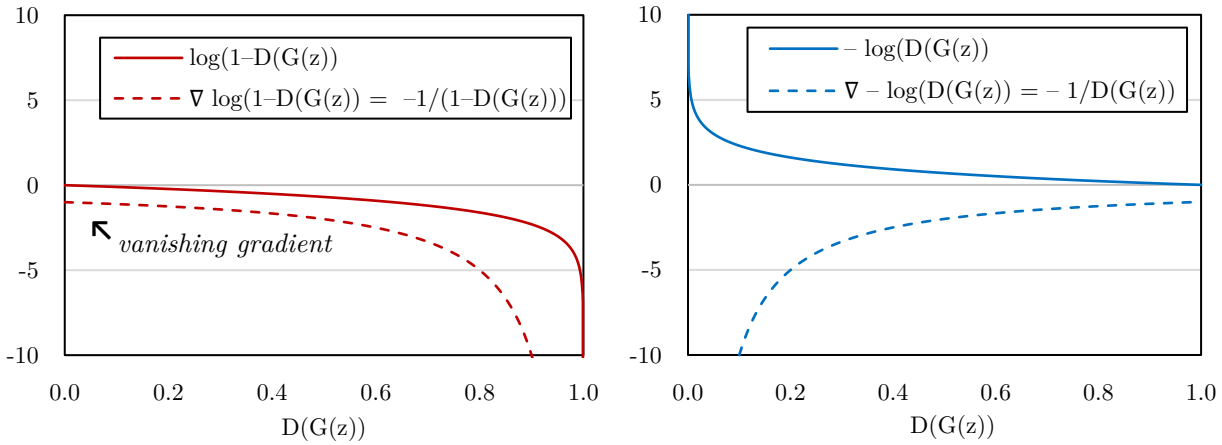
for n training iterations **do**
  for k steps **do**
    Sample a minibatch of m **noise** samples $\{\boldsymbol{z}^{(1)}, \dots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    Sample a minibatch of m **real** samples $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$ from $p_{data}(\boldsymbol{x})$.
    Update the **discriminator** by ascending its stochastic gradient:
$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D(\boldsymbol{x}^{(i)}) + \log\left(1 - D\left(G(\boldsymbol{z}^{(i)})\right)\right)\right]$$
  end for
  Sample a minibatch of m **noise** samples $\{\boldsymbol{z}^{(1)}, \dots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  Update the **generator** by descending its stochastic gradient:
$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G(\boldsymbol{z}^{(i)})\right)\right)$$
end for

2.5.1 GAN Challenges

There are several challenges in training GANs. This section will briefly cover the most common issues as well as proposed remedies.

Vanishing gradients. As pointed out in the original paper [46] and further investigated in [47], training the generator using the original loss, $\log\big(1 - D\big(G(\boldsymbol{z})\big)\big)$, may fail due to vanishing gradients. This occurs when the generator performs poorly, especially early on in training, or whenever the discriminator performs too well. This results in the saturation of the loss function, thus diminishing the gradients necessary for the generator to learn. One solution to this problem is training the generator to *maximise* $\log D\big(G(\boldsymbol{z})\big)$ instead of minimising $\log\big(1 - D\big(G(\boldsymbol{z})\big)\big)$ [46]. Both losses and their gradients are shown in **Figure 6**, where we can see the behaviour of the gradient in each case. Other approaches replace the original minimax loss with functions that have more desirable gradient properties. A notable example is the *Wasserstein loss* which has been implemented in the **Wasserstein GAN** (WGAN) [48]. In WGAN,

even when the discriminator (*critic*) is trained to optimality, it still provides usable gradients to train the generator.



(a) Original loss and its gradient

(b) Modified loss and its gradient

Figure 6. A comparison between the original and modified losses of the generator. In case of the original loss **(a)**, the gradient diminishes as $D\big(G(z)\big)$ approaches zero, while this is not the case with the modified loss **(b)**.

Instability and non-convergence. GANs are known to be hard to train and may fail to converge [49]. Unlike other models that are trained to minimise a loss function for a single network, GAN training aims to find an equilibrium between two players in a minimax game. This frequently results in the oscillation of the loss as each network is updated with gradient descent to counter the progress of the other. This requires a careful balance between the generator and the discriminator such that the game is not dominated by a single player. Even when equilibrium is reached, it is often unstable in such a way that further training may degrade the performance of the GAN. A few approaches have been proposed to improve the stability of GAN training. Examples include *regularising the discriminator* through penalising it weights [50], adding noise to its inputs [47], or smoothing the target labels [51]. Another technique is *historical averaging* which keeps track of parameter updates over time and penalises their divergence from the running average [51]. *Feature matching* and *virtual batch normalisation* are two other approaches proposed in [51]. Furthermore, variants of GAN such as WGAN offer improved training stability [48].

Mode collapse. Another manifestation of non-convergence in GANs is mode collapse, where the generator learns to produce the same output or a small subset of outputs with limited variability [49]. In other words, the generator learns only a few

modes of the data distribution and ends up mapping many inputs to the same output. This might not be a problem if the goal is to produce a small number of acceptable outputs or just *any* plausible output. However, the goal of using GANs is often to capture the whole target distribution and generate diverse outputs. To mitigate mode collapse, several approaches have been proposed. Unrolled GANs [52] aim to stabilise the generator by updating it based on multiple lookahead steps of training with the discriminator. This prevents the generator from being over-optimised for the current discriminator, and it was shown to reduce mode collapse. Another approach is minibatch discrimination [51] in which the discriminator penalises samples that are similar to others in a generated batch, thus encouraging diverse outputs. Furthermore, GAN variants such as WGAN are more resistant to mode collapse [48].

2.5.2 GAN Variants

Since their introduction in 2014, GANs have attracted much interest and inspired the development of numerous variants for several purposes. These variants introduce different architectures, objective functions, or training strategies. The following are a few notable examples.

As mentioned earlier, **WGANs** use the *Wasserstein loss* which allows stable training and alleviates mode collapse. In Deep Convolutional GANs (**DCGANs**) [53], D and G are *convolutional networks* (CNNs) where D has *convolutional* layers while G has *transposed convolutional* layers. In Conditional GANs (**CGANs**) [54], training is conditioned on additional information (such as a target label) that is appended to the inputs so that the network can learn a *conditional* distribution. Auxillary CGANs (**ACGANs**) [55] are an extension of CGANs where D not only outputs the probability of a sample being real but also learns to reconstruct the added label. Least-Squares GANs (**LSGANs**) [56] use the *least-squares loss* which has shown to improve the quality of the generated samples. **CycleGANs** [57] use two GANs for *cross-domain translation*, where one GAN translates a sample from one domain to another (forward cycle) while the other GAN translates it back to the original domain (backward cycle). **Progressive GANs (PGANs)** [58] start with small networks and incrementally add blocks of layers to allow generating large high-quality samples. The quality was further improved in **StyleGANs** [59] which added *style transfer* techniques to the generator.

3. Related Work

Numerous studies have investigated **evasion attacks against malware classifiers**. This chapter will briefly review many of them in terms of methods, target models, datasets, and results. We will then cover studies that have used **generative models**, including GANs, to generate AEs either in the image or malware domains.

3.1 Evasion Attacks against Malware Classifiers

Biggio et al. [14] used a PDF malware dataset to train different models (linear SVM, RBF SVM, neural network). They performed evasion attacks against the trained classifiers both in white-box and grey-box scenarios. AEs were crafted using gradient descent on an objective function with density estimation of the target class to maximise the confidence of the attack. Perturbations were limited to incrementing features in order to preserve functionality. Attacks showed varying degrees of success against the different models under different configurations.

Šrndić and Laskov [60] performed grey-box *mimicry* attacks against a PDF malware classifier that is based on random forests. They trained several surrogate classifiers reflecting varying levels of the attacker's knowledge of the feature set, training data, and target classifier. AEs were crafted by merging the feature vector of a benign sample with that of a malicious one, while limiting the set of modifiable features to preserve functionality. To convert AEs from the feature space into an actual file in the problem space, they exploited a *semantic gap* between the way features are extracted by the target classifier and the way files are parsed by PDF readers. This allowed them to inject features into regions that are only considered by the classifier, without affecting the validity of the resulting file as rendered by PDF readers. Attacks resulted in reducing the output score from 100% for the original samples to a median score of 24-42% for the adversarial samples, depending on the attacker's knowledge.

Xu et al. [25] used genetic programming to evade two PDF malware classifiers, one based on random forests and the other is an RBF SVM. With rough knowledge of the likely feature representation, attacks were performed in a black-box scenario requiring only the classification scores. To generate an AE, a malicious file (represented

as a logical tree-like structure) is used as a seed for a population of variants that keep mutating through generations until an evasive variant is found. Mutations included *deletion*, *insertion*, or *replacement,* using a pool of objects extracted from benign files. A *repacker* was used to convert the generated variants into files, and an *oracle* was used to ensure that the malicious behaviour was preserved. Using a set of 500 malicious samples as seeds for AEs, the evasion rate was 100% against each of the two target classifiers.

Grosse et al. [45] trained a deep neural network classifier on the DREBIN dataset, achieving an accuracy of 96-98%. AEs were crafted using an adapted Jacobian Saliency Map Attack (JSMA) against the trained classifier in a white-box scenario. To preserve functionality, perturbations were limited to only adding features with minimal footprint in the problem space application (`manifest` features). The attacks achieved an evasion rate of 63-69%.

Demontis et al. [61] investigated problem-space and feature-space evasion attacks against different SVM models trained on the DREBIN dataset. In the problem space, attacks were based on obfuscation, which generally showed limited effectiveness in evading the classifiers. In the feature space, two types of attacks were performed: mimicry and gradient-based attacks. Both attacks involved adding or removing features, but feature removal was limited to non-`manifest` features. Mimicry attacks were based on the probability distributions of malicious and benign samples in a surrogate dataset. Gradient-based attacks were performed in both white-box and grey-box scenarios, where each step adds (removes) the feature with the most weight in the direction of the benign class (malicious class). Mimicry attacks were less effective compared to gradient-based attacks, which showed varying degrees of success against the different models. The main contribution of the study was proposing a more secure classifier with *evenly distributed* feature weights, which would require an adversary to manipulate more features to achieve evasion.

Demontis et al. [62] used the DREBIN dataset to train several classifiers to investigate the transferability of evasion attacks across models. AEs were generated using projected gradient descent with maximum confidence for better transferability. Attacks were performed in white-box and grey-box (transfer) scenarios, and

perturbations were limited to only adding features to preserve functionality. The results showed that lower-complexity models were less vulnerable to evasion, and that non-linear models tended to be less vulnerable than linear ones. Also, surrogate models with lower complexity were shown to offer better transferability of attacks.

Pierazzi et al. [28] investigated the *inverse feature-mapping problem* mentioned in **Section 2.2.2**. They proposed a formalisation that provides necessary and sufficient conditions for problem-space attacks. The formalisation includes four sets of problem-space constraints, namely: available transformations, preserved semantics, plausibility, and absent artefacts. It also introduces the concept of side-effect features, which allow mapping attacks between feature and problem spaces. Based on this formalisation, they performed problem-space attacks on two variants of DREBIN classifier. Attacks were crafted with automated software transplantation using program slices from benign samples and following the gradient in the direction of the benign class. The attacks achieved an evasion rate of 100% against the two target classifiers under different configurations.

3.2 Using Generative Models to Generate AEs

Rather than solving an optimisation problem to generate an AE for each instance individually, **generative models** can be trained to generate AEs in a more efficient and generalisable way. In this section, we will briefly review studies that have used generative models for this purpose on either images or malware.[1]

Baluja and Fischer [64] proposed Adversarial Transformation Networks (ATNs) to generate AEs with minimal perturbations. They trained a neural network to optimise an objective function to achieve misclassification (using a *reranking function*) while being constrained by a similarity measure (using L_2 *loss*). Two different approaches were proposed to generate AEs: 1) Perturbation ATNs, and 2) Adversarial Autoencoding. A perturbation ATN is essentially a *residual network* G that takes a

[1] In particular, we will cover methods that are based on **additive perturbations in the input space**, which are most relevant to this project. This excludes other approaches such as those proposed in [10] and [63]. In [10], Zhao et al. used GANs to map input data into a dense semantic space where perturbations are applied to generate *natural* AEs. In [63], Song et al. proposed a totally different approach that is not based on perturbations, where GANs are used to generate AEs from scratch.

sample as input and generates perturbations that are *added to the input* to produce an AE: $\boldsymbol{x}' = \boldsymbol{x} + G(\boldsymbol{x})$. Adversarial autoencoding, on the other hand, uses an *autoencoder* to reconstruct the input $\boldsymbol{x}' = G(\boldsymbol{x})$, subject to the misclassification requirement which necessitates perturbations to cause misclassification. Perturbation ATNs were found to produce similar perturbations that worked on most samples regardless of the input, while autoencoding produced input-specific AEs. Attacks were performed in a white-box scenario where the error derivatives of the target model were used to train the ATN. Different ATN architectures and different weights of the two objectives (misclassification **vs** similarity) resulted in varying misclassification rates.

Poursaeed et al. [65] used two approaches that are similar to the perturbation ATNs in [64]. One approach transforms *noise* $\boldsymbol{z}$ into universal perturbations that can be added to any image to construct an AE: $\boldsymbol{x}' = \boldsymbol{x} + G(\boldsymbol{z})$. The other approach takes an image as input to generate image-dependent perturbations that are added to the input to produce an AE: $\boldsymbol{x}' = \boldsymbol{x} + G(\boldsymbol{x})$. Rather than including a similarity term in the loss function, perturbations were scaled to keep their magnitude within a certain limit. In addition to fooling image classifiers, AEs were also used against semantic segmentation models. Attacks were performed in a white-box setting and resulted in high misclassification rates against the target models, as well as varying degrees of transferability between models.

Hayes and Danezis [66] used a generative model to transform *noise* into universal perturbations, in a similar way to one of the approaches in [65]. They adapted the objective function of the Carlini-Wagner attack which incorporates a similarity term. They also scaled perturbations to impose an upper bound on their magnitude. Attacks were performed in a white-box scenario, and the results showed varying success rates and transferability across models.

Xiao et al. [67] proposed AdvGAN, a framework that generates AEs based on GANs. AdvGAN uses a generator that produces perturbations in a way similar to that of the perturbation ATNs in [64]. However, instead of using a similarity metric to constrain AEs as in ATNs, AdvGAN uses a discriminator network to keep AEs similar to the original samples. The objective function combines the losses of the GAN (similarity), the target classifier f (misclassification), as well a hinge loss to enforce an

upper bound on perturbations: $\mathcal{L} = \mathcal{L}^{f}_{adv} + \alpha\mathcal{L}_{\text{GAN}} + \beta\mathcal{L}_{hinge}$. Attacks generally resulted in high misclassification rates against different target models and defences in both white-box and black-box settings.

In the malware domain, Hu and Tan [68] designed MalGAN, a GAN-based model to perform grey-box evasion attacks against malware classifiers. Different target classifiers were trained and tested on a dataset of 180 thousand applications (70% benign, 30% malicious) represented using API features. MalGAN uses a generator that takes a malware sample and noise as input to generate perturbations. To preserve functionality, these perturbations are added to the input sample using a `max` operation to allow only feature addition. The resulting AE is labelled by the target classifier and fed to the discriminator, which acts as a surrogate classifier that the generator tries to evade. MalGAN relies on the GAN loss without setting an upper bound on perturbations, which was not justified by the authors. This resulted in perfect evasion rates against the different target classifiers. However, adding unbounded perturbations limits the feasibility of mapping the generated AEs into problem-space objects, which should be taken into account when interpreting the results.

A few other studies have investigated the use of GANs against security models. Examples include using GANs to evade intrusion detection systems [69], C&C domain name detectors [70], as well as phishing detectors [71].

4. Methodology and Implementation

4.1 The Dataset

The dataset consists of the feature vectors of 129,728 android applications, of which 12,735 (~10%) are malware and 116,993 are goodware. As mentioned in **Section 2.4**, these vectors are binary, representing static features that are extracted from either the manifest file (`AndroidManifest.xml`) or the bytecode (classes.dex).

The dataset produced a very large feature set, with over a million (1,140,816) distinct features. Feature vectors, however, are highly sparse, with each exhibiting only a few features (88 on average). **Table 2** shows some basic statistics on the number of features per sample in each class.

Table 2. Statistics on the number of features per application.

	Goodware	**Malware**
Mean	**88**	**94**
Standard deviation	105	70
Minimum	**0**	**6**
Q1 (25%)	44	45
Q2 (50%)	70	68
Q3 (75%)	110	118
Maximum	**6,591**	**1,296**

4.2 Feature Selection

Analysing the features and their distribution in the dataset reveals that the majority of the features provide irrelevant information that is not useful for classification. For example, about **82%** of the features are observed only once in the whole dataset. This makes them unique to the samples they appear in, rather than representative of their classes. Moreover, applying the **χ^2 test** to the features shows that about **90%** of them are *statistically independent* of class labels (at a significance level $\alpha = 0.05$). This means that, regardless of their frequency, the majority of features do not seem to be associated with a particular class, which makes them unsuitable for classification.

Consequently, feature selection has been applied to reduce the dimensionality of the feature space. In particular, the feature set has been reduced to the top (*more discriminative*) **10,000 features** according to the $\boldsymbol{\chi^2}$ **test**. Although the reduced feature set represents $< 1\%$ of the total features, the scores of training the classifier on the reduced set are comparable to those obtained when using all features. Also, training the classifier on the whole feature set resulted in more than 85% of the features being assigned zero weights, which nullifies their contribution to the classification decision. The results of training on the reduced feature set are discussed in the following section.

4.3 Training the Target Classifier

The target classifier f is a linear SVM model, based on `scikit-learn`'s `LinearSVC` class [72] which uses the `LIBLINEAR` library [73]. To train the classifier, the dataset was split into a 70% training set and a 30% test set. To avoid *spatial bias,* as suggested in [74], the split was *stratified* to maintain the class ratio (1 malware: 9 goodware) in each set. The hyperparameter C was determined using grid search with cross-validation on a range of values and class weights. Both $C = 0.1$ and $C = 1$ gave similar results that were better than those of other values, so $C = 1$ was chosen. Also, adjusting C to account for the class imbalance by adding more weight to malware misclassification showed better results. Class weights (2 malware: 1 goodware) were used.

Table 3. The evaluation scores for the trained classifier.

	Training Set	**Test Set**
Accuracy	98.83%	98.18%
Recall (TPR)	94.07%	**90.32%**
Precision	93.99%	91.03%
F1-Score	94.03%	90.67%
Fallout (FPR)	$00.65\% < 1\%$	$\mathbf{00.97\% < 1\%}$

The scores for evaluating the trained classifier on both the test and training sets are shown in **Table 3**. Since these scores are focused on the positive (malware) class, the confusion matrices are shown in **Figure 7** to shed light on the classifier's performance on the goodware class too. Evaluation on the training set (*resubstitution* evaluation) is given to show how much error was tolerated for the chosen C. Also,

comparing the scores on the training set to those on the test set allows the detection of overfitting, which is not apparent here. Of our interest is the true positive rate (**TPR**) on the test set, which is **90.32%** at a false positive rate (**FPR**) of **<1%.** The TPR is the metric we will be targeting in our evasion attacks. That is, we aim to generate AEs that will minimise the TPR as much as possible.

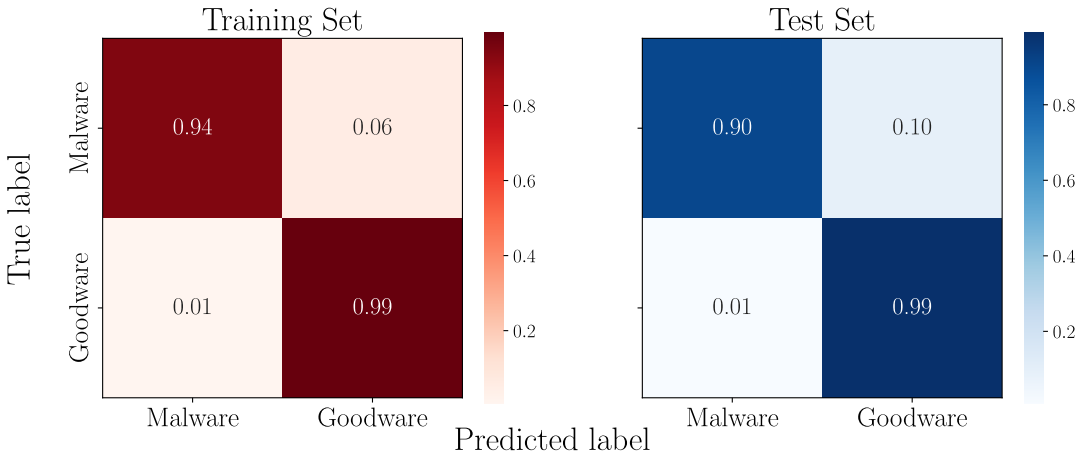
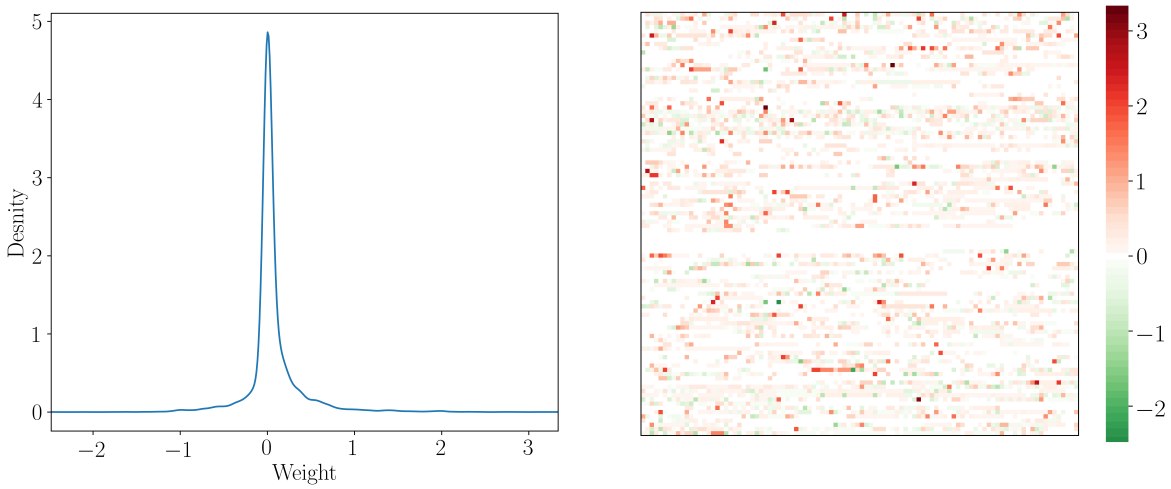


Figure 7. The confusion matrices on the training and test sets

To get an insight into the complexity of the classifier and understand its decisions, we can analyse its parameters. Feature weights are computed based on the support vectors of the classifier, which comprised around 16% of the training data. About 77% of the features are assigned non-zero weights, which means only these features contribute to the classification decision. Features with positive weights move the decision towards the malware class, while those with negative weights move it towards the goodware class. **Table 4** gives some statistics on the feature weights. **Figure 8.a** plots the weight density which shows that the majority of weights are close to zero. **Figure 8.b** shows a heatmap visualising features according to the sign and magnitude of their weights.

Table 4. Statistics on the feature weights.

Mean ± Std	**0.0653 ± 0.3137**
Minimum	**−2.4799**
Q1 (25%)	0.0
Q2 (50%)	0.0
Q3 (75%)	0.0807
Maximum	**3.3362**



(a) Weight density

(b) Weight heatmap (100×100=10K features)

Figure 8. Feature weights (model coefficients). The density plot in (a) shows that most weights are close to zero. The heatmap in (b) visualises features based on the sign and magnitude of their weights. This reflects what the classifier has learnt as well as the influence of each feature in the classification decision towards either the malware or the goodware class.

4.4 Threat Model

The goal of the attacks is to generate *minimally perturbed* AEs against the target classifier f. An attack starts with a sample that is classified by f as malware and applies *minimal perturbations* that cause f to misclassify it as goodware. Constraining perturbations to be *minimal* is a way of increasing the chances of mapping the perturbed AE to a valid *problem-space* object.[1] It also minimises the cost of the attack by reducing the number of changes the attacker has to make to evade the classifier. Another important constraint is that perturbations can only *add* features, in order to preserve the functionality of the original sample, as discussed in **Section 2.2.2**.

We will consider attacks in **grey-box settings,** where the attacker only has a *query* access to the target f, with no knowledge of its architecture or parameters. However, we assume the attacker has access to a surrogate dataset. As a baseline, we will consider the case where the attacker has access to the training dataset of f. Although this is hardly the case in the real world, we will consider this as a worst-case scenario. Furthermore, it has been shown that training a surrogate classifier on a different dataset[2] still gives a good approximation that allows attacking the target [62]. In [68], both approaches were used (training a surrogate classifier on the same or a

[1] In the image domain, minimising perturbations typically aims to increase their *imperceptibility*.

[2] Ideally from the same distribution as that of the target classifier's training data.

different training set) and both gave similar results. Therefore, we do not consider this to be a limitation, especially given that we will be using labels assigned by f as the ground truth in all cases (malware, goodware, and adversarial samples).

4.5 EvadeGAN Overview

EvadeGAN consists of a generator G and a discriminator D. On the one hand, D is trained to approximate the target f and act as a surrogate classifier. On the other hand, G is trained against D to learn *a mapping* that transforms any input into an AE that can evade f.

D is a feed-forward neural network with the structure shown in **Table 5**. Its output ranges from 0 (goodware) to 1 (malware). Feedback from the target f is used during training so that D can approximate it. In particular, we use f to label the original samples (malware or goodware) and the generated AEs. Since the goal is to evade f, EvadeGAN is trained only on samples that are correctly classified by f.

Table 5. The structure of the discriminator D.

Layer	**Shape**
Input	10000
Hidden (Fully-Connected + ReLU)	256
Output (Fully-Connected + Sigmoid)	1

G is also a feed-forward neural network, with three input configurations. That is, it can take as input one of the following:

1) a malware sample $\boldsymbol{x}$ to produce a perturbation vector $\boldsymbol{\delta} = G(\boldsymbol{x})$, [1]

2) a noise vector $\boldsymbol{z}$ (with elements $z \sim U(0,1)$) to produce $\boldsymbol{\delta} = G(\boldsymbol{z})$, [2] or

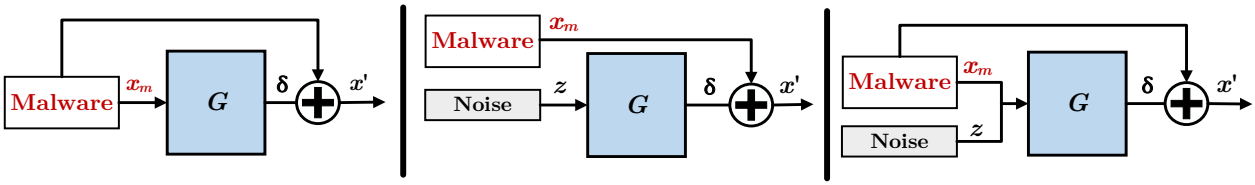
3) a combination of both, to produce $\boldsymbol{\delta} = G(\boldsymbol{x}, \boldsymbol{z})$. [3]

Figure 9 shows block diagrams for each of these configurations. In all cases, perturbations are produced by an output layer of the same dimensions as the malware samples. Sigmoid activations are used to give outputs in the range $[0, 1]$. As in [68], the

[1] G in this setting is similar to perturbation ATNs in [64].

[2] G in this setting is similar to one of the approaches in [65].

[3] G in this setting is similar to the generator in MalGAN [68].



(a) $\boldsymbol{\delta} = G(\boldsymbol{x})$ (b) $\boldsymbol{\delta} = G(\boldsymbol{z})$ (c) $\boldsymbol{\delta} = G(\boldsymbol{x}, \boldsymbol{z})$

Figure 9. The different types of the generator inputs

perturbations are added to the malware sample using a `max` operation to produce an AE. The resulting AE, therefore, preserves the features of the original sample along with other added ones. During training, these added features have sigmoid values, which need to be thresholded to give a valid AE with binary features during evaluation. The structure of G is shown in **Table 6**.

Table 6. The structure of the generator G.

Layer		**Shape**
Input	**1)** $\boldsymbol{x}$	10000
	2) $\boldsymbol{z}$	100
	3) $\boldsymbol{x} + \boldsymbol{z}$	10100
Hidden (Fully-Connected + ReLU)		256
Output (Fully-Connected + Sigmoid + Dropout)		10000

Based on which input is used to generate perturbations, EvadeGAN can operate in three modes: 1) **EvadeGAN$_{\mathbf{X}}$,** 2) **EvadeGAN$_{\mathbf{XZ}}$**, or 3) **EvadeGAN$_{\mathbf{Z}}$**. These modes can generate **two types of perturbations**:

A. Sample-dependent perturbations. This is the case when the sample $\boldsymbol{x}$ is used as an input to generate perturbations, as in **EvadeGAN$_{\mathbf{X}}$** and **EvadeGAN$_{\mathbf{XZ}}$**. The difference between the two approaches is that **EvadeGAN$_{\mathbf{X}}$** learns to map each given sample to one corresponding perturbation vector. On the other hand, the added noise in **EvadeGAN$_{\mathbf{XZ}}$** should introduce diversity in the generated vectors for any given sample.

B. Sample-independent (universal) perturbations. This is the case when only noise is used to generate perturbations, as in **EvadeGAN$_{\mathbf{Z}}$**. Using the same noise vector results in a universal perturbation vector that should ideally work on any sample. To generate various universal perturbation vectors, different noise vectors can be used.

4.6 EvadeGAN Training

In a typical GAN scenario, G is trained to produce *fake* samples that can fool D, which in turn is trained to distinguish them from *real* ones. The goal is for G to implicitly model the distribution of the *real* data. In our case, however, the training goals and dynamics are slightly different.

In EvadeGAN, D is meant to approximate a target f, via training with labels provided by f. This includes the correct labels for the original malware and goodware samples, as well as the labels given by f to the generated AEs.[1] Therefore, D is trained to minimise the cross-entropy loss between its predictions and the predictions of f:

$$\mathcal{L}_D = -\mathbb{E}_{\boldsymbol{x}|f(\boldsymbol{x})=1}[\log D(\boldsymbol{x})] - \mathbb{E}_{\boldsymbol{x}|f(\boldsymbol{x})=0}[\log(1 - D(\boldsymbol{x}))] \quad \textbf{(9)}$$

On the other hand, G is trained to generate perturbations that can be added to malware samples to produce evasive AEs against D with the goal of evading f. Therefore, EvadeGAN trains G to minimise the loss of classifying AEs as malware:

$$\mathcal{L}_G = \mathbb{E}_{\boldsymbol{x}'|\,\boldsymbol{x}'=\boldsymbol{x}+\boldsymbol{\delta}}[\log D(\boldsymbol{x}')], \text{ where } \boldsymbol{\delta} \text{ is either } G(\boldsymbol{x}), G(\boldsymbol{z}), \text{or } G(\boldsymbol{x}, \boldsymbol{z}) \quad \textbf{(10)}$$

Furthermore, to *minimise* perturbations, two penalties are added to $\mathcal{L}_G$. First, a *sparsity-inducing regulariser* (such as L_1, L_2, or a combination of both)[2] is added to sparsify $\boldsymbol{\delta}$ regardless of the malware sample. Second, a *hinge loss* penalty on the distance between $\boldsymbol{x}'$ and $\boldsymbol{x}$ is added to set an upper bound K on the number of changes. Each of these penalties is assigned a weight and added to **Eq. (10)** to form the total loss:

$$\mathcal{L}_G = \underbrace{\mathbb{E}_{\boldsymbol{x}'|\,\boldsymbol{x}'=\boldsymbol{x}+\boldsymbol{\delta}}[\log D(\boldsymbol{x}')]}_{\textit{Evasion loss}} + \underbrace{\alpha\| \boldsymbol{\delta} \|_1}_{\textit{Induce sparsity}} + \underbrace{\beta \max(0, \|\boldsymbol{x}' - \boldsymbol{x}\|_{\mathbf{1}} - K)}_{\textit{Impose an upper bound}} \quad \textbf{(11)}$$

[1] This is different from the approach in MalGAN [68], which trains D only on goodware and adversarial samples. Even though they present D as a substitute for the target model, our experimentation with their approach shows that D was not guaranteed to learn anything about the malware class. The reason appears to be that G starts to generate evasive AEs early on in training, which does not give D much exposure to the malware class. It was only after we lowered the learning rate for G that D was able to approximate f on the malware class. In our approach, D is trained on both malware and goodware samples, as well as the generated AEs.

[2] While L_1 *(lasso)* regularisation does induce sparsity, L_2 *(ridge)* regularisation only shrinks values closer to zero. However, it gives a similar effect in our case since perturbations are eventually thresholded to give binary values. Combining L_1 and L_2 regularisers is known as *elastic net* regularisation [75].

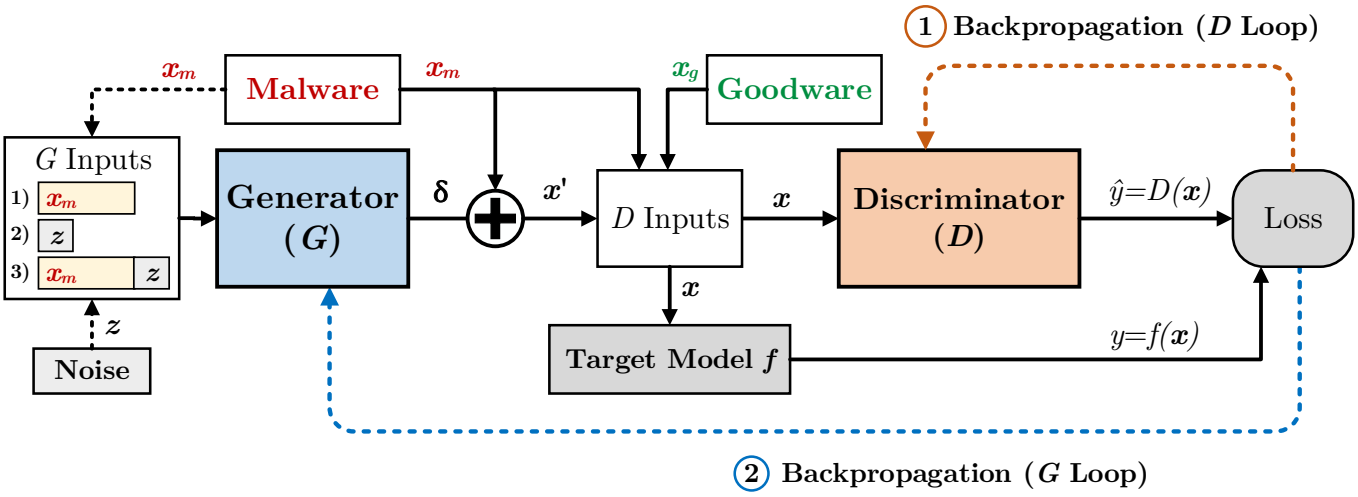


Figure 10. EvadeGAN training

EvadeGAN training is depicted in **Figure 10**, which shows the three input configurations for G. Regardless of G inputs, the general steps for training EvadeGAN are given in **Algorithm 2**. The training and validation data, as well as the specific training hyperparameters that were used in the experiments are described next.

Algorithm 2. EvadeGAN training.

---

repeat

  for k_D steps[1] **do**

    Sample a **goodware** minibatch (**as labelled by** f).

    Sample a **malware** minibatch (**as labelled by** f).

    Generate a minibatch of **AEs** and **label them by** f.

    Update D using SGD* on its loss on each of these minibatches[2]:

      $\nabla_{\theta_D}\mathcal{L}_D$, *where* $\mathcal{L}_D = -\mathbb{E}_{\boldsymbol{x}|f(\boldsymbol{x})=1}[\log D(\boldsymbol{x})] - \mathbb{E}_{\boldsymbol{x}|f(\boldsymbol{x})=0}[\log(1-D(\boldsymbol{x}))]$

  end for

  for k_G steps **do**

    Sample a **malware** minibatch (**as labelled by** f).

    Generate a minibatch of **AEs.**

    Update G using SGD* on its loss on the generated minibatch:

      $\nabla_{\theta_G}\mathcal{L}_G$, *where* $\mathcal{L}_G = \mathbb{E}_{\boldsymbol{x}'|\,\boldsymbol{x}'=\boldsymbol{x}+\boldsymbol{\delta}}[\log D(\boldsymbol{x}')] + \alpha\|\boldsymbol{\delta}\|_1 + \beta\max(0, \|\boldsymbol{x}'-\boldsymbol{x}\|_1 - K)$

  end for

until convergence OR reaching maximum N training epochs

---

*SGD: Stochastic gradient descent, the Nadam variant has been used.

[1] In the experiments, training alternates between *one* loop of D training and *one* loop of G training.

[2] During experimentation, training D on each minibatch separately gave better results than training on one mixed batch that combines all. So, in effect, each training loop updates D multiple times.

With f labels as the ground truth, the **training set** for EvadeGAN consisted of the correctly classified samples from the *training* set of f. This included 94.07% of the malware samples (*true positives*) and 99.35% of the goodware samples (*true negatives*). The **validation set** V for EvadeGAN consisted of the **true positives (3821 malware samples)** from the *test* set of f (90.32%of the *test* malware samples). The **noise vectors**, if used, are composed of random values sampled from a uniform distribution, $z \sim U(0,1)$.

To determine convergence, the training algorithm requires two hyperparameters: a target evasion rate R and a maximum number of allowed changes K. Convergence is defined as achieving the rate R against the target f on the validation set V within maximum K changes. The evasion rate is based on the true positive rate of f: $R = 1 - TPR_f$. Training experiments were conducted with a target evasion rate $R = 100\%$ and a maximum number of changes $K = 15$.

The optimiser used for both G and D was Nadam [76], with learning rates of 0.0005 for G and 0.001 for D. Optimising G at a slower rate than that of D was often found to better balance the training process. A minibatch size of 32 samples was used. Training stops either upon convergence or upon reaching a maximum of 500 epochs. In each epoch, malware minibatches are sampled such that all the malware samples in the training set are used in training G.

Two other hyperparameters are required for G loss in **Eq. (11)**: the weights α and β of the penalty terms. Their values have been empirically chosen from the ranges $\alpha \in [1, 100] \times 10^{-4}$ and $\beta \in [1,1000] \times 10^{-4}$. Taking into account the scale of each penalty, the weights are chosen such that the terms of the loss function are balanced in a way that achieves that desired results while offering a smooth optimisation. Unbalanced terms would result in the oscillation of the loss functions during optimisation.

EvadeGAN is implemented using `Keras 2.4.3` API on a `TensorFlow 2.2.0` backend. Therefore, the default settings of these two are used when a parameter is not explicitly defined (eg., weight initialization, optimiser weight decay, etc).

5. Evaluation and Results

In this chapter, we will present the results of evaluating EvadeGAN against the DREBIN classifier as the target model. We will also analyse the perturbations generated in each EvadeGAN mode to understand what it has learnt about AE generation. But first, we will describe a scenario of how a **gradient-based iterative algorithm** would generate an AE against our target f in a **white-box** setting. This will help us draw a comparison when evaluating the AEs generated by EvadeGAN in a **grey-box** setting.

5.1 An Iterative Algorithm Scenario

A gradient-based iterative algorithm typically generates an AE that is optimised for a given sample by following the gradient of the target classifier on that sample towards a target class. This requires white-box access to the classifier.

In our case, the target classifier f is a linear SVM with a decision function $f(\boldsymbol{x}) = b + \boldsymbol{w}^T\boldsymbol{x}$. With the features being binary, $x_i \in \{0, 1\}$, the decision function is effectively summing the weights of the features that are present in a given sample: $f(\boldsymbol{x}) = b + \sum_{i|x_i=1} w_i$. The gradient $\nabla f(\boldsymbol{x}) = \boldsymbol{w}$ means that the partial derivative of f with respect to an input feature x_i is the weight assigned to that feature: $\partial f/\partial x_i = w_i$.

A simple white-box iterative approach against f would go as follows. With the negative (goodware) class as the target, we need to *add* a feature x_i that has the *most influence* in moving the sample towards the negative class. This influence corresponds to the partial derivative $\partial f/\partial x_i = w_i$. In other words, we need to add the feature with the *largest-magnitude negative weight*. Alternatively, if *feature removal* is allowed, we could *remove* the feature with the *largest positive weight*. Both approaches have the effect of pushing the sample towards the negative class and away from the positive class. This process is repeated until the sample crosses the decision boundary to the side of the target class, thus evading the classifier. As a reference, and to analyse the performance of EvadeGAN, the top *negatively-weighted* and *positively-weighted* features according to f are listed in **Table 10** and **Table 11** in the appendix. These features are the most *salient*[1] for the outputs of f, either towards the negative or the positive class.

[1] This is roughly the idea behind the Jacobian Saliency Map Attack (JSMA) [17], which was adapted in [45] to evade the DREBIN classifier.

Ideally, we would want EvadeGAN to learn to perturb samples in a way similar to that of the iterative approaches. However, there are two main differences. First, EvadeGAN works in a grey-box setting as opposed to the white-box access of the iterative approaches. Therefore, its perturbations are not expected to match those of a white-box approach, but they should be similar enough to achieve evasion efficiently. This will be influenced by how well the discriminator D approximates the target f. The better D becomes at approximating f, the more likely it will weigh features similarly. Second, while iterative approaches *optimise* an objective function to generate *optimal* perturbations for a given sample, EvadeGAN is trained to optimise an objective function for a large training set so that it can *generalise* to any given input. Such differences between the two approaches are expected to result in different outcomes.

5.2 Evaluating EvadeGAN: Overall Results

As mentioned earlier, experiments were run with a target evasion rate $R = 100\%$ and a maximum number of changes $K = 15$. Convergence occurs when R is achieved on the validation set V (3821 malware samples) within an upper bound of K changes. That is, the goal of EvadeGAN is to evade f on 100% of the V samples.

Table 7. The convergence of EvadeGAN in three sample runs.

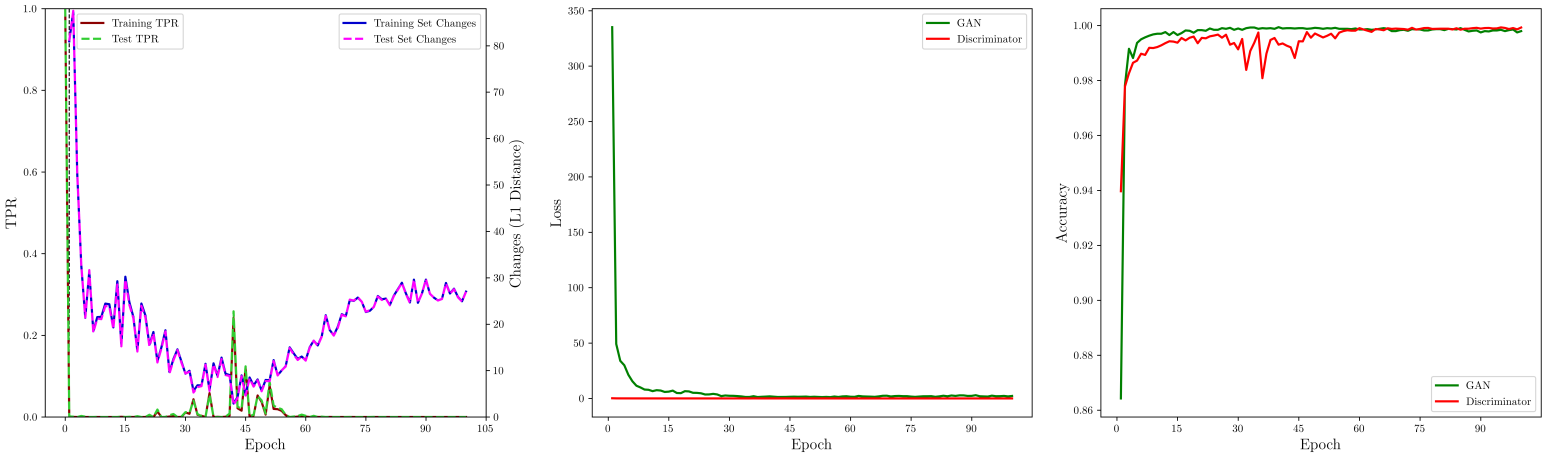
Mode	Evasion Rate (on the V set)	Number of Changes per Sample*					Epoch
		Avg ± SD	Q[25%, 50%, 75%]			Max	
EvadeGAN$_{XZ}$	**100%**	**9 ± 5.4**	8	9	10	91	28
	99%	**8 ± 6.5**	5	6	8	80	21
EvadeGAN$_{X}$	**100%**	**10 ± 4.6**	9	9	11	75	19
	99%	**8 ± 4.5**	6	8	9	62	10
EvadeGAN$_{Z}$	**100%**	**8 ± 1.6**	7	8	8	32	46
	99%	**6 ± 1.1**	**5**	**6**	**7**	23	43

In all three modes of EvadeGAN, **convergence was typically achieved in under 50 epochs**. The results of a sample run of each mode are shown in **Table 7**,[1] where the *highlighted* row for each mode indicates convergence with 100% evasion rate. Convergence is based on the *average* number of changes per sample in the whole validation set (highlighted **Avg** column). As reflected by the quantiles and the standard deviation, most samples were close to the average in each case. We should, however, expect that

[1] Sample runs of training each of the three EvadeGAN modes are included as `IPython` notebooks in the submitted code files.

some outliers would require more changes, as observed in the **Max** column. To alleviate the impact of such outliers, we can *lower* the target evasion rate R. As shown in the second row of each mode, an evasion rate of 99%, which is still very high, corresponds to a *smaller number of changes* overall (with the outliers ignored). This better reflects the capability of EvadeGAN to generate *minimal* perturbations. Another way of dealing with outliers is to penalise them to mitigate their impact on convergence.

In the following sections, we will first evaluate EvadeGAN modes that generate **sample-dependent perturbations**: **EvadeGAN$_{\mathbf{xz}}$** and **EvadeGAN$_{\mathbf{x}}$**. Since these two modes were very similar in their training and outcomes, we will focus on the more general **EvadeGAN$_{\mathbf{xz}}$** and highlight the differences where applicable. We will then evaluate **EvadeGAN$_{\mathbf{z}}$**, which generates **sample-independent (universal) perturbations.** In each case, we will briefly describe the training process and how learning progresses throughout training. We will also analyse the perturbations generated by the *trained* model to interpret what EvadeGAN has learnt in that mode.



(a) Minimising both TPR$_f$ & pertrubations **(b)** Training Loss **(c)** Accuracy

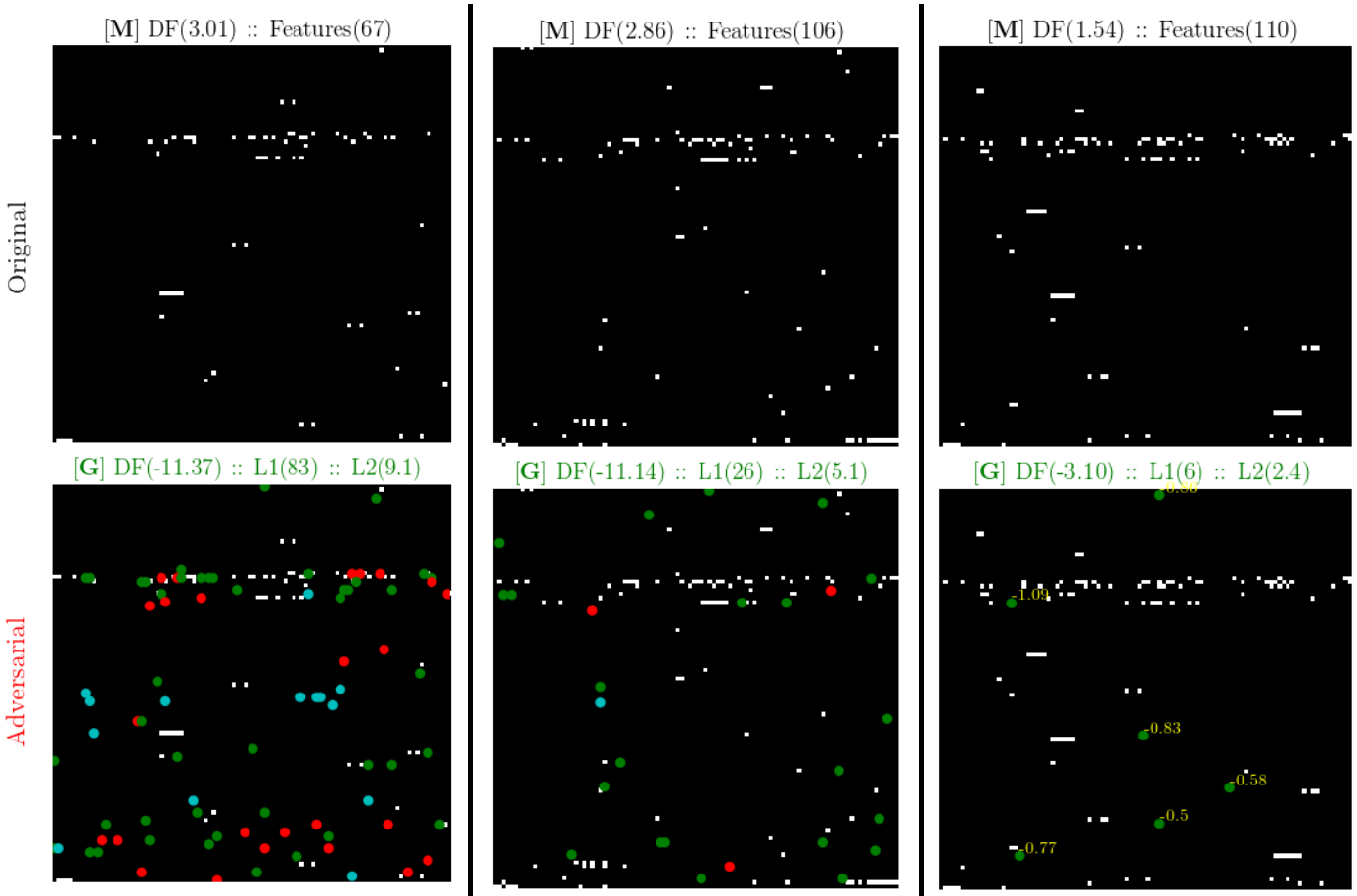
Figure 11. EvadeGAN$_{\mathbf{xz}}$ training

5.3 Sample-Dependent Perturbations: EvadeGANxz & EvadeGANx

In both **EvadeGAN$_{\mathbf{xz}}$** and **EvadeGAN$_{\mathbf{x}}$**, **perfect evasion** was always achieved **within 2-3 epochs** of training with *a reasonably small number of changes*. Training then keeps optimising the added features until **convergence** within the upper bound K, typically **within 20-50 epochs** depending on the training hyperparameters. To show the behaviour of **EvadeGAN$_{\mathbf{xz}}$** during training in terms of maximising evasion (i.e., minimising TPR_f) while minimising perturbations, **Figure 11.a** plots the two conflicting objectives throughout 100 epochs of training. We can infer that the training hyperparameters in this particular

run added more weight to the evasion objective. Convergence, in this run, was achieved after **38 epochs** with an evasion rate of **100%** and an average of **8 changes per sample**. The loss and accuracy of both D and G during training are plotted in **Figure 11.b** and **Figure 11.c** respectively.

To give a glimpse of the learning progress of **EvadeGAN$_{\mathbf{xz}}$** during training, we sample a few AEs that were generated early on in training and compare them with other AEs that were further optimised later. In **Figure 12**, we visualise three AEs that were generated at different training epochs, along with the original malware samples. Each sample is visualised as a 100×100 image (10K features), with white pixels indicating the features present in the original malware sample. Coloured pixels in the AE images represent the added features according to their weights.



(a) A sample AE after 1 epoch. (b) A sample AE after 2 epochs. (c) A sample AE at convergence.

Figure 12. Sample AEs generated by **EvadeGAN$_{\mathbf{xz}}$** during training. The original malware samples are shown in the top row, while their corresponding AEs are in the bottom row. Perturbations are coloured to indicate the sign of their weights: *green* for negative, *cyan* for zero, and *red* for positive. **[M]** and **[G]** are the labels assigned by f (malware and goodware, respectively). **DF:** Decision Function value by f. **L1:** ℓ_1 distance from the original sample = number of added features. **L2**: ℓ_2 distance.

A few observations can be made from this visualisation. First, from **(a)** we notice that **EvadeGAN$_{\mathbf{xz}}$** generated evasive AEs, albeit not optimal, early on in training with a reasonable number of changes (considering the 10K possibilities). Second, comparing **(a)** to **(b)** and then **(c)** shows how **EvadeGAN$_{\mathbf{xz}}$** is learning to optimise perturbations by focusing on *negatively-weighted features* (green) and gradually dropping *zero-weighted* (cyan) and *positively-weighted* (red) features from perturbations. Third, we notice from the weight annotations in **(c)** that the perturbations are minimised to include some of the *top negatively-weighted features*, as per **Table 10** in the appendix. **Figure 12** also shows the ℓ_1 and ℓ_2 norms of the distance between each AE and its corresponding malware sample, to demonstrate how far the AE moved in the feature space. Furthermore, the decision function values for both the malware and the AE are shown, which give an idea of how close the malware sample is from the decision boundary, and how far the AE has moved into the target class region.

After training, the generator of **EvadeGAN$_{\mathbf{xz}}$** can produce AEs instantly for any given input. **Figure 13** shows a few AEs with **sample-dependent perturbations** produced by the trained **EvadeGAN$_{\mathbf{xz}}$** generator. These AEs were produced from malware samples with different features and sizes, and we can see how the generated perturbations are unique to each sample.

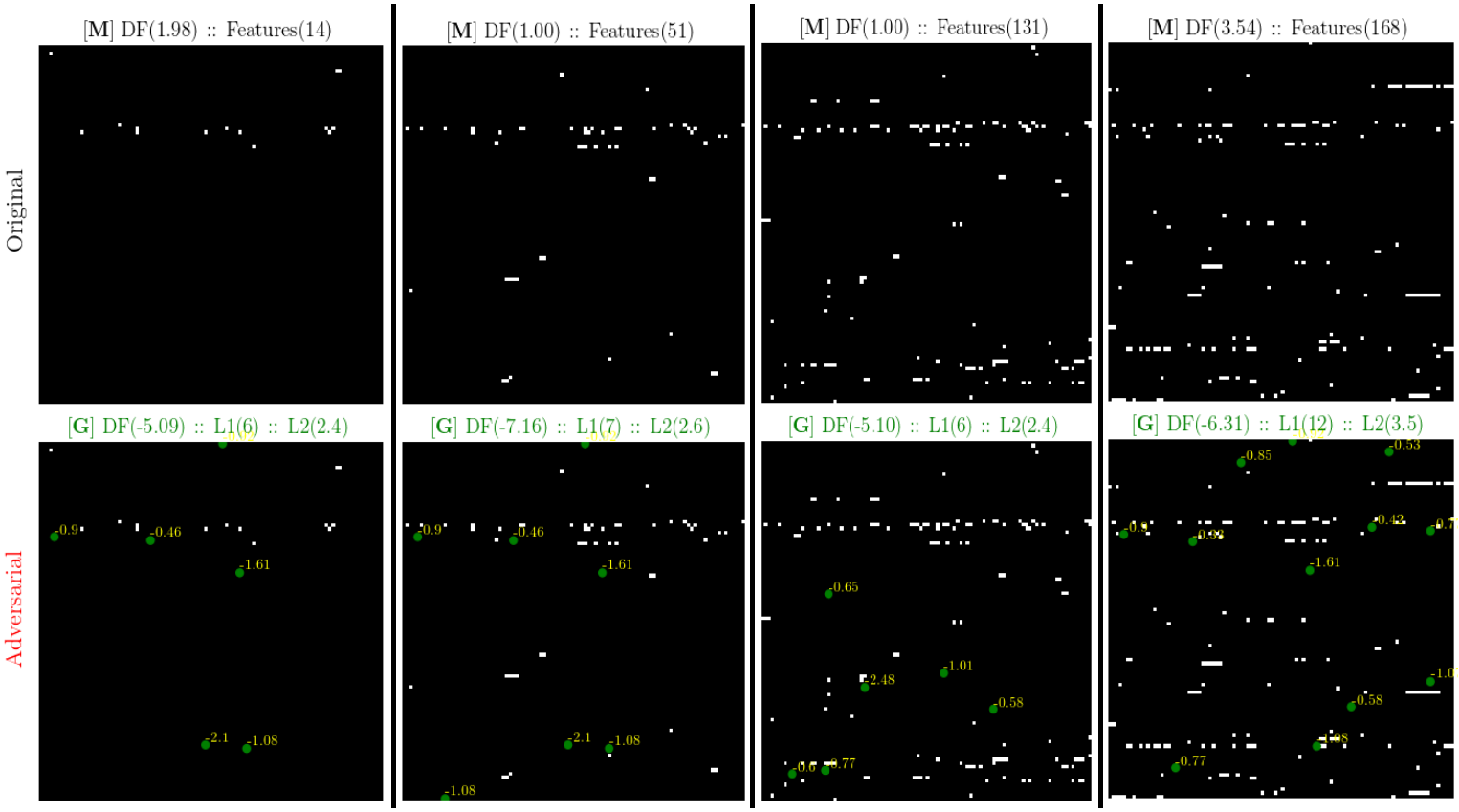


Figure 13. AEs with **sample-dependent perturbations** produced by the *trained* generator of **EvadeGAN$_{\mathbf{xz}}$**.

To get more insight into what perturbations **EvadeGAN$_{\mathbf{xz}}$** has learnt to generate, **Table 8** shows the top features that were used to achieve perfect evasion on the validation set (of 3821 samples). It also shows their weights according to f. While **EvadeGAN$_{\mathbf{xz}}$** has no knowledge of these weights, we can see that many of these features are among the *top negatively-weighted* features as per **Table 10** in the appendix.

Table 8. The top most frequently used features in **EvadeGAN$_{\mathbf{xz}}$** perturbations.

Rank	**Feature** x_i	Frequency*	Weight w_i by f
1	`urls::https://app_adjust_com`	3451	**-0.773**
2	`urls::http://test_urbanairship_com`	3450	**-0.582**
3	`api_permissions::android_permission_AUTHENTICATE_A…`	3442	**-0.648**
4	`urls::https://docs_google_com/spreadsheet/formResp…`	3426	**-0.601**
5	`urls::https://twitter_com/%s/status/%d…`	3222	**-1.045**
6	`intents::android_intent_action_PACKAGE_CHANGED…`	3038	**-0.774**
7	`urls::http://www_startappexchange_com/1_3/trackdow…`	2723	**-2.104**
8	`activities::com_ccx_xm_AdActivity`	2424	**-0.854**
9	`activities::_AndromoDashboardActivity…`	1796	**-0.919**
10	`urls::http://www_youtube_com`	1413	**-1.082**
11	`activities::com_payeco_android_plugin_PayecoPlugin…`	1204	**-0.237**
12	`interesting_calls::Cipher(RSA/None/PKCS1Padding)…`	1158	**-1.094**
13	`activities::com_qbiki_seattleclouds_SettingsActivi…`	909	**-0.874**
14	`urls::http://portre_yemonisoni_com/mcla…`	794	**-2.480**
15	`activities::_IntroActivity`	599	**-0.865**
16	`urls::10_0_2_2`	564	**-0.520**
17	`urls::https://device-api_urbanairship_com/…`	516	**-0.619**
18	`intents::android_intent_action_SEND_MULTIPLE…`	439	**-0.900**
19	`urls::http://tdcv3_talkingdata_net/g/d…`	433	**-0.924**
20	`urls::https://ssl_gstatic_com/accessibility/javasc…`	384	**-0.607**
21	`urls::http://maps_google_com/maps/api/geocode/json…`	282	**-0.826**
22	`urls::http://my_mobfox_com/request_php…`	276	**-1.009**
23	`urls::http://play_google_com/store/apps/details…`	252	**-1.074**
24	`urls::https://www_googleapis_com/oauth2/v1/certs…`	237	**-1.080**
25	`app_permissions::name='com_android_browser_permiss…`	230	**-0.418**
26	`urls::http://www_ngs_ac_uk/tools/jcepolicyfiles…`	212	**-0.501**
27	`activities::cn_jpush_android_ui_PushActivity…`	199	**-0.530**
28	`urls::http://www_google_com/loc/json…`	188	**-0.446**
29	`activities::android_support_v7_widget_TestActivity…`	149	**-0.725**
30	`intents::android_intent_action_ACTION_POWER_DISCON…`	146	**-0.423**

* Based on 3821 validation samples.

As pointed out earlier, what EvadeGAN can learn is influenced by how well the surrogate D approximates the target f. With our training setup, the trained D was able to fit f on 99–100% of the training and test samples. These samples were the ones correctly classified by f from the original training and test sets.

One final note on the difference between **EvadeGAN$_{\mathbf{xz}}$** and **EvadeGAN$_{\mathbf{x}}$**. While **EvadeGAN$_{\mathbf{x}}$** performed similarly to **EvadeGAN$_{\mathbf{xz}}$** in most aspects, we have hypothesised earlier that the **added noise vector** in **EvadeGAN$_{\mathbf{xz}}$** would allow generating diverse AEs for the same malware sample. This turned out to be the case. To test this, a malware sample was randomly selected and fed to the trained generator of **EvadeGAN$_{\mathbf{xz}}$** to produce 100 AEs using 100 different *noise vectors*. Testing on different malware samples resulted in **10-20 unique AEs** (out of the generated 100) for each malware sample. We assume that this diversity could be further increased by increasing the size of the noise vectors $\boldsymbol{z}$ in proportion to the size of the input samples $\boldsymbol{x}$. **EvadeGAN$_{\mathbf{x}}$**, on the other hand, produces the same output given the same input. **Figure 14** shows a few *unique* AEs that were generated by **EvadeGAN$_{\mathbf{xz}}$** for the *same malware sample* using *different noise vectors*.

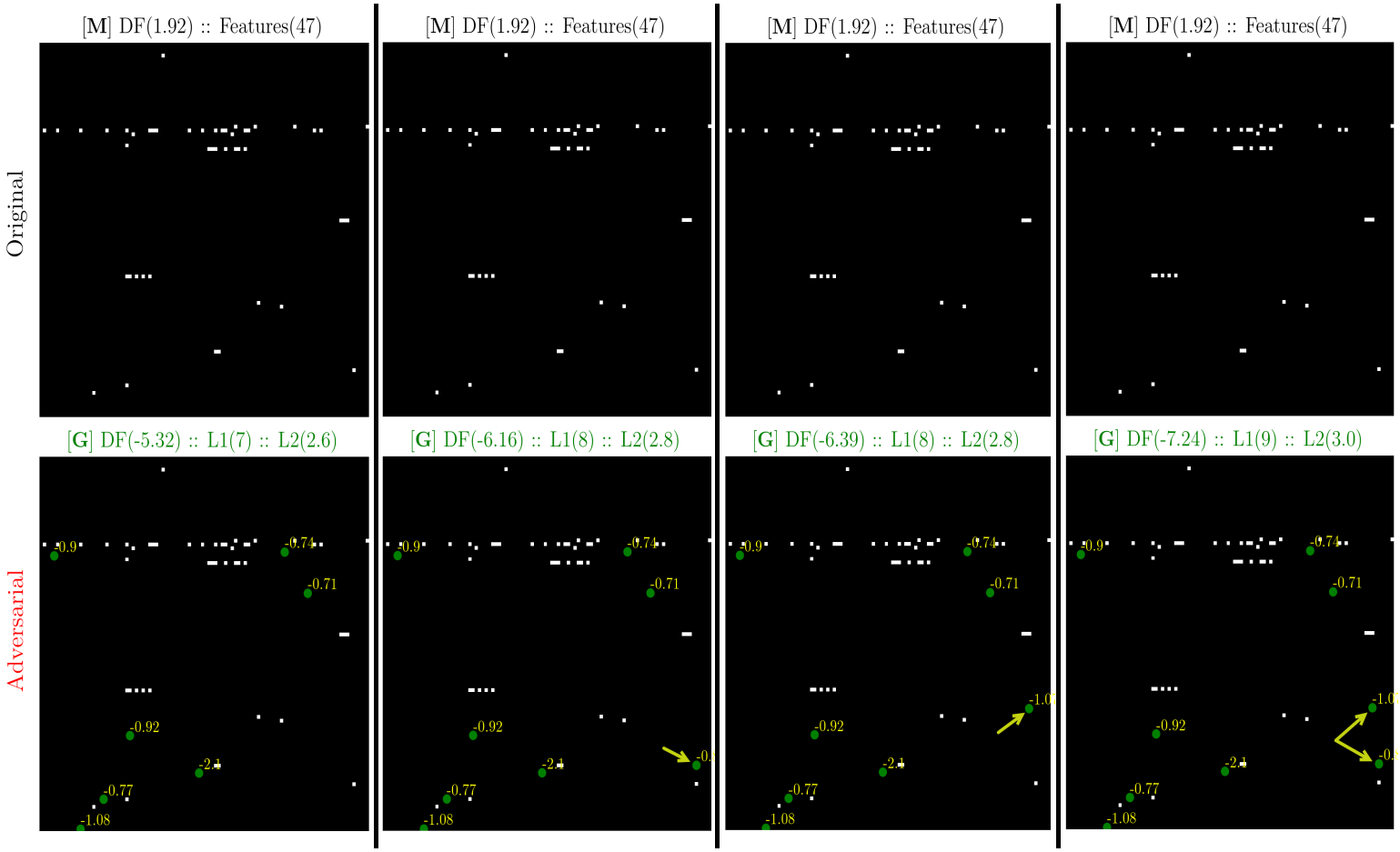
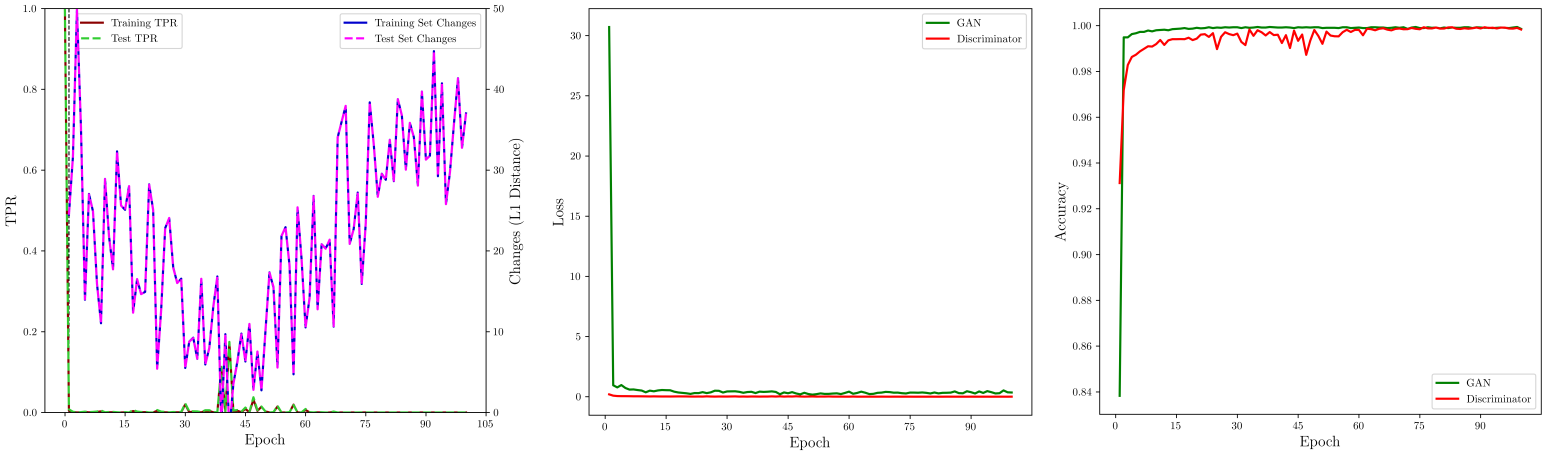


Figure 14. A few unique AEs produced by **EvadeGAN$_{\mathbf{xz}}$** for *the same malware sample* using different noise vectors. The differences between the AEs are reflected in the **DF**, **L1**, and **L2** values of each. Added features that are unique to each are also highlighted.

5.4 Sample-Independent (Universal) Perturbations: EvadeGANz

The generator of **EvadeGAN$_{\mathbf{Z}}$** only takes *noise* as input to produce **universal perturbations** that can be added to any malware sample to cause evasion. Training **EvadeGAN$_{\mathbf{Z}}$** typically took longer to converge with perfect evasion, compared to the other two modes. **Figure 15.a** plots the two objectives of **EvadeGAN$_{\mathbf{Z}}$** training (*maximising* evasion while *minimising* perturbations) throughout 100 epochs of a sample run. In this particular run, convergence was achieved after **50 epochs** with an evasion rate of **100%** and an average of **8 changes per sample**. The loss and accuracy of both D and G during training are plotted in **Figure 15.b** and **Figure 15.c** respectively.



(a) Minimising both TPR$_f$ & pertrubations **(b)** Training Loss **(c)** Accuracy

Figure 15. EvadeGAN$_{\mathbf{Z}}$ training

The trained **EvadeGAN$_{\mathbf{Z}}$** generator can produce many universal perturbations by changing the noise every time. Using 100 different noise vectors to produce 100 perturbation vectors resulted in 10-40% of them being unique. We assume that this percentage could be further increased by increasing the dimensionality of the input in proportion to that of the output. In our experiments, we used noise vectors of 100 dimensions to output perturbation vectors of 10,000 dimensions. **Figure 16** shows a few AEs with the same universal perturbations produced by the trained **EvadeGAN$_{\mathbf{Z}}$** generator for malware samples with different features and sizes.

As for the perturbations **EvadeGAN$_{\mathbf{Z}}$** has learnt to add to achieve evasion, **Table 9** lists the most frequently added features using various noise vectors as input. Also shown in the table are the weights of these features according to f. As shown in **EvadeGAN$_{\mathbf{XZ}}$**, most of these features are among the *top negatively-weighted* features as per **Table 10** in the appendix. One main difference is that **EvadeGAN$_{\mathbf{Z}}$** uses a smaller set of features in its perturbations compared to the other two modes, hence the high frequency counts in **Table 9**.

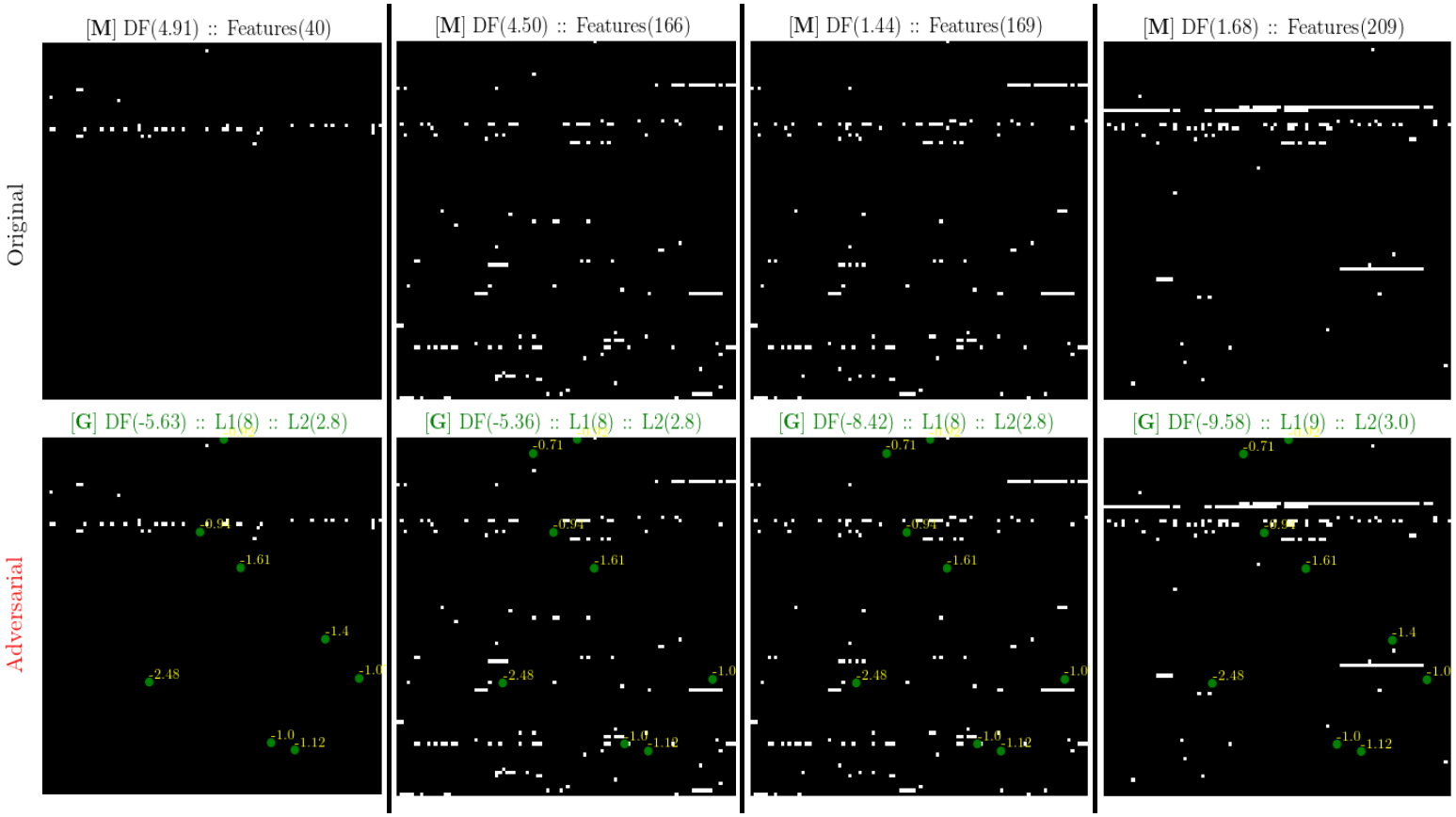


Figure 16. AEs with **universal perturbations** produced by the *trained* generator of **EvadeGAN$_{\mathbf{Z}}$**. Note that some of the added features *already exist* in some of the original samples so they are not highlighted in their corresponding AEs.

Table 9. The top most frequently used features in **EvadeGAN$_{\mathbf{Z}}$** perturbations.

Rank	**Feature** x_i	Frequency*	Weight w_i by f
1	`urls::http://portre_yemonisoni_com/mcla`	3445	-2.480
2	`activities::com_qbiki_seattleclouds_SettingsAc...`	3445	-0.874
3	`activities::com_ccx_xm_AdActivity`	3445	-0.854
4	`urls::http://my_mobfox_com/request_php`	3440	-1.009
5	`activities::android_support_v7_widget_TestActi...`	3435	-0.725
6	`intents::android_intent_action_PACKAGE_CHANGED`	3429	-0.774
7	`interesting_calls::Cipher(RSA/None/PKCS1Padding)`	3428	-1.094
8	`urls::http://api_w_inmobi_com/showad/v2`	3426	-1.045
9	`activities::_AndromoDashboardActivity`	3421	-0.919
10	`activities::_IntroActivity`	3412	-0.865
11	`urls::https://api_stripe_com`	3395	-0.551
12	`urls::https://app_adjust_com`	3362	-0.773
13	`urls::http://www_startappexchange_com/1_3/trac...`	3345	-2.104
14	`urls::http://play_google_com/store/apps/details`	3320	-1.074
15	`urls::http://www_ngs_ac_uk/tools/jcepolicyfiles`	3319	-0.501
16	`urls::http://www_youtube_com`	3304	-1.082
17	`urls::https://docs_google_com/spreadsheet/form...`	3042	-0.601
18	`urls::http://admin_appnext_com/`	2955	-1.614
19	`urls::http://ads_heyzap_com/in_game_api/ads`	2931	-0.712
20	`urls::https://ad_mail_ru/mobile/`	2891	-1.116

* Based on 3821 validation samples.

6. Conclusion and Future Work

In this project, we have proposed **EvadeGAN**, a new GAN-based framework for performing evasion attacks against malware classifiers. **EvadeGAN** is a novel framework that implements different approaches to generate adversarial examples (AEs) in **grey -box** settings. We have shown and compared the results of the different **EvadeGAN** modes which can generate either **sample-dependent** or **universal** perturbations. The results have shown that **EvadeGAN**, in all its modes, was able to generate **minimally perturbed AEs** that achieved a **perfect evasion rate** against a state-of-the-art malware classifier. We have also drawn a comparison between the *generative* approach of **EvadeGAN** and that of a typical *iterative white-box method*. **EvadeGAN** has two main advantages over such iterative approaches: **efficiency** and **generalisability**. Once trained, **EvadeGAN** can generate AEs instantly for any given input.

As for future work, we would like to investigate incorporating *problem-space* constraints to ensure that the generated AEs in the feature space correspond to valid malware applications in the problem space. We could build on the formalisation proposed by Pierazzi et al. [28] which was described in **Section 3.1**. Their implementation could be incorporated as an *oracle* that feeds **EvadeGAN** during training, after which **EvadeGAN** should be able to automatically generate AEs that satisfy problem-space constraints.

Furthermore, we would like to investigate the use of **EvadeGAN** as a defence framework for machine learning models. With its ability to instantly generate AEs for any given input, **EvadeGAN** could be used to *adversarially train* target models with datasets that are augmented with AEs. We could also further analyse what **EvadeGAN** learns about a target model and use such knowledge to regularise and harden the model. The benefits of these potential uses of **EvadeGAN** could be maximised by training in *white-box settings*. This would result in *better-optimised* outputs and would lead to *more accurate knowledge* of the target model. ■

7. References

[1] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of android malware in your pocket," in *The Network and Distributed System Security Symposium (NDSS)*, 2014, vol. 14, pp. 23-26.

[2] A. D. Joseph, B. Nelson, B. I. P. Rubinstein, and J. D. Tygar, *Adversarial machine learning*. Cambridge University Press, 2018.

[3] M. Barreno, B. Nelson, A. D. Joseph, and J. D. Tygar, "The security of machine learning," *Machine Learning,* vol. 81, no. 2, pp. 121-148, 2010.

[4] B. Biggio and F. Roli, "Wild patterns: Ten years after the rise of adversarial machine learning," *Pattern Recognition,* vol. 84, pp. 317-331, 2018.

[5] C. Chio and D. Freeman, *Machine Learning and Security: Protecting Systems with Data and Algorithms*. " O'Reilly Media, Inc.", 2018.

[6] N. Papernot, P. McDaniel, A. Sinha, and M. P. Wellman, "SoK: Security and privacy in machine learning," in *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2018: IEEE, pp. 399-414.

[7] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical black-box attacks against machine learning," in *Proceedings of the 2017 ACM on Asia conference on computer and communications security*, 2017, pp. 506-519.

[8] D. Hendrycks, K. Zhao, S. Basart, J. Steinhardt, and D. Song, "Natural adversarial examples," *arXiv preprint arXiv:1907.07174,* 2019.

[9] C. Laidlaw and S. Feizi, "Functional adversarial attacks," in *Advances in Neural Information Processing Systems*, 2019, pp. 10408-10418.

[10] Z. Zhao, D. Dua, and S. Singh, "Generating natural adversarial examples," *arXiv preprint arXiv:1710.11342,* 2017.

[11] I. Goodfellow, "Defense against the dark arts: An overview of adversarial example security research and future research directions," *arXiv preprint arXiv:1806.04169,* 2018.

[12] C. Szegedy *et al.*, "Intriguing properties of neural networks," *arXiv preprint arXiv:1312.6199,* 2013.

[13] N. Carlini and D. Wagner, "Towards Evaluating the Robustness of Neural Networks," in *2017 IEEE Symposium on Security and Privacy (SP)*, 22-26 May 2017 2017, pp. 39-57.

[14] B. Biggio *et al.*, "Evasion attacks against machine learning at test time," in *Joint European conference on machine learning and knowledge discovery in databases*, 2013: Springer, pp. 387-402.

[15] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," *arXiv preprint arXiv:1412.6572,* 2014.

[16] A. Kurakin, I. Goodfellow, and S. Bengio, "Adversarial examples in the physical world," *arXiv preprint arXiv:1607.02533,* 2016.

[17] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, "The Limitations of Deep Learning in Adversarial Settings," in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, 21-24 March 2016 2016, pp. 372-387.

[18] S.-M. Moosavi-Dezfooli, A. Fawzi, and P. Frossard, "Deepfool: a simple and accurate method to fool deep neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2574-2582.

[19] S.-M. Moosavi-Dezfooli, A. Fawzi, O. Fawzi, and P. Frossard, "Universal adversarial perturbations," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 1765-1773.

[20] P.-Y. Chen, H. Zhang, Y. Sharma, J. Yi, and C.-J. Hsieh, "Zoo: Zeroth order optimization based black-box attacks to deep neural networks without training substitute models," in *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, 2017, pp. 15-26.

[21] W. Brendel, J. Rauber, and M. Bethge, "Decision-based adversarial attacks: Reliable attacks against black-box machine learning models," *arXiv preprint arXiv:1712.04248,* 2017.

[22] J. Uesato, B. O'Donoghue, A. v. d. Oord, and P. Kohli, "Adversarial risk and the dangers of evaluating against weak attacks," *arXiv preprint arXiv:1802.05666,* 2018.

[23] J. Su, D. V. Vargas, and K. Sakurai, "One Pixel Attack for Fooling Deep Neural Networks," *IEEE Transactions on Evolutionary Computation,* vol. 23, no. 5, pp. 828-841, 2019.

[24] A. Ilyas, L. Engstrom, A. Athalye, and J. Lin, "Black-box Adversarial Attacks with Limited Queries and Information," presented at the Proceedings of the 35th International Conference on Machine Learning, Proceedings of Machine Learning Research, 2018. [Online]. Available: http://proceedings.mlr.press.

[25] W. Xu, Y. Qi, and D. Evans, "Automatically evading classifiers," in *Proceedings of the 2016 network and distributed systems symposium*, 2016, vol. 10.

[26] M. Alzantot, Y. Sharma, S. Chakraborty, H. Zhang, C.-J. Hsieh, and M. B. Srivastava, "Genattack: Practical black-box attacks with gradient-free optimization," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2019, pp. 1111-1119.

[27] R. Mosli, "Crafting Adversarial Examples using Particle Swarm Optimization," 2020.

[28] F. Pierazzi, F. Pendlebury, J. Cortellazzi, and L. Cavallaro, "Intriguing Properties of Adversarial ML Attacks in the Problem Space," *arXiv preprint arXiv:1911.02142,* 2019.

[29] A. C. Serban, E. Poll, and J. Visser, "Adversarial examples-a complete characterisation of the phenomenon," *arXiv preprint arXiv:1810.01185,* 2018.

[30] K. Grosse, P. Manoharan, N. Papernot, M. Backes, and P. McDaniel, "On the (statistical) detection of adversarial examples," *arXiv preprint arXiv:1702.06280,* 2017.

[31] X. Li and F. Li, "Adversarial examples detection in deep networks with convolutional filter statistics," in *Proceedings of the IEEE International Conference on Computer Vision*, 2017, pp. 5764-5772.

[32] R. Feinman, R. R. Curtin, S. Shintre, and A. B. Gardner, "Detecting adversarial samples from artifacts," *arXiv preprint arXiv:1703.00410,* 2017.

[33] W. Xu, D. Evans, and Y. Qi, "Feature squeezing: Detecting adversarial examples in deep neural networks," *arXiv preprint arXiv:1704.01155,* 2017.

[34] Z. Gong, W. Wang, and W.-S. Ku, "Adversarial and clean data are not twins," *arXiv preprint arXiv:1704.04960,* 2017.

[35] D. Meng and H. Chen, "Magnet: a two-pronged defense against adversarial examples," in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 135-147.

[36] C. Xie, J. Wang, Z. Zhang, Z. Ren, and A. Yuille, "Mitigating adversarial effects through randomization," *arXiv preprint arXiv:1711.01991,* 2017.

[37] C. Guo, M. Rana, M. Cisse, and L. Van Der Maaten, "Countering adversarial images using input transformations," *arXiv preprint arXiv:1711.00117,* 2017.

[38] J. Buckman, A. Roy, C. Raffel, and I. Goodfellow, "Thermometer encoding: One hot way to resist adversarial examples," in *International Conference on Learning Representations*, 2018.

[39] P. Samangouei, M. Kabkab, and R. Chellappa, "Defense-GAN: Protecting Classifiers Against Adversarial Attacks Using Generative Models," in *International Conference on Learning Representations*, 2018.

[40] N. Papernot, P. D. McDaniel, X. Wu, S. Jha, and A. Swami, "Distillation as a defense to adversarial perturbations against deep neural networks, 2016," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2015.

[41] A. Athalye, N. Carlini, and D. Wagner, "Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples," *arXiv preprint arXiv:1802.00420,* 2018.

[42] C. J. Burges, "A tutorial on support vector machines for pattern recognition," *Data mining and knowledge discovery,* vol. 2, no. 2, pp. 121-167, 1998.

[43] J. Platt, "Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods," *Advances in large margin classifiers,* vol. 10, no. 3, pp. 61-74, 1999.

[44] C.-W. Hsu and C.-J. Lin, "A comparison of methods for multiclass support vector machines," *IEEE transactions on Neural Networks,* vol. 13, no. 2, pp. 415-425, 2002.

[45] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel, "Adversarial examples for malware detection," in *European Symposium on Research in Computer Security*, 2017: Springer, pp. 62-79.

[46] I. Goodfellow *et al.*, "Generative adversarial nets," in *Advances in neural information processing systems*, 2014, pp. 2672-2680.

[47] M. Arjovsky and L. Bottou, "Towards principled methods for training generative adversarial networks," *arXiv preprint arXiv:1701.04862,* 2017.

[48] M. Arjovsky, S. Chintala, and L. Bottou, "Wasserstein GAN," *arXiv preprint arXiv:1701.07875,* 2017.

[49] I. Goodfellow, "NIPS 2016 tutorial: Generative adversarial networks," *arXiv preprint arXiv:1701.00160,* 2016.

[50] K. Roth, A. Lucchi, S. Nowozin, and T. Hofmann, "Stabilizing training of generative adversarial networks through regularization," in *Advances in neural information processing systems*, 2017, pp. 2018-2028.

[51] T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen, "Improved techniques for training gans," in *Advances in neural information processing systems*, 2016, pp. 2234-2242.

[52] L. Metz, B. Poole, D. Pfau, and J. Sohl-Dickstein, "Unrolled generative adversarial networks," *arXiv preprint arXiv:1611.02163,* 2016.

[53] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," *arXiv preprint arXiv:1511.06434,* 2015.

[54] M. Mirza and S. Osindero, "Conditional generative adversarial nets," *arXiv preprint arXiv:1411.1784,* 2014.

[55] A. Odena, C. Olah, and J. Shlens, "Conditional Image Synthesis With Auxiliary Classifier GANs.," *arXiv preprint arXiv:1610.09585,* vol. 10, 2016.

[56] X. Mao, Q. Li, H. Xie, R. Lau, and Z. Wang, "Least Squares Generative Adversarial Networks. cite," *arXiv preprint arxiv:1611.04076,* vol. 4, 2016.

[57] J.-Y. Zhu, T. Park, P. Isola, and A. A. Efros, "Unpaired image-to-image translation using cycle-consistent adversarial networks," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 2223-2232.

[58] T. Karras, T. Aila, S. Laine, and J. Lehtinen, "Progressive growing of gans for improved quality, stability, and variation," *arXiv preprint arXiv:1710.10196,* 2017.

[59] T. Karras, S. Laine, and T. Aila, "A Style-Based Generator Architecture for Generative Adversarial Networks, 2019 IEEE," in *CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018, pp. 4396-4405.

[60] N. Šrndić and P. Laskov, "Practical evasion of a learning-based classifier: A case study," in *2014 IEEE symposium on security and privacy*, 2014: IEEE, pp. 197-211.

[61] A. Demontis *et al.*, "Yes, Machine Learning Can Be More Secure! A Case Study on Android Malware Detection," *IEEE Transactions on Dependable and Secure Computing,* vol. 16, no. 4, pp. 711-724, 2019.

[62] A. Demontis *et al.*, "Why do adversarial attacks transfer? explaining transferability of evasion and poisoning attacks," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 321-338.

[63] Y. Song, R. Shu, N. Kushman, and S. Ermon, "Constructing unrestricted adversarial examples with generative models," presented at the Proceedings of the 32nd International Conference on Neural Information Processing Systems, Montréal, Canada, 2018.

[64] S. Baluja and I. Fischer, "Adversarial transformation networks: Learning to generate adversarial examples," *arXiv preprint arXiv:1703.09387,* 2017.

[65] O. Poursaeed, I. Katsman, B. Gao, and S. Belongie, "Generative adversarial perturbations," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 4422-4431.

[66] J. Hayes and G. Danezis, "Learning universal adversarial perturbations with generative models," in *2018 IEEE Security and Privacy Workshops (SPW)*, 2018: IEEE, pp. 43-49.

[67] C. Xiao, B. Li, J.-Y. Zhu, W. He, M. Liu, and D. Song, "Generating adversarial examples with adversarial networks," *arXiv preprint arXiv:1801.02610,* 2018.

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

[69] Z. Lin, Y. Shi, and Z. Xue, "IDSGAN: Generative adversarial networks for attack generation against intrusion detection," *arXiv preprint arXiv:1809.02077,* 2018.

[70] H. S. Anderson, J. Woodbridge, and B. Filar, "DeepDGA: Adversarially-tuned domain generation and detection," in *Proceedings of the 2016 ACM Workshop on Artificial Intelligence and Security*, 2016, pp. 13-21.

[71] A. AlEroud and G. Karabatis, "Bypassing Detection of URL-based Phishing Attacks Using Generative Adversarial Deep Neural Networks," in *Proceedings of the Sixth International Workshop on Security and Privacy Analytics*, 2020, pp. 53-60.

[72] "Scikit-learn's LinearSVC Class." https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html (accessed August 1, 2020).

[73] "LIBLINEAR: A Library for Large Linear Classification." https://www.csie.ntu.edu.tw/~cjlin/liblinear/ (accessed August 1, 2020).

[74] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro, "{TESSERACT}: Eliminating experimental bias in malware classification across space and time," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 729-746.

[75] T. Hastie, R. Tibshirani, and J. Friedman, *The elements of statistical learning: data mining, inference, and prediction*. Springer Science & Business Media, 2009.

[76] T. Dozat, "Incorporating Nesterov Momentum into Adam," 2016.

8. Appendix

Table 10. The **top 100 (1%) negatively weighted** features according to the target classifier f.

Rank	**Feature** x_i	Weight w_i	$Freq(x_i\|\boldsymbol{M})$	$P(x_i\|\boldsymbol{M})$	$Freq(x_i\|\boldsymbol{G})$	$P(x_i\|\boldsymbol{G})$
1	urls::http://portre_yemonisoni_com/…	**-2.4799**	3	0.0004	377	0.0046
2	urls::http://www_startappexchange_c…	**-2.1041**	229	0.0273	147	0.0018
3	activities::com_revmob_FullscreenAc…	**-1.7739**	28	0.0033	25	0.0003
4	urls::http://pms_mb_qq_com/rsp204	**-1.6652**	10	0.0012	12	0.0002
5	urls::http://admin_appnext_com/	**-1.6138**	48	0.0057	57	0.0007
6	urls::http://www_google-analytics_c…	**-1.5723**	1316	0.1570	25337	0.3114
7	urls::http://ads_mobilecore_com/?pa…	**-1.4495**	154	0.0184	872	0.0107
8	interesting_calls::Cipher(Lorg/chro…	**-1.4106**	2	0.0002	329	0.0040
9	urls::http://i_ytimg_com/vi/	**-1.4011**	442	0.0527	1026	0.0126
10	urls::https://sites_google_com/site…	**-1.3277**	40	0.0048	3	0.0000
11	activities::_ads_vast_VideoAdsLoade…	**-1.2956**	71	0.0085	18	0.0002
12	urls::http://opds-spec_org/2010/cat…	**-1.2848**	28	0.0033	40	0.0005
13	urls::http://www_youtube_com/embed/	**-1.2557**	447	0.0533	1128	0.0139
14	urls::http://mobads_baidu_com/cpro/…	**-1.2240**	3	0.0004	2	0.0000
15	urls::http://loc_map_baidu_com/tcu_…	**-1.2199**	129	0.0154	366	0.0045
16	urls::http://www_startappexchange_c…	**-1.2168**	327	0.0390	220	0.0027
17	api_permissions::android_permission…	**-1.1957**	4397	0.5244	15540	0.1910
18	urls::http://iframe_ip138_com/ic_as…	**-1.1513**	13	0.0016	28	0.0003
19	activities::_MAME4all	**-1.1318**	110	0.0131	5	0.0001
20	urls::https://ad_mail_ru/mobile/	**-1.1157**	185	0.0221	74	0.0009
21	activities::_MonacaSScreenActivity	**-1.1028**	8	0.0010	1	0.0000
22	urls::http://data_altbeacon_org/and…	**-1.0951**	94	0.0112	371	0.0046
23	interesting_calls::Cipher(RSA/None/…	**-1.0940**	68	0.0081	168	0.0021
24	urls::http://strategy_beacon_qq_com…	**-1.0917**	7	0.0008	7	0.0001
25	activities::org_jufh_xmvo_woeiwej_S…	**-1.0903**	5	0.0006	2	0.0000
26	urls::http://www_youtube_com	**-1.0824**	6	0.0007	929	0.0114
27	urls::https://www_googleapis_com/oa…	**-1.0797**	5	0.0006	382	0.0047
28	urls::http://play_google_com/store/…	**-1.0740**	97	0.0116	3447	0.0424
29	urls::http://api_w_inmobi_com/showa…	**-1.0452**	30	0.0036	69	0.0009
30	urls::https://twitter_com/%s/status…	**-1.0450**	0	0.0000	386	0.0047
31	app_permissions::name='getui_permis…	**-1.0364**	25	0.0030	7	0.0001
32	urls::http://openmobile_qq_com/oaut…	**-1.0314**	167	0.0199	411	0.0051
33	urls::http://debugtbs_qq_com/	**-1.0246**	8	0.0010	8	0.0001
34	urls::https://api_weibo_com/2/statu…	**-1.0241**	76	0.0091	292	0.0036
35	urls::https://api_weibo_com/2/statu…	**-1.0241**	76	0.0091	295	0.0036
36	urls::https://api_weibo_com/2/statu…	**-1.0149**	73	0.0087	270	0.0033
37	urls::http://my_mobfox_com/request_…	**-1.0087**	31	0.0037	925	0.0114
38	urls::https://api_parse_com/1/	**-1.0082**	8	0.0010	352	0.0043
39	activities::_ISniperActivity	**-1.0000**	3	0.0004	0	0.0000
40	urls::http://kairopark_jp/j7hjT4ua/…	**-1.0000**	2	0.0002	1	0.0000
41	urls::https://graph_facebook_com/%1…	**-1.0000**	267	0.0318	0	0.0000
42	activities::_krazyracers	**-1.0000**	2	0.0002	0	0.0000
43	urls::http://www_youtube_com/watch	**-1.0000**	0	0.0000	488	0.0060
44	urls::https://apigee_com/oauth_call…	**-1.0000**	1	0.0001	1	0.0000
45	urls::https://www_growingio_com/oau…	**-1.0000**	4	0.0005	1	0.0000
46	activities::com_forthblue_aq_Main	**-1.0000**	1	0.0001	0	0.0000
47	urls::http://m_iqiyi_com	**-1.0000**	3	0.0004	2	0.0000
48	activities::_activity_WelActivity	**-1.0000**	2	0.0002	0	0.0000

49	activities::com_fw_toth_activity_Fw…	**-1.0000**	6	0.0007	0	0.0000
50	app_permissions::name='android_perm…	**-1.0000**	1	0.0001	305	0.0038
51	urls::http://cq01-cm-et01_cq01_baid…	**-0.9953**	4	0.0005	5	0.0001
52	interesting_calls::Cipher(PBEWithSH…	**-0.9915**	74	0.0088	25	0.0003
53	urls::https://www_googleapis_com/au…	**-0.9902**	1718	0.2049	30261	0.3719
54	urls::https://openmobile_qq_com/	**-0.9796**	341	0.0407	957	0.0118
55	activities::com_unionpay_upomp_tbow…	**-0.9717**	13	0.0016	5	0.0001
56	api_permissions::android_permission…	**-0.9701**	174	0.0208	710	0.0087
57	activities::_SoftMainActivity	**-0.9604**	19	0.0023	1	0.0000
58	urls::https://www_mopub_com/optout	**-0.9480**	260	0.0310	460	0.0057
59	intents::cn_jpush_android_intent_NO…	**-0.9419**	165	0.0197	416	0.0051
60	urls::http://loc_map_baidu_com/fenc…	**-0.9391**	94	0.0112	407	0.0050
61	urls::http://mobads-logs_baidu_com/…	**-0.9390**	9	0.0011	3	0.0000
62	interesting_calls::Cipher(out)	**-0.9327**	99	0.0118	115	0.0014
63	urls::http://api_weibo_com/oauth2/d…	**-0.9314**	5	0.0006	3	0.0000
64	urls::http://v_youku_com/	**-0.9264**	5	0.0006	0	0.0000
65	urls::http://tdcv3_talkingdata_net/…	**-0.9240**	50	0.0060	88	0.0011
66	activities::_AndromoDashboardActivi…	**-0.9186**	0	0.0000	544	0.0067
67	urls::http://140_206_74_66:9322/ums…	**-0.9178**	5	0.0006	3	0.0000
68	urls::https://116_228_21_162:8602/u…	**-0.9178**	5	0.0006	3	0.0000
69	urls::http://ex_mobmore_com/api/wap…	**-0.9061**	75	0.0089	148	0.0018
70	intents::android_intent_action_SEND…	**-0.9003**	15	0.0018	623	0.0077
71	urls::https://8b2824c2cb184ce0ac78b…	**-0.8981**	160	0.0191	66	0.0008
72	activities::com_apps_pop_activity_P…	**-0.8948**	8	0.0010	3	0.0000
73	activities::com_ishow4s_web_activit…	**-0.8847**	9	0.0011	4	0.0001
74	intents::android_intent_category_SA…	**-0.8815**	38	0.0045	28	0.0003
75	urls::http://loc_map_baidu_com/wloc	**-0.8807**	67	0.0080	136	0.0017
76	urls::http://loc_map_baidu_com/iofd…	**-0.8807**	67	0.0080	131	0.0016
77	activities::com_qbiki_seattleclouds…	**-0.8738**	11	0.0013	417	0.0051
78	urls::http://maps_google_com/maps/a…	**-0.8682**	0	0.0000	317	0.0039
79	activities::_IntroActivity	**-0.8646**	4	0.0005	378	0.0047
80	activities::com_ccx_xm_AdActivity	**-0.8543**	13	0.0016	65	0.0008
81	urls::http://sns_whalecloud_com/sin…	**-0.8504**	50	0.0060	176	0.0022
82	urls::http://localhost:3000	**-0.8491**	46	0.0055	1496	0.0184
83	app_permissions::name='com_fede_lau…	**-0.8443**	21	0.0025	74	0.0009
84	urls::http://www_jivesoftware_com/x…	**-0.8429**	93	0.0111	561	0.0069
85	activities::_UserInfoActivity	**-0.8408**	20	0.0024	11	0.0001
86	urls::http://pingmid_qq_com:80/	**-0.8350**	34	0.0041	135	0.0017
87	urls::http://books_google_	**-0.8344**	54	0.0064	1553	0.0191
88	urls::https://plus_google_com/share…	**-0.8297**	149	0.0178	978	0.0120
89	urls::http://maps_google_com/maps/a…	**-0.8261**	0	0.0000	274	0.0034
90	urls::http://qzonestyle_gtimg_cn/qz…	**-0.8246**	15	0.0018	9	0.0001
91	intents::android_intent_action_SCRE…	**-0.8205**	42	0.0050	189	0.0023
92	urls::https://e_crashlytics_com/spi…	**-0.8174**	36	0.0043	4675	0.0575
93	urls::https://api_weibo_com/2/frien…	**-0.8147**	69	0.0082	238	0.0029
94	urls::http://adslist_wandoujia_com/…	**-0.7989**	4	0.0005	1	0.0000
95	intents::cn_jpush_android_intent_NO…	**-0.7957**	183	0.0218	462	0.0057
96	urls::http://android_heyzap_com/mob…	**-0.7884**	55	0.0066	89	0.0011
97	activities::kr_rain_WallPaperBeauti…	**-0.7768**	5	0.0006	1	0.0000
98	activities::_BatteryWidgetFrame	**-0.7768**	5	0.0006	1	0.0000
99	urls::https://api_weibo_com/2/frien…	**-0.7763**	192	0.0229	739	0.0091
100	intents::android_intent_action_PACK…	**-0.7744**	44	0.0053	223	0.0027

* The frequencies $Freq(\cdot)$ and probabilities $P(\cdot)$ are based on the *training* dataset. $\boldsymbol{M}$=Malware, $\boldsymbol{G}$=Goodware**.**

Table 11. The **top 100 (1%) positively weighted** features according to the target classifier f.

Rank	**Feature** x_i	Weight w_i	$Freq(x_i\|\boldsymbol{M})$	$P(x_i\|\boldsymbol{M})$	$Freq(x_i\|\boldsymbol{G})$	$P(x_i\|\boldsymbol{G})$
1	activities::vn_adflex_ads_AdsActivi…	**3.3362**	11	0.0013	0	0.0000
2	activities::com_phonegap_plugins_vi…	**3.2230**	3	0.0004	0	0.0000
3	urls::https://collector_mobile_cnzz…	**3.1074**	33	0.0039	0	0.0000
4	app_permissions::name='com_xiaomi_s…	**2.8688**	33	0.0039	1	0.0000
5	app_permissions::name='com_legion2a…	**2.7963**	3	0.0004	0	0.0000
6	urls::http://ads_yemonisoni_com/?pa…	**2.7284**	157	0.0187	377	0.0046
7	urls::https://android_bcfads_com	**2.6774**	14	0.0017	0	0.0000
8	activities::_Act_Home	**2.4482**	2	0.0002	0	0.0000
9	activities::com_apperhand_device_an…	**2.4293**	7	0.0008	13	0.0002
10	urls::http://api_jiagu_360_cn/s_htm…	**2.3559**	21	0.0025	0	0.0000
11	urls::http://push_apaddown_com/reg_…	**2.3161**	9	0.0011	4	0.0001
12	activities::_SFGameSplashActivity	**2.3048**	5	0.0006	0	0.0000
13	urls::http://play_google_com/store/…	**2.3040**	2	0.0002	0	0.0000
14	urls::http://&domain/function/ping_…	**2.2556**	3	0.0004	0	0.0000
15	urls::https://android_revmob_com	**2.1655**	937	0.1118	47	0.0006
16	urls::http://android_marsorstudio_c…	**2.1491**	9	0.0011	0	0.0000
17	urls::http://android_marsorstudio_c…	**2.1491**	9	0.0011	0	0.0000
18	urls::http://m_qy_u-kor_cn/	**2.1284**	11	0.0013	0	0.0000
19	activities::com_feiwofour_coverscre…	**2.1078**	64	0.0076	0	0.0000
20	urls::http://www_xiami_com/web/spar…	**2.0460**	3	0.0004	0	0.0000
21	activities::com_qihoo_util_appupdat…	**2.0412**	188	0.0224	0	0.0000
22	activities::com_adwo_adsdk_AdwoAdBr…	**2.0327**	7	0.0008	0	0.0000
23	urls::http://imgcache_qq_com/bosswe…	**2.0135**	3	0.0004	1	0.0000
24	activities::_HeyzapPublisherActivit…	**2.0001**	4	0.0005	0	0.0000
25	activities::_KumoAppActivity	**2.0000**	4	0.0005	0	0.0000
26	urls::http://payment_umpay_com/hfwe…	**2.0000**	4	0.0005	0	0.0000
27	urls::http://www_baidu_com/search/s…	**2.0000**	2	0.0002	0	0.0000
28	urls::http://www_startappexchange_c…	**2.0000**	3	0.0004	0	0.0000
29	urls::http://www_startappexchange_c…	**2.0000**	3	0.0004	0	0.0000
30	urls::http://appapi_neoline_biz/app…	**2.0000**	22	0.0026	0	0.0000
31	urls::http://relation_gamebox_360_c…	**2.0000**	2	0.0002	0	0.0000
32	urls::http://www_searchmobileonline…	**2.0000**	5	0.0006	0	0.0000
33	activities::com_feiwoone_coverscree…	**1.9830**	83	0.0099	0	0.0000
34	urls::http://appnext_com/ver/images…	**1.9635**	37	0.0044	33	0.0004
35	urls::http://ip_taobao_com/service/…	**1.9596**	8	0.0010	4	0.0001
36	urls::https://iddata-commons_org/IP…	**1.9520**	11	0.0013	1	0.0000
37	activities::MonacaPageActivity	**1.9520**	11	0.0013	1	0.0000
38	urls::http://www_javaeye_com/custom	**1.9292**	6	0.0007	2	0.0000
39	activities::com_bangcle_everisk_stu…	**1.9085**	45	0.0054	0	0.0000
40	urls::http://image_cauly_co_kr:1515…	**1.9070**	29	0.0035	0	0.0000
41	activities::net_youmi_android_AdBro…	**1.9046**	32	0.0038	0	0.0000
42	urls::http://iphone_com2us_com/app/…	**1.8975**	3	0.0004	0	0.0000
43	urls::http://www_amazon_com/gp/mas/…	**1.8943**	3	0.0004	0	0.0000
44	urls::http://app	**1.8885**	17	0.0020	3	0.0000
45	activities::com_ryg_dynamicload_DLP…	**1.8633**	5	0.0006	0	0.0000
46	activities::com_feiwo_appwall_WA	**1.8582**	52	0.0062	0	0.0000
47	urls::https://userinfo_revmob_com/a…	**1.8451**	20	0.0024	12	0.0002
48	urls::http://sdk_open_phone_igexin_…	**1.8257**	127	0.0152	15	0.0002
49	activities::org_nexage_sourcekit_va…	**1.8162**	351	0.0419	42	0.0005
50	urls::https://bugsense_appspot_com/…	**1.7812**	70	0.0084	0	0.0000
51	urls::http://ditu_google_cn/maps/ge…	**1.7704**	13	0.0016	3	0.0000

52	activities::rdr_y_pdb_k_m_aio	**1.7565**	5	0.0006	0	0.0000
53	urls::http://beekn_net/wp-content/u…	**1.7530**	51	0.0061	1	0.0000
54	urls::http://weixin_qq_com	**1.7219**	13	0.0016	7	0.0001
55	urls::http://image_baidu_com/i?ct=2…	**1.7060**	3	0.0004	0	0.0000
56	activities::com_cc_activity_CcActiv…	**1.6892**	4	0.0005	0	0.0000
57	urls::http://api_pingstart_com/plat…	**1.6885**	3	0.0004	0	0.0000
58	urls::http://cdn_monsterdevs_com/in…	**1.6832**	3	0.0004	0	0.0000
59	activities::_BrowserActivity	**1.6562**	73	0.0087	49	0.0006
60	activities::com_fivefeiwo_coverscre…	**1.6524**	573	0.0683	0	0.0000
61	activities::com_fendou_activity_Fen…	**1.6495**	11	0.0013	0	0.0000
62	activities::pygo_ypy_demx_vvguj	**1.6384**	3	0.0004	0	0.0000
63	urls::http://sns_vserv_mobi/sns/get…	**1.6337**	79	0.0094	0	0.0000
64	urls::https://github_com/litesuits/…	**1.6273**	4	0.0005	0	0.0000
65	activities::jvxn_ara_hml_ldn_ahc	**1.5989**	4	0.0005	0	0.0000
66	urls::https://play_google_com/store…	**1.5951**	2	0.0002	0	0.0000
67	activities::org_sshfo_xvhoiw_a	**1.5949**	7	0.0008	0	0.0000
68	activities::_ForgetPasswordActivity	**1.5886**	1	0.0001	3	0.0000
69	urls::http://revmob_com_	**1.5676**	952	0.1135	64	0.0008
70	urls::https://api_kaixin001_com	**1.5451**	23	0.0027	49	0.0006
71	urls::http://admin_appnext_com/Admi…	**1.5448**	90	0.0107	239	0.0029
72	activities::com_mshkf_jkkjkl_hkka	**1.5436**	4	0.0005	0	0.0000
73	urls::http://d1byvlfiet2h9q_cloudfr…	**1.5339**	662	0.0790	1437	0.0177
74	urls::http://b_tile_openstreetmap_o…	**1.5228**	0	0.0000	204	0.0025
75	urls::http://cloud_hexage_net	**1.5101**	8	0.0010	1	0.0000
76	urls::http://image_cauly_co_kr:1515…	**1.5077**	7	0.0008	0	0.0000
77	activities::co_lvdou_ldu_ALD	**1.4957**	3	0.0004	0	0.0000
78	urls::http://common_wacai_com/updat…	**1.4873**	3	0.0004	0	0.0000
79	urls::http://apk_hiapk_com/html/201…	**1.4873**	2	0.0002	0	0.0000
80	urls::http://www_gyswad_com:90/push…	**1.4820**	15	0.0018	0	0.0000
81	activities::com_pbjnzxeozsyglqbzb_A…	**1.4806**	3	0.0004	0	0.0000
82	urls::https://ssl_google-analytics_…	**1.4675**	1320	0.1574	25296	0.3109
83	urls::http://files_doudouad_com/upd…	**1.4665**	3	0.0004	0	0.0000
84	urls::http://ads	**1.4580**	8	0.0010	0	0.0000
85	urls::https://play_google_com/store…	**1.4532**	543	0.0648	180	0.0022
86	app_permissions::name='android_webk…	**1.4452**	59	0.0070	15	0.0002
87	interesting_calls::Cipher(PBEWITHMD…	**1.4393**	47	0.0056	9	0.0001
88	urls::https://www_growingio_com/mob…	**1.4388**	5	0.0006	1	0.0000
89	urls::https://www_growingio_com/mob…	**1.4388**	5	0.0006	1	0.0000
90	urls::https://www_growingio_com/mob…	**1.4388**	5	0.0006	1	0.0000
91	urls::http://schemas_android_com/ap…	**1.4324**	433	0.0516	429	0.0053
92	urls::http://os_scmpacdn_com/files/…	**1.4323**	279	0.0333	759	0.0093
93	interesting_calls::Cipher(anadirAmp…	**1.4109**	29	0.0035	26	0.0003
94	urls::http://www_nagapt_com_	**1.4108**	3	0.0004	0	0.0000
95	urls::http://www_startappexchange_c…	**1.4098**	5	0.0006	0	0.0000
96	urls::http://www_startappexchange_c…	**1.4098**	5	0.0006	0	0.0000
97	urls::http://www_startappexchange_c…	**1.4098**	5	0.0006	0	0.0000
98	urls::http://diguo-test_s3_amazonaw…	**1.4092**	2	0.0002	0	0.0000
99	api_calls::java/lang/Runtime;->exec	**1.4041**	4396	0.5243	15471	0.1902
100	urls::http://m_soopay_net:8080/wire…	**1.3897**	15	0.0018	18	0.0002

* The frequencies $Freq(\cdot)$ and probabilities $P(\cdot)$ are based on the *training* dataset. $\boldsymbol{M}$=Malware, $\boldsymbol{G}$=Goodware. ■