DANdroid: A Multi-View Discriminative Adversarial Network for Obfuscated Android Malware Detection

Anonymous Author(s)

ABSTRACT

We present DANdroid, a novel Android malware detection model using a deep learning Discriminative Adversarial Network (DAN) that classifies both obfuscated and unobfuscated apps as either malicious or benign. Our method, which we empirically demonstrate is robust against a selection of four prevalent and real-world obfuscation techniques, makes three contributions. Firstly, an innovative application of discriminative adversarial learning results in malware feature representations with a strong degree of resilience to the four obfuscation techniques. Secondly, the use of three feature sets; raw opcodes, permissions and API calls, that are combined in a multi-view deep learning architecture to increase this obfuscation resilience. Thirdly, we demonstrate the potential of our model to generalize over rare and future obfuscation methods not seen in training. With an overall dataset of 68,880 obfuscated and unobfuscated malicious and benign samples, our multi-view DAN model achieves an average F-score of 0.973 that compares favourably with the state-of-the-art, despite being exposed to the selected obfuscation methods applied both individually and in combination.

CCS CONCEPTS

• **Security and privacy → Malware and its mitigation**; • **Computing methodologies → Adversarial learning**; **Multi-task learning**; **Neural networks**;

KEYWORDS

Deep Learning; Convolutional Neural Networks; Android Malware Detection; Obfuscation

1 INTRODUCTION

Android usage statistics from Google show there are over 2 billion active devices in use globally each month, with 82 billion apps and games downloaded every year [40]. With momentum building for its use in IoT-connected devices and vehicles, Android implementations are a frequent and increasing target of malware attacks. Hence, there is a significant and pressing need for Android malware detection techniques given the sheer volume of potentially dangerous apps that may negatively affect financial, technical and social reputations. With this huge number of apps to review, manual inspection cannot scale. Traditional machine learning (ML) systems using statistical methods and feature sets hand-crafted by malware experts can be slow to react to changing and new threats. Unless this laborious manual engineering and ranking of features continually takes place over time, these detection methods risk only being fully effective in the short-term. In addition, relatively few research publications have advanced the classification of obfuscated Android malware. This presents an opportunity for new techniques that do not need expert malware domain insight to generate or rank features, and can display a level of resilience to obfuscation.

Obfuscation is a challenging problem for current detection systems, with Android malware authors regularly using techniques like encryption, reflection and reference renaming. These aim to disguise and camouflage malicious functionality in an app, tricking a model into classifying it as benign. For example, obfuscation is almost universally employed to hide use of APIs [45], and the use of encryption algorithms is five times higher in malicious apps than in benign [4, 23]. Furthermore, authors in [43] analyse 76 malware families, finding almost 80% of apps use at least one obfuscation technique. This increasing use of obfuscation generally was noted in a study of almost half a million Android apps [23] . The problem is further compounded by the fact that many benign apps are obfuscated to protect intellectual property. While it is nearly impossible for detectors to be resistant to any and all obfuscation techniques, due the difficulty of the problem and a lack of quality obfuscated datasets of sufficient size, it is crucial emerging detection methods are robust with respect to whatever obfuscated samples might exist at that point in time, and that they purposefully consider the presence of obfuscation in their design.

We propose several approaches to addressing the obfuscation problem. Firstly, we present a deep learning Discriminative Adversarial Network (DAN) with two cost functions, one to minimise classification error for malware, and the other to maximise classification error for obfuscation, ensuring the internal features learned are useful for malware detection whilst simultaneously being ignorant of obfuscation caused by four commonly used techniques. The DAN employs the adversarial learning aspect of a GAN [16, 17] but instead of training a generative model, we train two discriminators, one for malware and another for obfuscation. This learning algorithm is inspired by previous work from other fields in domain adaptation [13] and feature disentanglement [24, 41], with obfuscation considered a form of bias to remove from the learning process. Secondly, we adopt a multi-view learning approach that uses obfuscation-resilient feature sets of permissions and API calls, in addition to raw opcode sequences that are extracted directly from each Android app. The model's resilience to obfuscation is tested against four specific techniques applied individually, and in combination - class encryption, API calls obfuscation, string encryption and resource encryption. We also provide evidence the features learned by our multi-view DAN model are less affected by obfuscation compared to without using DAN. Lastly, through considering obfuscation as a form of data augmentation, we augment the original training data with the obfuscated versions of the same malicious and benign apps, annotating each sample with two labels; either malicious or benign, and either obfuscated or unobfuscated.

Our contributions in this paper are:

- A multi-view deep learning Android malware detector that demonstrates excellent performance despite the presence of four selected obfuscation techniques. This DAN model

gives an F-score of 0.973 in classifying obfuscated and unobfuscated samples, comparable with the state-of-the-art, and an F-score of 0.948 in classifying samples obfuscated with all four techniques at once.
- An innovative DAN which learns a shared representation from three feature sets - raw opcodes, permissions and API calls - via a bespoke adversarial cost function that negates the obfuscation cost, ensuring this learning process is resilient to, and ignorant of, the selected obfuscation techniques.
- A demonstration that our detector has the potential to generalize to rare and future obfuscation techniques not seen in training.

The rest of this paper is as follows - Section 2 discusses related work on Android malware detection, obfuscation-resilience of existing detectors, and adversarial learning. Section 3 outlines how we extract input features from our Android samples, Section 4 contains our multi-view DAN model architecture, the discriminative adversarial learning process, and a detailed description of the DAN, while Section 5 discusses the four encryption methods used and the creation of the training and testing dataset splits. Results, a state-of-the-art comparison and a commercial anti-malware engine comparison are in Section 6, Section 7 verifies the obfuscation-resilience of the features learned using DAN compared to not using DAN, and Section 8 contains our conclusions.

2 RELATED WORK

2.1 Android Malware Detection

Android malware detection uses either static analysis, using derived signatures with no code execution, or dynamic analysis, where an attempt is made to execute malware in real-time for behavioural study. Early research incorporating traditional ML algorithms included k-means clustering, kNN [35, 39], SVM [5, 28, 50], decision trees [1, 7, 8, 14, 47], and naive Bayes [47]. These ML algorithms usually have manually selected or ranked features as input, such as malicious system call traces [6], permissions [34, 37], APIs [1, 27, 32, 37, 39, 50], network addresses [5], network traffic [22, 42] and embedded call graphs [15]. However, a reliance on expert knowledge for feature engineering can render a model more vulnerable to change than if the model learns features itself. Others derived features using feature selection techniques [1, 2, 47] and n-grams [42] though these methods were often evaluated with unbalanced datasets that create uncertainty about how they would generalize in the wild.

Recently, deep learning methods have been proposed that allow feature extraction and classification to occur simultaneously in an end-to-end architecture, with notable progress in using Convolutional Neural Networks (CNNs). DroidDetector [49] and MalDozer [20] were two of the first, with the former using a deep belief network, and the latter presenting a detection and family attribution framework using API sequence classification via a CNN. Similarly, authors in [30] use a CNN to learn opcode patterns akin to natural language processing. In [29], a CNN is used with system calls extracted via dynamic analysis and treated as sentences for input. [31] helped introduce recurrency using a CNN with a Long Short-Term Memory unit (LSTM) to learn program flows of system API call sequences. A combination of different feature types is attempted in [21] using multimodal deep learning with a complex collection of neural nets. None of these approaches were tested on obfuscated samples to assess specific resistance to such techniques.

2.2 Obfuscation Resilience

Despite existing detectors from established vendors having a poor resilience to simple obfuscation [33], research methods are rarely evaluated on obfuscated malware, due to the difficulty of the problem and lack of suitable samples. TIRO addresses language-based and runtime-based obfuscation by developing a de-obfuscator [45]. With a dataset including apps provided by Google, they find obfuscation techniques are present in 80% of those apps and indicate the susceptibility of IntelliDroid [44] to obfuscation. FalDroid [12] requires extra tools to evaluate its resilience to obfuscation, with the authors listing handling advanced obfuscation techniques as future work. RevealDroid [14] made early progress in dealing with basic obfuscation techniques, whilst DroidSieve [38] incorporated features such as permissions and artefacts introduced by obfuscation. DeepRefiner [46] employs LSTM units to analyse Android bytecode, however there may be a complexity issue in that approach with the model containing at least 18 million parameters. [25] uses semantic labels for classification determined from API methods specifically invoked within loops from an app's code. This is tested against control flow graph obfuscation and reflection. Results remained high for the former, but for the latter, when reflection is applied to all API calls across all malicious and benign apps in their dataset, every app is classified malicious, meaning a 100% false positive rate.

A useful study undertaken in [26] reports detection rates of thirteen different commercial anti-malware engines when faced with obfuscated apps. Notably, when attempting to detect malware where each app is obfuscated with multiple techniques, the commercial engines only detect 35% at most, with some detecting none at all. Lastly, [9] state it is not possible to analyse malicious code that is thoroughly obfuscated. While this is true in static analysis, for instance when using highly advanced encryption algorithms such as AES, we aim to show that deep learning has the potential to encapsulate its own feature representations inside a model that are discriminative against lesser but very common forms of encryption [36], instead of the traditional ML method of hand-crafting them. Indeed, authors in [9] also indicate obfuscation may leave detectable traces, and again deep learning architectures such as CNNs can learn from these motifs to aid classification.

2.3 Adversarial Learning

Adversarial learning takes place when two opposing tasks are played off against each other and learnt in parallel. Seminal research in [16] and [17] proposed this adversarial process of training two models at the same time. One is generative, representing a data distribution, the other is discriminative, trying to predict if a sample is from the generator, or is from that same true data distribution. This is a Generative Adversarial Net (GAN), an adversarial game with the two models competing to improve their methods of generation and prediction respectively. GANs have mostly been used in computer vision for image generation, using two differing and competing cost functions. In this paper, we view malware detection and obfuscation as two opposing tasks that need to be considered

together, however a GAN in its original form is not applicable since there is no generative or synthetic aspect to malware classification. We explain how GANs influence our DAN architecture and the treatment of obfuscation as a form of bias in Section 4.2.2. An interesting example from a very different domain saw an adversarial network trained to remove racial bias from predicting recidivism [41] in the US.

3 FEATURE EXTRACTION

From each sample we extract three input feature sets - 1) opcode instructions; previously shown to be an effective feature set [30], 2) permissions; which cannot be obfuscated without rendering the app useless, and 3) the presence of a selection of API calls, Android commands and Linux terminal commands [48]. The latter two feature sets provide useful information for Android malware classification [5, 28, 32, 34, 50], since a malware detector learning only from opcodes is likely to be cheated easily with obfuscation, especially if such evasive techniques are not considered in the learning process. We point out that these features are simply extracted, and expert malware knowledge is not used to rank or engineer them using statistical methods. Indeed, it is the model itself that learns their weighting, encapsulating the choice of how and if these input features are used in the internal feature representation.

To create these input feature sets, we first disassemble each sample's *.dex* file into various *.smali* files using Apktool [3] and baksmali [19]. Each *.smali* file represents a single Java class and contains its methods. A method consists of a set of instructions, and each instruction comprises one single opcode and various operands. The opcode sequences are extracted from each method and the operands are discarded. These opcode sequences from all Java classes are concatenated to give one single sequence of opcodes that represent one sample. Each sample also contains an *AndroidManifest.xml* file with permissions requested by that sample. Permissions are extracted via parsing this manifest. The various API calls, Android commands, and Linux terminal commands are extracted from the *.smali* files using a tool provided by [48]. The output of this feature extraction are the input feature sets as seen in Figure 1.

4 METHODOLOGY

We propose a Discriminative Adversarial Network (DAN) that exploits the presence of four selected obfuscation techniques in the training data to make malware detection resilient to those same techniques. In deep learning, obfuscation of apps can be considered a data augmentation technique [18]. Several obfuscation techniques are similar to augmentation techniques used in deep learning for computer vision, such as adding noise, changing brightness, reflections or rotations. Therefore, we augment our original unobfuscated training data with obfuscated apps to enable the learning of feature representations that display resilience to the obfuscation techniques in question. We develop a novel deep learning detection architecture comprising a CNN and two neural nets that, in combination, learn from the three input feature sets.

4.1 Multi-View Feature Learning

Our deep learning architecture is shown in Figure 1. We select this multi-view approach because the representation for each input feature set is implicitly not identical, due to their modalities being significantly different in nature. Our design decision in not combining all feature sets into one at the initial input stage allows us to tune the model more effectively, and enables future in-depth analysis of each section of architecture, which is beyond the scope of this paper. Moreover, with the operands being discarded from opcode instructions, we hypothesise exploiting the presence of permissions and various API calls in addition to the opcodes will provide more information to the model.

4.1.1 Opcode Sequence CNN. The CNN aims to learn discriminative patterns from each sample's sequence of raw opcode instructions that can be used to differentiate between malicious and benign. This decoupling of expert knowledge means these concepts are learnt directly and encapsulated inside the CNN. Thus, effort needed to handle changes introduced by opcode authoring variations over time is significantly less compared to repeating complex mathematical ranking and feature engineering. This is a differentiator between our method and previous work where more manual, hand-engineered features are required. We need only to extract the raw opcodes from each sample, which is purely an implementation task and needs no Android malware domain knowledge. The CNN does not need to be informed what the most malicious opcode patterns are, nor which ones are safe to be overlooked. The input opcode sequence I_o for a given sample is a series of one-hot vectors, denoted as $I_o = \{x_1 ... x_n\}$, with x_n being the nth opcode in the sequence. The one-hot vector x_n has d=218 dimensions to represent all potential opcode instructions, with a 1 at the element index corresponding to the opcode used at position n in I_o, and 0 in all other elements. CNNs use filters, also known as kernels or sliding windows, to learn feature representations from spatial topologies using relationships between individual nearby elements. This allows the learning of discriminate patterns of consecutive opcode instructions.

However, initially the one-hot vector encoding produces a sparse topology with little meaningful information for a filter to learn from, as illustrated in Figure 2. Since the useful non-zero elements are so far apart, the filter size would need to be inefficiently large. To remove this data sparsity and allow the model to better learn relationships between opcodes, the one-hot vectors are projected into a dense k-dimensional embedding space. The information-rich embedding projection P is created by multiplying each one-hot vector in I_o by an embedding weight matrix W_E of size $d \times k$.

W_E has values that are randomly initialised at the start of training and updated during the learning process. The resultant P is a matrix of size $n \times k$, processed by a single convolutional layer with m filters. Each filter is of size $s \times k$, with s being the number of consecutive opcodes analysed by each filter. Each of the m filters perform a 1D convolution over the full matrix P generating an activation map a of size n as follows:

$$a_m = ReLU(Conv(P)_{W_m b_m}) \tag{1}$$

where W_m and b_m are the weight and bias parameters of the mth convolutional filter learned during training. The rectified linear

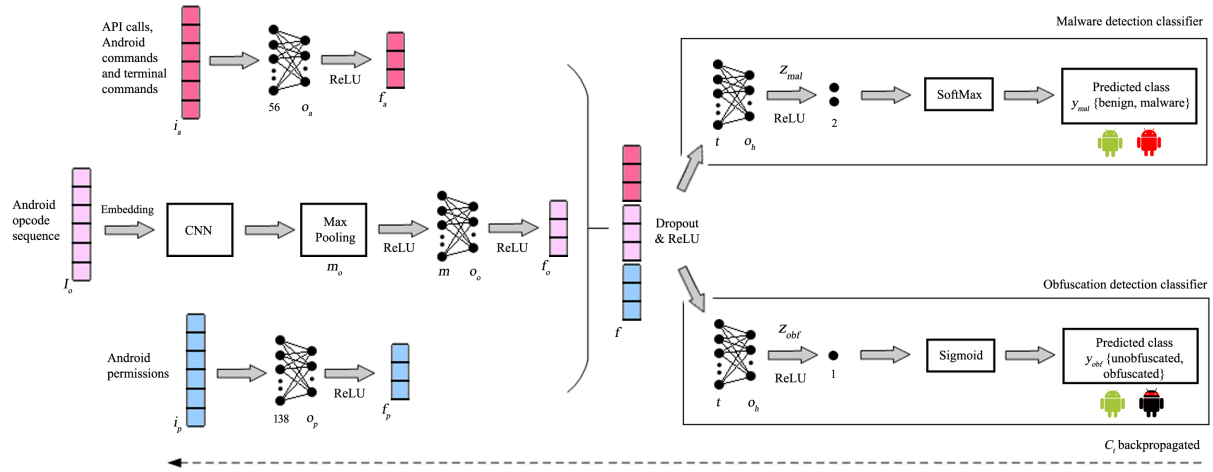

Figure 1: Multi-view Discriminative Adversarial Network (DAN) architecture for Android malware detection

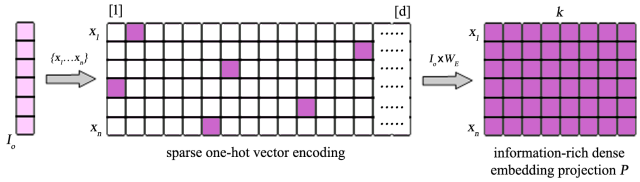

Figure 2: Creating the embedding projection P

activation function $ReLU(x) = max\{0, x\}$ is used throughout. All resulting activation maps are stacked to produce the activation matrix A, of size $n \times m$, denoted as:

$$A = [a_1|a_2|...|a_m] \tag{2}$$

Global max-pooling produces a vector m_o of length m containing the maximum activation of each learned convolutional filter over the program length n, formally:

$$m_o = [max(a_1)|max(a_2)|...|max(a_m)] \tag{3}$$

This focuses the classifier's attention on parts of the opcode sequence that generate the maximum activation of each filter, i.e. the opcode sections the model deems most interesting and relevant for detection, regardless of their location in the sequence. Max-pooling also ensures an input program of arbitrary length is represented as a fixed-length vector. The output from max-pooling then passes through a fully-connected layer with o_o output neurons, producing the first feature vector f_o:

$$f_o = ReLU(W_o m_o + b_o) \tag{4}$$

where W_o and b_o are the weight and bias parameters of the hidden layer learned during training.

4.1.2 Permissions Neural Net. The permissions complement features derived from opcodes as they contain useful data for malware detection and can be deemed to provide resilience against obfuscation - permissions in the manifest cannot be obfuscated if they are to be used. For the permissions input, given a sample and its corresponding manifest file, a multi-hot vector of length 138 is generated, where each element maps to one of 138 Android permissions e.g. SEND_SMS, with the element value being 1 if a permission is found in the manifest, or 0 if not. This permissions vector i_p is input to a fully-connected layer to allow the model to learn relationships between permissions for classification, giving the second feature vector f_p, denoted as:

$$f_p = ReLU(W_p i_p + b_p) \tag{5}$$

where W_p and b_p are the weight and bias parameters of the permissions input layer learned during training. The output layer in this neural net has o_p neurons.

4.1.3 API Calls Neural Net. For the API calls input, which is also complimentary to the other two feature sets of opcodes and permissions, a samples's *.smali* files are parsed for the presence of 56 API calls, Android commands and Linux terminal commands listed by [48]. This includes API calls made to, for example, the SMS manager API and telephony manager API. Android commands are, for example, *pm install* that can be used for installation of packages. The terminal commands are, for example, Linux CLI commands like *chmod* or *mount*. For brevity this is referred to as the API calls neural net. After parsing, a multi-hot vector i_a of length 56 is generated, which is input to a fully-connected layer to allow the model to learn relationships between these various calls and commands, resulting in the third feature vector f_a denoted as:

$$f_a = ReLU(W_a i_a + b_a) \tag{6}$$

where W_a and b_a are the weight and bias parameters of the API calls input layer learned during training. The output layer in this neural net has o_a neurons.

4.1.4 Classification Layer. As per Figure 1, each learned feature vector is concatenated into a single feature representation f. The make-up of f depends on which permutation of input feature sets are used. It can be denoted as:

$$f = [f_o|f_p|f_a] \tag{7}$$

Dropout is applied at a rate of 0.5, then f is input to a multi-layer perceptron (MLP) with an input layer of size t=o_o+o_p+o_a, a hidden layer h_{mal} of size o_h neurons, and an output layer of size 2 neurons, since our malware classification is a two-class problem, either malicious or benign. z_{mal} can be denoted as:

$$z_{mal} = ReLU(W_{h_{mal}} f + b_{h_{mal}}) \tag{8}$$

where $W_{h_{mal}}$ and $b_{h_{mal}}$ are the weight and bias parameters of the hidden layer learned during training. This final output from the MLP, z_{mal}, is a vector with each element giving a score that the test sample is associated with each class. z_{mal} is passed to the SoftMax classification layer, which outputs the normalized probability of the test sample belonging to each class in the problem, formally:

$$p(y_{mal} = i|z_{mal}) = \frac{\exp(w_i^\intercal z_{mal} + b_i)}{\sum_{i'}^{I} \exp(w_{i'}^\intercal z_{mal} + b_i)} \tag{9}$$

where w_i and b_i are the weight and bias parameters of the SoftMax layer learned during training for each of the i classes, $w_i^\intercal z_{mal}$ is the inner product of w_i and z_{mal}, with y_{mal} being the predicted label.

4.2 The Discriminative Adversarial Learning Process

4.2.1 Android Malware Classification. In training, the cost function is minimized by iteratively updating the network's parameters, that is, all the weights and biases present in the model. These learnable parameters encapsulate the weighting of each feature and their contribution towards malware detection. The objective is to reduce the difference between the prediction and ground truth of each sample. The more the difference, or the cost, is minimized, the better the learning. The better the learning, the more confident we can be in the model's predictive power. This is a supervised approach as for training we possess ground truth labels for every sample as to whether it is malicious or not. For malware detection, the cost function C_{mal} to be minimized for a batch of B training samples, $\{I_{(1)}, ..., I_{(B)}\}$, can be written as:

$$C_{mal} = -\frac{1}{B} \sum_{j=1}^{B} \sum_{i=1}^{c} 1\{y'_{mal(j)} = i\} \log p(y_{mal(j)} = i|z_{mal(j)}) \tag{10}$$

where c is the number of classes in the task, $y'_{mal(j)}$ is the ground truth for sample $I_{(j)}$ and $z_{mal(j)}$ is the resulting output after the forward pass of $I_{(j)}$ through the network. Learning is performed by stochastic gradient descent which updates the network parameters via backpropagation after each sample is forwarded using the gradient of the cost function with respect to these parameters. During training the network is repeatedly presented with batches of training samples in random order until the parameters converge, and the cost function is minimized. A learning rate α is used in this process.

4.2.2 Discriminative Adversarial Network (DAN). The aim of the DAN is to ensure learned features are not biased by the presence of obfuscation and malware detection remains accurate. This is achieved by modifying a GAN architecture and replacing the generator with another discriminator, such that the replacement discriminator learns to detect malware whilst the other discriminator ensures ignorance to obfuscation. Consider this problem space one of domain adaptation, where malware detection is the main task, and obfuscated and unobfuscated malware are two similar, but different, domains. In this setting, we deliberately reverse the gradient of the obfuscation learning process, meaning the model generalises from one domain to another, being discriminative for the main task of malware detection, and indiscriminate between the obfuscated and unobfuscated domains.

As per Figure 1, f is a shared feature representation fed into both discriminator subnets simultaneously during the forward pass of the data. Each discriminator subnet is an MLP, with one responsible for malware detection and the other for obfuscation detection. They output the vectors z_{mal} and z_{obf} that measure the likelihood of a sample being malware and obfuscated respectively, which are passed to SoftMax and Sigmoid layers, leading to predictions y_{mal} and y_{obf}. The formal SoftMax output is already given in Equation 9, and the Sigmoid function in the obfuscation subnet outputs the normalized probability of the test sample being obfuscated, formally:

$$p(y_{obf} = i|z_{obf}) = \frac{\exp(wo_i^\intercal z_{obf} + bo_i)}{\exp(wo_{i'}^\intercal z_{obf} + bo_i) + 1} \tag{11}$$

where wo_i and bo_i are the the weight and bias parameters of the Sigmoid layer learned during training for each of the i classes, $wo_i^\intercal z_{obf}$ is the inner product of wo_i and z_{obf}, with y_{obf} being the predicted label. Using a single output neuron with Sigmoid means i = 1. This is supervised learning using separate sets of ground truth labels for both tasks. Formally, the cost functions C_{mal} and C_{obf} are:

$$C_{mal} = -\frac{1}{B} \sum_{j=1}^{B} \sum_{i=1}^{c} 1\{y'_{mal(j)} = i\} \log p(y_{mal(j)} = i|z_{mal(j)}) \tag{12}$$

$$C_{obf} = -\frac{1}{B} \sum_{j=1}^{B} \sum_{i=1}^{c} 1\{y'_{obf(j)} = i\} \log p(y_{obf(j)} = i|z_{obf(j)}) \tag{13}$$

Both C_{mal} and C_{obf} are used to calculate C_t. Thus our bespoke adversarial cost function can be written as:

$$C_t = (1 - \lambda)C_{mal} - (\lambda)C_{obf} \tag{14}$$

where the parameter λ allows the errors to be weighted, preventing one discriminator from dominating the cost function and reducing the performance of the other. Note our negation of C_{obf}, which is crucial. This results in the final learned features encapsulated inside the model containing little to no salient characteristics regarding obfuscation. Thus the DAN architecture obtains discriminative features that are full of useful information for malware detection whilst having resilience to the selected obfuscation techniques. In Section 7 we verify to what degree those learned malware characteristics are correct, and hence how resilient the model is to the four obfuscation techniques. Algorithm 1 shows how the DAN training phase is carried out.

5 EXPERIMENTAL SETUP

The benchmark dataset Drebin [5] is used in its unobfuscated form augmented by five other versions of the same apps obfuscated with

Algorithm 1 Multi-view DAN model training

Require: Cost function C_{mal}, cost function C_{obf}, randomly initialised model parameters θ, number of epochs e, weighting parameter λ, batch size B, learning rate α.

procedure DAN Training(C_{mal}, C_{obf}, θ, e, λ, B, α)
  $\lambda \leftarrow \in (0, 1)$ ▷ Initialise weighting parameter
  $\alpha \leftarrow \in (0, 1)$ ▷ Initialise learning rate
  $B \leftarrow \mathbb{N}$ ▷ Initialise batch size
  $e \leftarrow \mathbb{N}$ ▷ Initialise number of epochs
  for all $epoch \in e$ **do**
    $B \leftarrow samples$ ▷ Build a batch of B samples
    for all $x \in B$ **do**
      $y_{malx} \leftarrow \{0, 1\}$ ▷ Set malware label
      $y_{obfx} \leftarrow \{0, 1\}$ ▷ Set obfuscation label
      $C_{mal} \leftarrow \mathbb{R} \leq 0$ ▷ Calculate cost C_{mal}
      $C_{obf} \leftarrow \mathbb{R} \leq 0$ ▷ Calculate cost C_{obf}
      $C_t \leftarrow (1-\lambda)C_{mal}$ - $(\lambda)C_{obf}$ ▷ Calculate total cost C_t
    end for
    for all $\theta_i \in \theta$ **do**
      $\theta_i \leftarrow \theta_i - \alpha(\frac{\partial}{\partial \theta_i})C_t(\theta_i \ldots \theta_n)$ ▷ Calculate each update
    end for
    $\theta \leftarrow (\theta_i \ldots \theta_n)$ ▷ Execute all updates at once
  end for
end procedure

the real-world tool DexProtector, which is used by corporations on the Fortune 500 and Forbes 2000 lists [11]. This dataset was chosen due to its continued use throughout published work on Android malware detection, and DexProtector was chosen as it is enterprise-grade, compared to other tools which have less capability and are also open-sourced. Our use of the same apps in obfuscated and unobfuscated states allow direct comparisons on the difficulties of various scenarios. The original Drebin dataset, containing 5,560 apps from 179 families, is augmented with a further five obfuscated versions of same apps. The four obfuscation techniques, applied separately to create four of these five new versions, are:

- **Class encryption** - encrypts the entire *classes.dex*, which stores all classes used in a sample. Classes protected by this function are encrypted and moved from the original *classes.dex* file, becoming invisible to feature extraction tools.
- **API calls obfuscation** - masks the calls of library methods and application methods with reflection and dynamic functions. It also hides fields types and access to fields.
- **String encryption** - uses dynamic key cryptography to hide the contents of string constants. Keys cannot be extracted from the code base, as they are calculated in real-time. If code is reverse engineered, the true contents of strings are hidden and are inaccessible for static analysis.
- **Resource encryption** - encrypts the internal resources and assets of a sample plus obfuscates resource names to prevent malicious copying and modification.

The fifth new version of the Drebin dataset is created by subjecting each app to all four obfuscation techniques applied together. This layered combination of all four techniques makes detection much more difficult [26]. We balance each of these malicious datasets with benign samples. Firstly, 5,920 unobfuscated benign apps were collected from Google Play. Then, to help mitigate bias in the learning process, these benign apps are also obfuscated using DexProtector, with the identical application of each obfuscation technique separately producing four new versions, plus again a fifth version with all four techniques applied to each app. All our obfuscated apps will be made available to the community for facilitating future research.

Table 1: Detection performance of a single-view model trained with only unobfuscated samples

Obfuscation	Acc.	Prec.	Recall	F-score
None	0.977	0.98	0.973	0.976
Resource encryption	0.937	0.905	0.982	0.942
String encryption	0.916	0.896	0.945	0.92
API calls obfuscation	0.672	0.726	0.578	0.644
Class encryption	0.503	0.503	1	0.669
All four obfuscations applied	0.51	0.514	1	0.679
Average	0.596	0.754	0.913	0.805

After balancing, each dataset is then split into 80% for training, 10% for validation, and 10% for testing. It is ensured the malicious and benign apps in the testing sets are different to those in the training and validation sets, and the balance of malware and benign apps in the training, validation and testing sets are the same as in the whole dataset, that is, almost 50/50. Note that as obfuscating each app produces five further versions, and we consider each version as a separate sample. With these obfuscated and unobfuscated versions of the same app used together as a form of data augmentation, it is ensured an original app and each of its obfuscated versions are only present in one split. That is, all versions of an app are used either in training or in testing. This avoids inflated performance and overfitting as any potentially obfuscation-resilient input features seen in training are not used for testing.

Models are trained for e=75 epochs with the same, unseen held-out validation set tested every five epochs. Validation set results from the original unobfuscated Drebin dataset are used to tune the hyperparameters in the CNN and the rest of the model. This prevents tuning the model to test sets and artificially improving results. The optimal hyperparameter values are B=1, n=8192, s=8, k=8, m=64, o_o=16, o_p=64, o_a=16, o_h=16, α=0.001 and a kernel stride of 1. We empirically find using a dimensionality k=8 gives an optimal balance between performance and computation time. When training the multi-view DAN model we set λ=0.5 so neither task has unfair influence in the calculation. These hyperparameter values are the same in all our experiments to avoid overfitting.

6 RESULTS

6.1 Single-view model trained with only unobfuscated samples

We start by evaluating the performance of an existing detector [30] against the obfuscation techniques to give a baseline against which to compare our contributions. The single-view model in this experiment comprises only the opcodes CNN. For training we use the training set of the original unobfuscated dataset. The model is tested against the test set of each type of obfuscation, plus the unobfuscated test set. Overall, obfuscation diminishes performance for this single-view model trained only with unobfuscated samples. Table 1 shows these poor results, worsening with obfuscation complexity.

Although the model can classify unobfuscated malware and benign samples well with an F-score of 0.976, performance decreases once obfuscation is introduced, radically so as the techniques become stronger, with API calls obfuscation and class encryption producing F-scores of 0.644 and 0.669 respectively. Furthermore, when faced with either class encryption or the application of all four obfuscations, every single sample in testing was classified as malicious. The resulting false positive rate of 100% in both cases is a strong indicator of the power of these techniques to confuse this baseline model.

6.2 Multi-view model trained with both obfuscated and unobfuscated samples

In this experiment we extend the single-view approach to a multi-view model that is trained with the combined training sets of each type of obfuscated malware and original unobfuscated malware, balanced with both obfuscated and unobfuscated benign samples. The architecture is as per Figure 1 though only uses C_{mal} as the cost function. The same testing sets as in Section 6.1 are used for comparison.

The inclusion of obfuscated benign samples helps mitigate against learning bias and creates a more challenging scenario - without using obfuscated benign samples, if a model silently learned only obfuscation features, seemingly good results could be attained by classifying any obfuscated sample as malware. Thus, adding obfuscated benign samples is a sterner test but it also allows the learned features to generalize over the selected four obfuscation techniques if successfully trained. It is important to note that in the wild, benign samples are obfuscated to protect Intellectual Property and prevent reverse engineering.

The breakdown of all 68,880 samples used is:

- 5,560 original unobfuscated Drebin malware samples.
- 22,240 obfuscated Drebin malware samples, comprising the four individual different obfuscation techniques.
- 5,560 obfuscated Drebin malware samples with all four obfuscation techniques applied together to each sample.
- 5,920 unobfuscated benign samples.
- 23,680 obfuscated benign samples, comprising the four individual different obfuscation techniques.
- 5,920 obfuscated benign samples will all four obfuscation techniques applied together to each sample.

Recall this dataset was originally split into training, validation and testing sets, with balance between malware and benign samples in each set, and all versions of a single app only belonging to one set. This avoids artificial performance in testing where the same input features for a sample, which could be unchanged despite obfuscation, are used in training, and then also appear in testing. In other words, it is ensured all versions of an app, unobfuscated or obfuscated, only appear in one set.

Using the multi-view model with the two extra feature sets improves malware detection performance across all metrics, as per Table 2, with an average F-score of 0.975. Compared to the Table 1 baseline results, performance is much improved in every case, even when faced with difficult obfuscations, with the API calls obfuscation F-score rising to 0.991 from 0.644, the class encryption F-score rising to 0.951 from 0.669, and the multiple obfuscations F-score rising to 0.952 from 0.679. The average F-score also rises to 0.975 from 0.805. Whilst these results are promising, it should be ensured our model is not silently learning obfuscation characteristics, thereby becoming an obfuscation detector, as opposed to truly being a malware detector. This is mitigated in our next experiment with the introduction of the DAN.

Table 2: Detection performance of a multi-view model trained with both obfuscated and unobfuscated samples

Obfuscation	Acc.	Prec.	Recall	F-score
None	0.982	0.968	0.996	0.982
Resource encryption	0.987	0.994	0.978	0.986
String encryption	0.988	0.992	0.983	0.987
API calls obfuscation	0.99	0.99	0.99	0.991
Class encryption	0.951	0.955	0.946	0.951
All four obfuscations applied	0.953	0.952	0.952	0.952
Average	0.975	0.975	0.974	0.975

Table 3: Detection performance of the multi-view DAN model trained with both obfuscated and unobfuscated samples

Obfuscation	Acc.	Prec.	Recall	F-score
None	0.987	0.984	0.989	0.986
Resource encryption	0.99	0.991	0.991	0.991
String encryption	0.988	0.989	0.987	0.988
API calls obfuscation	0.977	0.982	0.973	0.977
Class encryption	0.947	0.964	0.93	0.947
All four obfuscations applied	0.948	0.97	0.927	0.948
Average	0.973	0.98	0.966	0.973

6.3 Multi-view DAN model trained with both obfuscated and unobfuscated samples

Here we use the multi-view DAN model, now using both C_{mal} and C_{obf}, with the negation of the obfuscation cost, which is trained with the combined training sets of each type of obfuscated malware and original unobfuscated malware, balanced with both obfuscated and unobfuscated benign samples. Using the DAN in addition to the multi-view allows us to verify the obfuscation-resilience of the learned malware features, and check if the DAN affects detection performance itself. Without the DAN, in theory a model could learn obfuscation itself as a malware indicator, rather than learning useful features for malware detection. Again, the same testing sets as in Section 6.1 are used for comparison.

Results in Table 3 show excellent malware detection performance across all metrics using the multi-view DAN model despite the presence of obfuscation, with an average F-score of 0.973. Overall we achieve strong performance by using DAN and compared to the baseline results in Table 1, one can see significant performance improvement in all cases, including classifying samples under complex obfuscations, with the API calls obfuscation F-score rising to 0.977 from 0.644, the class encryption F-score rising to 0.947 from 0.669, and the multiple obfuscations F-score rising to 0.948 from 0.679. The average F-score also rises to 0.973 from 0.805. This is due to augmentation of the training data with obfuscated samples, and because the two extra input feature sets in this multi-view DAN model are less affected, or not affected at all, by each obfuscation technique, displaying a valuable level of obfuscation-resilience. For

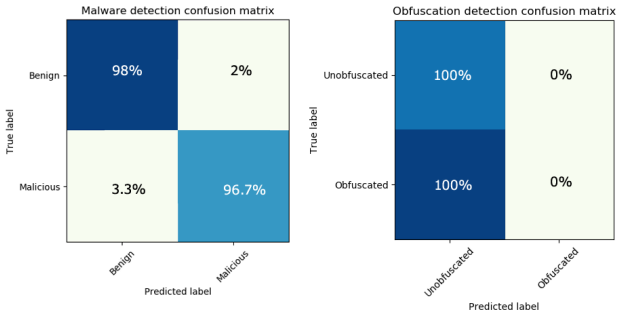

Figure 3: Multi-view DAN model malware detection and obfuscation detection confusion matrices

example API calls extraction uses regular expressions with random or null values for obfuscated APIs, but the model still performs well due to the other complementary features. Also, when comparing Table 3 to Table 2, we can see that malware detection performance is maintained with the introduction of DAN, and it does not have a detrimental effect.

We believe these results are representative of real-world performance in overcoming these four obfuscation techniques as:

- the apps in the testing set are totally unseen and are not present in any form, original or obfuscated, in the training set.
- our model is not unnecessarily complex, with the number of learnable parameters (15,506) much smaller than size of the training set (42,608 samples), due to our augmenting of the original apps with their obfuscated versions.
- the use of dropout and the ReLU activation function introduces noise and non-linearity to the learning process.
- the model hyperparameters were not tuned directly to the test set, with validation set performance used for selecting their final values, as per Section 5.

When the obfuscation predictions from the model are studied in Figure 3, we find that - as expected - every single sample is classified as unobfuscated. This confirms our desired outcome has been reached, in that we have successfully removed obfuscation bias from the learning process thanks to our bespoke adversarial cost function with a negated obfuscation cost. We highlight DAN isn't necessarily about better malware detection performance but removes bias to obfuscation.

These results further show that this multi-view DAN architecture can successfully learn from the three input feature sets of opcodes, permissions and API calls in order to generate a shared feature representation f which is characteristic of malware and not obfuscation, meaning the model has not been tricked or misled. Were this not the case, malware detection performance would be much lower. Thus, we can say our multi-view DAN model is resilient to the four professional obfuscation techniques used in this paper.

6.4 Generalisation capability of DANdroid to unseen obfuscations

Our results are compelling when considering the selected four obfuscation techniques. However we should not presume these are the only four techniques in existence - rather, we selected these due to their prevalence both in the real-world and the literature concerning Android obfuscation. Ideally, our model will generalise to future and rarer obfuscation techniques for which currently there is no training data for, or variations of current techniques. To evaluate this, each one of the four obfuscation types in turn is held-out for testing only to keep it totally unseen during the learning process. Then our multi-view DAN model is trained with all the unobfuscated and obfuscated malicious and benign samples from the three remaining obfuscation techniques. The process is repeated four times, once for each technique, allowing us to assess the model's generalisation capability to such unseen obfuscations. This setting is particularly challenging as not only have the test set samples never been seen by the model, not even under a different obfuscation as per Section 5, but each held-out test set is also obfuscated with a technique never seen in training.

Table 5 shows using the multi-view DAN model deals very well with unseen obfuscations, with an average F-score of 0.94. As one may expect, the F-score decreases as the obfuscation becomes more powerful, reaching a low of 0.845 when faced with class encryption, compared to 0.977 for resource encryption. On balance, we assert this is a promising indicator of generalisation capability. For future work we plan to build new datasets of rarer techniques to study this further.

6.5 State-of-the-art comparison

In Section 2, we found there is little research directly testing Android malware detectors against obfuscated apps. To complicate comparisons, different datasets are used and the code for many detectors is either not available, does not include all required parameter settings, or cannot be shared for legal or territorial reasons. Nevertheless, we compare our multi-view DAN model to Drebin [5] and provide comparisons of other published results specifically for detection of obfuscated samples in Table 4. Note some works do not report the same metrics as us for their experiments using obfuscated samples, though where possible we accurately calculate them from those metrics that are published.

From studying metrics only, we achieve close to the state-of-the-art performance of both Drebin and DeepRefiner [46] - however, it is felt Drebin has perhaps an advantage due to its hand-coded feature extraction method, and there may be some matters of contention in both the experimental setup and methodology of DeepRefiner, as well as the other state-of-the-art approaches, which we will discuss. We also make a direct comparison with commercial malware detectors in Section 6.5.

Whilst Drebin gives an F-Score of 0.986 compared to our F-score of 0.973 using the same training and testing splits as in our experiments, it requires considerably more input features that need to be defined by a malware expert. Drebin uses eight different feature sets (hardware features, requested permissions, app components, filtered intents, restricted API calls, used permissions, suspicious API calls and network addresses) compared to the use of only three by DANdroid. It is this type of expert malware insight and feature engineering occurring outside a classifier that renders it vulnerable to change, as these feature sets could easily change over time, and

Table 4: State-of-the-art comparison - evaluation details of DANdroid versus other work

Model	# obf. samples	Acc.	Prec.	Recall	F-score	Comments
DANdroid (DAN, CNN, Neural Nets)	57,400	0.973	0.965	0.981	0.973	Four real-world obfuscation techniques used, enterprise tool. Dataset balanced and includes obfuscated benign apps. Expert malware knowledge decoupled, no major feature engineering.
Drebin [5] (SVM)	57,400	0.985	0.975	0.997	0.986	Average metrics using same training and testing splits as in our experiments. Manual hand-crafted feature selection, searches for usage of eighteen pre-categorised suspicious Android and suspicious Java APIs.
LoopMC [25] (RF)	24,690	0.295	0.295	1	0.455	Simulated reflection obfuscation. All obfuscated apps classified as malicious.
RevealDroid [14] (SVM)	1,188	N/A	0.86	0.85	0.85	No obfuscated benign samples. Manual hand-crafted feature selection. Simpler obfuscation.
DroidSieve [38] (Extra Trees)	1,260	0.997	N/A	N/A	N/A	No obfuscated benign apps. Manual feature engineering and ranking.
DeepRefiner [46] (Neural Nets & LSTMs)	11,000	0.999	1	0.999	1	No obfuscated benign apps. Model complexity issue (upward of 18 million parameters). Simpler obfuscation.

Table 5: Detection performance of the multi-view DAN model tested against each unseen, held-out obfuscation

Held-out Obfuscation	Acc.	Prec.	Recall	F-score
Resource encryption	0.976	0.991	0.963	0.977
String encryption	0.986	0.991	0.982	0.986
API calls obfuscation	0.95	0.964	0.938	0.951
Class encryption	0.864	0.987	0.739	0.845
Average	0.944	0.983	0.906	0.94

as a consequence the given detector would likely suffer a dip in performance. Drebin's slightly better performance could be expected since its authors are the same as those that released the Drebin dataset, and therefore the used features are tailored to the whole training and testing data. ~~This, however, is expected to drastically drop for new apps not included in that dataset.~~ On the other hand, our use of deep learning means the model encapsulates internally how suspicious or not it deems input features - for example, we simply extract raw opcode sequences, neither carrying out feature engineering nor categorising any as suspicious. The Drebin paper states ML spares them from manually constructing detection rules for the extracted features, however rules have still already been applied to categorise these suspicious APIs. It is this decoupling of human expertise from the DANdroid learning process we believe makes up for the marginal difference in performance compared to Drebin, given it has a larger reliance on feature engineering.

DeepRefiner reports an F-score of 1 compared to our F-score of 0.973, however their system was not exposed to obfuscated benign samples in training or testing, which could lead to silent learning of any obfuscation as a sign of malware. This creates problems in deployment when obfuscated benign samples are seen as malicious with high volumes of false alarms. On the contrary, we actively contemplate this problem and mitigate it using obfuscated benign samples in training. With DeepRefiner having 18 million learnable parameters in three hidden layers alone, much greater than the number of training samples used, there is a distinct possibility it is too complex and may not generalize well in production. In addition, simple obfuscation techniques from [33] were used to evaluate DeepRefiner, rather than our more challenging selection.

We achieve similar accuracy to DroidSieve [38], however obfuscated benign samples are again not included in their training, and only 1,260 obfuscated samples are present in their whole dataset, compared to 57,400 in ours. DroidSieve also appears to require expert malware knowledge for feature engineering and selection.

Our approach outperforms RevealDroid [14], which achieves an F-score of only 0.85 using traditional ML with an SVM classifier. We feel this work may have issues around class imbalance, a reliance on manually engineered features, the use of too few obfuscated samples and again using simpler obfuscation provided by [33].

Similarly, authors in [25] use expert-derived loop features and a conventional Random Forest (RF) classifier. Furthermore, they conduct an obfuscation test across their whole dataset by using reflection on all their malicious and benign apps. This use of reflection results in all the apps being classified as malicious, again showing susceptibility to false alarms, and the reflection obfuscation was simulated by the authors rather than being applied by a professional tool.

Again, overall we decouple expert knowledge from our model to a greater degree than other methods, with the multi-view DAN architecture learning for itself what features contribute to malware detection with no laborious feature engineering. In summary, we feel our approach is more robust than the state-of-the-art due to:

- encapsulated DAN feature learning that is resilient to the four selected obfuscations.
- use of more obfuscation techniques of an enterprise-grade standard compared to other work.
- inclusion of obfuscated benign samples to reduce bias and prevent silent learning of obfuscation as a malware indicator, thus reducing false positives.
- evidence of generalisation potential to future, unseen obfuscation techniques.
- an augmented dataset of considerable size.

Table 6: Detection performance of commercial engines compared to DANdroid on the PRAGuard dataset

Obfuscation	Median of commercial engines [26]	DANdroid
None	0.99	0.99
Reflection	0.97	**0.993**
Trivial obfuscation	0.9	**0.989**
String encryption	0.65	**0.978**
Class encryption	0.35	**0.987**
All four obfuscations applied	0.28	**0.984**
Average	0.695	**0.987**

Table 7: Individual SVM performance for malware detection and obfuscation detection

SVM	Details	Acc.
$SVMnoDAN_{mal}$	Malware SVM using features from multi-view model without DAN.	0.984
$SVMDAN_{mal}$	Malware SVM using features from multi-view DAN model.	0.965
$SVMnoDAN_{obf}$	Obfuscation SVM using features from multi-view model without DAN.	0.823
$SVMDAN_{obf}$	Obfuscation SVM using features from multi-view DAN model.	**0.178**

6.6 Commercial anti-malware engine comparison

For a comparison with commercial anti-malware detectors, the authors of [26] kindly provided us with their publicly available PRAGuard dataset. This includes four versions of the Genome dataset, each obfuscated with one of four techniques, plus another version with all four obfuscation techniques applied in combination, totalling 6,253 samples. They carried out a study of commercial anti-malware engines from thirteen different vendors such as McAfee, AVG, Norton and Kaspersky, reporting the median detection rate of these engines for each type of obfuscation. The authors imply using their combined obfuscation techniques, implemented with a different professional tool [10] to ours, is effective against any anti-malware engine. Their four techniques are class encryption, string encryption, reflection, and trivial obfuscation. To evaluate the performance of our DAN architecture against their obfuscations and allow a comparison with the results from the commercial anti-malware detectors, we test our already trained model from Section 6.2 directly against all these PRAGuard samples. Table 6 shows the median detection rate of all the commercial engines against each obfuscation type, as reported in [26], plus the detection rate of DANdroid. It can be seen DANdroid achieves superior performance in every case. Detection by commercial engines suffers a major decrease as the obfuscation becomes stronger. For example, with class encryption the median detection rate is 0.35 compared our detection rate of 0.987. Comparing the averages, the commercial engines score only 0.695 compared to our average of 0.984. DANdroid produces a major improvement, especially considering the model was neither exposed to these held-out multiple-obfuscated PRAGuard samples, nor any sample obfuscated with this other professional tool [10] during its training phase. This performance gain is likely to be because our static analysis of a sample is much more in-depth compared to these commercial signature-based detection systems. The higher performance in Table 6 compared to Table 3 is due to a less difficult experimental setup with less samples and less malware families.

7 FURTHER ANALYSIS

7.1 Verifying obfuscation-resilience in learned malware features

If in theory a model learned obfuscation as being indicative of malware, it would classify obfuscated benign samples as malicious, producing poor malware detection performance. Results in Table 3 show this is not the case but nevertheless we now validate the learning process by seeking further proof that the learning of malware features has not been hindered by obfuscation.

To study the ability of the multi-view DAN model to ensure this silent learning of obfuscation is not taking place, we compare the usefulness of its feature representations with those from a multi-view model without DAN. This comparison involves measuring the accuracy of different SVMs trained with these features in the separate settings of detecting malware and detecting obfuscation.

Firstly, the respective feature vectors to be used need extracted from each model. If the features designed to detect malware are largely successful in the task of detecting obfuscation, it can be understood therefore that those features are not fully independent of obfuscation. For our multi-view DAN model in Section 6.3, we remove both classifiers and forward each test set sample through the remaining network, storing each resulting feature representation, i.e. the vector f in Figure 1, in a feature matrix. In such a matrix, each row represents one sample in the test set, and a row has the same length as $f = 96$.

Next, using the multi-view model without DAN from Section 6.2, that is, using C_{mal} as the only cost function, the process repeated to extract and store its resultant features for the test set samples in a second feature matrix. In both the multi-view DAN feature matrix and the multi-view without DAN feature matrix, the order of the rows, and hence the samples, is the same.

Lastly, using the malware and obfuscation labels for each sample, 90% of the multi-view DAN feature matrix is used to train two SVMs, one a malware detector and the other an obfuscation detector, whilst 90% of the multi-view without DAN feature matrix is used to train another two SVMs, again for malware detection and obfuscation detection respectively.

For the evaluation, we test each of these SVMs with the remaining 10% of the feature matrix it was created from. In this way, we can compare the accuracy of each SVM to assert how well the features extracted with and without DAN can perform in the two separate settings of malware detection and obfuscation detection. Table 7 shows both sets of features are very useful for malware detection, with $SVMnoDAN_{mal}$ having an accuracy of 0.984 and $SVMDAN_{mal}$ having an accuracy of 0.965. However performance for obfuscation detection is considerably different - the $SVMnoDAN_{obf}$ accuracy is 0.823, with the $SVMDAN_{obf}$ accuracy dropping to only 0.178. Given this major difference, and with an accuracy of 0.5 being chance, we conclude the learned malware features encapsulated in the multi-view DAN model are of little value for detecting obfuscation, thus the learning process is robust and exhibits obfuscation-resilience.

7.2 Correlation between learned features

The feature representations from both the multi-view DAN model and the multi-view model without DAN can be directly compared to further investigate how similar they are. With the two separate feature matrices extracted in the previous experiment, we calculate Pearson's correlation co-efficient between them. Recall that one matrix contains learned features from the multi-view DAN model, and the other matrix contains learned features from the multi-view model without DAN. Again, in both matrices, the order of the rows, and therefore the order of the samples, is the same. A score of 0 indicates absolutely no linear correlation and a score of 1 indicates total correlation. The resulting correlation is only 0.27, matching the conclusion of Section 7.1.

7.3 Multi-view model permutations and impact on detection performance

Here we analyse the change in performance caused by using the multi-view featuring learning that includes the extra input features of permissions and API calls, in addition to opcodes. We perform an ablation study using unobfuscated samples in order to analyse the contribution of each input feature set, with separate models being trained using each permutation of one, two, or three input feature sets, using only C_{mal} instead of the DAN. For training, the same unobfuscated training set is used as in Section 6.1. We choose not to use obfuscated samples in this ablation study to avoid optimising our architecture specifically for the obfuscated scenario.

Table 8 shows detection performance on this unobfuscated test set. While using any two feature sets from opcodes, permissions and API calls boosts performance compared to using them in isolation - for example, combining permissions and API calls to give an F-score of 0.978 - optimal performance is reached by combining all three feature sets, increasing the F-score to 0.991 from 0.976 when using just the single-view opcodes model. We can therefore deduce that this multi-view approach with all three input feature sets improves malware detection.

Table 8: Detection performance of each feature set permutation on unobfuscated samples

Feature Set	Acc.	Prec.	Recall	F-score
Opcodes	0.977	0.98	0.973	0.976
Permissions	0.933	0.93	0.935	0.932
API Calls	0.954	0.953	0.953	0.953
Opcodes + Permissions	0.981	0.985	0.976	0.981
Opcodes + API Calls	0.99	0.991	0.989	0.99
Permissions + API Calls	0.979	0.98	0.976	0.978
Opcodes + Permissions + API Calls	0.991	0.987	0.995	0.991

8 CONCLUSIONS

A novel deep learning approach that performs strongly in detecting obfuscated and unobfuscated Android malware is presented, which we name as DANdroid. Results compare favorably with the state-of-the-art. Using a multi-view DAN architecture we remove obfuscation bias from the learning process via negation of the obfuscation cost in a bespoke adversarial cost function. A balanced, augmented dataset of both obfuscated and unobfuscated malicious and benign apps, combined with a multi-view architecture learning from opcodes, permissions and API calls, provides more confidence in the discriminative power of learned features when compared to state-of-the-art methods that attempt to deal with obfuscation. Further analysis proves our model is resilient to the four selected obfuscation methods used in training, and we also demonstrate its potential to generalize over rare and future obfuscation methods not seen in training. The model is highly effective in detecting malicious samples obfuscated with multiple techniques, including those which thirteen commercial anti-malware engines struggled with.

REFERENCES

[1] Yousra Aafer, Wenliang Du, and Heng Yin. 2014. DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android. https://doi.org/10.1007/978-3-319-04283-1_6

[2] Mohammed K. Alzaylaee, Suleiman Y. Yerima, and Sakir Sezer. 2017. EMULATOR vs REAL PHONE: Android Malware Detection Using Machine Learning. In Proceedings of the 3rd ACM on International Workshop on Security And Privacy Analytics (IWSPA '17). ACM, New York, NY, USA, 65–72. https://doi.org/10.1145/3041008.3041010

[3] Apktool. 2019. Apktool - a tool for reverse engineering 3rd party, closed, binary Android aps. https://ibotpeaches.github.io/Apktool/. (2019).

[4] Axelle Apvrille and Tim Strazzerer. 2012. Reducing the Window of Opportunity for Android Malware Gotta catch 'em all. In Journal in Computer Virology Vol. 8. 61–71. https://doi.org//10.1007/s11416-012-0162-3

[5] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. 2014. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket.. In NDSS. The Internet Society. http://dblp.uni-trier.de/db/conf/ndss/ndss2014.html#ArpSHGR14

[6] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. 2011. Crowdroid: Behavior-based Malware Detection System for Android. In Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM '11). ACM, New York, NY, USA, 15–26. https://doi.org/10.1145/2046614.2046619

[7] Sen Chen, Minhui Xue, Zhushou Tang, Lihua Xu, and Haojin Zhu. 2016. StormDroid: A Streaminglized Machine Learning-Based System for Detecting Android Malware. In Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (ASIA CCS '16). ACM, New York, NY, USA, 377–388. https://doi.org/10.1145/2897845.2897860

[8] Wei Chen, David Aspinall, Andrew D. Gordon, Charles Sutton, and Igor Muttik. 2016. More Semantics More Robust: Improving Android Malware Classifiers. In Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec '16). ACM, New York, NY, USA, 147–158. https://doi.org/10.1145/2939918.2939931

[9] A. Demontis, M. Melis, B. Biggio, D. Maiorca, D. Arp, K. Rieck, I. Corona, G. Giacinto, and F. Roli. 2018. Yes, Machine Learning Can Be More Secure! A Case Study on Android Malware Detection. IEEE Transactions on Dependable and Secure Computing (2018), 1–1. https://doi.org/10.1109/TDSC.2017.2700270

[10] DexGuard. 2019. DexGuard. https://www.guardsquare.com/en/products/dexguard. (2019).

[11] DexProtector. 2019. DexProtector. https://dexprotector.com. (2019).

[12] M. Fan, J. Liu, X. Luo, K. Chen, Z. Tian, Q. Zheng, and T. Liu. 2018. Android Malware Familial Classification and Representative Sample Selection via Frequent Subgraph Analysis. IEEE Transactions on Information Forensics and Security 13, 8 (Aug 2018), 1890–1905. https://doi.org/10.1109/TIFS.2018.2806891

[13] Yaroslav Ganin, Evgeniya Ustinova, Hana Ajakan, Pascal Germain, Hugo Larochelle, François Laviolette, Mario Marchand, and Victor Lempitsky. 2016. Domain-adversarial Training of Neural Networks. J. Mach. Learn. Res. 17, 1 (Jan. 2016), 2096–2030. http://dl.acm.org/citation.cfm?id=2946645.2946704

[14] Joshua Garcia, Mahmoud Hammad, and Sam Malek. 2018. Lightweight, Obfuscation-Resilient Detection and Family Identification of Android Malware. ACM Trans. Softw. Eng. Methodol. 26, 3, Article 11 (Jan. 2018), 29 pages. https://doi.org/10.1145/3162625

[15] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. 2013. Structural Detection of Android Malware Using Embedded Call Graphs. In Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security (AISec '13). ACM, New York, NY, USA, 45–54. https://doi.org/10.1145/2517312.2517315

[16] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative Adversarial Nets. In Advances in Neural Information Processing Systems 27, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 2672–2680. http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf

[17] Ian J. Goodfellow. 2016. NIPS 2016 Tutorial: Generative Adversarial Networks. CoRR abs/1701.00160 (2016).

[18] Yoshua Bengio Ian Goodfellow and Aaron Courville. 2017. Deep Learning: Data Encoding (1 ed.). MIT Press, Cambridge, MA.

[19] Jesus Freke. 2019. Baksmali. https://github.com/JesusFreke/smali. (2019).

[20] ElMouatez Billah Karbab, Mourad Debbabi, Abdelouahid Derhab, and Djedjiga Mouheb. 2018. MalDozer: Automatic framework for android malware detection using deep learning. Digital Investigation 24 (2018), S48–S59.

[21] T. Kim, B. Kang, M. Rho, S. Sezer, and E. G. Im. 2019. A Multimodal Deep Learning Method for Android Malware Detection Using Various Features. IEEE Transactions on Information Forensics and Security 14, 3 (March 2019), 773–788. https://doi.org/10.1109/TIFS.2018.2866319

[22] A. H. Lashkari, A. F. A.Kadir, H. Gonzalez, K. F. Mbah, and A. A. Ghorbani. 2017. Towards a Network-Based Framework for Android Malware Detection and Characterization. In 2017 15th Annual Conference on Privacy, Security and Trust (PST). 233–23309. https://doi.org/10.1109/PST.2017.00035

[23] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor van der Veen, and Christian Platzer. 2014. ANDRUBIS - 1, 000, 000 Apps Later: A View on Current Android Malware Behaviors. In Third International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security, BADGERS@ESORICS 2014, Wroclaw, Poland, September 11, 2014. 3–17. https://doi.org/10.1109/BADGERS.2014.7

[24] Yang Liu, Zhaowen Wang, Hailin Jin, and Ian Wassell. 2018. Multi-Task Adversarial Network for Disentangled Feature Learning. In The IEEE Conference on Computer Vision and Pattern Recognition (CVPR).

[25] Aravind Machiry, Nilo Redini, Eric Gustafson, Yanick Fratantonio, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. 2018. Using Loops For Malware Classification Resilient to Feature-unaware Perturbations. In Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC '18). ACM, New York, NY, USA, 112–123. https://doi.org/10.1145/3274694.3274731

[26] Davide Maiorca, Davide Ariu, Igino Corona, Marco Aresu, and Giorgio Giacinto. 2015. Stealth Attacks: An Extended Insight into the Obfuscation Effects on Android Malware. Comput. Secur. 51, C (June 2015), 16–31. https://doi.org/10.1016/j.cose.2015.02.007

[27] Davide Maiorca, Francesco Mercaldo, Giorgio Giacinto, Corrado Aaron Visaggio, and Fabio Martinelli. 2017. R-PackDroid: API Package-based Characterization and Detection of Mobile Ransomware. In Proceedings of the Symposium on Applied Computing (SAC '17). ACM, New York, NY, USA, 1718–1723. https://doi.org/10.1145/3019612.3019793

[28] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon J. Ross, and Gianluca Stringhini. 2017. MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models. In 24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017.

[29] Fabio Martinelli, Fiammetta Marulli, and Francesco Mercaldo. 2017. Evaluating Convolutional Neural Network for Effective Mobile Malware Detection. Procedia Comput. Sci. 112, C (Sept. 2017), 2372–2381.

[30] Niall McLaughlin, Jesus Martinez del Rincon, BooJoong Kang, Suleiman Yerima, Paul Miller, Sakir Sezer, Yeganeh Safaei, Erik Trickel, Ziming Zhao, Adam Doupé, and Gail Joon Ahn. 2017. Deep Android Malware Detection. In Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy (CODASPY '17). ACM, New York, NY, USA, 301–308.

[31] Robin Nix and Jian Zhang. 2017. Classification of Android apps and malware using deep neural networks. In 2017 International Joint Conference on Neural Networks, IJCNN 2017, Anchorage, AK, USA, May 14-19, 2017. 1871–1878. https://doi.org/10.1109/IJCNN.2017.7966078

[32] L. Onwuzurike, M. Almeida, E. Mariconti, J. Blackburn, G. Stringhini, and E. De Cristofaro. 2018. A Family of Droids-Android Malware Detection via Behavioral Modeling: Static vs Dynamic Analysis. In 2018 16th Annual Conference on Privacy, Security and Trust (PST). 1–10. https://doi.org/10.1109/PST.2018.8514191

[33] V. Rastogi, Y. Chen, and X. Jiang. 2014. Catch Me If You Can: Evaluating Android Anti-Malware Against Transformation Attacks. IEEE Transactions on Information Forensics and Security 9, 1 (Jan 2014), 99–108. https://doi.org/10.1109/TIFS.2013.2290431

[34] Borja Sanz, Igor Santos, Carlos Laorden, Xabier Ugarte-Pedrero, Pablo García Bringas, and Gonzalo Álvarez Marañón. 2012. PUMA: Permission Usage to Detect Malware in Android.. In CISIS/ICEUTE/SOCO Special Sessions (Advances in Intelligent Systems and Computing), Álvaro Herrero, Václav Snásel, Ajith Abraham, Ivan Zelinka, Bruno Baruque, Héctor Quintián-Pardo, José Luś Calvo-Rolle, Javier Sedano, and Emilio Corchado (Eds.), Vol. 189. Springer, 289–298. http://dblp.uni-trier.de/db/conf/softcomp/soco2012s.html#SanzSLUBA12

[35] A. Saracino, D. Sgandurra, G. Dini, and F. Martinelli. 2018. MADAM: Effective and Efficient Behavior-based Android Malware Detection and Prevention. IEEE Transactions on Dependable and Secure Computing 15, 1 (Jan 2018), 83–97. https://doi.org/10.1109/TDSC.2016.2536605

[36] Michael Sikorski and Andrew Hong. 2012. Data Encoding. In Practical Malware Analysis. No Starch Press Inc, US, 281–285, 294.

[37] A. Skovoroda and D. Gamayunov. 2017. Automated Static Analysis and Classification of Android Malware using Permission and API Calls Models. In 2017 15th Annual Conference on Privacy, Security and Trust (PST). 243–24309. https://doi.org/10.1109/PST.2017.00036

[38] Guillermo Suarez-Tangil, Santanu Kumar Dash, Mansour Ahmadi, Johannes Kinder, Giorgio Giacinto, and Lorenzo Cavallaro. 2017. DroidSieve: Fast and Accurate Classification of Obfuscated Android Malware. In Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy (CODASPY '17). ACM, New York, NY, USA, 309–320. https://doi.org/10.1145/3029806.3029825

[39] M. Sun, X. Li, J. C. S. Lui, R. T. B. Ma, and Z. Liang. 2017. Monet: A User-Oriented Behavior-Based Malware Variants Detection System for Android. IEEE Transactions on Information Forensics and Security 12, 5 (May 2017), 1103–1112. https://doi.org/10.1109/TIFS.2016.2646641

[40] Sundar Pichai. 2017. Google I/0 2017 Keynote Speech. https://www.youtube.com/watch?v=Y2VF8tmLFHw. (2017).

[41] Christina Wadsworth, Francesca Vera, and Chris Piech. 2018. Achieving Fairness through Adversarial Learning: an Application to Recidivism Prediction. CoRR abs/1807.00199 (2018).

[42] S. Wang, Q. Yan, Z. Chen, B. Yang, C. Zhao, and M. Conti. 2018. Detecting Android Malware Leveraging Text Semantics of Network Flows. IEEE Transactions on Information Forensics and Security 13, 5 (May 2018), 1096–1109. https://doi.org/10.1109/TIFS.2017.2771228

[43] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. 2017. Deep Ground Truth Analysis of Current Android Malware. In DIMVA.

[44] Michelle Wong and David Lie. 2016. IntelliDroid: A targeted input generator for the dynamic analysis of Android malware. In Proceedings of the 2016 Symposium on Network and Distributed System Security (NDSS). (Acceptance: 60/389, 15%, 38 citations).

[45] Michelle Wong and David Lie. 2018. Tackling runtime-based obfuscation in Android with TIRO. In Proceedings of the 27th USENIX Security Symposium.

[46] K. Xu, Y. Li, R. H. Deng, and K. Chen. 2018. DeepRefiner: Multi-layer Android Malware Detection System Applying Deep Neural Networks. In 2018 IEEE European Symposium on Security and Privacy (EuroS P). 473–487. https://doi.org/10.1109/EuroSP.2018.00040

[47] Suleiman Y. Yerima, Sakir Sezer, and Igor Muttik. 2014. Android Malware Detection Using Parallel Machine Learning Classifiers. In Proceedings of the 2014 Eighth International Conference on Next Generation Mobile Apps, Services and Technologies (NGMAST '14). IEEE Computer Society, Washington, DC, USA, 37–42. http://dx.doi.org/10.1109/NGMAST.2014.23

[48] S. Y. Yerima, S. Sezer, and I. Muttik. 2015. High accuracy android malware detection using ensemble learning. IET Information Security 9, 6 (2015), 313–320. https://doi.org/10.1049/iet-ifs.2014.0099

[49] Z. Yuan, Y. Lu, and Y. Xue. 2016. Droiddetector: android malware characterization and detection using deep learning. Tsinghua Science and Technology 21, 1 (Feb 2016), 114–123. https://doi.org/10.1109/TST.2016.7399288

[50] J. Zhu, Z. Wu, Z. Guan, and Z. Chen. 2015. API Sequences Based Malware Detection for Android. In 2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom). 673–676.