**QUEEN'S
UNIVERSITY
BELFAST**

Multi-view Deep Learning for Zero-day Android Malware Detection

Millar, S., McLaughlin, N., Martinez-del-Rincon, J., & Miller, P. (2021). Multi-view Deep Learning for Zero-day Android Malware Detection. *Journal of Information Security and Applications*, 58. <https://doi.org/10.1016/j.jisa.2020.102718>

Published in:
Journal of Information Security and Applications

Document Version:
Peer reviewed version

Queen's University Belfast - Research Portal:
[Link to publication record in Queen's University Belfast Research Portal](#)

Publisher rights

Copyright 2021 Elsevier.

This manuscript is distributed under a Creative Commons Attribution-NonCommercial-NoDerivs License

(<https://creativecommons.org/licenses/by-nc-nd/4.0/>), which permits distribution and reproduction for non-commercial purposes, provided the author and source are cited.

General rights

Copyright for the publications made accessible via the Queen's University Belfast Research Portal is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy

The Research Portal is Queen's institutional repository that provides access to Queen's research output. Every effort has been made to ensure that content in the Research Portal does not infringe any person's rights, or applicable UK laws. If you discover content in the Research Portal that you believe breaches copyright or violates any law, please contact openaccess@qub.ac.uk.

Manuscript Details

Manuscript number	JISA_2020_558
Title	Multi-view Deep Learning for Zero-day Android Malware Detection
Article type	Full Length Article

Abstract

Zero-day malware samples pose a considerable danger to users as implicitly there are no documented defences for previously unseen, newly encountered behaviour. Malware detection therefore relies on past knowledge to attempt to deal with zero-days. Often such insight is provided by a human expert hand-crafting and pre-categorising certain features as malicious. However such tightly coupled feature-engineering, based on previous domain knowledge, risks not being effective when faced with a new threat. In this work we decouple this human expertise, instead encapsulating knowledge inside a deep learning neural net with no prior understanding of malicious characteristics. Raw input features consist of low-level opcodes, app permissions and proprietary Android API package usage. Our method makes three main contributions. Firstly, a novel multi-view deep learning Android malware detector with no specialist malware domain insight used to select, rank or handcraft input features. Secondly, a comprehensive zero-day scenario evaluation using the Drebin and AMD benchmarks, with our model achieving weighted average detection rates of 91% and 81% respectively, an improvement of up to 57% over the state-of-the-art. Thirdly, a 77% reduction in false positives on average compared to the state-of-the-art, with excellent F1 scores of 0.9928 and 0.9963 for the general detection task again on the Drebin and AMD benchmark datasets respectively.

Keywords	Android malware detection; zero-day; cybersecurity; deep learning; convolutional neural networks; multi-view learning.
Manuscript category	Network and mobile security
Corresponding Author	Stuart Millar
Corresponding Author's Institution	Centre for Secure Information Technologies (CSIT) - Queen's University Belfast
Order of Authors	Stuart Millar, Niall McLaughlin, Jesus Martinez del Rincon, Paul Miller
Suggested reviewers	Leandros Maglaras, Sotiris Moschoyiannis

Submission Files Included in this PDF

File Name [File Type]

cover_letter.pdf [Cover Letter]

MultiviewDeepLearningForZerodayAndroidMalwareDetection_JISA.pdf [Manuscript File]

declaration-of-competing-interests.docx [Conflict of Interest]

CReditT_author_statement.docx [Author Statement]

To view all the submission files, including those not included in the PDF, click on the manuscript title on your EVISE Homepage, then click 'Download zip file'.

Centre for Secure Information Technologies (CSIT)
Queen's University Belfast
Northern Ireland Science Park
Belfast
BT3 9DT

Friday 1st May 2020

Dear Prof. Anthony T.S. Ho, Editor-in-Chief

I am writing to confirm the submission of a manuscript titled “Multi-view Deep Learning for Zero-day Android Malware Detection” to be considered for publication in Elsevier Journal of Information Security and Applications.

The Android OS from Google is widely used with over 2.5 billion active Android devices, including smartphones, wearables and in-car implementations. This creates a plethora of attack opportunities for malware authors, and consequently a need for effective Android malware detection techniques. Zero-day malware is the most dangerous due to there being no established defence for the malicious activity in question.

This paper proposes a deep neural network for Android malware detection incorporating convolutional methods, with multiple views learning from three raw input feature sets of low-level opcodes, app permissions and proprietary Android API packages. We provide evidence that these feature sets in combination with our multi-view architecture outperform the state-of-the-art, both in a zero-day scenario - arguably the sternest test of a malware detector - and for the conventional detection case. Specifically, our work makes the following contributions:

- 1) A multi-view deep learning Android malware detector with a novel CNN-based approach for analysing API sequence calls. In contrast to previous work, this requires no specialist domain insight to rank or hand-craft these input features.
- 2) The use of proprietary Android API packages as raw input features, rather than generating API and method dictionaries of arbitrary size.
- 3) A comprehensive zero-day scenario evaluation, using the Drebin and AMD benchmark datasets, with our model achieving a weighted average detection rate of 91% and 81% respectively, up to 57% better than the state-of-the-art.
- 4) Excellent detection performance, with an F1 of 0.9928 and 0.9963 using the Drebin and AMD benchmark datasets respectively, and a 77% reduction in false positives on average compared to the state-of-the-art.

Based on the above, we believe that this manuscript aligns well with the research focus of JISA.

Thank you for your consideration.

Yours sincerely, on behalf of all authors

Stuart Millar

smillar09@qub.ac.uk

Multi-view Deep Learning for Zero-day Android Malware Detection

Stuart Millar, Niall McLaughlin, Jesus Martinez del Rincon, and Paul Miller

Abstract—Zero-day malware samples pose a considerable danger to users as implicitly there are no documented defences for previously unseen, newly encountered behaviour. Malware detection therefore relies on past knowledge to attempt to deal with zero-days. Often such insight is provided by a human expert hand-crafting and pre-categorising certain features as malicious. However such tightly coupled feature-engineering, based on previous domain knowledge, risks not being effective when faced with a new threat. In this work we decouple this human expertise, instead encapsulating knowledge inside a deep learning neural net with no prior understanding of malicious characteristics. Raw input features consist of low-level opcodes, app permissions and proprietary Android API package usage. Our method makes three main contributions. Firstly, a novel multi-view deep learning Android malware detector with no specialist malware domain insight used to select, rank or hand-craft input features. Secondly, a comprehensive zero-day scenario evaluation using the Drebin and AMD benchmarks, with our model achieving weighted average detection rates of 91% and 81% respectively, an improvement of up to 57% over the state-of-the-art. Thirdly, a 77% reduction in false positives on average compared to the state-of-the-art, with excellent F1 scores of 0.9928 and 0.9963 for the general detection task again on the Drebin and AMD benchmark datasets respectively.

Index Terms—Android malware detection, zero-day, cybersecurity, deep learning, convolutional neural networks, multi-view learning, neural networks.

I. INTRODUCTION

The Android OS from Google is widely used with over 2.5 billion active Android devices [1], including smartphones, wearables and in-car implementations. This creates a plethora of attack opportunities for malware authors, and consequently a need for effective Android malware detection techniques. Malicious apps that successfully compromise a device can cause major harm, for example with regards finances, exfiltration of sensitive data, personal information and identity fraud. Vehicles that use Android for automated driving or in-car systems [2] are also a target, with attacks potentially compromising driver and pedestrian safety. Given the vast amount of apps in the marketplace, it is not possible to ensure the legitimacy of each one by hand. Previous research adopted machine learning (ML) techniques in an effort to automate detection at scale. However, expert Android malware knowledge is regularly required to hand-craft input features to such detectors. This leaves them potentially more vulnerable to zero-day attacks, until such time as the new malware can firstly be detected and reverse-engineered, after which features for detection are updated. In this paper, we aim to decouple these domain insights from the input features and the learning process. As a result our method is attractive in terms of

real-world use and maintenance, particularly from a zero-day perspective. Zero-day malware is the most dangerous due to there being no established defence for the malicious activity in question, meaning it is only discovered at the time of the attack, hence the term 'zero-day' [3]. In the Android domain this translates into circumstances where a given malware family has not been encountered before.

We propose a deep neural network incorporating convolutional methods, with multiple views learning from three raw input feature sets of low-level opcodes, app permissions and proprietary Android API packages, all of which are extracted from each app directly. Our motivation for using convolutional methods comes from their success in Natural Language Processing (NLP). These methods usually involve vocabulary selection, building a dictionary of the most frequent words and subwords in training data, however such arbitrary thresholding decisions can create the possibility of missing out meaningful words. For example, in the benchmark datasets used in our study, whilst there are only a fixed number of unique opcodes in use, we find tens of thousands of unique APIs from a wide array of libraries. We ultimately avoid vocabulary selection by only considering Android APIs, which are limited in number to 210 [4], forming a small yet potentially salient set. Using only this proprietary information removes the need to contemplate wider, likely limitless, API usage. Otherwise, as new samples are introduced, arbitrary dictionary mappings need updated and datasets scanned again. Furthermore, bespoke dictionaries coupled to specific datasets present a risk in zero-day situations.

Using two benchmark datasets, the usefulness of each feature set is tested to show its effectiveness despite not requiring domain insight for feature ranking or selection. We provide evidence that these feature sets in combination with a multi-view architecture outperform the state-of-the-art, both in a zero-day scenario - arguably the sternest test of a malware detector - and for the conventional detection case.

Our contributions are:

- A multi-view deep learning Android malware detector with a novel CNN-based approach for analysing API sequence calls. In contrast to previous work, this requires no specialist domain insight to rank or hand-craft these input features.
- The use of proprietary Android API packages as raw input features, rather than generating API and method dictionaries of arbitrary size.
- A comprehensive zero-day scenario evaluation, using the Drebin and AMD benchmark datasets, with our model

achieving a weighted average detection rate of 91% and 81% respectively, up to 57% better than the state-of-the-art.

- Excellent detection performance, with an F1 of 0.9928 and 0.9963 using the Drebin and AMD benchmark datasets respectively, and a 77% reduction in false positives on average compared to the state-of-the-art.

This paper builds on our previous research in [5] and [6] by introducing multi-view network architecture, including permissions and proprietary Android APIs as input, and carrying out both zero-day and conventional detection evaluations using the Drebin and AMD benchmark datasets. Section II discusses related work, Section III details the feature extraction process, whilst our methodology and model architecture are in Section IV. Section V explains the experimental setup including benchmark datasets used, hyper-parameter tuning is in Section VI, detailed results are in Section VII, and Section VIII contains our conclusions.

II. RELATED WORK

A. Traditional ML Android malware detection techniques

Detecting Android malware uses techniques in two categories, either static analysis, with no code execution, or dynamic analysis, with an app executed in real-time and its behaviour studied. Traditional ML algorithms were adopted in early work, including SVMs [7], [8], [9], naive Bayes [10], kNN, k-means clustering [11], [12] and decision trees [10], [13], [14], [15], [16]. Such methods tend to have hand-crafted, manual rankings or selections of input features [10], [15], [17], [18]. These include pre-selected APIs [12], [8], [15], [19], [20], [21], permissions [19], [22], network traffic [23], [24], network addresses [7] and malicious system call traces [25]. The drawback to these approaches is that depending on expert knowledge for feature engineering makes a detector more vulnerable to change, rather than if such knowledge was encapsulated inside the detector itself. There is also a problem with the use of too small or unbalanced datasets, giving rise to some uncertainty regarding effectiveness in the wild. None of these methods have been thoroughly tested in arguably the toughest of scenarios, that is, a comprehensive zero-day situation.

B. Neural network Android malware detection techniques

More recent detection methods have begun to implement neural networks and deep learning (DL) architectures. DL-based systems benefit from a data-centric approach where features are learned from training examples rather than being manually derived by experts. This allows creation of models that are easier to train, update and maintain when new data or evidence is available. They are also able to extract discriminative features that are not obvious to human experts. DroidDetector [26] was an early work using a series of autoencoders in a deep belief network. [27] built a neural network that included an LSTM, though with 18 million learnable parameters in three hidden layers alone, a question remains over how well it might generalise given the amount

of training data was many orders of magnitude smaller, and no zero-data experiment was reported. [28] also presented a deep neural network to include multi-modal features such as opcodes or APIs, though seemingly offered no significant improvement over their baseline. [29] experimented with both an LSTM and a BERT model for the malware detection task, showing promising results, however the dataset used was a set of dynamic analysis traces produced by Palo Alto Networks using their Wildfire [30] tool. These are not easily reproducible given Wildfire is not open-sourced and the apps used are not publicly available.

Convolutional Neural Networks (CNNs), known for their initial success in computer vision [31], have also produced compelling performance in NLP tasks [32], [33], [34], [35]. In addition, CNNs have achieved strong results in intrusion detection [36], fake social media profile classification [37], Windows malware detection [38] and software code review [39]. However very few publications have used CNNs effectively in Android malware detection - of those that do, work in [5] used a CNN to learn opcode patterns, and [40] built a CNN to learn from system calls extracted via dynamic analysis. [41] pre-selected 245 sensitive APIs as input to their system but only reported results on part of a benchmark dataset. Evaluations in [42] showed strong detection performance though their CNN had several hundred thousand trainable parameters plus an arbitrary API dictionary with one hundred thousand entries. No existing approach has used more than one CNN in a multi-view setting, and again these neural network methods were not tested in a true zero-day scenario.

C. Use of APIs as features in Android malware detection

APIs are regularly used as input features for Android malware detection, however they are heavily hand-crafted using expert knowledge. For example, an app can be statically scanned for a select number of predefined sensitive or malicious APIs [43], either for their presence, or to measure their frequency of use. Alternatively this analysis can be dynamic, for example when building sequences of temporal API call invocations [44]. However considering a benchmark dataset can have tens of thousands of apps, this is time-consuming, with upwards of several minutes execution time per app. In addition, anti-emulation and anti-sandbox techniques are used by malware authors to keep malicious functionality dormant if instrumentation is detected [17], so these dynamic sequences may not be as useful as one would think.

Regarding static analysis of APIs, the Drebin detector [7] has a hand-engineered mapping of APIs in a total of eight feature sets, including suspicious API calls and restricted API calls. Many other state-of-the-art detectors used APIs in some form as features too. [13] built a group of 240 sensitive API calls that were then ranked to select a top 90. [19] created a list of hand-engineered protected API calls plus a list of unprotected API calls, totalling 384 API features. [45] adopted a weighted sensitive API call-based matching approach to detection, with a pool of 26,322 sensitive API calls. [28] used method invocation within APIs as a feature, parsing apps for

hand-chosen dangerous APIs, both for existence and frequency of use. [14] also counted various API invocations as features, with a package-level count of occurrences, grouped into purpose, in addition to a method-level count. [43] produced a listing of 56 malicious API calls, Android commands and Linux terminal commands, while [46] engineered sensitive Android-specific and Java-specific API features for detecting repackaged malware.

Of methods that adopt dynamic analysis of APIs, [44] focused on what APIs were used, their frequency and their dynamic sequence, forming one-hot vectors derived from control flow graphs. They considered all APIs both in the JDK and Android SDK, likely to run into the thousands. [47] engineered features from a behavioural characterisation study, using security-relevant APIs they described as sensitive. This involved hand-crafting 122 different features, with feature selection pruning this list to 70. [17] used API call signatures, part of a wider ranked list of 100 features, generated during dynamic instrumentation. However they admit anti-emulator techniques meant it was possible for some API calls not to be logged. [48] used a list of 62 targeted Android APIs, whilst [49] traced invocations of APIs in specified lists, using call frequencies in classification, and [50] extracted API calls as a one-hot encoding using a heterogeneous information network, again requiring significant expertise.

It can clearly be seen the use of APIs as features in previous work has required considerable feature-engineering and domain insight. We avoid this effort by using a CNN to learn from a limited set of only 210 proprietary Android API packages that have no expert pre-categorisation as sensitive or otherwise. Lastly, in contrast to the previously explained methods, we evaluate our approach in a zero-day scenario.

III. FEATURE EXTRACTION

From each app in our datasets the following raw input feature sets are extracted:

- 1) opcode instruction sequences, shown as useful in [5], [6].
- 2) Android permissions usage, previously identified as effective in [19], [22].
- 3) arbitrary API package usage.
- 4) proprietary Android API package [4] usage.

Note we are simply carrying out raw feature extraction, and not undertaking feature-selection or ranking used by other approaches to categorise input features. This extraction does not include any further statistical analysis to create a weighting or pre-categorisation. Hence domain insight is not required, with the relative importance of these features and their contribution to classification learnt and encapsulated within the model during the learning process. The desire to decouple specialist knowledge, lessening the involvement of the human analyst, is key in our approach. Each app's *.dex* file is first disassembled into *.smali* files using baksmali [51] and apktool [52]. The creation of each feature set is as follows:

1) *Opcodes*: Each method in a *.smali* file is made up of a set of opcode instructions, for example *move* or *goto*. We extract the opcode instructions and discard the operands. The opcodes from all the *.smali* files are concatenated to give a sequence representing one single app.

2) *Permissions*: An app contains an *AndroidManifest.xml* file that lists its requested Android permissions in order for the app to run correctly, for example INTERNET or INSTALL_PACKAGES. We extract permissions for each app by parsing this manifest, generating a multi-hot vector representing 138 Android permissions.

3) *Arbitrary API packages*: Two variations of arbitrary API usage, with feature sets we name as APIs and GoogleAPIs respectively, are used for comparison to the proprietary Android APIs:

- *APIs*: A single Java class in an app has a corresponding *.smali* file that contains the methods of that class. For each app in turn, all its *.smali* files are parsed line-by-line for API usage. Given a line of code, if the term 'invoke' is present, the API invoked is extracted and added to a dictionary as a key-value mapping. In such a mapping, the key is a unique API, and its value a unique integer ID.
- *GoogleAPIs*: This is similar to the creation of the APIs feature set, except only APIs invoked which start with 'com.google.' [53] are added to the dictionary.

These dictionary-based approaches are inspired by NLP methods, and bear similarities to others [42]. After an initial dictionary is created from a given dataset as per above, the total number of times each API was invoked can be calculated. By selecting the most frequently used APIs as either possible common malicious calls, or perhaps indicators of normality, plus the less frequent as potential anomalous indicators, a new dictionary of relevant APIs is created automatically. To ensure such a dictionary represents APIs used both often and seldom, but not anecdotally, a threshold is set for a minimum number of invocations that must be reached for an API to be included. After setting the threshold T , the $S/2$ most used and $S/2$ least used APIs are selected, generating a final dictionary of size S . Once the dictionary has been created, each app is re-scanned, and the corresponding API sequence to be input to the network is then adjusted, leaving out the APIs that are not in the thresholded dictionary. This is finally translated to a sequence of integers, where each value matches the dictionary integer ID of the API invoked at a given point in the sequence. In Section VI-B the dictionary size S and threshold T are varied see how this type of vocab selection impacts performance.

4) *Proprietary Android API packages*: Since these are at the heart of the Android platform [4], it is reasonable to expect they may hold a degree of predictive power. Given proprietary Android APIs are less likely to change over time compared to the wider selection that is available to app developers generally, we hypothesise this is of benefit in zero-day scenarios defending against future, unseen apps. This rationale, along with their lack of use in previous research, motivates us to experiment with them, and avoids both arbitrary decisions

Fig. 1: Android APIs sequence creation.

about which APIs to consider from other vendors, as well as library dependency issues. When extracting them, a dictionary data structure is first built using all 210 Android API packages, eliminating the need to define a dictionary size or a threshold. Each entry is a key-value mapping, where each key is a unique Android API, and its value an integer between 1 and 210. Android API usage is registered if in a given line of code the term 'invoke' is used and the API matches one in the Android API dictionary. This results in a sequence of integers for each app, where each value matches the dictionary integer ID of the Android API invoked at a given point in the sequence, as illustrated in Figure 1. This feature set is referred to as Android APIs throughout.

IV. METHODOLOGY

This paper proposes an innovative multi-view deep learning model for Android malware detection, both under zero-day conditions and in the general detection case, with malware domain insight decoupled. Our architecture consists of three views; one CNN learning from opcodes, a fully-connected neural net learning from permissions, and a second CNN learning from APIs. These feature sets are then combined together in a fully connected network that gives a final classification as to whether an Android app is malicious or benign. We adopt a multi-view approach since the representations for each feature set are inherently different due to their modalities, and allows analysis of each part of the architecture separately. This design decision also enables more refined tuning of the model, and performance comparisons between each view permutation.

A. Deep Learning Network Architecture

1) *Opcodes CNN*: Outside computer vision, many successful applications of convolutional architectures occur when the task can be modelled as a Natural Language Processing (NLP) problem [32], [33], [35]. Our work is connected to these methods however instead of using pre-trained embeddings, our dense embedding layer is trained from the data simultaneously with the rest of the network parameters, considering each opcode as a word or a token, and attempts to capture their semantics. The purpose of this CNN is to automatically learn salient patterns from each app's sequence of opcodes that discriminate between malicious and benign. In this way knowledge is encapsulated inside the architecture, rather than being provided by a human expert via feature engineering and ranking.

A raw input opcode sequence I_o for a single app is a series of one-hot vectors and can be written as $I_o = \{x_1 \dots x_n\}$, where x_n represents the n th opcode in a sequence of length

Fig. 2: Creating the embedding projection P_o .

n_o . Each vector x has $d_o = 218$ elements to represent the range of all opcodes. The element corresponding to the opcode used at position n in I_o is set to 1, and all other elements are set to 0. CNN filters learn feature representations containing relationships between nearby opcodes, which enables learning of sub-patterns. However, due to the use of one-hot vectors, I_o has a degree of sparsity, reducing the amount of useful information for a filter to learn from without that filter being inefficiently large from a computational standpoint. To mitigate this sparsity and improve learning, the one-hot vectors are projected into a dense, information rich k_o -dimensional embedding space P_o . This is shown in Figure 2. In doing so, each vector in I_o is multiplied by an embedding weight matrix W_{Eo} of size $d_o \times k_o$. The weights in W_{Eo} are randomly initialised at the start of the training procedure and updated during the learning process. The resulting embedding projection P_o is a matrix of size $n_o \times k_o$, processed by one convolutional layer with r filters. Each filter has size $s_o \times k_o$, with s_o being the number of consecutive opcodes analysed by each filter. Each of the r filters perform a 1D convolution over the full embedding matrix P_o generating an activation map a_{or} of size n_o as follows:

$$a_{or} = \text{ReLU}(\text{Conv}(P_o)_{W_r, b_r}) \quad (1)$$

where W_r and b_r are the trainable weight and bias parameters of the r th filter. The rectified linear activation function $\text{ReLU}(x) = \max\{0, x\}$ is used post-convolution. The activation maps produced from each filter are stacked to produce the activation matrix A_o of size $n_o \times r$, denoted as:

$$A_o = [a_{o1} | a_{o2} | \dots | a_{or}] \quad (2)$$

Global max-pooling subsamples A_o and produces a vector q_o of length r containing the maximum activation of each filter over the opcode sequence length n_o , written as:

$$q_o = [\max(a_{o1}) | \max(a_{o2}) | \dots | \max(a_{or})] \quad (3)$$

The sections of the opcode sequence that cause the maximum activation of each filter can be thought of those that model deems most important in classifying an app, regardless of the location of that section in the overall sequence. Max-pooling also ensures an input opcode sequence of arbitrary length is represented as a fixed-length vector ahead of being passed to the classification layer. q_o is fed through a fully-connected layer with o_o output neurons, producing the feature vector f_o :

$$f_o = \text{ReLU}(W_o q_o + b_o) \quad (4)$$

Fig. 3: Multi-view Deep Learning Network Architecture for Android malware detection.

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

2) *Permissions neural net*: The permissions contain useful information pertaining to malware detection that complements features derived from the opcodes CNN and the APIs CNN. For the permissions input, given an app and its manifest file, a multi-hot 1D array, or vector, of length 138 is generated, where each element maps to one of 138 Android permissions, with its value being 1 if a permission is 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 feature vector f_p , denoted as:

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

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

3) *APIs CNN*: This CNN is concerned with learning discriminative patterns from API input sequences. Its architecture is similar to that of the opcodes CNN. Again, these API sequences are only extracted, with no ranking or feature-engineering being undertaken, and we do not manually identify APIs, or patterns of APIs, that are indicative of malware or benign apps. Like the opcodes CNN, a raw input API sequence I_g for an app is a series of one-hot vectors, $I_g = \{x_1 \dots x_n\}$, representing a sequence of length n_g . In this case each x has length d_g elements to represent all potential APIs.

Similar to I_o , I_g also is a sparse matrix, and is projected into a k_g -dimensional embedding space P_g of size $n_g \times k_g$, processed by a single convolutional layer with m filters. Each filter has size $s_g \times k_g$, with s_g being the number of consecutive APIs analysed by each filter. Each of the m filters perform a 1D convolution operation over the full embedding matrix P_g generating an activation map a_{gm} of size n_g as follows:

$$a_{gm} = \text{ReLU}(\text{Conv}(P_g)W_m b_m) \quad (6)$$

where W_m and b_m are the trainable weight and bias parameters of the m th filter. Again ReLU is used post-convolution. The activation maps produced from each filter are stacked to produce the activation matrix A_g , of size $n_g \times m$, denoted as:

$$A_g = [a_{g1}|a_{g2}|\dots|a_{gm}] \quad (7)$$

Global max-pooling subsamples A_g and produces a vector q_g of length m containing the maximum activation of each filter over the opcode sequence length n_g , written as:

$$q_g = [\max(a_{g1})|\max(a_{g2})|\dots|\max(a_{gm})] \quad (8)$$

q_g is fed through a fully-connected layer with o_g output neurons, producing the feature vector f_g :

$$f_g = \text{ReLU}(W_g q_g + b_g) \quad (9)$$

where W_g and b_g are the weight and bias parameters of the hidden layer learned during training.

4) *Classification layer*: As per Figure 3, each 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:

$$f = [f_g|f_o|f_p] \quad (10)$$

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 $t=o_g+o_o+o_p$, a hidden layer h of size o_h neurons, and an output layer of size 2 neurons, since this malware classification problem is two-class, either malicious or benign. z can be denoted as:

$$z = \text{ReLU}(W_h f + b_h) \quad (11)$$

where W_h and b_h are the trainable weight and bias parameters of the hidden layer. This final output from the MLP, z , is a vector with each element giving a score that the app is associated with each class. z is passed to the SoftMax

classification layer, which outputs the normalized probability of the app belonging to each class in the problem, formally:

$$p(y = i|z) = \frac{\exp(w_i z + b_i)}{\sum_{i=1}^I \exp(w_i z + b_i)} \quad (12)$$

where w_i and b_i are the trainable weight and bias parameters of the SoftMax layer learned for each of the i classes, $w_i z$ is the inner product of w_i and z , with y being the predicted label.

B. Cost function

Our model's cost function C to be minimized for a batch of B training samples, $\{I_{(1)}, \dots, I_{(B)}\}$, can be written as:

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

where c is the number of classes in the task, $y'_{(j)}$ is the ground truth for sample $I_{(j)}$ and $z_{(j)}$ 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. A learning rate α is used in this process. 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.

V. EXPERIMENTAL SETUP

Our experiments are presented in four stages. The first covers hyper-parameter tuning for the opcodes CNN and the APIs CNN, with the second an analysis of features learned by the permissions network. The third evaluates the single and multi-view settings for malware detection to prove our model is effective in a conventional detection setting. The fourth is a series of zero-day experiments to recreate a challenging scenario where a malware detector is tested against a new family of malware it has never encountered before. To do this each family is excluded in turn from training, keeping it held out only for testing. State-of-the-art comparisons are provided throughout and the focus is on F1 score and false positive rate, i.e. the false alarm rate, as per convention in previous related literature [7], [5], [42], [9], [6].

A. Datasets

Four different datasets are used - 1) Malgenome [54], which contains 1,260 malware apps across 49 different families 2) an Intel Security dataset of 9,902 malware apps provided to us from the vendor's internal repository 3) Drebin [7], with 5,560 malware apps from 179 families, and 4) AMD [55], consisting of 24,553 malware apps from 71 families. The Malgenome dataset is only used for designing the initial architecture and hyperparameter tuning, not in the detection or zero-day experiments given its small size, and the Intel dataset is only used for investigating dictionary parameters since it is a proprietary set that does not allow comparison. For the evaluations and

state-of-the-art comparison, the Drebin and AMD benchmark datasets are selected. This avoids overfitting and artificially inflated performance since we do not use Drebin or AMD in the tuning phase. These datasets are publicly available, and are chosen due to their continual use in the Android malware detection domain, allowing direct comparisons between our approach and others, both in this work and in future. Every dataset is balanced with the addition of equivalent numbers of benign apps collected from Google Play, and then each dataset is split into 80% for training, 10% for validation, and 10% for testing, with the 50/50 balance of malware and benign apps maintained in each split. In Section VI, the Malgenome and Intel validation sets are used for tuning, and in Section VII the Drebin and AMD test set evaluation results are reported. For the avoidance of doubt, these test apps are kept totally unseen and are not used in training or validation at any stage.

VI. EXPERIMENTAL RESULTS: TUNING AND LEARNING ANALYSIS

In this section, a thorough set of ablation experiments were performed to define the design of our architecture as well as investigate the influence of each of the hyperparameters. All tested architectures used in both the opcodes CNN and the APIs CNN have only a single convolutional layer, as in [5]. Given the relatively small datasets available, larger numbers of layers are likely to be prone to overfitting and have a greater amount of trainable parameters with little performance improvement [34].

A. Opcodes CNN

These hyperparameters were empirically set using the Malgenome dataset. In summary, they are: $s_o = 8$, $r = 64$, $k_o = 8$ and $n_o = 8, 192$.

1) *Filter length s_o and number of filters r* : The filter length s_o was varied from $s_o = 2$ to $s_o = 32$, while the number of convolutional filters was held constant at $r = 64$ and the filter stride held at 1. Longer filter lengths let the network learn longer patterns of opcodes and detect long-range dependencies between them. Figure 4a shows an increase in F1 using longer filters, until around $s_o = 8$. This result may be unsurprising, as very short opcode sequences are presumably statistically common and hence less discriminative, while very long sequences are too specific to a single sample. There likely exists an in-between length where opcode sequences both discriminate and generalise well. We set $s_o = 8$, given the F1 improvement is marginal using filters twice or three times longer. In addition, since Dalvik is optimised for devices with constrained resources, it has a more powerful instruction set than conventional x86, meaning less use of the stack and registers. With over 1,000 x86 opcodes [56] compared to 218 opcodes in Dalvik [57], fewer instructions are likely to be needed by Dalvik for standard code constructs such as switch statements or for loops. These architectural differences motivate us further to select $s_o = 8$ over a unnecessarily longer filter length of 16 or 32.

(a) F1 on Malgenome dataset with varying s_o (filter length).

(b) F1 on Malgenome dataset with varying r (number of filters).

(c) F1 on Malgenome dataset with varying k_o (embedding dimension).

(d) F1 on Malgenome dataset with varying l (number of convolutional layers).

Fig. 4

The number of filters r affects the network's ability to learn a diverse set of features from the opcode sequences. This may be especially important if different malware families do not share many common characteristics, thus requiring different sequences of opcodes to be detected. All other network hyperparameters were held constant while the number of filters varied from $r = 2$ to $r = 128$ in powers of two. Note that regardless of the number of filters used, they were all of length $s_o = 8$. As per Figure 4b, the trend is for increasing F1 as the number of filters is increased. Approximately a 10% improvement in F1 is seen when moving from $r = 2$ to $r = 128$ filters. Performance plateaus at around 32 filters. Therefore, $r = 64$ filters were used to ensure that satisfactory performance while keeping the number of parameters small, thus reducing the chance of over fitting.

2) *Embedding dimension k_o* : The embedding layer takes a list of raw opcodes and learns to represent each opcode as a vector in an embedding space. The dimension of the embedding space k_o affects the network's internal representation of opcodes. In this experiment the dimension of the embedding space was varied from $k_o = 2$ to $k_o = 16$ in powers of two. Furthermore, to validate the use of the embedding layer, an experiment with no embedding layer was performed - note that in this case each opcode was simply represented using its corresponding index value, i.e. a real number in the range 1 to 218, and thus filters of size $s_o \times 1$ were used. From Figure 4c firstly it can be seen that use of the embedding layer is crucial to good performance. Once an embedding layer is used, there only a weak correlation between increasing the embedding dimension and increasing F1. This may be due to the fact that there are only a relatively small number of possible opcodes, with many opcodes representing similar underlying instructions. This contrasts with human language,

where high dimensional embeddings are needed to encode the rich semantic relationships between words [33]. The results of this experiment indicate that, while using an embedding layer produces much better F1 performance, the use of a high-dimensional embedding is not required. Therefore a small embedding can be used, and k_o is set to 8, which in turn reduces the number of parameters in the opcodes CNN, and hence the network may generalize better to new data.

3) *Number of convolutional layers*: In this experiment the number of convolutional layers, l , was varied from $l = 1$ to $l = 6$, while the number of convolutional filters and their length were held constant at $r = 64$ and $s_o = 8$ for all layers. Increasing the number of layers potentially allows the network to learn increasingly complex relationships between widely separated parts of the opcode sequence, by increasing the receptive field of the deeper filters.

However, the results in Figure 4d show that performance does not significantly improve when adding more layers. This suggests that, at least in the Malgenome dataset being used for tuning, features based on sequences of consecutive opcodes are sufficient for the task of malware detection. Higher level features, based on the combinations and ordering of sets of lower level features, appear not to be required. For static analysis this may be expected as the large-scale ordering of instructions, classes and other program constructs when coded originally is only loosely related to how those instructions are then executed at run-time after compilation. As a result of this experiment, we use only one layer in the opcodes CNN, since adding more layers will only increase the number of parameters, hence increasing the likelihood of over-fitting, while also reducing computational efficiency.

4) *Effect of max-pooling layer*: Using max-pooling after the convolutional layer serves two main purposes. Firstly, it

Fig. 5: F1 on Malgenome dataset with varying e (number of pooling segments).

allows any program, regardless of the length, to be represented as a fixed-length feature vector, which is a prerequisite to enable classification. Given that each Android app has a different number of instructions, this normalisation is required. Secondly, by only selecting the maximum activation of each convolution, the presence of a relevant sequence of opcodes can be detected anywhere in the program regardless of its location. However, compressing the full feature map, which gives magnitude of activation and location of each filter response, down to just the maximum activation value from each filter may discard too much potentially relevant information, such as the repetition of the relevant opcode sequence over the program length. To validate the max-pooling approach and explore alternatives that may retain additional information, we compare the performance of max-pooling over the full length, also known as global max-pooling, to max-pooling applied over regular program segments. This alternative max-pooling approach aims to preserve more information as well as enable the network to detect repetitions of filter activations.

For this experiment, the full program length is divided into a given number of segments e , where $e = 1$ is equivalent to the default of max-pooling over all instructions, and $e = n$ corresponds to dividing the full program length into n equal sized blocks, before applying max-pooling to each block. The length of the feature vector passed to the classifier is thus $e \times m$. Note each app has a different number of opcodes, and each program is divided into the same number of segments, therefore the segment length will vary depending on each individual program length. Figure 5 indicates changing the number of segments, e , does not significantly affect performance. Hence these results justify our architectural choice of applying the max-pooling over the full program length.

5) *Input length n_o* : We hypothesize that an accurate classification decision may not require the entire sequence of opcodes from an app. Instead it may be possible to detect malware by examining only a short sample of contiguous instructions from the complete program. This will not only improve computational efficiency, but could also be useful in the future design of a cascade classifier [58] where certain programs likely to be benign are quickly excluded from further consideration, while programs that appear to be malicious are examined more thoroughly. To study how performance is affected by limiting the number of opcodes n_o seen by the network, the maximum length of the opcode sequence taken

Fig. 6: F1 on Malgenome dataset with varying n_o (opcode input length).

from each program is varied from $n_o = 256$ to $n_o = 524,288$, increasing two-fold for each step. Where the number of opcodes in the full program is less than the maximum length, the full program is used. Figure 6 shows using longer lengths of opcode instructions substantially improves performance. However it can also be seen even very short opcode sequences can be used to detect malware effectively, which may be useful in resource constrained environments, such as smartphones or in-vehicle systems running Android OS. We find $n_o = 8,192$ is an optimal balance of performance and computation, as much higher orders of magnitude do not produce a large enough corresponding improvement.

B. APIs CNN

Similar ablation studies were performed for the APIs CNN. As with the opcodes CNN, the model hyperparameters were empirically set using the Malgenome dataset. In summary they are $s_g = 8$, $m = 64$ and $k_g = 8$. The Intel dataset was then used to empirically set the dictionary hyperparameters of $S = 250$, $T = 1$ and $n_g = 10,000$.

In the first of these experiments, APIs sequences created from an unthresholded arbitrary dictionary are used, before then introducing S and T . It is interesting to note some hyperparameters are the same as for the opcodes CNN, though we stress both were tuned independently. Perhaps this is to be expected on reflection, given the opcodes sequences and API sequences used in these ablation studies are derived from the same *.smali* files. For this reason a single convolutional layer and global max-pooling were again used, as both were already varied when tuning the opcodes CNN and thus these experiments were not repeated for the APIs CNN.

1) *Filter length s_g and number of filters m* : Like the opcodes CNN, increasing the length of the convolutional filters allows the network to learn longer patterns of APIs and learn long-range dependencies between APIs, while the number of filters impacts the diversity of features learnt from the API sequences. The Malgenome dataset is used in this experiment and a grid search performed. The range of hyperparameters in the grid search are: $s_g = [4, 8, 16, 32]$ and $m = [4, 8, 16, 32, 64]$. The filter stride is held at 1. Figure

Fig. 7: F1 on Malgenome dataset with varying s_g (filter length) and m (number of filters).

Fig. 8: Avg. F1 on Malgenome dataset across all T (invocation threshold) with varying k_g (embedding dimension). Bars show σ over k_g .

7, with $k_g = 4$ and $n_g = 50$, shows a noticeable F1 increase when s_g increases from 4 to 8, and when $m = 64$. Hence $s_g = 8$ and $m = 64$.

2) *Embedding dimension k_g* : The embedding layer learns to represent each API as a vector in an embedding space and the dimension of this space affects the model’s internal representation of APIs. Another grid search is performed using $k_g = [2, 4, 8]$ and $T = [1, 10, 50, 100, 250, 500]$. The dictionary size S is held at 250 and for each dictionary threshold T the APIs sequences are adjusted to remove APIs not present in each resulting dictionary. The average length of a sequence across both malware and benign samples in the Malgenome dataset was around 6,500 APIs. Therefore the input length n_g was set to 10,000, including zero padding where needed. As per Figure 8, when the F1 is averaged across all values of T , with the variability over k_g represented as a standard deviation bar around the mean, $k_g = 8$ performs best overall. Thus with a high-dimensional embedding not required, k_g is set to 8.

3) *Dictionary size S , invocation threshold T and input length n_g* : In order to better investigate the influence of these hyperparameters, the Intel dataset is used as it is much larger than Malgenome. Here the grid search has the following hyper-parameters: $S = [125, 250, 500, 750, 1000]$ and $T = [1, 10, 50, 100, 250, 500, 1000, 1500]$. As per Figure 9, again with the variability over T represented as a standard deviation bar around the mean, there is no clear choice for T since the impact of varying it is minimal. Therefore we select $S = 250$

Fig. 9: Avg. F1 on Intel dataset across all S (dictionary size) with varying T (invocation threshold). Bars show σ over T .

Fig. 10: F1 on Intel dataset with varying n_g (API input length).

as this is the closest size to the opcodes dictionary of $d_o = 218$ and saves computation compared to using a larger S , plus we set an arbitrary $T = 1$. S and T were then held constant, with n_g varied to study the effect of longer API input lengths. For the malicious samples in the Intel dataset, the average length of an API sequence varied from 8,500 to 26,500 APIs, and for the benign samples the average length varied from 7,000 to 20,200 APIs. The average length across both malware and benign samples varied from 8,200 to 23,300 APIs. In this grid search $n_g = [10000, 15000, 20000]$. However Figure 10 shows the improvement for API input lengths larger than $n_g = 10,000$ is minimal, and incurs extra computational cost. Since this real-world dataset is provided to us by Intel, we assert these dictionary and input length hyper-parameters are sufficient for our experiments.

C. Analysis of features learned by the permission view

For the permissions network, no hyperparameters need to be tuned, since all 138 specified permissions are used as input. However, an analysis is performed to both validate that our architecture has learned something meaningful, and to explore how the network’s internal knowledge could be used to inform the Android malware domain in general. Note doing the same for the opcodes CNN or Android APIs CNN is difficult as the input information is at a very low level - while reverting to the sequences of opcodes or APIs that max-activate each filter is possible, linking back opcodes and API usage directly to give meaningful domain insights is highly challenging, so we leave this CNN analysis as future work. By comparison, the permission view weights can be more easily associated to the importance of each permission for detecting malware or benign samples, and we have a selection of previous Android malware detection literature to cross-reference our insights with.

Fig. 11: Android permissions ranked by relevance for malware classification.

The importance of each permission for discriminating between malware or benign samples is ranked, giving insight into what our model has learned. This also acts as a sanity check for the approach, as the important permissions can be checked against the literature. If we can see the neural network using features in a similar way to an expert human malware analyst, this builds confidence in the decisions made by the model. To create this ranking, we average the weight associated with each input neuron of the permission view. Since the permissions are input as an ordered vector, each input neuron corresponds directly to one single permission. By averaging the weights in matrix W_p connected to each input neuron, the importance of each permission can be found. Stronger connections, i.e. larger weight values, mean the network ascribes more importance to a given permission. Figure 11 lists all permissions ordered by their average weight. To prove the permissions view is learning meaningful features from the data, the permissions deemed to be important by the model are compared with those from the previously published literature. The permissions SEND_SMS and RECEIVE_SMS, ranked first and third most important by the model, have been previously found to have high mutual-information scores for malware detection [43], [59]. These two permissions were also ranked in the top-ten most important permissions in several other analyses [22], [60], [61], [62], [63], [64], [65]. Similarly, it is noted that permissions with highly negative importance values according to our model were also found to be important in the literature: READ_PHONE_STATE and ACCESS_NETWORK_STATE [43], [59], [54], [66], CALL_PHONE [60], [66], [43], [59] and INTERNET [22], [54], [67], [66], [68], [65], [43], [59]. Finally, SYSTEM_ALERT_WINDOW, which was given the second largest value by our model, has been recently linked to ‘Cloak and Dagger’ attacks [69].

VII. EXPERIMENTAL RESULTS: FINAL MODEL AND STATE-OF-THE-ART COMPARISONS

In this section, a thorough evaluation of our proposed multi-view approach is performed using the two standard publicly available benchmark datasets Drebin [7] and AMD [55]. A comparison with the state-of-the-art is presented and a zero-day setting used to evaluate our network under challenging realistic conditions. Hyperparameter values used in all experiments are as set in Section VI, that is $n_o = 8,192$, $s_o = 8$, $k_o = 8$, $m = 64$, $o_o = 16$, $n_g = 10,000$, $s_g = 8$, $k_g = 8$, $r = 64$, $o_g = 16$, $S = 250$, $T = 1$, $o_p = 64$, and $o_h = 16$, plus $B = 1$, $\alpha = 0.001$ and a filter stride in both CNNs of 1. No further tuning is carried out, and each model is trained for 75 epochs. The metrics reported are on the unseen Drebin and AMD testing sets, as outlined in Section V-A.

A. Single-view model detection performance

We start by evaluating the performance of a series of single-view models learning from only one of the input feature sets. Similarly, the three presented API feature extraction methods are compared. This creates a baseline to compare our multi-view models to, and allows us to confirm to what degree each of these feature sets are of use in isolation. For Drebin and AMD separately, each single-view model is trained using the training set and then tested with the unseen testing set. Results in Table I and Figure 12 show each of the feature sets are indeed effective for malware detection, with Figure 13 showing the FP% rate comparison. For the Drebin apps, while performance is strong across the board, it can be seen that the APIs methods which use GoogleAPIs and AndroidAPIs give the best F1 scores and FP% rates. For the AMD apps, the GoogleAPIs and AndroidAPIs again perform best, with particularly low FP% rates. We conclude each single-view model performs well on both the Drebin and AMD datasets.

TABLE I: Detection performance and FP % rate of single-view models

Dataset	Model	F1	FP %
Drebin	Opcodes	0.9559	4.98
Drebin	Perms	0.9217	5.62
Drebin	APIs	0.9456	6.68
Drebin	GoogleAPIs	0.9688	4.57
Drebin	AndroidAPIs	0.9679	4.39
AMD	Opcodes	0.9745	1.03
AMD	Perms	0.9514	3.38
AMD	APIs	0.9653	2.58
AMD	GoogleAPIs	0.9928	0.49
AMD	AndroidAPIs	0.9922	0.76

Fig. 12: single-view F1 comparison.

Interestingly these results show the discriminative power of the AndroidAPIs, despite using the limited proprietary set of only 210 APIs, much smaller than the other API methods, resulting in similar or better detection performance with lower computational cost. It is worth recalling too that the dictionaries used in APIs and GoogleAPIs need some arbitrary and/or empirical decision-making. The benefit of AndroidAPIs is that these decisions are avoided, as there only are 210 APIs to start with.

B. Multi-view model detection performance

Next we combine our single-view features in a multi-view model to assess how performance is impacted versus using only one feature set in isolation. Having established the input feature sets are of use individually in single-view, it is expected there may be some difference in performance if more than one is used, which would show the feature sets are complementary. Thus in this experiment our feature sets are combined to see if detection can be improved and the FP rate reduced.

Results in Table II, Figure 14 and Figure 15 show that for both the Drebin and AMD datasets, combining feature

Fig. 13: single-view FP% rate comparison.

TABLE II: Detection performance and FP% rate of multi-view models

Dataset	Model	F1	FP %
Drebin	Opcodes + Perms	0.9759	3.51
Drebin	Opcodes + Perms + APIs	0.9837	1.23
Drebin	Opcodes + Perms + GoogleAPIs	0.9838	1.93
Drebin	Opcodes + Perms + AndroidAPIs	0.9928	1.05
AMD	Opcodes + Perms	0.989	0.53
AMD	Opcodes + Perms + APIs	0.9951	0.53
AMD	Opcodes + Perms + GoogleAPIs	0.9971	0.27
AMD	Opcodes + Perms + AndroidAPIs	0.9963	0.08

Fig. 14: multi-view F1 comparison.

sets always results in a performance improvement, since the network learns to use them in a complementary way during training. Among the three API feature extraction methods, AndroidAPIs results in the most successful combination with a significant decrease in false positives, i.e. false alarms, which is a critical parameter for real-world redeployment. Again, AndroidAPIs also does not involve arbitrary decisions regarding API selection as it is a limited, pre-defined set of 210, reducing the number of hyperparameters to use and simplifying updates to the model, since the dictionary does not need to be revised or recalculated for new samples or datasets.

C. State-of-the-art comparison: Malware detection

Table III shows a comparison of our approach against state-of-the-art methods using the Drebin and AMD benchmark datasets. Direct comparisons were performed using the Drebin detector [7], Maldozer [42] and [10] (referred to as APIs56). Drebin was implemented using available code and the other two with our own implementation, since the code was unavailable. It is ensured the same experimental setting and dataset splits used in our experiments are also used to evaluate these other methods. Note the Maldozer paper does

Fig. 15: multi-view FP% rate comparison.

TABLE III: State-of-the-art comparison in general detection setting

Dataset	Model	F1	Acc.	Recall	FP %	Comment
Drebin	Martinelli et al. [40]	-	0.78	-	-	Hand-crafted features. Only used 10 families.
Drebin	DroidAPIMiner [15]	0.32	-	-	-	Hand-crafted features. As reported in [9].
Drebin	MaMaDroid [9]	0.96	-	0.97	-	Hand-crafted features.
Drebin	Drebin detector [7]	0.9868	0.9866	0.9736	2.64	Hand-crafted features e.g. <i>suspiciousAPIs</i> .
Drebin	Maldozer [42]	0.9875	0.9875	0.9964	2.11	Word2Vec embedding. 244,098 params.
Drebin	APIs56 [10]	0.9378	0.9385	0.9403	6.33	Hand-crafted features e.g. API calls related.
Drebin	Our AndroidAPIs	0.9679	0.9679	0.9801	4.39	No hand-crafted features, 6,922 params.
Drebin	Our Multi-View	0.9928	0.9929	0.9964	1.05	No hand-crafted features. 23,226 params.
AMD	Ma et al. [44]	-	-	0.95	6.2	Hand-crafted features. API usage, decision tree.
AMD	Ma et al. [44]	-	-	0.97	9.1	Hand-crafted features. API frequency, neural net.
AMD	Ma et al. [44]	-	-	0.99	2.9	Hand-crafted features. API sequence, LSTM.
AMD	Drebin detector [7]	0.9975	0.9928	0.988	1.2	Hand-crafted features e.g. <i>suspiciousAPIs</i> .
AMD	Maldozer [42]	0.9936	0.9939	0.9914	0.38	Word2Vec embedding. 244,098 params.
AMD	APIs56 [10]	0.7458	0.8005	0.6104	2.43	Hand-crafted features e.g. API calls related.
AMD	Our AndroidAPIs	0.9922	0.9925	0.9926	0.76	No hand-crafted features. 6,922 params.
AMD	Our Multi-View	0.9963	0.9964	0.9934	0.08	No hand-crafted features. 23,226 params.
Average	Drebin detector [7]	0.9922	0.9897	0.9808	1.92	Average of Drebin dataset & AMD dataset results.
Average	Maldozer [42]	0.9906	0.9907	0.9939	1.25	Average of Drebin dataset & AMD dataset results.
Average	APIs56 [10]	0.8418	0.8695	0.7754	4.38	Average of Drebin dataset & AMD dataset results.
Average	Our AndroidAPIs	0.9801	0.9802	0.9864	2.58	Average of Drebin dataset & AMD dataset results.
Average	Our Multi-View	0.9946	0.9947	0.9949	0.57	Average of Drebin dataset & AMD dataset results.

not disclose its model’s input sequence length or batch size, and not enough detail is present to recreate their API dictionary of 100,000 APIs with embedding size 64. Thus we replicated the Maldozer network architecture for detection, including the use of word2vec [33] when training the embedding layer, with the input features being our AndroidAPIs. In all other cases, comparisons are made versus the reported results in the respective papers. On the Drebin dataset, our multi-view model achieves the highest F1 and the lowest FP rate of only 1.05%. This is 60% less than the Drebin FP rate, 50% less than the Maldozer FP rate, and 83% less than APIs56. The MaMaDroid [9] and DroidAPIMiner [15] F1 scores are 0.96 and 0.32 respectively, with [40] using only 10 of the Drebin families in evaluating their model, giving an accuracy of 0.78. Our performance gains are due to the multi-view model learning better from the three input feature sets, and the fact there are no hand-crafted features, allowing the model to encapsulate internally its own representation of malware and benign characteristics without expert human analysis beforehand. With the AMD dataset, our multi-view model again achieves a near-perfect F1 whilst giving a FP rate of only 0.08%. This is an 80% reduction compared to the best performing other model in that regard, which is Maldozer, and compared to the neural net in [44], this is a 99% reduction. Our FP rate is 72% lower than the best FP rate using the LSTM reported in [44]. Also, our multi-view model is likely to train faster than LSTMs, since recurrent architectures are known to be comparatively slow to train [70]. Notably, the F1 of APIs56 on AMD samples is lower than its F1 on the Drebin samples, and indeed much lower than our model. Given the Drebin dataset consists of samples discovered from 2010 to 2012, with the AMD dataset consisting of samples discovered from 2010 to 2016, it could be said APIs56 is a possible example of a detection system that, since it uses manually hand-crafted

features from a snapshot in time, is vulnerable to aging and becoming out-of-date. Lastly, considering the overall average figures it can be seen our multi-view model achieves the highest F1 and the lowest FP rate which as explained is much improved compared to the other methods.

In addition to our compelling quantitative performance, there are further qualitative comparisons to be made regarding the differences in the amount of human domain expertise used in creating the input feature sets. Our use of opcodes, permissions and AndroidAPIs implicitly means malware domain expertise has been decoupled, since these three input feature sets are only extracted in their raw form, not statistically ranked, refined or pre-categorised by an expert. The AndroidAPIs feature set is very likely to perpetuate as they are intrinsically linked to the core of the Android platform - they are proprietary and are much less likely to change. By comparison, from the Drebin detector codebase there appears to be handcrafting of input features to create eight different feature sets: *suspicious API calls*, *restricted API calls*, *hardware features*, *requested permissions*, *app components*, *filtered intents*, *used permissions*, and *network addresses*. The Drebin paper appears to state ML avoids the manual construction of detection rules for their extracted features, though detection-orientated logic was probably still applied, for example to populate the *suspiciousAPIs* list. On the other hand, our approach does not require any malware detection insights.

D. Zero-day scenario evaluation

The original Drebin paper [7] included a zero-day experiment where the top 20 largest families in the Drebin benchmark dataset were selected, and then each one was in turn left out of the training process. That same unseen held-out family was used in testing. Here, using both the Drebin and AMD datasets, we recreate this experiment with

TABLE IV: State-of-the-art comparison in zero-day scenario: detection performance on Drebin dataset

Average	Drebin detector [7]	Maldozer [42]	Our Multi-View
Largest 20 families	52%	90%	90%
All families (Weighted)	58%	86%	91%

TABLE V: State-of-the-art comparison in zero-day scenario: detection performance on AMD dataset

Average	Drebin detector [7]	Maldozer [42]	Our Multi-View
Largest 20 families	85%	81%	81%
All families (Weighted)	75%	75%	81%

Fig. 16: Num. of samples per family in the Drebin dataset.

Fig. 17: Num. of samples per family in the AMD dataset.

the multi-view model architecture using opcodes, permissions and Android APIs from Section VII-B. For the state-of-the-art comparison we use the Drebin detector and Maldozer. We extend the original zero-day experiment to evaluate not only the top 20 largest families as proposed in [7], but in fact all 178 families in the Drebin dataset, and all 71 families in the AMD dataset. The detection rate is used as the metric, calculated as the % of malicious samples tested that are correctly classified as malware. A weighted average is used when reporting results across all families since the number of samples in each family varies widely, and it is very small for a large number of families, as per Figures 16 and 17. This could considerably skew the results and is likely to be the reason why [7] reported results on solely the 20 largest families. The detection rate for each family is therefore weighted by the number of available samples for that family in relation to the total number of samples in the dataset. The normal average detection rate is used when using only the 20 largest families to allow comparison with previously reported results.

E. State-of-the-art comparison: Zero-day scenario

1) *Drebin dataset zero-day evaluation:* Table IV shows the results for the Drebin dataset, with detection performance calculated across the 20 largest families and also calculated across all 178 families. For the largest 20 families, our multi-view model using opcodes, permissions and Android APIs has a weighted detection average of 90%, a significant improvement on Drebin and Maldozer. Furthermore, when this experiment is expanded across the whole Drebin dataset of 178 families, strong performance is maintained and our model performs best when considering the variation in family size by using the weighted average.

2) *AMD dataset zero-day evaluation:* Table V contains the detection performance for the AMD dataset top 20 largest families plus the weighted average across the full 71 families. It can be seen for our multi-view model has a detection rate of 81% for the largest 20 families, better than the Drebin detector and Maldozer. When considering all 71 families in the AMD dataset, this performance gain is maintained, with our model scoring 81% compared to the Drebin and Maldozer scores, both of which are only 75%.

VIII. CONCLUSIONS

A novel multi-view CNN-based neural network is presented for static Android malware analysis learning from opcodes, permissions and proprietary Android APIs. It achieves state-of-the-art detection performance in a challenging zero-day scenario compared to the Drebin and Maldozer detectors. Our method reduces false positives by 77% on average compared to other methods, an important metric for potential real-world deployment. This network also achieves state-of-the-art performance in the conventional task of malware detection whilst decoupling domain insights. As future work we intend to carry out analysis of the activations in both CNNs, similar to the study presented of the permissions view.

REFERENCES

- [1] Stephanie Cuthbertson - Director, Android - Google, "Google I/O 2019 Keynote Speech," <https://www.youtube.com/watch?v=lyRPyRKHO8M>, [Online: last accessed April 2020].
- [2] Volvo Cars, "Android Auto," <https://www.volvocars.com/en-bh/support/manuals/xc40/2020/sound-media-and-internet/available-in-car-apps/android-auto>, [Online: last accessed April 2020].
- [3] K. Radhakrishnan, R. R. Menon, and H. V. Nath, "A survey of zero-day malware attacks and its detection methodology," in *TENCON 2019 - 2019 IEEE Region 10 Conference (TENCON)*, 2019, pp. 533–539.

- [4] Google, "Android API Developer Reference," <https://developer.android.com/reference/packages>, [Online: last accessed April 2020].
- [5] N. McLaughlin, J. Martinez del Rincon, B. Kang, S. Yerima, P. Miller, S. Sezer, Y. Safaei, E. Trickle, Z. Zhao, A. Doupe, and G. Joon Ahn, "Deep android malware detection," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, ser. CODASPY '17, 2017.
- [6] S. Millar, N. McLaughlin, J. Martinez del Rincon, P. Miller, and Z. Zhao, "Dandroid: A multi-view discriminative adversarial network for obfuscated android malware detection," in *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '20, 2020.
- [7] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "Drebin: Effective and explainable detection of android malware in your pocket," in *24th Annual Network and Distributed System Security Symposium, NDSS 2014*.
- [8] J. Zhu, Z. Wu, Z. Guan, and Z. Chen, "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)*, Aug 2015, pp. 673–676.
- [9] E. Mariconti, L. Onwuzurike, P. Andriotis, E. D. Cristofaro, G. J. Ross, and G. Stringhini, "Mamadroid: Detecting android malware by building markov chains of behavioral models," in *24th Annual Network and Distributed System Security Symposium, NDSS 2017*.
- [10] S. Y. Yerima, S. Sezer, and I. Muttik, "Android malware detection using parallel machine learning classifiers," in *Proceedings of the 2014 Eighth International Conference on Next Generation Mobile Apps, Services and Technologies*, ser. NGMAST '14, 2014, pp. 37–42.
- [11] A. Saracino, D. Sgandurra, G. Dini, and F. Martinelli, "Madam: Effective and efficient behavior-based android malware detection and prevention," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 1, pp. 83–97, 2018.
- [12] M. Sun, X. Li, J. C. S. Lui, R. T. B. Ma, and Z. Liang, "Monet: A user-oriented behavior-based malware variants detection system for android," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 5, pp. 1103–1112, May 2017.
- [13] S. Chen, M. Xue, Z. Tang, L. Xu, and H. Zhu, "Stormdroid: A streaming machine learning-based system for detecting android malware," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '16, 2016, pp. 377–388.
- [14] J. Garcia, M. Hammad, and S. Malek, "Lightweight, obfuscation-resilient detection and family identification of android malware," *ACM Trans. Softw. Eng. Methodol.*, vol. 26, no. 3, pp. 11:1–11:29, Jan. 2018.
- [15] Y. Aafer, W. Du, and H. Yin, "Droidapiminer: Mining api-level features for robust malware detection in android," in *Security and Privacy in Communication Networks. 9th International ICST Conference, SecureComm 2013*.
- [16] W. Chen, D. Aspinall, A. D. Gordon, C. Sutton, and I. Muttik, "More semantics more robust: Improving android malware classifiers," in *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, ser. WiSec '16, 2016, pp. 147–158.
- [17] M. K. Alzaylaee, S. Y. Yerima, and S. Sezer, "Emulator vs real phone: Android malware detection using machine learning," in *Proceedings of the 3rd ACM on International Workshop on Security And Privacy Analytics*, ser. IWSPA '17, 2017, pp. 65–72.
- [18] P. Faruki, V. Laxmi, A. Bharmal, M. Gaur, and V. Ganmoor, "Androsimilar: Robust signature for detecting variants of android malware," *Journal of Information Security and Applications*, vol. 22, pp. 66 – 80, 2015, special issue on Security of Information and Networks.
- [19] A. Skovoroda and D. Gamayunov, "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)*, Aug 2017, pp. 243–24309.
- [20] L. Onwuzurike, M. Almeida, E. Mariconti, J. Blackburn, G. Stringhini, and E. De Cristofaro, "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)*, Aug 2018, pp. 1–10.
- [21] D. Maiorca, F. Mercaldo, G. Giacinto, C. A. Visaggio, and F. Martinelli, "R-packdroid: Api package-based characterization and detection of mobile ransomware," in *Proceedings of the Symposium on Applied Computing*, ser. SAC '17, 2017.
- [22] B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, P. G. Bringas, and G. A. Marañón, "Puma: Permission usage to detect malware in android," in *CISIS/ICEUTE/SOCO Special Sessions 2012, Advances in Intelligent Systems and Computing*, vol. 189. Springer, 2012, pp. 289–298.
- [23] A. H. Lashkari, A. F. A. Kadir, H. Gonzalez, K. F. Mbah, and A. A. Ghorbani, "Towards a network-based framework for android malware detection and characterization," in *2017 15th Annual Conference on Privacy, Security and Trust (PST)*, Aug 2017, pp. 233–23309.
- [24] S. Wang, Q. Yan, Z. Chen, B. Yang, C. Zhao, and M. Conti, "Detecting android malware leveraging text semantics of network flows," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 5, pp. 1096–1109, May 2018.
- [25] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: Behavior-based malware detection system for android," in *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, ser. SPSM '11, 2011, pp. 15–26.
- [26] Z. Yuan, Y. Lu, and Y. Xue, "Droiddetector: android malware characterization and detection using deep learning," *Tsinghua Science and Technology*, vol. 21, no. 1, pp. 114–123, Feb 2016.
- [27] K. Xu, Y. Li, R. H. Deng, and K. Chen, "Deeprefiner: Multi-layer android malware detection system applying deep neural networks," in *2018 IEEE European Symposium on Security and Privacy (Euro SP)*, 2018, pp. 473–487.
- [28] T. Kim, B. Kang, M. Rho, S. Sezer, and E. G. Im, "A multimodal deep learning method for android malware detection using various features," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 3, pp. 773–788, March 2019.
- [29] R. Oak, M. Du, D. Yan, H. Takawale, and I. Amit, "Malware detection on highly imbalanced data through sequence modeling," in *Proceedings of the 12th ACM Workshop on Artificial Intelligence and Security*, ser. AISec'19. Association for Computing Machinery, 2019, p. 37–48.
- [30] Palo Alto Networks, "WildFire malware analysis service," <https://docs.paloaltonetworks.com/wildfire.html>, [Online: last accessed April 2020].
- [31] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105.
- [32] Y. Kim, "Convolutional neural networks for sentence classification," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.
- [33] T. Mikolov, K. Chen, G. S. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *CoRR*, vol. abs/1301.3781, 2013.
- [34] Y. Zhang and B. C. Wallace, "A sensitivity analysis of (and practitioners' guide to) convolutional neural networks for sentence classification," in *IJCNLP*, 2015.
- [35] X. Zhang, J. Zhao, and Y. LeCun, "Character-level convolutional networks for text classification," in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'15, 2015.
- [36] M. A. Ferrag, L. Maglaras, S. Moschogiannis, and H. Janicke, "Deep learning for cyber security intrusion detection: Approaches, datasets, and comparative study," *Journal of Information Security and Applications*, vol. 50, 2020.
- [37] P. Wanda and H. J. Jie, "Deepprofile: Finding fake profile in online social network using dynamic cnn," *Journal of Information Security and Applications*, vol. 52, 2020.
- [38] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. K. Nicholas, "Malware detection by eating a whole exe," in *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.
- [39] J. Moore, B. Gelman, and D. Slater, "A convolutional neural network for language-agnostic source code summarization," in *14th International Conference on Evaluation of Novel Approaches to Software Engineering*, 2019, pp. 15–26.
- [40] F. Martinelli, F. Marulli, and F. Mercaldo, "Evaluating convolutional neural network for effective mobile malware detection," *Procedia Comput. Sci.*, vol. 112, no. C, pp. 2372–2381, Sep. 2017.
- [41] Y. Ding, W. Zhao, Z. Wang, and L. Wang, "Automatically learning features of android apps using cnn," in *2018 International Conference on Machine Learning and Cybernetics (ICMLC)*, vol. 1, July 2018, pp. 331–336.

- [42] E. B. Karbab, M. Debbabi, A. Derhab, and D. Mouheb, "Maldozer: Automatic framework for android malware detection using deep learning," *Digital Investigation*, vol. 24, pp. S48–S59, 2018.
- [43] S. Y. Yerima, S. Sezer, and I. Muttik, "High accuracy android malware detection using ensemble learning," *IET Information Security*, vol. 9, no. 6, pp. 313–320, 2015.
- [44] Z. Ma, H. Ge, Y. Liu, M. Zhao, and J. Ma, "A combination method for android malware detection based on control flow graphs and machine learning algorithms," *IEEE Access*, vol. 7, pp. 21 235–21 245, 2019.
- [45] M. Fan, J. Liu, X. Luo, K. Chen, Z. Tian, Q. Zheng, and T. Liu, "Android malware familial classification and representative sample selection via frequent subgraph analysis," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 8, pp. 1890–1905, Aug 2018.
- [46] K. Tian, D. Yao, B. G. Ryder, G. Tan, and G. Peng, "Detection of repackaged android malware with code-heterogeneity features," *IEEE Transactions on Dependable and Secure Computing*, vol. 17, no. 1, pp. 64–77, 2020.
- [47] H. Cai, N. Meng, B. Ryder, and D. Yao, "Droidcat: Effective android malware detection and categorization via app-level profiling," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 6, pp. 1455–1470, June 2019.
- [48] M. Wong and D. Lie, "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)*, Feb. 2016.
- [49] V. Afonso, M. Amorim, A. Grégio, G. Junquera, and P. De Geus, "Identifying android malware using dynamically obtained features," *Journal of Computer Virology and Hacking Techniques*, vol. 11, pp. 9–17, 02 2014.
- [50] Y. Ye, S. Hou, L. Chen, J. Lei, W. Wan, J. Wang, Q. Xiong, and F. Shao, "Out-of-sample node representation learning for heterogeneous graph in real-time android malware detection," in *Twenty-Eighth International Joint Conference on Artificial Intelligence*, ser. IJCAI-19, 08 2019, pp. 4150–4156.
- [51] Jesus Freke, "Baksmali," <https://github.com/JesusFreke/smali>, [Online: last accessed April 2020].
- [52] Apktool, "Apktool - a tool for reverse engineering 3rd party, closed, binary Android apps," <https://ibotpeaches.github.io/Apktool/>, [Online: last accessed April 2020].
- [53] Google, "Google APIs for Android," <https://developers.google.com/android/reference/packages>, [Online: last accessed April 2020].
- [54] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *2012 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2012, pp. 95–109.
- [55] F. Wei, Y. Li, S. Roy, X. Ou, and W. Zhou, "Deep ground truth analysis of current android malware," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'17)*. Bonn, Germany: Springer, 2017, pp. 252–276.
- [56] Intel Developer Zone, "Intel 64 and IA-32 Architectures Software Developer's Manual: Volume 2," <https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4>, [Online: last accessed April 2020].
- [57] Google, "Dalvik," <https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html>, [Online: last accessed April 2020].
- [58] P. Viola and M. J. Jones, "Rapid object detection using a boosted cascade of simple features," in *IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 2001, pp. 511–518.
- [59] S. Yerima, S. Sezer, G. McWilliams, and I. Muttik, "A new android malware detection approach using bayesian classification," in *Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference on*. IEEE, 2013, pp. 121–128.
- [60] X. Liu and J. Liu, "A two-layered permission-based android malware detection scheme," in *2nd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*. IEEE, 2014, pp. 142–148.
- [61] S. Rosen, Z. Qian, and Z. M. Mao, "Appprofiler: A flexible method of exposing privacy-related behavior in android applications to end users," in *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '13, 2013, pp. 221–232.
- [62] S. Zonouz, A. Houmansadr, R. Berthier, N. Borisov, and W. Sanders, "Seccloud: A cloud-based comprehensive and lightweight security solution for smartphones," *Computers and Security*, vol. 37, pp. 215 – 227, 2013.
- [63] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, "A survey of mobile malware in the wild," in *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*. ACM, 2011, pp. 3–14.
- [64] M. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock android smartphones," in *NDSS Symposium 2012*, 2012.
- [65] H. Peng, C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy, "Using probabilistic generative models for ranking risks of android apps," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. ACM, 2012, pp. 241–252.
- [66] N. Peiravian and X. Zhu, "Machine learning for android malware detection using permission and api calls," in *2013 IEEE 25th International Conference on Tools with Artificial Intelligence*, 2013, pp. 300–305.
- [67] D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji, "A methodology for empirical analysis of permission-based security models and its application to android," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*. ACM, 2010, pp. 73–84.
- [68] A. P. Felt, K. Greenwood, and D. Wagner, "The effectiveness of application permissions," in *Proceedings of the 2Nd USENIX Conference on Web Application Development*, 2011, pp. 7–7.
- [69] Y. Fratantonio, C. Qian, S. P. Chung, and W. Lee, "Cloak and dagger: From two permissions to complete control of the ui feedback loop," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 1041–1057.
- [70] I. Goodfellow, Y. Bengio, and A. Courville, "Sequence modelling: Recurrent and recursive nets," in *Deep Learning*. MIT Press, 2016, pp. 371–372.

Declaration of interests

☒ **X** The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

☐ The authors declare the following financial interests/personal relationships which may be considered as potential competing interests:

--

Multi-view Deep Learning for Zero-day Android Malware Detection

Stuart Millar: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Resources, Data Curation, Writing – Original Draft, Writing – Review & Editing

Niall McLaughlin: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Resources, Data Curation, Writing – Review & Editing

Jesus Martinez del Rincon: Conceptualization, Methodology, Writing – Review & Editing, Supervision

Paul Miller: Conceptualization, Methodology, Writing – Review & Editing, Supervision