NF-GNN: Network Flow Graph Neural Networks for Malware Detection and Classification

Julian Busch
busch@dbs.ifi.lmu.de
LMU Munich
Munich, Germany

Anton Kocheturov
anton.kocheturov@siemens.com
Siemens Technology
Princeton, NJ, USA

Volker Tresp
volker.tresp@siemens.com
Siemens AG
Munich, Germany

Thomas Seidl
seidl@dbs.ifi.lmu.de
LMU Munich
Munich, Germany

ABSTRACT

Malicious software (malware) poses an increasing threat to the security of communication systems as the number of interconnected mobile devices increases exponentially. While some existing malware detection and classification approaches successfully leverage network traffic data, they treat network flows between pairs of endpoints independently and thus fail to leverage rich communication patterns present in the complete network. Our approach first extracts flow graphs and subsequently classifies them using a novel edge feature-based graph neural network model. We present three variants of our base model, which support malware detection and classification in supervised and unsupervised settings. We evaluate our approach on flow graphs that we extract from a recently published dataset for mobile malware detection that addresses several issues with previously available datasets. Experiments on four different prediction tasks consistently demonstrate the advantages of our approach and show that our graph neural network model can boost detection performance by a significant margin.

CCS CONCEPTS

• Security and privacy → Intrusion/anomaly detection and malware mitigation; **• Computing methodologies → Supervised learning by classification**; **Neural networks**; **Anomaly detection**.

KEYWORDS

Graph Neural Networks, Malware Detection

ACM Reference Format:
Julian Busch, Anton Kocheturov, Volker Tresp, and Thomas Seidl. 2021. NF-GNN: Network Flow Graph Neural Networks for Malware Detection and Classification. In *33rd International Conference on Scientific and Statistical Database Management (SSDBM 2021), July 6–7, 2021, Tampa, FL, USA*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3468791.3468814

1 INTRODUCTION

Malicious software (malware) poses a significant threat to the security of information technology (IT) and operational technology (OT) in private and corporate environments. Along with an increasing degree of digitalization and the rise of new technologies such as the Internet of Things (IoT), an increasing number of devices, including mobile devices, such as smartphones, and Industrial Control Systems (ICS), become connected and thus are potential targets for attacks. Accurate detection and classification of malware is thus a vital task to ensure the security of such systems.

In this work, we focus on dynamic malware detection and classification. In contrast to static methods, which analyze a candidate application's source code or the structure of its executable, dynamic methods execute the application in a controlled environment and analyze dynamic behavior that can usually not be extrapolated from static data. More specifically, we consider network traffic data generated during the execution of the application, which provides valuable insights into the dynamic and potentially malicious behavior affecting that application. However, in contrast to existing works, which classify individual network flows between two endpoints, i.e., aggregated communication between the two endpoints during some time frame, we construct a communication graph from all recorded network flows between any two endpoints during that time frame to obtain a rich representation of communication in the network. To the best of our knowledge, no existing work has considered this setting so far.

While classical machine learning methods have proven successful for malware detection and classification [15], deep learning approaches have not been studied as extensively. Deep learning methods offer an additional advantage of automatically learning suitable feature representations of the input data optimized for the task at hand, in contrast to traditional feature engineering approaches. Our novel edge feature-based graph neural network model learns suitable representations from extracted network flow graphs. Along with a base model for learning graph representations, we propose three derived model variants, a graph classifier, a graph autoencoder, and a one-class graph neural network, which are able to perform supervised malware detection and classification and unsupervised malware detection, respectively.

We evaluate our approach on a graph dataset that we extract from network flow features obtained from traffic generated by android applications executed on real mobile devices. The original flow features are provided by [26]. The data was collected in a carefully designed environment to account for several common defects observed in previously used datasets. Instead of classifying individual network flows, as the original work proposes, we follow our graph-based approach and construct a flow graph for each execution of a candidate application. Experiments on four different prediction tasks, including supervised binary, category and family classification, and unsupervised detection, consistently demonstrate the significantly superior detection performance of our approach. Our neural network model additionally boosts performance compared to baseline models, even in settings with unlabeled data and small amounts of available training data.

We summarize our contributions as follows:

- We propose, to the best of our knowledge, the first graph-based approach to network traffic-based malware detection and classification.
- We propose a novel method for extracting directed edge-attributed flow graphs from sets of network flows recorded in a monitored network.
- We propose a novel graph neural network model that effectively learns representations from these graphs, utilizing the graph topology and edge attributes.
- We provide an extensive experimental evaluation on a novel flow graph dataset considering four different supervised and unsupervised detection tasks.

2 RELATED WORK

While static malware detection and classification methods analyze a candidate application's source code or the structure of its executable file, dynamic methods execute the application in a controlled environment and analyze its behavior. Such behavior is difficult and often impossible to extrapolate from static data. Further, static methods are commonly vulnerable to obfuscation techniques modifying the code or structure of the executable. Our approach is a dynamic one, focusing on network traffic data generated by a particular candidate application during execution.

A comprehensive overview of existing machine learning methods for static and dynamic malware detection and classification is provided by a recent survey [15]. Methods relying on network traffic data mainly differ by which specific features are extracted and which machine learning algorithm is used. To the best of our knowledge, all existing approaches focus on classifying individual network traffic between two endpoints in a network, possibly at different resolutions. For instance, [5] describes different resolution levels, ranging from single transactions over sessions to flows and conversations. Thereby, a flow describes traffic associated with a specific (`Source-IP, Source-Port, Destination-IP, Destination-Port, Protocol`)-tuple within a particular time frame and a conversation aggregates flows with the same Source- and Destination-IP endpoints. Instead of focusing on bilateral flows or conversations between two endpoints, our approach extracts a graph from flows between any two endpoints in a monitored network during a specific time frame to obtain a richer representation of communication in the network.

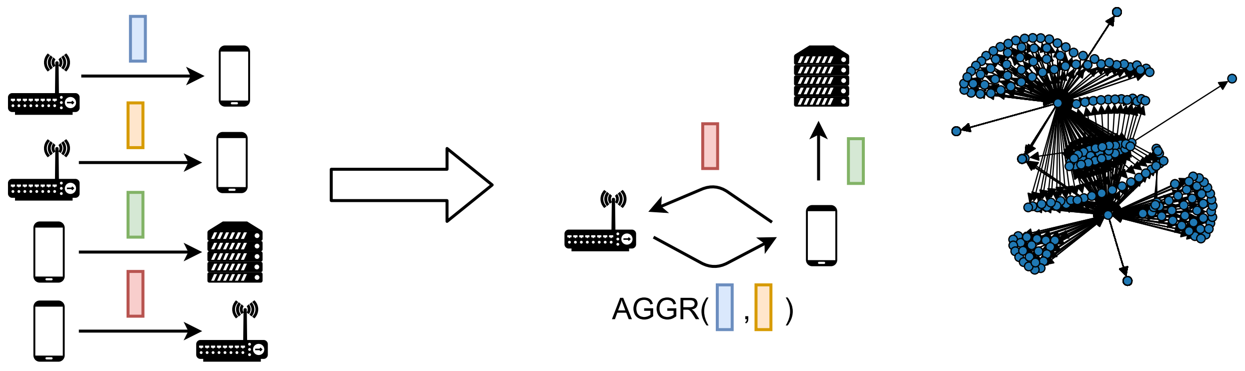
Existing methods typically report high performance on datasets that exhibit various common defects. The recently published *CICAndMal2017* dataset [26] has addressed these issues by providing a sufficient number of malware samples from diverse malware categories and families with a realistic distribution of benign and malicious applications. Additionally, each application is executed on an actual physical device instead of an emulator or a virtual machine to accurately capture its actual behavior. Instead of classifying network flows individually as proposed by the authors, we instead extract network flow graphs from the given network traffic features.

The results reported in [26] have been improved by adding new dynamic API-call features and combining a dynamic prediction model with a static prediction model [34]. Our model could be combined with a static prediction model in a similar fashion. Improved accuracy could also be achieved by predicting labels for conversations instead of individual flows [1], further mitigating the effect of port randomization techniques. Our approach takes an additional step ahead by classifying flow graphs modeling the communication between all endpoints instead of only pairs of endpoints. Two further works [6, 12] reported malware detection results only for a subset of the whole dataset.

Existing graph-based approaches to malware detection and classification mostly focus on static analysis, considering function call graphs [18, 20, 23] or control graphs [4, 14]. Dynamic graph-based approaches include [2], where graphs are constructed from instruction traces collected during execution, and [22], where system call graphs are considered. The latter work is most related to our approach since a *Graph Convolutional Neural Network (GCN)* [25] is used to learn node features. To the best of our knowledge, no existing work has considered graph-based approaches based on dynamically generated network traffic data.

Deep learning methods have not been investigated as extensively as classical machine learning methods for network traffic-based approaches. Existing methods consider detection of endpoints generating malicious traffic [29], intrusion detection [28], or ransomware detection [6]. To the best of our knowledge, no existing deep learning-based malware detection and classification method considers network flow graphs.

While our proposed approach includes a novel graph neural network model for malware detection and classification, it could be more generally employed to solve other graph classification and graph anomaly detection tasks. Graph neural networks [3, 7, 8, 16, 25, 36–38, 40] have recently become a de-facto standard for machine learning on graphs. The majority of these methods can be described in a message-passing framework, as detailed in [16]. The main idea is to iteratively propagate feature vectors through the graph by passing messages between nodes such that a node's representation depends on feature vectors appearing in its neighborhood. While most existing graph neural network models consider only node attributes, our model focuses on edge attributes instead. Some existing models consider edge attributes [17, 21, 24, 31, 33, 39], but none of these models is directly applicable to our setting since they either consider multi-relational graphs or focus on node attributes and only utilize edge features to improve message passing between nodes. Other existing models [27, 32, 42] focus on more specific tasks which differ from our setting.



(a) Network Flow Graph Extraction (b) Real Network Flow Graph

Figure 1: From a set of network flows recorded during a specific time frame, a flow graph is constructed by adding directed edges between all pairs of endpoints that communicated with at least one flow. Each flow is described by a feature vector summarizing its corresponding network traffic. Edges in the flow graph are annotated with these feature vectors, where feature vectors of parallel flows are aggregated. The topology of a real flow graph extracted from network traffic generated during execution of a *FakeAV* Scareware application is shown in (1b).

3 NETWORK FLOW GRAPH EXTRACTION

To decide whether a particular candidate instance of an application is malicious, we collect all network traffic generated during the execution of that application instance within a given time interval after installation. The resulting data consists of a set of network flows described by feature vectors that can be extracted from pcap-files using tools such as *CICFlowMeter* [13]. Thereby, each flow F describes network traffic associated with one particular `(Source-IP, Source-Port, Destination-IP, Destination-Port, Protocol)`-tuple during the considered time frame and has a feature vector $f \in \mathbb{R}^d$ attached to it. Typical features include the number of packets sent, mean and standard deviation of the packet length, or minimum and maximum interarrival time of the packets.

From the resulting set of flows $\mathcal{F}$, we extract a flow graph, where the nodes correspond to endpoints in the network and edges model communication between these endpoints. Instead of considering `(IP, Port)`-tuples, we factor out the port information and consider IP-endpoints for two main reasons:

(1) Apart from standard ports, port selection is often arbitrary and could even be subject to port randomization techniques, leading to arbitrary and potentially misleading graph structure.
(2) Empirically, we found `(IP, Port)`-graphs to be very sparse and rather uninformative. In comparison, IP-graphs exhibit much more interesting topologies.

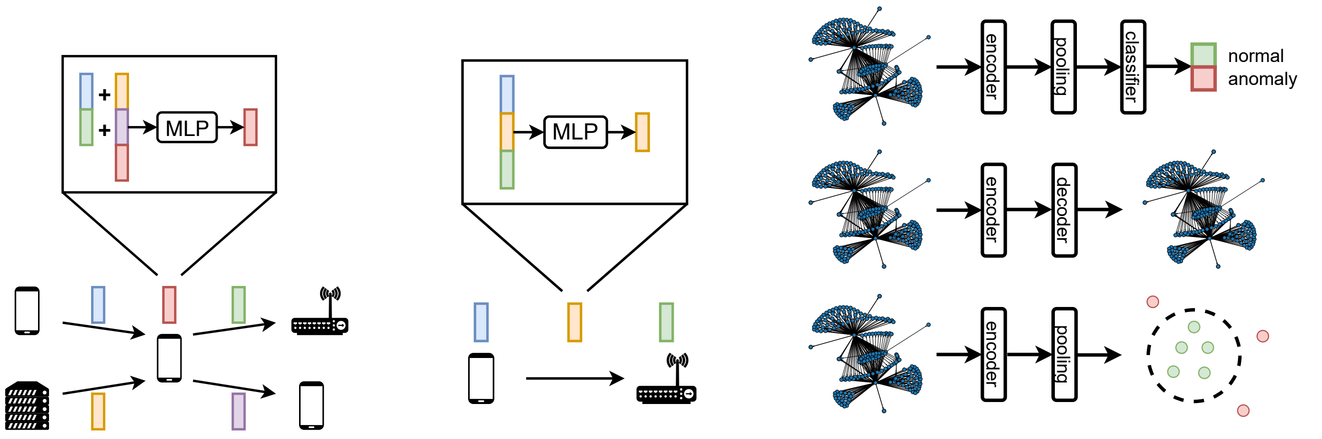
More specifically, from a set of flows $\mathcal{F}$, we extract a directed graph $G = (V, E)$ where the nodes correspond to endpoints involved in any flow $F \in \mathcal{F}$ and a directed edge is added for all pairs (s_i, t_i) for which there exists a flow $F_i \in \mathcal{F}$ with source and target IP s_i and t_i, respectively. The feature vector assigned to this edge aggregates the feature vectors $f_i \in \mathbb{R}^d$ of all flows F_i along this edge using a set of five aggregation functions. For each feature, the shape of the distribution of values along the edge is described using the first four moments, namely the mean, standard deviation, skew and kurtosis, and the median value. The aggregate values are computed for each feature and then concatenated, resulting in a feature vector $x_i \in \mathbb{R}^{5d}$ for each edge $e_i \in E$. The flow graph extraction procedure is illustrated in Figure 1a. Figure 1b shows an exemplary graph extracted from real data.

Intuitively, the resulting graph captures how network traffic flows between different endpoints in the monitored network during a specific time frame. The graph structure reveals where traffic is flowing, and the additional edge attributes describe how it is flowing. Connecting individual flows in a graph provides a much richer relational representation compared to treating flows individually. Thus, we expect models learning from these graphs to perform significantly better at detection and classification tasks than models which classify individual flows. Our experimental results confirm this intuition.

We wish to note that such graphs could potentially be used for further applications, such as intrusion detection or identifying devices generating malicious traffic.

4 NETWORK FLOW GRAPH NEURAL NETWORKS

From a machine learning perspective, we consider two different problems: Supervised graph classification and unsupervised graph anomaly detection. To solve these problems, we introduce a novel graph neural network model, which learns expressive representations from network flow graphs, along with three different variants of that base model, a supervised graph classifier, an unsupervised graph autoencoder, and an unsupervised one-class graph neural network. The different model variants are illustrated in Figure 2c.



(a) Node Update **(b) Edge Update** **(c) Model Variants**

Figure 2: Our model learns how endpoints in the network communicate with each other by sequentially updating node (2a) and edge feature vectors (2b) by neural message passing. The learned representations are used by our three model variants, a graph classifier, a graph autoencoder, and a graph one-class neural network, to solve supervised and unsupervised detection tasks (2c).

While we focus on malware detection and classification in this paper, the proposed models can be employed for general learning problems on (directed) graphs with edge attributes.

4.1 Problem Setting

Formally, we are given a set of graphs $\mathcal{G} = \{G_1, \ldots, G_N\}$, a set of edge feature matrices $X_i \in \mathbb{R}^{m_i \times d}$, $i = 1, \ldots, N$, where $m_i = |E_i|$, and a label matrix $\hat{Y} \in \{0,1\}^{N \times c}$ providing a class label for each graph, where c is the number of classes. For the sake of simplicity, we restrict ourselves to directed graphs in the following, though our models can be applied to undirected graphs in a straightforward fashion. Further, it is not required that the input graphs share the same topology,

In a supervised anomaly detection setting, class labels could be binary (normal vs. anomalous) or multi-class (normal class and different categories or families of anomalies). For supervised classification, a labeled training dataset is given as described above, and the task is to train a model which accurately predicts labels for new graphs not seen by the model during training. For unsupervised anomaly detection, the training set is not labeled and consists of normal data and (usually a relatively small fraction of) anomalous data. The model is required to learn a concept of normality from the training data and correctly classify new graphs as either being normal or anomalous.

4.2 Learning Representations of Network Flow Graphs

Each input instance for our model is a directed graph $G = (V, E)$ with adjacency matrix $A \in \{0,1\}^{n \times n}$ and an edge feature matrix $X \in \mathbb{R}^{m \times d}$, where $m := |E|$. The representation learning part of our model computes latent representations of the edges and nodes in the graph and finally outputs a latent feature vector $h^{(1)} \in \mathbb{R}^h$ for each node in the graph. Such a vector intuitively describes how the corresponding endpoint interacts with other endpoints in the network. Depending on the availability of labels, variants of our model compute either predictions or an anomaly score for an input graph from its latent node feature vectors. The model is trained end-to-end such that the latent representations are optimized towards the specific task. Given an input graph with edge attributes, our model performs the following feature transformation and propagation steps to sequentially compute latent representations of the graph's nodes and edges:

$$E^{(0)} = f_1(X) \qquad \in \mathbb{R}^{m \times h} \tag{1}$$

$$H^{(0)} = f_2\left(\left[\hat{\hat{\mathrm{B}}}_{in}\, E^{(0)}, \hat{\hat{\mathrm{B}}}_{out}\, E^{(0)}\right]\right) \qquad \in \mathbb{R}^{n \times h} \tag{2}$$

$$E^{(1)} = f_3\left(\left[\hat{\hat{\mathrm{B}}}_{in}^{T}\, H^{(0)}, \hat{\hat{\mathrm{B}}}_{out}^{T}\, H^{(0)}, E^{(0)}\right]\right) \qquad \in \mathbb{R}^{m \times h} \tag{3}$$

$$H^{(1)} = f_4\left(\left[\hat{\hat{\mathrm{B}}}_{in}\, E^{(1)}, \hat{\hat{\mathrm{B}}}_{out}\, E^{(1)}, H^{(0)}\right]\right) \qquad \in \mathbb{R}^{n \times h}, \tag{4}$$

where $[\cdot,\cdot]$ denotes concatenation and $f_1, \ldots, f_4$ are Multi-Layer Perceptrons (MLPs) with appropriate input and output dimensions. As per default, we use single-layer MLPs

$$f_i(X) = q\left(XW_i + b_i\right), \tag{5}$$

where W_i and b_i are the learnable parameters of the model and q is a non-linear activation. We use *ReLU* activations and add batch normalization. The propagation matrices $\hat{\hat{\mathrm{B}}}_{in}, \hat{\hat{\mathrm{B}}}_{out} \in \mathbb{R}^{n \times m}$ are obtained from the node-edge incidence matrices $B_{in}, B_{out} \in \{0,1\}^{n \times m}$ with

$$(B_{in})_{ij} = \begin{cases} 1 & \text{if} \quad \exists v_k \in V : e_j = (v_k, v_i) \\ 0 & \text{else} \end{cases} \tag{6}$$

and

$$(B_{out})_{ij} = \begin{cases} 1 & \text{if} \quad \exists v_k \in V : e_j = (v_i, v_k) \\ 0 & \text{else} \end{cases}, \tag{7}$$

indicating in- and out-going edges for each node, by substituting non-zero entries with normalized edge weights. Normalization is applied to preserve the scale of the feature vectors. In particular, we apply symmetric normalization to the adjacency as in [25] before computing the node-edge incidence matrices, where the normalized adjacency matrix is given as $\hat{\tilde{A}} = \tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}$ with $\tilde{A} = A + I$ and degree matrix $\tilde{D}$. Self-loops added for normalization are removed again such that the graph structure remains unchanged. Illustrations of the performed node and edge feature updates are provided in Figure 2a and 2b, respectively.

The first network layer learns how endpoints interact with each other directly by first applying a learnable feature transformation to the original edge feature vectors (Equation 1) and subsequently computing node representations by aggregating feature vectors from neighboring edges (Equation 2). Notably, incoming and outgoing traffic is modeled separately for each node.

The second layer enables the model to learn how endpoints communicate indirectly with their 2-hop neighbors. In a first step, the edge features are updated again by transforming the concatenated feature vectors of the source and destination node and the edge features from the previous layer (Equation 3). Concatenating the edge features from the previous layer as residual connections [19] gives this layer direct access to previously learned features and aids in optimization. Such skip-connections have shown to improve the performance of graph neural networks when applied to node features [41], motivating us to apply them to edge features as well. The edge feature update is followed by an update of the node features using features of incoming and outgoing edges and skip-connected node features from the first layer (Equation 4). These node representations constitute the final output of our representation learning module.

In principle, one could add more layers to the model in a similar fashion to model interaction between more distant endpoints. However, the flow graphs considered in this paper usually have a relatively small diameter, such that additional layers might not result in improved performance but rather lead to over-fitting. In our experiments, we observed the best performance with either one or two layers.

4.3 Network Flow Graph Classifier

For supervised graph classification, we append two more layers to the representation learning module. First, a pooling layer aggregates all node feature vectors in the input graph to a single vector describing the whole graph. The second layer predicts the graph label from the pooled graph representation,

$$h = \text{pool}\left(H^{(1)}\right) \qquad \in \mathbb{R}^{h} \tag{8}$$

$$y = \text{softmax}\left(Wh + b\right) \qquad \in \mathbb{R}^{c}\,. \tag{9}$$

Above, pool denotes a pooling function, such as element-wise mean or maximum, which aggregates all node representations into a single embedding vector for the whole graph. Predictions are computed by a dense prediction layer with learnable parameters $W \in \mathbb{R}^{c\times h}$ and $b \in \mathbb{R}^{c}$ and a softmax activation. The model parameters are then optimized w.r.t. the *cross-entropy* loss

$$\mathcal{L}_{CLF}(\mathcal{G}) = \frac{1}{N \cdot c} \sum_{\substack{G_i \in \mathcal{G} \\ j \in 1,\ldots,c}} -y_{ij} \log \hat{y}_{ij}\,. \tag{10}$$

We denote this model as *NF-GNN-CLF*.

4.4 Network Flow Graph Autoencoder

For unsupervised anomaly detection, autoencoder models often perform well in practice [11]. In general, an autoencoder consists of two neural network modules. While an encoder learns compact and expressive representations of the model inputs, a decoder is supposed to reconstruct the original inputs from their learned representations. If the model is trained with exclusively or mostly normal data, the reconstruction loss can be interpreted as an anomaly score, where instances incurring a larger reconstruction loss are considered more anomalous.

We propose a graph autoencoder model where our representation learning module acts as an encoder. The latent node representations $H^{(1)}$ are then used to reconstruct the original edge feature vectors of the graph using a decoder, which is a mirrored version of the encoder:

$$E^{(2)} = f_5\left(\left[\hat{\tilde{B}}_{in}^{T} H^{(1)}, \hat{\tilde{B}}_{out}^{T} H^{(1)}\right]\right) \qquad \in \mathbb{R}^{m\times h} \tag{11}$$

$$H^{(2)} = f_6\left(\left[\hat{\tilde{B}}_{in} E^{(2)}, \hat{\tilde{B}}_{out} E^{(2)}, H^{(1)}\right]\right) \qquad \in \mathbb{R}^{n\times h} \tag{12}$$

$$E^{(3)} = f_7\left(\left[\hat{\tilde{B}}_{in}^{T} H^{(2)}, \hat{\tilde{B}}_{out}^{T} H^{(2)}, E^{(2)}\right]\right) \qquad \in \mathbb{R}^{m\times h} \tag{13}$$

$$E^{(4)} = f_8\left(E^{(3)}\right) \qquad \in \mathbb{R}^{m\times h} \tag{14}$$

If the encoder uses only a single layer, the first layer of the decoder (Equation 11–12) is dropped and the node embeddings $H^{(0)}$ are used as input instead. The model parameters are optimized w.r.t. a reconstruction loss

$$\mathcal{L}_{AE}(\mathcal{G}) = \frac{1}{N} \sum_{G_i \in \mathcal{G}} \frac{1}{m_i} \left\| X_i - E_i^{(4)} \right\|_F^2, \tag{15}$$

where $|| \cdot ||_F$ denotes the Frobenius norm. A similar loss was used in [10] to reconstruct node attributes, whereas our model operates on edge-attributed graphs. We denote this variant of our model as *NF-GNN-AE*.

4.5 One-Class Network Flow Graph Neural Network

Though autoencoder models perform well in practice, they don't optimize an anomaly detection objective directly. *Deep SVDD* [30] combines *Support Vector Data Description (SVDD)* [35] with a neural network for anomaly detection in a learned latent space. The main idea is to learn a transformation into a latent space such that most instances are mapped into a hypersphere in that space, and anomalous instances will fall outside of the hypersphere.

We propose a one-class graph neural network consisting of our representation learning module and an additional pooling layer, which summarizes each input graph into a single feature vector, similarly as for the supervised graph classifier. The model parameters are optimized w.r.t. a one-class loss

$$\mathcal{L}_{OC}(\mathcal{G}) = \frac{1}{N} \sum_{G_i \in \mathcal{G}} \|h_i - \mu\|_2^2 + \frac{\lambda}{2} \sum_{i=1}^{4} \|W_i\|_F^2 , \tag{16}$$

where $\mu \in \mathbb{R}^h$ denotes the center of the hyper-sphere in latent space. The center is initialized with the mean embedding vector of all graphs in $\mathcal{G}$ after the first forward-pass and does not change thereafter. The second part of the loss regularizes the model parameters to limit model complexity. Bias vectors have been removed from all layers to prevent trivial solutions [30]. We denote this variant of our model as *NF-GNN-OC*.

5 EXPERIMENTS

We evaluate our approach on a graph dataset that we extract from the network traffic data provided by the *CICAndMal2017* dataset [26]. We focus on this datasets since it addresses several common defects shared by other existing datasets, allowing for a realistic and meaningful evaluation. These issues are addressed by providing a sufficient number of malware samples from diverse malware categories and families with a realistic distribution of benign and malicious applications. To accurately capture dynamic behavior, each application is executed on an actual physical device instead of an emulator or a virtual machine.

5.1 Dataset Preparation

Our extracted dataset consists of 2126 samples, where each sample corresponds to one instance of an android application installed and executed on a mobile phone. For each sample, all network flows within the network during execution are captured. For each flow, 80 features are recorded, including, e.g., the number of packets sent, mean and standard deviation of the packet length, and minimum and maximum interarrival time of the packets. For a more detailed description of the data collection process, we refer to [26].

For each sample, 3 different labels are available, a binary label indicating whether the application is malicious or not, a category label with 5 possible values indicating the general class of malware, and a family label with 36 different values indicating the specific type of malware. Malware families with fewer than 9 samples have been removed to ensure a reasonable split into train, validation, and test sets. Consequently, for the family prediction task, only 2071 samples are available.

For each sample, we extract a graph as described in Section 3 and remove the flow-id, timestamp, and endpoint IP and port information from the feature set. Additionally, we remove all features that are constant among all edges of all graphs, leaving 330 edge features.

To be able to compare against baselines apart from flow- and conversation-based methods and our graph neural network models, we construct additional datasets, which represent each sample by a single feature vector instead of a graph. We consider three different feature sets:

(1) **Flow Features** For each sample, we aggregate the features of all flows for this sample using the same aggregation functions we used for deriving the edge feature vectors and concatenate the aggregates to a 318-dimensional feature vector.
(2) **Graph Features** To evaluate the importance of the graph topology for the baseline models, we extract a set of structural features from each graph. In particular, we extract 2 global features (global clustering coefficient and assortativity) and 8 local node-features (degree, number of 2-hop neighbors, clustering coefficient, avg. neighbor degree, avg. neighbor clustering coefficient, number of edges in egonet, number of edges leaving egonet, betweenness centrality) that are aggregated over all nodes in the graph, again using the same aggregation functions. All extracted graph features are concatenated, leading to a 40-dimensional feature vector.
(3) **Combined Features** To provide access to both types of features, we concatenate the flow and graph features to a combined 358-dimensional feature vector for each sample.

Again, all constant feature columns have been removed. All feature matrices, including the edge feature matrices for our model, are standardized before training.

5.2 Supervised Malware Detection and Classification

We consider three supervised tasks, binary, category, and family classification. To ensure a fair and unbiased comparison, we follow a rigorous evaluation protocol.

5.2.1 Experimental Setup. First, we split the dataset into a train, validation, and test part. Each model is trained on the training set, hyper-parameters are chosen based on validation set performance using a grid search, and results are reported on the test set using the optimal hyper-parameter values. All experiments are repeated on 30 randomly generated splits, and mean and standard deviation of the results are reported.

To ensure a balanced training set, we sample 100, 25, and 5 samples per class for training for binary, category, and family classification, respectively. For binary and category classification, 5% of the remaining samples are sampled in a stratified fashion for validation. The remaining samples are used for testing. For family prediction, 20% of the non-training samples are chosen for validation to account for smaller class sizes. Stratification ensures that smaller classes are represented appropriately in the validation set.

Again, to account for class imbalance, we report weighted precision, recall, and F1 scores. In all considered settings, we checked that our models outperform the respective best competitor with statistical significance at $P < 1e-5$ according to a Wilcoxon signed-rank test.

We compare our model against 7 baseline algorithms with different inductive bias, *Support Vector Machine (SVM)* with linear and RBF-kernel, *k-Nearest Neighbor Classifier (KNN)*, *Decision Tree (DT)*, *Random Forest (RF)*, *Adaboost (ADA)* and a *Multi-Layer Perceptron (MLP)* with up to two dense layers and *ReLU* and *softmax* activations. Additionally, we compare our graph-based approach against two existing flow-based approaches [26, 34] and one conversation-based approach [1]. Since no source code has been made available,

	Weighted Recall			Weighted Precision		
	Binary	**Category**	**Family**	**Binary**	**Category**	**Family**
Flows [26]	88.30	48.50	25.50	85.80	49.90	27.50
Flows + Static + API Calls [34]	95.30	81.00	61.20	95.30	83.30	59.70
Conversations [1]	89.00	79.64	66.59	86.65	80.20	67.21
SVM-LIN	96.26 ± 2.12	72.83 ± 10.03	28.74 ± 15.56	96.72 ± 1.49	87.31 ± 0.80	90.35 ± 0.37
SVM-RBF	96.20 ± 1.51	85.42 ± 3.84	49.96 ± 16.20	96.64 ± 1.06	89.13 ± 2.18	91.52 ± 0.98
KNN	95.71 ± 2.05	78.10 ± 12.29	42.53 ± 17.86	95.75 ± 1.96	87.08 ± 2.40	90.92 ± 0.76
DT	92.65 ± 4.13	73.42 ± 12.19	45.98 ± 30.81	93.79 ± 2.80	85.63 ± 6.59	85.72 ± 18.57
RF	95.85 ± 2.00	84.30 ± 8.24	56.06 ± 19.60	96.24 ± 1.60	90.32 ± 2.60	91.67 ± 0.99
ADA	96.38 ± 1.62	76.02 ± 12.67	42.17 ± 28.82	96.67 ± 1.31	83.97 ± 6.72	87.88 ± 13.35
MLP	97.19 ± 1.19	85.69 ± 4.46	49.56 ± 17.75	97.29 ± 1.07	89.28 ± 2.49	90.23 ± 4.32
NF-GNN-CLF	**99.42 ± 0.45**	**95.41 ± 1.48**	**91.37 ± 8.39**	**99.44 ± 0.44**	**96.14 ± 1.07**	**93.62 ± 2.52**

Table 1: Weighted recall and precision scores for the three supervised tasks. For competing methods, we report the scores provided by the respective authors. For our baselines and our model, scores are reported in terms of mean and standard deviation over 30 independent data splits.

we report the scores provided by the respective authors. Results have been provided only for a single data split.

For all neural network models, we use early stopping based on validation set performance using a patience of 20 epochs and a maximum of 1000 epochs. The default learning rate for all MLP-models is fixed as $1e-3$. For our model, we additionally apply dropout regularization before each dense layer and on the propagation matrices. All considered hyper-parameter values for all models are provided in Appendix A.

Since the original feature vectors are rather high-dimensional, we introduce an additional hyper-parameter for each baseline model indicating whether or not to perform *Principle Component Analysis (PCA)* as feature reduction before training where 95% of the variance in the data is retained.

5.2.2 Detection and Classification Performance. Detection and classification results for all three tasks are reported in terms of weighted recall and precision in Table 1. For each of our baseline model, the best feature set (Flow, Graph, or Combined) has been chosen based on validation set performance.

While some baselines perform notably weaker than others, we can observe that all baseline models still exceed the best results reported for flow classification in [26] in terms of both recall and precision, in most cases by a large margin.

Flow classification with an added static detection model and additional API-call features [34] performs better than some of our weaker baselines, especially in terms of recall. However, this approach is outperformed in terms of precision by all our baselines on category and family prediction. Our stronger baselines can outperform this approach, sometimes even by a large margin, except in terms of recall for family classification. It is important to note that [34] use additional data compared to our baselines and our model. Our approach could, in principle, be extended to also use this additional data and thus further boost performance.

Conversation-level detection [1] performs worse than [34], except for family classification. This competitor outperforms all of our baselines in terms of recall on the family classification task. In terms of precision, however, all of our baselines outperform all competitors by a significant margin.

The competitive and sometimes even vastly superior performance of our baselines already supports the main motivation of our approach to detect and classify malware using network flow graphs. Notably, this performance has been achieved even under a rigorous evaluation protocol and using relatively small training sets. In comparison, the competitors have used training set sizes between 60% and 80%.

Our proposed graph neural network model NF-GNN-CLF can further boost performance significantly compared to the baselines. It is able to significantly exceed the performance of all competitors on all three tasks in terms of both recall and precision. In particular, compared to the best competitor, recall can be improved by 4.12%, 14.41%, and 24.78% for binary, category, and family classification,

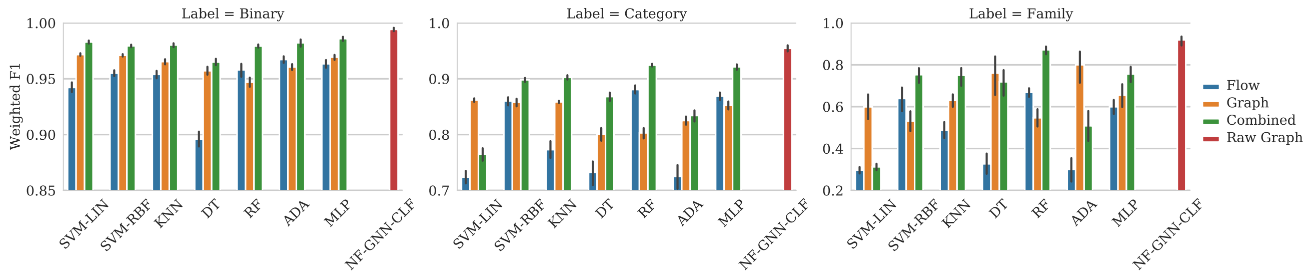


Figure 3: Weighted F1 scores for the three supervised tasks using different feature sets. Results are reported in terms of mean and standard deviation over 30 independent data splits.

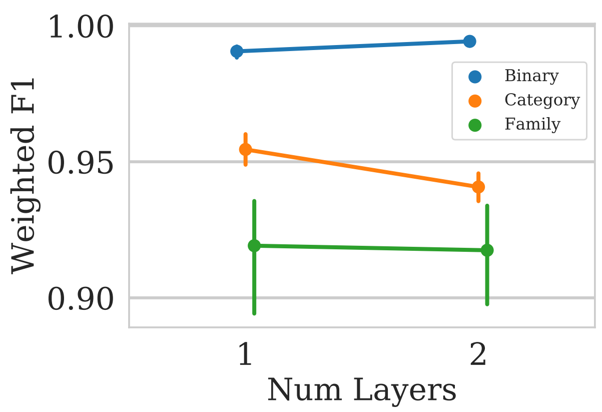


Figure 4: Performance of our model on supervised tasks with different numbers of layers.

respectively. Precision is improved by 4.14%, 12.84%, and 26.41%. We wish to note that in most cases, the best competitor [34] uses additional data that is not available to our baselines and model. Compared to the other two competitors, the performance gain is even more significant. The largest improvement is achieved for the family classification task with a performance increase of over 65% compared to flow classification [26].

5.2.3 *Importance of Different Feature Sets.*

To get more insight into the importance of different feature sets for our baseline models, we compare their performance in terms of weighted F1 score on different feature sets in Figure 3. We can observe that almost all baseline models perform best on the combined flow and graph feature set.

As a notable exception, the linear SVM performs best on category and family classification using only the graph features, and adding flow features significantly hurts performance. However, even using only graph features, this baseline still performs worst among all baselines. Similarly, DT and ADA perform best using only graph features for family classification but are also outperformed by other baselines using the combined feature set. In most cases, the combination boosts performance significantly over the best individual feature set.

Our model operating on the raw graphs still outperforms all baselines.

5.2.4 *Influence of the Number of Network Layers.*

To further examine the influence of the number of layers on our model's performance, we compare different choices for the supervised classification tasks in Figure 4. We can observe that modeling 2-hop interactions between endpoints can boost performance on the binary prediction tasks, while direct interactions are more crucial for the remaining two tasks. In general, performance remains relatively stable for different numbers of layers.

5.3 Unsupervised Malware Detection

We further evaluate our approach in a more realistic unsupervised setting, where no labels are available for training.

5.3.1 *Experimental Setup.*

Our experimental setup is the same as in the supervised case with a few adjustments. To ensure a realistic distribution of benign and malicious samples, we perform stratified sampling to first split 20% of the samples for training and then 10% of the remaining samples for validation. The remaining samples are used for testing. All algorithms obtain access to the labeled validation set for hyper-parameter optimization, but training is still unsupervised. We evaluate detection performance using AUROC since it inherently adjusts for class imbalance [9].

We evaluate against a set of popular baseline algorithms for anomaly detection, *One-class SVM (OC-SVM)* with linear and RBF-kernel, *Local Outlier Factor (LOF)*, *Kernel Density Estimation (KDE)*, *Isolation Forest (IF)*, *Autoencoder with dense layers (MLP-AE)* and *One-Class Neural Network with dense layers (MLP-OC)*. Again, all considered hyper-parameter values are provided in the supplement. Since the flow- and conversation-based competitors [1, 26, 34] only consider supervised detection and classification, we are not able to include them for comparison.

5.3.2 *Detection Performance.*

Table 2 shows the detection results for different feature sets. Results for different feature sets are additionally visualized in Figure 6. We can observe that several models report better prediction performance using only structural graph features. Thus, compared to the supervised setting, it is even more important to consider the topology of the network flow graph. While both neural network baselines, as well as KDE and IF already exhibit high detection performance, our models can again boost performance by a significant margin, demonstrating the importance

	OC-SVM-LIN	OC-SVM-RBF	LOF	KDE	IF	MLP-AE	MLP-OC	NF-GNN-AE	NF-GNN-OC
Flow	57.17 ± 4.81	70.01 ± 1.90	73.15 ± 1.22	72.50 ± 1.03	70.39 ± 1.12	71.64 ± 1.20	81.29 ± 4.07	|	|
Graph	54.60 ± 19.32	67.19 ± 2.89	58.57 ± 5.12	91.79 ± 0.66	90.73 ± 1.31	89.56 ± 0.77	94.00 ± 1.32	95.34 ± 0.85	**96.75 ± 1.22**
Combined	58.21 ± 4.83	76.07 ± 1.78	77.94 ± 1.56	86.60 ± 1.07	85.73 ± 2.21	83.04 ± 0.78	94.03 ± 4.47	|	|

Table 2: AUROC scores for unsupervised malware detection in terms of mean and standard deviation over 30 independent data splits. Our models use the raw graphs as input instead of the features extracted for the baseline models.

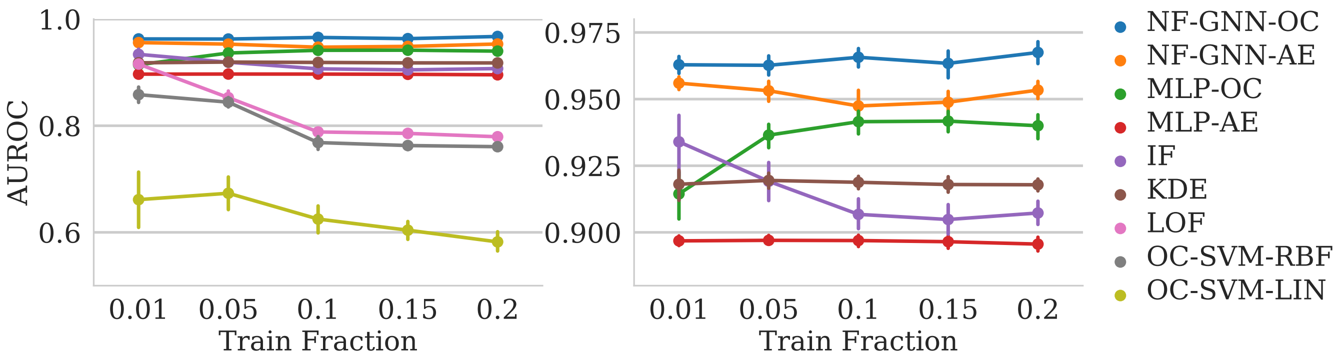


Figure 5: AUROC scores for unsupervised malware detection using different fractions of training samples. The right-hand figure provides an enlarged view of the upper part of the left-hand figure.

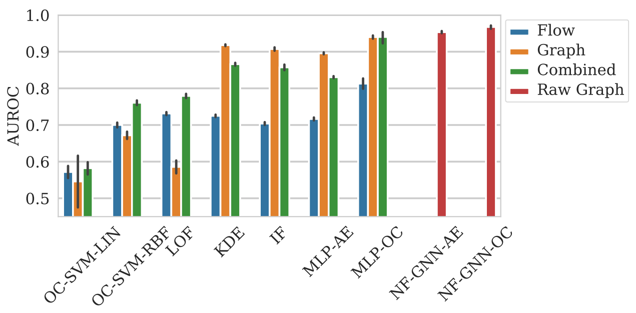


Figure 6: AUROC scores for unsupervised malware detection using different feature sets.

of learning suitable representations from network flow graphs. We wish to emphasize that the performance of our unsupervised models, NF-GNN-AE and NF-GNN-OC, almost matches that of their supervised binary classification counterpart, NF-GNN-CLF, even without any labels provided for training.

5.3.3 *Influence of the Training Set Size.* In practice, there is often a shortage of available training data, even unlabeled data. Thus, we further investigate performance using different amounts of training data where we gradually reduce the fraction of training samples from 20% to 1%. For each training set size, a new grid search is performed to determine the best hyper-parameters for each model. Figure 5 shows that while some models perform worse with more available training data, possibly due to overfitting to anomalies in the training set, especially MLP-OC and NF-GNN-OC benefit from more training data. For all training set sizes, both of our models consistently outperform all competing baselines.

6 CONCLUSION

We proposed a novel network flow graph-based approach to malware detection and classification, where we monitor network traffic generated by a candidate application and extract a flow graph, which models communication between devices during the considered time frame. In addition, we proposed a novel edge feature-based graph neural network model along with three different model variants for supervised and unsupervised settings. Empirically, we found that even baseline models operating on manually extracted graph and flow features perform very well in all settings. In addition, our proposed models automatically learn suitable representations of network flow graphs and can boost performance even further, significantly outperforming all competitors on all tasks. Ablation studies examined the influence of different feature sets, the number of network layers, and training set size on detection and classification performance. In future work, we plan to consider additional network architectures, such as attention, model temporal dynamics, and consider explainability of model decisions.

ACKNOWLEDGMENTS

This work was done during an internship at CT RDA BAM IBI-US, Siemens Technology, Princeton, NJ, USA.

REFERENCES

[1] Mohammad Abuthawabeh and Khaled Mahmoud. 2020. Enhanced android malware detection and family classification using conversation-level network traffic features. *International Arab Journal of Information Technology* 17 (2020), 607–614.

[2] Blake Anderson, Daniel Quist, Joshua Neil, Curtis Storlie, and Terran Lane. 2011. Graph-based malware detection using dynamic analysis. *Journal in computer Virology* 7, 4 (2011), 247–258.

[3] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. 2018. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261* (2018).

[4] Zahra Bazrafshan, Hashem Hashemi, Seyed Mehdi Hazrati Fard, and Ali Hamzeh. 2013. A survey on heuristic malware detection techniques. In *The 5th Conference on Information and Knowledge Technology*. IEEE, 113–120.

[5] Dmitri Bekerman, Bracha Shapira, Lior Rokach, and Ariel Bar. 2015. Unknown malware detection using network traffic classification. In *2015 IEEE Conference on Communications and Network Security (CNS)*. IEEE, 134–142.

[6] Iram Bibi, Adnan Akhunzada, Jahanzaib Malik, Ghufran Ahmed, and Mohsin Raza. 2019. An effective android ransomware detection through multi-factor feature filtration and recurrent neural network. In *2019 UK/China Emerging Technologies (UCET)*. IEEE, 1–4.

[7] Michael M Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. 2017. Geometric deep learning: going beyond euclidean data. *IEEE Signal Processing Magazine* 34, 4 (2017), 18–42.

[8] Julian Busch, Jiaxing Pi, and Thomas Seidl. 2020. PushNet: Efficient and Adaptive Neural Message Passing. In *24th European Conference on Artificial Intelligence (ECAI)*.

[9] Guilherme O Campos, Arthur Zimek, Jörg Sander, Ricardo JGB Campello, Barbora Micenková, Erich Schubert, Ira Assent, and Michael E Houle. 2016. On the evaluation of unsupervised outlier detection: measures, datasets, and an empirical study. *Data mining and knowledge discovery* 30, 4 (2016), 891–927.

[10] Keting Cen, Huawei Shen, Jinhua Gao, Qi Cao, Bingbing Xu, and Xueqi Cheng. 2020. ANAE: Learning Node Context Representation for Attributed Network Embedding. *arXiv preprint arXiv:1906.08745* (2020).

[11] Raghavendra Chalapathy and Sanjay Chawla. 2019. Deep learning for anomaly detection: A survey. *arXiv preprint arXiv:1901.03407* (2019).

[12] Rong Chen, Yangyang Li, and Weiwei Fang. 2019. Android malware identification based on traffic analysis. In *International Conference on Artificial Intelligence and Security*. Springer, 293–303.

[13] Gerard Draper-Gil, Arash Habibi Lashkari, Mohammad Saiful Islam Mamun, and Ali A Ghorbani. 2016. Characterization of encrypted and vpn traffic using time-related. In *Proceedings of the 2nd international conference on information systems security and privacy (ICISSP)*. 407–414.

[14] Parvez Faruki, Vijay Laxmi, Manoj Singh Gaur, and P Vinod. 2012. Mining control flow graph as api call-grams to detect portable executable malware. In *Proceedings of the Fifth International Conference on Security of Information and Networks*. 130–137.

[15] Daniel Gibert, Carles Mateu, and Jordi Planes. 2020. The rise of machine learning for detection and classification of malware: Research developments, trends and challenges. *Journal of Network and Computer Applications* 153 (2020), 102526.

[16] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. 2017. Neural message passing for quantum chemistry. In *International Conference on Machine Learning*. PMLR, 1263–1272.

[17] Liyu Gong and Qiang Cheng. 2019. Exploiting edge features for graph neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 9211–9219.

[18] Mehadi Hassen and Philip K Chan. 2017. Scalable function call graph-based malware classification. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*. 239–248.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[20] Haodi Jiang, Turki Turki, and Jason TL Wang. 2018. DLGraph: Malware detection using deep learning and graph embedding. In *2018 17th IEEE international conference on machine learning and applications (ICMLA)*. IEEE, 1029–1033.

[21] Xiaodong Jiang, Ronghang Zhu, Sheng Li, and Pengsheng Ji. 2020. Co-embedding of Nodes and Edges with Graph Neural Networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2020).

[22] Teenu S John, Tony Thomas, and Sabu Emmanuel. 2020. Graph Convolutional Networks for Android Malware Detection with System Call Graphs. In *2020 Third ISEA Conference on Security and Privacy (ISEA-ISAP)*. IEEE, 162–170.

[23] Joris Kinable and Orestis Kostakis. 2011. Malware classification based on call graph clustering. *Journal in computer virology* 7, 4 (2011), 233–245.

[24] Thomas Kipf, Ethan Fetaya, Kuan-Chieh Wang, Max Welling, and Richard Zemel. 2018. Neural relational inference for interacting systems. In *International Conference on Machine Learning*. PMLR, 2688–2697.

[25] Thomas N Kipf and Max Welling. 2017. Semi-supervised classification with graph convolutional networks. In *5th International Conference on Learning Representations (ICLR)*.

[26] Arash Habibi Lashkari, Andi Fitriah A Kadir, Laya Taheri, and Ali A Ghorbani. 2018. Toward developing a systematic approach to generate benchmark android malware datasets and classification. In *2018 International Carnahan Conference on Security Technology (ICCST)*. IEEE, 1–7.

[27] Lu Lin and Hongning Wang. 2020. Graph Attention Networks over Edge Content-Based Channels. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 1819–1827.

[28] AL-Hawawreh Muna, Nour Moustafa, and Elena Sitnikova. 2018. Identification of malicious activities in industrial internet of things based on deep learning models. *Journal of information security and applications* 41 (2018), 1–11.

[29] Paul Prasse, Lukáš Machlica, Tomáš Pevnỳ, Jiří Havelka, and Tobias Scheffer. 2017. Malware detection by analysing network traffic with neural networks. In *2017 IEEE Security and Privacy Workshops (SPW)*. IEEE, 205–210.

[30] Lukas Ruff, Robert Vandermeulen, Nico Goernitz, Lucas Deecke, Shoaib Ahmed Siddiqui, Alexander Binder, Emmanuel Müller, and Marius Kloft. 2018. Deep one-class classification. In *International conference on machine learning*. PMLR, 4393–4402.

[31] Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne Van Den Berg, Ivan Titov, and Max Welling. 2018. Modeling relational data with graph convolutional networks. In *European semantic web conference*. Springer, 593–607.

[32] Chao Shang, Qinqing Liu, Ko-Shin Chen, Jiangwen Sun, Jin Lu, Jinfeng Yi, and Jinbo Bi. 2018. Edge attention-based multi-relational graph convolutional networks. *arXiv preprint arXiv:1802.04944* (2018).

[33] Martin Simonovsky and Nikos Komodakis. 2017. Dynamic edge-conditioned filters in convolutional neural networks on graphs. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 3693–3702.

[34] Laya Taheri, Andi Fitriah Abdul Kadir, and Arash Habibi Lashkari. 2019. Extensible android malware detection and family classification using network-flows and API-calls. In *2019 International Carnahan Conference on Security Technology (ICCST)*. IEEE, 1–8.

[35] David MJ Tax and Robert PW Duin. 2004. Support vector data description. *Machine learning* 54, 1 (2004), 45–66.

[36] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2018. Graph attention networks. In *6th International Conference on Learning Representations (ICLR)*.

[37] Felix Wu, Amauri Souza, Tianyi Zhang, Christopher Fifty, Tao Yu, and Kilian Weinberger. 2019. Simplifying graph convolutional networks. In *International conference on machine learning*. PMLR, 6861–6871.

[38] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. 2020. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems* (2020).

[39] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. 2020. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems* (2020).

[40] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How powerful are graph neural networks?. In *7th International Conference on Learning Representations (ICLR)*.

[41] Keyulu Xu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken-ichi Kawarabayashi, and Stefanie Jegelka. 2018. Representation learning on graphs with jumping knowledge networks. In *International Conference on Machine Learning*. PMLR, 5453–5462.

[42] Xikun Zhang, Chang Xu, Xinmei Tian, and Dacheng Tao. 2019. Graph edge convolutional neural networks for skeleton-based action recognition. *IEEE transactions on neural networks and learning systems* 31, 8 (2019), 3047–3060.

A HYPER-PARAMETER OPTIMIZATION

For the sake of a fair comparison, hyper-parameters of all models are optimized by a grid search based on performance on a separate validation set. For each algorithm, we consider a set of possible values for each of its adjustable hyper-parameters. While some common choices can be found in the literature, for the remaining hyper-parameters, we determine potential values, which seem most promising within the available computational budget. The considered hyper-parameter values for all supervised models can be found in Table 3. The values considered for the unsupervised models are provided in Table 4.

Algorithm	Parameter	Values
Support Vector Machine (SVM)	C	$[2^{-7}, 2^{-6}, \ldots, 2^{7}]$
	γ (RBF-kernel)	$[2^{-7}, 2^{-6}, \ldots, 2^{7}]$
k-Nearest Neighbor Classifier (KNN)	num. neighbors	$[1, 2, 3, 5, 8, 13, 21]$
Decision Tree (DT)	max. depth	$[2, 5, 10, None]$
	max. features	$[sqrt, None]$
Random Forest (RF)	num. estimators	$[10, 100, 1000]$
	criterion	$[entropy, gini]$
	max. features	$[sqrt, None]$
Adaboost (ADA)	num. estimators	$[10, 100, 1000]$
	learning rate	$[1e-3, 1e-2, 1e-1, 1]$
Multi-Layer Perceptron (MLP)	num. layers	$[1, 2]$
	num. hidden	$[16, 32, 64, 128]$
	L2-reg.	$[0, 1e-1, 1e-2, 1e-3, 1e-4]$
NF-GNN-CLF (ours)	num. layers	$[1, 2]$
	num. hidden	$[16, 32, 64, 128]$
	learning rate	$[1e-3, 1e-2]$
	dropout prob.	$[0, 0.2, 0.4, 0.6]$
	pool	$[mean, add, max]$

Table 3: Hyper-parameter values used in grid search for supervised algorithms.

Algorithm	**Parameter**	**Values**
One-class SVM (OC-SVM)	ν	$[1e-2, 1e-1]$
	γ (RBF-kernel)	$[2^{-10}, 2^{-9}, \ldots, 2^{10}]$
Local Outlier Factor (LOF)	num. neighbors	$[1, 2, 3, 5, 8, 13, 21]$
Kernel Density Estimation (KDE)	bandwidth	$[2^{0.5}, 2, \ldots, 2^{5}]$
Isolation Forest (IF)	num. estimators	$[10, 100, 1000]$
	max. features	$[256, None]$
Autoencoder (MLP-AE)	num. layers	$[1, 2]$
	num. hidden	$[16, 32, 64, 128]$
	L2-reg.	$[0, 1e-1, 1e-2, 1e-3, 1e-4]$
One-class MLP (MLP-OC)	num. layers	$[1, 2]$
	num. hidden	$[16, 32, 64, 128]$
	L2-reg.	$[0, 1e-1, 1e-2, 1e-3, 1e-4]$
NF-GNN-AE (ours)	num. layers	$[1, 2]$
	num. hidden	$[16, 32, 64, 128]$
	learning rate	$[1e-3, 1e-2]$
	dropout prob.	$[0, 0.2, 0.4, 0.6]$
	pool	$[mean, add, max]$
NF-GNN-OC (ours)	num. layers	$[1, 2]$
	num. hidden	$[16, 32, 64, 128]$
	learning rate	$[1e-3, 1e-2]$
	dropout prob.	$[0, 0.2, 0.4, 0.6]$
	pool	$[mean, add, max]$

Table 4: Hyper-parameter values used in grid search for unsupervised algorithms.