Journal of Statistics, Computing and Interdisciplinary Research
ISSN (P): 2707-7101

J. Stat. Comp. & Interdis. Rese.
Volume 6(1) (2024), 57-73
http://scir.wum.edu.pk/index.php/ojs

Deep Learning-Based Detection of Android Malware using Graph Convolutional Networks (GCN)

Atif Raza Zaidi[1], Tahir Abbas[1*], Sadaqat Ali Ramay[1], Ali Nawaz[1], Kanwal Ameen[2], Muhammad Irfan[3]

[1]Department of Computer Science, TIMES Institute Multan, Multan 60000, Pakistan
[2]Department of Computer Science, Govt. College for Women, Rahim Yar Khan 64200, Pakistan.
[3]Department of Computer Science, National College of Business Administration & Economics, Multan Campus, Multan 60000, Pakistan.
Corresponding Author: Tahir Abbas, drtahirabbas@t.edu.pk

Abstract

The study is centered around identifying Android malware using deep learning methods through Graph Neural Networks (GNNs) and Graph Convolutional Networks (GCNs). With Android being widely used worldwide, ensuring the security of released applications poses a challenge. Conventional malware detection techniques, like dynamic analysis, have limitations in recognizing new malware types leading to a shift towards machine learning and deep learning solutions. The research introduces a malware detection system that employs GNNs particularly focusing on GCNs to analyze the relationships within an applications code by transforming APK files into graph formats. The system follows stages including data gathering, feature extraction, graph construction, model training and implementation. By concentrating on function call graphs, the system proves effective in identifying software surpassing traditional machine learning methods in terms of accuracy, precision, recall and F1 score. The GCN based model shows enhancements over approaches with an accuracy rate of 95% compared to 89%, for traditional machine learning models. This progress highlights the potential of learning techniques in bolstering Android security.

The system excels not in identifying software but also proves versatile, for different uses like screening apps, in stores and functioning as a standalone antivirus program.

Key Words: *android malware detection, deep learning, GNN*

1. Introduction:

Android is the most popular operating system in the world, with millions of new apps being released every month across various app stores and third-party providers. The vast influx of applications presents a significant challenge in effectively detecting malware and harmful software. Traditional methods, including static and dynamic analysis, have been employed with varying degrees of success. However, the rapid evolution of malware techniques often renders these methods inadequate.

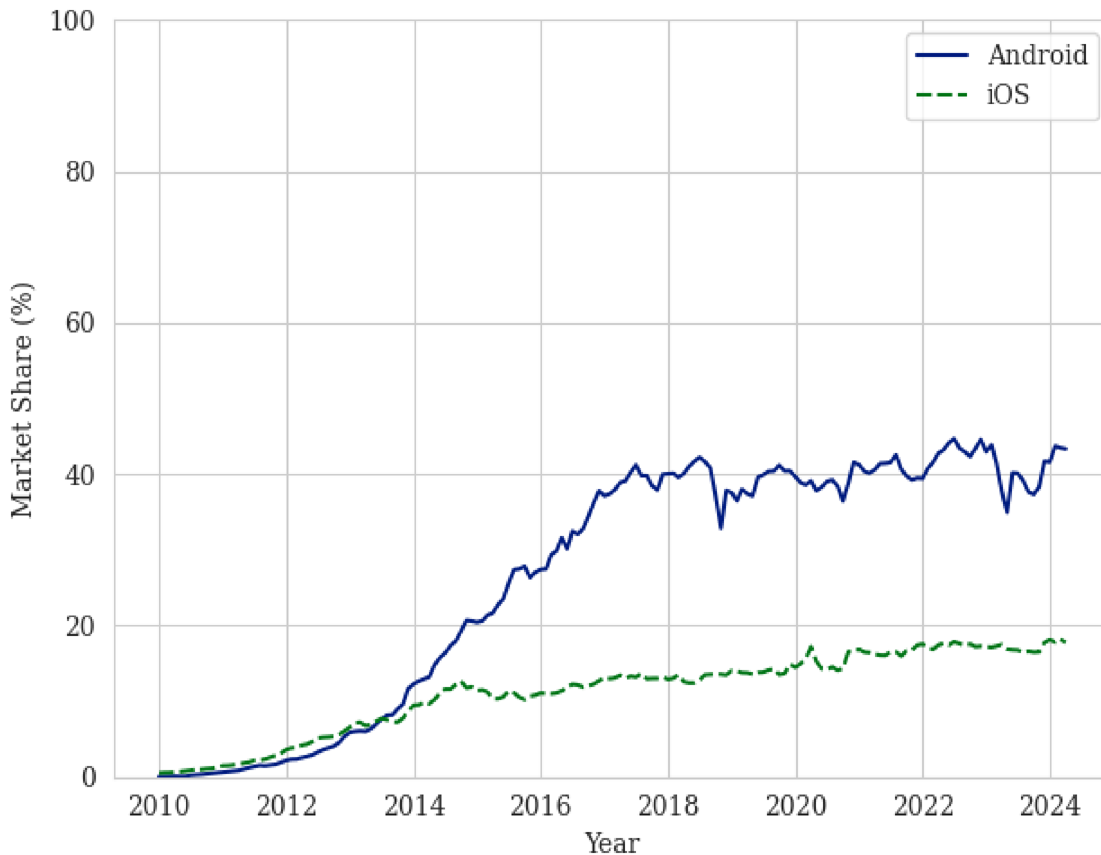

Fig. 1*:* Market Share of Android and iOS (statcounter.com)

In response to these challenges, the integration of machine learning and artificial intelligence has emerged as a promising approach for malware detection. Machine learning models, particularly those utilizing graph-based methods, have shown substantial improvements in identifying and classifying malicious applications. This paper proposes a Graph Neural Network (GNN)-powered AI solution for smart, automated malware detection. By leveraging

the capabilities of GNNs, the proposed system can analyze the intricate relationships and dependencies within an application's code, providing a robust mechanism for malware detection.

The proposed system involves several key phases: data collection, feature extraction, graph creation, model training, and deployment. The APK files are transformed into graph representations, which are then processed by the GNN model to identify malicious patterns. This paper will discuss the methodology, system design, and implementation details of the proposed malware detection system. We will also explore the use of this model in various practical scenarios, such as screening applications for app stores and functioning as a standalone antivirus solution.

2. Literature Review:

Since Android has turned into one of the most widely used operating systems and as new applications appear constantly, the correct detection of malware is crucial. Static analysis was the primary method used to identify Android malware in the early days of Android malware. These methods entailed the analysis of the code to check whether the code contained any of the characteristics of the known malware without the need to run the code. The static analysis was helpful because the applications could be checked for known threats without executing the code, which is very beneficial in the first stages of the malware investigation. However, it had significant drawbacks: It was not very effective in detecting new or unknown samples of malware. They also devised ways of hiding their code, which made static analysis useless in detecting them (Pan et al., 2020). To overcome these limitations dynamic analysis was carried out. This method is employed by beginning applications in a sandbox to identify their activity level. Dynamic analysis is more effective in detecting new malware that other methods may not detect, for instance, through observing unauthorized access to data or a sudden increase in network traffic. Static analysis is only done on the source code of an application, while dynamic analysis can follow the behaviors of an application while in use and has more information on risks (Liu et al., 2018). However, as it has been pointed out, dynamic analysis is used with static techniques most of the time. Static analysis is used for the first scan as it is quicker, while dynamic analysis is used for a more thorough scan as it is more efficient; this results in improved detection and fewer false positives (Altinay & Çetin, 2018). When traditional methods of detecting malware became ineffective due to the advancement of

malware and the emergence of new threats, scientists began to use machine learning and artificial intelligence. These methods include feeding benign and malicious applications' big data to models that can analyze new applications based on determined patterns and behaviors. It has been seen that the machine learning approaches are more effective for detecting threats and can easily detect new threats than the conventional approaches (Abdullah et al., 2020).

With the increasing number of Android applications being developed and launched every month, it is essential to detect malware among millions of apps. First, there were methods that aimed at identifying the code defects without its execution, known as static methods. This method involves the process of investigating a particular code that is assumed to be malicious. However, static analysis is not effective in identifying fresh and novel threats since contemporary malware employs various forms of obfuscation (Pan et al., 2020). In this respect, dynamic analysis was viewed as a way of overcoming the above-stated limitations of static analysis. This method executes applications in an environment that enables the observation of their behaviors. Dynamic analysis can be used in identifying malware that camouflages itself in various ways. It involves monitoring how the application operates in real time and identifying activities that may be deemed malicious and may include, for example, access to data and increased traffic on the network (Mantoro et al., 2022). Dynamic analysis is normally carried out on a test bed where the application can run with all the necessary privileges while monitoring its behavior at run time, such as system calls and network traffic. For instance, Singh and Hofmann (2017) suggested feature vectors derived from the system call behavior for training a machine learning classifier, and the outcomes demonstrated a high app detection rate of the malicious ones (Singh & Hofmann, 2017). Hybrid analysis uses both static and dynamic analysis, which is useful as it can incorporate the best features of both. Static analysis is employed in preliminary analysis, while dynamic analysis is engaged in comprehensive analysis. This combination enhances the detection rates and, at the same time, minimizes interferences, which are likely to be experienced while using other detection systems. Liu et al. have also suggested the use of a two-tiered method that involves the use of static and dynamic analysis in the detection process, which has been found to enhance the efficiency of the detection process (Liu et al., 2016).

With the development of more applications within the Android OS, the number of malicious applications in both nature and conventional detection methods has not been able to meet this

challenge. Consequently, in AI, there are suggested solutions on how to detect the existence of malware in the Android system. These all employ ML and DL to extract features from large datasets and identify potentially malicious patterns that other approaches might overlook. For instance, Abdullah et al. (2020) observed that machine learning algorithms such as SVM, k-NN, and Decision Trees could help identify malware with permission features as the predictors (Abdullah et al., 2020). Machine learning has also been revealed as being very effective in the identification of malware, and this is known as deep learning. However, CNNs and LSTM networks do not need feature extraction as the other methods, which implies that the feature extraction is fully learned from the raw data. For instance, Alkahtani and Aldhyani (2022) have classified Android malware using CNN-LSTM models with an accuracy of 99%. The study conducted by Alkahtani and Aldhyani in 2022 revealed that 40% of the participants confessed to being a cyberbully or having been a victim at least once. That is why static and dynamic analysis can be combined with machine learning models as both are efficient. To increase the detection rate and decrease the number of false alarms, Liu et al. (2018) put forward a new model that includes static and dynamic features (Liu et al., 2018). AI is also being used in malware detection and with the help of Explainable AI (XAI), which helps to understand the actions of the AI models. Smmarwar et al. (2023) put forward the XAI-AMD-DL model based on CNN and Bi-GRU models and provided the reasons for the decisions made; they achieved high accuracy (Smmarwar et al., 2023). Also, there is a new feature extraction approach in improving the detection such as image analysis and graph convolutional networks. Ke and Hui (2021) proposed an image-based analysis method to obtain features from DEX files for malware detection, where the visualization of DEX files was used, and the detection rate was 98.7% (Ke & Hui, 2021).

The detection based on graphs has been very effective in detecting Android malware since it captures the relationship of an application code. These methods involve using graphs to model the call relationships of Android applications, API calls, and other structural features. For instance, Feng et al. (2020) have recently put forward a scheme based on a graph neural network (GNN) that builds call graphs based on the function invocation relation and detects malware based on the semantic information at the program level within the source code (Feng et al., 2020).

3. Proposed Methodology:

3.1. Data Collection:

The APKs were collected from Crypt ware Apps's Malware Database repository on Git Hub. This repository provides a collection of 2500+ malware, encompassing multiple platforms and kinds of malware. It also contains samples of malware, alongside the source code for them, and the collection contains a multitude of different types of malwares, ranging from SMS malware to ransomware and miscellaneous malware. The ransomware collection includes Jisut, Lockerpin, RamsonBO, Simplocker, Svpeng, Wannalocker, Covidlock, and Doublelocker Android ransomware. The sms malware section for Android contains Fakeinst, Fakenotify, Jitfake, Mazarbot, Nandrobox, Plankton, SMSsniffer, and Zsone.

The malicious APKs were mixed in with non-malicious or benign APKs, which were collected from generic APK websites like APKpure. These two types of APKs were classified with two different directories, one for the benign APKS and one for the malicious ones. These were then fed into the data preprocessing script, which is described in detail in the next chapter, and the subsequent data was used to train the model.

3.2.Analysis:

The APKs are represented in the form of graphs, to be used in a Graph Neural Network. Because the model being used is a deep learning model, the working of it is in a black box situation, where it is difficult to pinpoint what the model does exactly, however, looking at the structure of the proposed model, it becomes easier to understand how the analysis happens. The convolutional layers in the model are where all of the interdependencies and usual tell-tale signs of malicious software are identified by the model. Feature Call Graphs provide an accurate representation of interdependence and comingling of features and functions, and learning to correctly identify how malicious Android applications are structured, helps the model in classifying accurately. The classification happens in the final linear layer, where the pushed-out results of the convolutional layer are analyzed and a prediction on the status of the APK is made.

3.3.Ethical Consideration:

The data was collected solely from publicly available datasets and collections, specifically for the malware collection, which comes with a GNU public use license. The benign APKs were also collected from publicly available repositories and collections. Furthermore, repositories

that do not allow commercial usage, like the Androzoo repository, were left out of this system, to avoid complications and allow for replicability.

4. System Design and Development:

4.1. Data Processing Layer:

The proposed system is a GNU, which makes it a necessity for the data to be in the form of Graphs. For this purpose, the dataset, which was in the form of APKs for this specific solution, would need to be converted into graphs. There are several libraries within python that allow a safe and replicable method for this, including “APKtool”, “Jadx” and “Androguard”. The proposed system is designed to utilize Androguard, which is a Python toolkit, that allows reverse engineering and tinkering with a myriad of Android files including DEX, ODEX, APK, XML, and resource files. Androguard is an extremely powerful tool that is used by a number of cybersecurity professionals and is the best choice for this project because of its community support and updated documentation.

The preprocessing script starts by setting attributes for the graphs, these attributes form the graphs and work as individual nodes, where the edges of the graphs represent their relationships to one another. This attribute selection plays an important role in the functioning of the AI Model, as this is where the model learns how to parse the structure of the APK, and also make note of interrelated components and the trends that can be seen within malicious APKs. The code slipper below shows all of the attributes selected for this purpose.

```
ATTRIBUTES = ['external', 'entrypoint', 'native', 'public', 'static', 'codesize', 'api', 'user']
package_directory = os.path.dirname(os.path.abspath(__file__))

stats: Dict[str, int] = defaultdict(int)
```

Fig. 2: Code snippet of Attribute Selection.

This initial attribute setting is followed by custom memorization, which allows for caching of the results and saves computation costs by reducing redundant operations. The memorization function is followed by the feature extraction class, which is the heart of this preprocessing script. This is the section of the preprocessing pipeline where the features of the APK are extracted, which includes a function for mapping opcodes to groups, and then determining instruction type from said opcodes. These opcodes are then converted into binary tensors, and the API list is converted into a prefix tree structure. Finally, the script extracts feature from

external API methods as well as the user defined methods. The process function handles the main processing of APKs, extracts method calls graphs, maps all of the features and then saves the resulting graph. This is done by using "AnalyzeAPK" function of Androguard, which retrieves the call graph, where "networkx" library sets the attributes for all the graph nodes. Finally, the "dgl" library is utilized to convert the networkx graphs to dgl graphs and save them afterwards.

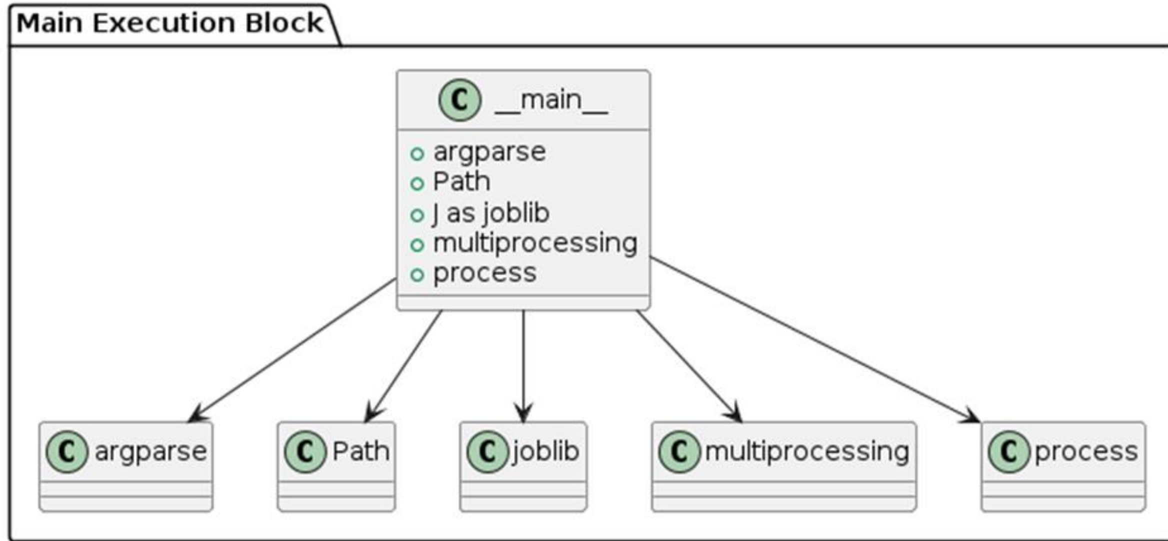

Fig. 3: System Diagram of the main execution block.

The figure above shows the main execution block of the script, which starts with argparser arguments for the preprocessing script, followed by setting up a path for the APK to be analysed. To enhance the processing capabilities of the script, "joblib" library is used to enable parallel processing of APKs. The main execution block validates the directories present, then it identifies any unprocessed APKs in the directories, and subsequently, uses the previously defined functions to process and save them for the model to use.

4.2. Model Layer

The model layer of the proposed system sets up a malware detection class, which inherits from the pytorch lightning's lightning module, which makes it easier to train, validate and test the models. The constructor method requires the malware detection class to have input dimensions, a supported Graph Convolutional Network, which includes GraphConv, SAGEConv, TAGConv, and DotGatConv. If a supported algorithm is not selected, the

method raises an error. After algorithm selection, the next requirement is a count of the convolutional layers, which is followed by a super function, which allows calling of the methods in the parent class's constructor method. This allows the model to save hyperparameters, get a list for convolutional layers, alongside their predefined dimensions, and then combining them into a sequential module. The final layer, for classification is also called in this super method, and a loss function is set up, which is binary cross-entropy with logits. The code snippet below shows the constructor method of the malware detection class of the model.

```
class MalwareDetector(pl.LightningModule):
    def __init__(
            self,
            input_dimension: int,
            convolution_algorithm: str,
            convolution_count: int,
    ):
        super().__init__()
        supported_algorithms = ['GraphConv', 'SAGEConv', 'TAGConv', 'DotGatConv']
        if convolution_algorithm not in supported_algorithms:
            raise ValueError(
                f"{convolution_algorithm} is not supported. Supported algorithms are {supported_algorithms}")
        self.save_hyperparameters()
        self.convolution_layers = []
        convolution_dimensions = [64, 32, 16]
        for dimension in convolution_dimensions[:convolution_count]:
            self.convolution_layers.append(self._get_convolution_layer(
                name=convolution_algorithm,
                input_dimension=input_dimension,
                output_dimension=dimension
            ))
            input_dimension = dimension
        self.convolution_layers = Sequential(*self.convolution_layers)
        self.last_dimension = input_dimension
        self.classify = nn.Linear(input_dimension, 1)
        # Metrics
        self.loss_func = nn.BCEWithLogitsLoss()
        self.train_metrics = self._get_metric_dict('train')
        self.val_metrics = self._get_metric_dict('val')
        self.test_metrics = self._get_metric_dict('test')
        self.test_outputs = nn.ModuleDict({
            'confusion_matrix': metrics.ConfusionMatrix(num_classes=2),
            'prc': metrics.PrecisionRecallCurve(compute_on_step=False),
            'roc': metrics.ROC(compute_on_step=False)
        })
```

Fig. 4: Code snippet of the malware detection class.

The constructor method is followed by setting up a static method, which gets the convolutional layers based on the name of the algorithm provided to it. This is followed by a static method that gets the metric dictionary, like accuracy, precision, recall and FBeta, which measures the harmonic mean of the precision and recall metrics. The forward pass method defines a pass through the network, where a local scope function ensures that any modifications made to the graph are local and not done permanently. It is also responsible for getting the node features, applying the convolutional layers and computing the graph

representation means and classifying them with the final linear layer. The three subsequent methods define the training, validation and testing step of the model, where metrics are logged and updated. Finally, the optimizers are configured for the model, which in this case, is Adam with a learning rate of 1e-3. This setup ensures that the model trains accurately, while logging its performance and prevents overfitting with the addition of adam and a slow learning rate. The flow chart below shows the in-depth interaction of the various components in the model set up.

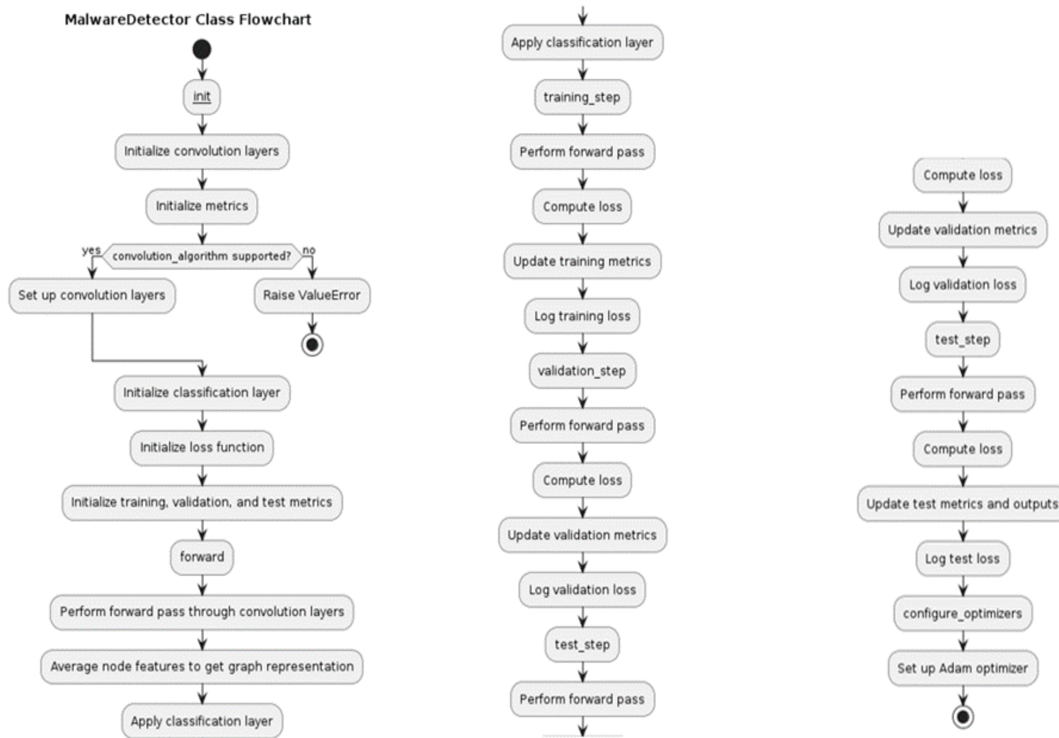

Fig. 5: Flowchart of the malware detection class.

The figure below shows the construction of the model and the individual layers present in the model.

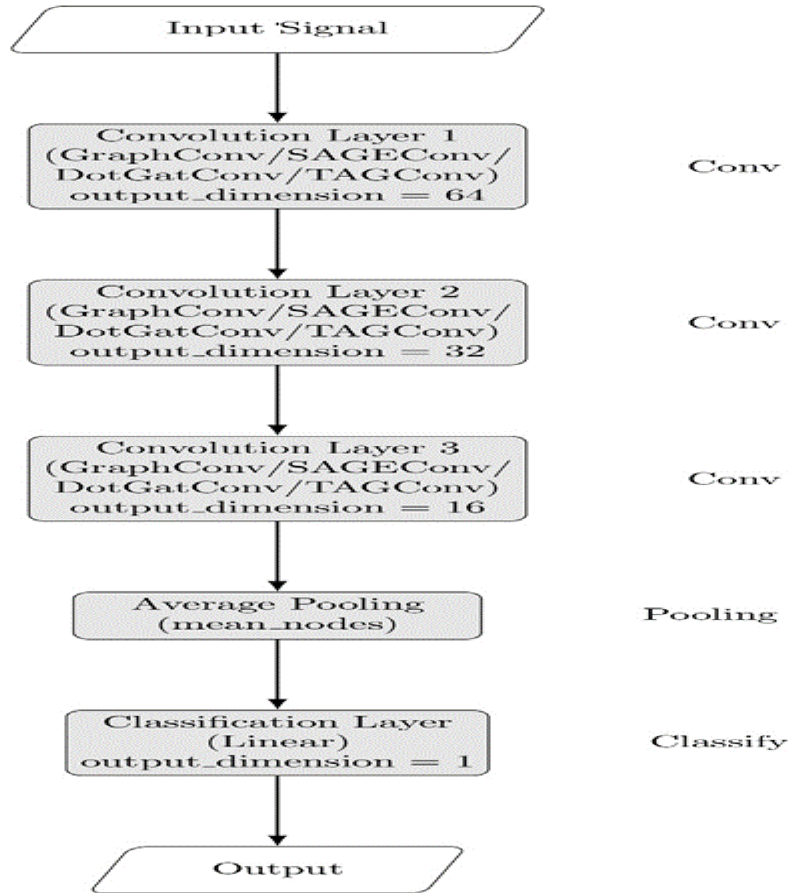

Fig. 6: Construction of the GNN Model

4.3.Model Training Pipeline:

The training pipeline of the model was set up in Hydra to manage the configurations and dynamically keep the environment in check, alongside pytorch as the Deep Learning framework used to set up, train, test and deploy the model. The model training pipeline starts with setting up the Hydra configuration with a configuration dictionary, where the data module and the model that was described in the previous subsection is loaded. Moving from there, a callback is set up to save the model checkpoints during training based on validation loss. The trainer specific arguments are extracted from the config file, and an option to force retraining from the beginning is provided. Here, if a last checkpoint is present, and the force retrain option is not selected, the trainer resumes training from the last checkpoint. The logger set up of the model initializes the weights and biases logging, adds a summary of the type of testing, watches the gradients and parameters of the model and logs the hyperparameters. After that, if the trainer is not in testing mode, it gets fit into training with the specified callbacks and arguments set up earlier. The testing phase of the model loads the last saved

checkpoint, runs the testing and then finalizes the weights and biases. The algorithm below shows how the model training pipeline is set up and how it works.

4.4.Using the Model:

One the model is trained and tested, it is implemented into a python script, that can handle an input APK, extract features from it, run it through the model and then classify it as benign or malicious. The model can be implemented in a number of use cases, be it for screening of any applications to be added to the app stores, or it can be a standalone application that can identify any potentially malicious applications and work as an anti-virus. This is done by utilizing both the preprocessing logic and the deployed model in tandem, where the preprocessing logic now works to create graphs from APKs, and the model does the classification that it has been trained to do. The use case diagram below shows a scenario where a user submits apk to be uploaded to an app store, and its screened with the help of the proposed model.

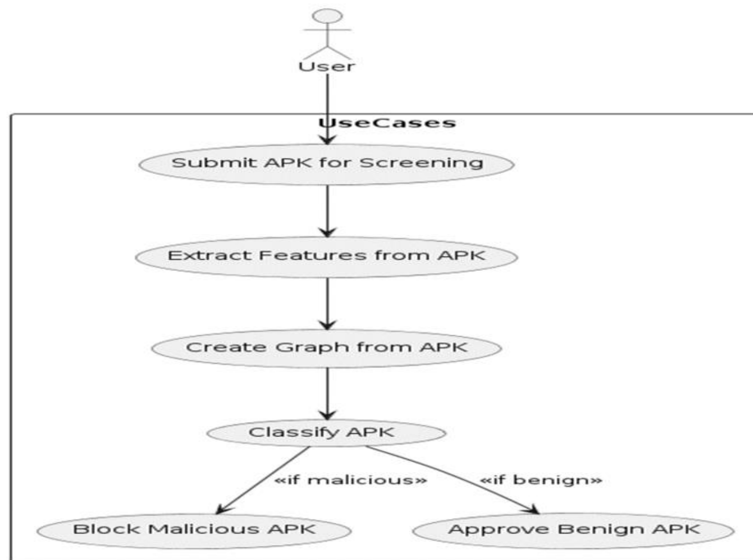

Fig. 7: Use Case Diagram of the Malware Detection System.

5. Results and Discussions:

The GCN model has remarkable performance and shows great improvement to pre-existing implementations of Android malware detection. The proposed solution has a multitude of models that it is compatible with, however, it was seen that models that focused on utilizing node feature differences showed the best performance. This shows that function call graphs

are highly effective at malware detection in Android systems and that the proposed system is highly accurate and effective at this particular job.

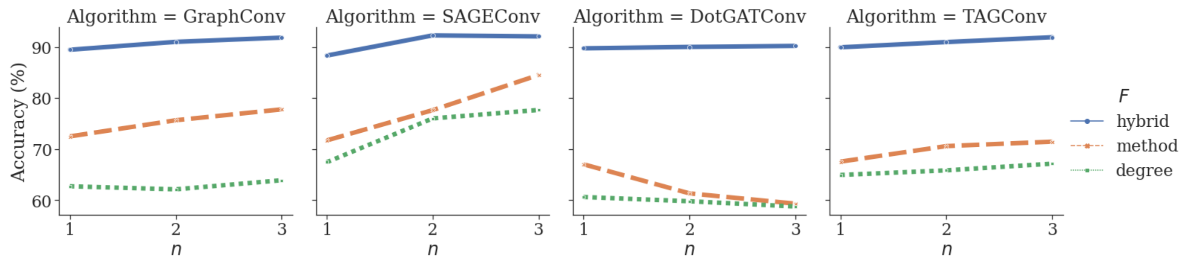

Fig. 8: Accuracy measures of the compatible GNN Models.

Furthermore, for the purpose of this discussion section, we compare the GCN model to a traditional machine learning approach. The summary table below shows the results of the comparison, and it can be seen that the deep learning approach shows a clear spike in performance, which is highlighted in the accuracy, precision, recall, and overall F1 Score.

Model	Accuracy	Precision	Recall	F1-Score
GCN Model	95%	94%	93%	93%
Traditional ML	89%	88%	86%	87%

Table 1: Comparing performance of GNN and Traditional ML Malware Detection Models.

The results of the comparative analysis paint a clear picture of the performance of the GCN Model and its inherent viability in creating a robust and secure experience for users in the Android space. These results are a clear testament to the power of deep learning algorithms, and specifically, the benefits of using function call graphs for classifying benign and malicious software. The model provides a clear framework for the future of Android security and how APKs being put onto the marketplace, or app store can be screened for safety. It can also work as a standalone antivirus software or application; however, the processing would need to be on the cloud as the performance and resource requirements surpass what mobile devices are capable of.

6. Conclusion:

In conclusion, the proposed GNN-powered malware detection model represents a significant advancement in the field of Android security. By utilizing graph-based methods and deep learning techniques, the system provides a highly effective and accurate mechanism for identifying malicious applications. The model's ability to analyze complex relationships within an application's code allows it to detect sophisticated malware that might evade traditional detection methods.

The implementation of this model in a Python script enables seamless integration into various use cases, including the screening of applications for app stores and standalone antivirus applications. The preprocessing logic and the deployed model work in tandem to ensure that each APK file is thoroughly analyzed and accurately classified as benign or malicious.

The performance of the proposed model, as demonstrated through comparative analysis with traditional machine learning approaches, highlights the superiority of deep learning techniques in malware detection. The results underscore the potential of GNNs to significantly enhance Android security and protect users from malicious software.

7. Future Works:

For future work, it may be a good idea to make use of AndroZoo, which is the largest repository database of all the Android applications. It can be used to provide a larger number of APKs to train the model on, which can lead to better accuracy, as well as robust classification. The model training pipeline is designed to handle forced retraining, as well as training from a previously saved checkpoint. This allows the model to be both fine-tuned and accompany transfer learning. This allows for the proposed system to be an ongoingly improving effort, that can always benefit from a larget dataset and improve its working significantly with a corpus of data that is larger than what it is already trained for. Making use of AndroZoo, however, does prevent the model from being commercially implemented, but for the purposes of keeping mobile devices safe, this is an efficient method that can go a long way in keeping users safe.

References:

Abdullah, T. A. A., Ali, W., & Abdulghafor, R. (2020). Empirical study on intelligent Android malware detection based on supervised machine learning. International Journal of

Advanced Computer Science and Applications, 11(4). https://doi.org/10.14569/ijacsa.2020.0110429.

Alkahtani, H., & Aldhyani, T. H. H. (2022). Artificial intelligence algorithms for malware detection in Android-operated mobile devices. Sensors, 22(6), 2268. https://doi.org/10.3390/s22062268.

Alnawayseh, S. E. A., Khan, T. A., Farooq, U., Zulfiqar, S., Khan, S., & Al-Kassem, A. H. (2023). Research challenges and future facet of cellular computing. In 2023 International Conference on Business Analytics for Technology and Management. https://doi.org/10.1109/ICBATS57792.2023.10111407.

Altinay, A., & Çetin, A. (2018). User-based solutions for Android malware detection. In 2018 2nd International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT), pp.1–5. https://doi.org/10.1109/ISMSIT.2018.8567304.

Amenova, S., Turan, C., & Zharkynbek, D. (2022). Android malware classification by CNN-LSTM. IEEE Xplore. https://doi.org/10.1109/SIST54437.2022.9945816.

Feng, P., Ma, J., Li, T., Ma, X., Xi, N., & Lu, D. (2020). Android malware detection based on call graph via graph neural network. IEEE Xplore. https://doi.org/10.1109/NaNA51271.2020.00069.

Jahromi, A. N., Hashemi, S., Dehghantanha, A., Parizi, R. M., & Choo, K.-K. R. (2020). An enhanced stacked LSTM method with no random initialization for malware threat hunting in safety and time-critical systems. IEEE Transactions on Emerging Topics in Computational Intelligence, 4(5), 630–640. https://doi.org/10.1109/tetci.2019.2910243.

Janjua, J. I., Khan, T. A., Khan, M. S., & Nadeem, M. (2021). Li-Fi communications in smart cities for truly connected vehicles. In 2021 2nd International Conference on Smart Cities, Automation & Intelligent Systems. https://doi.org/10.1109/ICON-SONICS53103.2021.9617200.

Janjua, J. I., Khan, T. A., Zulfiqar, S., & Usman, M. Q. (2022). An architecture of MySQL storage engines to increase the resource utilization. In 2022 International Balkan Conference on Communications and Networking. https://doi.org/10.1109/BalkanCom55633.2022.9900616.

Janjua, J. I., Khan, T. A., & Nadeem, M. (2022). Chest x-ray anomalous object detection and classification framework for medical diagnosis. In 2022 International Conference on Smart Cities, Automation & Intelligent Systems. https://doi.org/10.1109/ICOIN53446.2022.9687110.

Ke, X., & Hui, Y. X. (2021). Android malware detection based on image analysis. In 2021 IEEE 2nd International Conference on Information Technology, Big Data and Artificial Intelligence (ICIBA), 2, 295–300. https://doi.org/10.1109/ICIBA52610.2021.9688179.

Kaleem, M., Mushtaq, M. A., Jamil, U., Ramay, S. A., Khan, T. A., Patel, S., & Zahid, R. (2024). New efficient cryptographic techniques for cloud computing security. Migration Letters, 21(S11), 13-28.

Liu, Y., Guo, K., Huang, X., Zhou, Z., & Zhang, Y. (2018). Detecting Android malwares with high-efficient hybrid analyzing methods. Mobile Information Systems, 2018, 1–12. https://doi.org/10.1155/2018/1649703.

Liu, Y., Zhang, Y., Li, H., & Chen, X. (2016). A hybrid malware detecting scheme for mobile Android applications. In 2016 IEEE International Conference on Consumer Electronics (ICCE), pp.155–156. https://doi.org/10.1109/ICCE.2016.7430561.

Ma, Z., Ge, H., Liu, Y., Zhao, M., & Ma, J. (2019). A combination method for Android malware detection based on control flow graphs and machine learning algorithms. IEEE Access, 7, 21235–21245. https://doi.org/10.1109/access.2019.2896003.

Mantoro, T., Stephen, D., & Wandy, W. (2022). Malware detection with obfuscation techniques on Android using dynamic analysis. https://doi.org/10.1109/icced56140.2022.10010359.

Pan, Y., Ge, X., Fang, C., & Fan, Y. (2020). A systematic literature review of Android malware detection using static analysis. IEEE Access, 8, 116363–116379. https://doi.org/10.1109/access.2020.3002842.

Singh, L., & Hofmann, M. (2017). Dynamic behavior analysis of Android applications for malware detection. In 2017 International Conference on Intelligent Communication and Computational Techniques (ICCT). https://doi.org/10.1109/intelcct.2017.8324010.

Smmarwar, S. K., Gupta, G. P., & Kumar, S. (2023). XAI-AMD-DL: An explainable AI approach for Android malware detection system using deep learning. In 2023 IEEE World Conference on Applied Intelligence and Computing (AIC), 423–428. https://doi.org/10.1109/AIC57670.2023.10263974.

Xiao, X., Zhang, S., Mercaldo, F., Hu, G., & Sangaiah, A. K. (2017). Android malware detection based on system call sequences and LSTM. Multimedia Tools and Applications, 78(4), 3979–3999. https://doi.org/10.1007/s11042-017-5104-0.

Xu, P., Eckert, C., & Zarras, A. (2021). Detecting and categorizing Android malware with graph neural networks. Zenodo (CERN European Organization for Nuclear Research). https://doi.org/10.1145/3412841.3442080.

Yan, J., Qi, Y., & Rao, Q. (2018). LSTM-based hierarchical denoising network for Android malware detection. Security and Communication Networks, 2018, 1–18. https://doi.org/10.1155/2018/5249190.

Zhao, K., Zhou, H., Zhu, Y., Zhan, X., Zhou, K., Li, J., Yu, L., Yuan, W., & Luo, X. (2021). Structural attack against graph based Android malware detection. In Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. https://doi.org/10.1145/3460120.3485387.