Towards Scalable Android Malware Detection Through Function Call Graph Pruning

Pascal Duncan Lek Hou U[1]

MSc Information Security

Dr. Lorenzo Cavallaro

Submission date: 8 September 2026

Abstract

Function call graph representations of Android applications can become computationally expensive to operate on by graph neural networks (GNNs) to process as programs contain increasing numbers of functions and function invocations. Graph pruning offers one avenue for reducing this complexity by removing large portions of the graph. However, the effects on detection performance, scalability, sensitivity to graph size and quality of explanation remain underexplored topics. This dissertation evaluates whether graph pruning can provide a practical basis for scalable Android malware detection while preserving predictive performance.

Two pruning methods are considered; sensitive function call graph pruning (sFCG) and Leaf Pruning (LP). They are evaluated against unpruned FCGs using two GNN architectures: GCN and GraphSAGE. This study makes use of a temporally ordered subset of the Hypercube dataset under spatio-temporal constraints laid out by TESSERACT. We propose the use of a node embedding structure combining opcode2vec embeddings, FastText-based API embeddings, and one-hot permission features as an extensive node enrichment technique. Evaluation considers a suite of metrics including malware F1, Area Under Time (AUT), size-stratified harmonic score and fidelity characterisation, along with training time and graph size reduction rates.

The results demonstrate that graph pruning does not induce a uniform improvement in detection performance, but is rather a trade-off between scalability, performance and explainability. sFCG is found to reduce computational cost by decreasing average epoch training time from 1771 to 586 seconds for GCN and 1675 to 574 seconds for GraphSAGE, though neither pruning method improves upon the unpruned baseline malware F1 scores. Further, our results show no convincing evidence that pruning becomes increasingly beneficial as original FCG size grows. However, we find a substantial effect of pruning on explanation faithfulness under GNNExplainer, with the GCN – sFCG pairing achieving a characterisation score of 76.4% at $K = 1,000$ explanation edges, compared to the baseline GCN – unpruned pairing achieving 15.8%. These findings show that graph pruning should not be treated as a lossless preprocessing step, but rather as a balancing choice between scalability, predictive performance, and explanation faithfulness.

Declaration

I, Pascal Duncan Lek Hou U, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the thesis.

Signed 

Date September 5, 2026

Statement of Use on Generative AI

Generative AI was used throughout this dissertation in the following capacities:

- Generating and improving existing code for the following purposes:
  - Assisting with in debugging code.
  - Assisting with converting pseudocode into Python code.
  - Converting sequential code into parallelised processes.
  - Implementing efficient CUDA processing for faster GPU inference.
  - Suggesting and implementing efficient methods for generating and handling large-scale dataset pipelines.
  - Assisting with generating scripts for processing data points.
- Help in writing the report in an assistive capacity for the following purposes:
  - Gathering and suggesting literature to review.
  - Assisting with constructing the overall direction and flow of chapters.
  - Assisting with formatting equations.
  - Generating tables and graphs, with figures and values manually verified.
  - Criticising and evaluating prior drafts of this report.

Contents

Chapter 1

Introduction

The study of Android malware and its detection is a widely researched field, yet malware persists as a significant problem within the cybersecurity space. One of the central issues with malware detection is that malware authors frequently employ obfuscation tactics to hide malicious executions among abstract code enabling classifiers to misidentify them as benign software [19].

To that end, malware detectors must be able to identify indicators of malicious behaviour under such adversarial circumstances. Many modern methods employ machine learning models trained on abstract representations of programs to capture the fundamental features of their underlying objectives [20]. One technique in this regard is graph representation learning (GRL) [31, 6], which trains a classifier to learn graph representations of programs to identify structural and semantic features of different malware programs as opposed to benign software. One of the most common representation techniques is to identify a program's structure as a graph of functions invoking other functions, commonly known as function call graphs (FCGs).

The adversarial dynamic between malware authors and researchers naturally leads to degradation of detection techniques over time, known in machine learning as concept drift [15, 35, 16], requiring new methods to compete against evolving forms of malware. Further, programs can have highly complex structures that need to be represented by many features, requiring expensive computations. This complexity can also be counterproductive when an adversary is able to hide malicious portions of a malware within a sea of obfuscatory data.

One approach to reducing the computational burden of large graphs is graph pruning, where nodes and edges considered less relevant to the downstream task are removed. The central challenge is sufficiently reducing the graph while preserving struc-

tural and semantic information critical to its classification. Some methods implement rules for retaining nodes and edges, and removing parts of the graph which do not fit those rules; this is known as graph sparsification [21, 24, 30].

While pruning can reduce the computational complexity of training and inference, removing excess information not relevant to the classification is also hypothesized to improve the model's ability to identify malware. However, this has yet to be demonstrated conclusively for malware detection.

1.1 Motivation

Motivated by the need for a more comprehensive understanding of how graph pruning can affect GNN-based malware classifiers, this dissertation studies the effects of pruning on FCGs and evaluates their practical use with modern large-scale Android datasets. Many works incorporate graph pruning in some form; however, only a few implement it in a way that is both scalable and explainable. Further, many works suffer from biased and unrealistic evaluation settings which are not translatable to real-world malware distributions.

The work presented in this dissertation evaluates two pruning techniques – Sensitive Function Call Graphs (as formulated by Liu et al. [21]) and Leaf Pruning (as formulated by Mohammadian et al. [24]) – against the baseline unpruned FCG. To ensure a fair evaluation, each pruning model uses the same graph generation and node embedding pipeline, and is evaluated with two GNN models: GraphSAGE [10] and GCN [17].

1.1.1 Threat Model

We consider Android malware authors as the identified adversary. Thus, our threat model suggests the following capabilities:

- The author controls the application's own code and alters internal functions, call structures, and opcode sequences.
- They may use obfuscation tactics and increase the complexity of the code to hide malicious functionality.
- They can select which Android APIs and permissions their application uses, provided they maintain the intended malicious behaviour.

Under these circumstances, we cannot arbitrarily select dataset partitions which may be favourable for our classification task. We must utilise a dataset representative of the overall Android ecosystem. Thus, our techniques must be generalisable across different Android malware structures and resistant to obfuscation.

1.2 Summary of Work

We consider each pruning method using several key metrics, including F1 score, Area Under Time (AUT), false-positive rate, recall and precision. Further, we evaluate the predictive ability of each model using stratified buckets of similarly sized FCGs, and propose a novel metric to indicate the model's classification sensitivity to graph size. We evaluate the usefulness of the graph explanations generated by GNNExplainer by analysing their suitability for identifying malicious subgraphs using fidelity+ and fidelity− metrics. To the best of our knowledge, we have found no prior work which evaluates an Android malware detection framework against the fidelity metrics outlined by GraphFramEx [2].

1.2.1 Hypothesis

Our primary hypothesis is that pruning FCGs reduces the information available for training and will thus reduce classification performance. However, the impact of pruning may be nuanced and depend not only on the type of pruning implemented, but also the structure of the sample being classified.

1.2.2 Research Questions

The following research questions are explored in this dissertation.

- **RQ1** – **Scalability**: How much do sFCG and Leaf Pruning reduce FCG size and computational cost relative to unpruned FCGs?
- **RQ2** – **Performance**: What effect does pruning have on temporally realistic malware-detection performance?
- **RQ3** – **Sensitivity to Graph Size**: Does pruning alter the classification performance when graph sizes differ?

- **RQ4 – Explainability**: Can pruning affect GNN explanations such that they alter the faithfulness to the model's classification?

1.2.3 Contributions

The main contributions of this dissertation are as follows:

- A unified methodology for applying and directly comparing pruned and unpruned FCGs under the same graph-generation, embedding and GNN pipeline.
- A graph representation model combining Word2Vec, FastText, and one-hot permission features to enrich FCG node embeddings.
- An empirical evaluation of graph pruning for Android malware detection, comparing sensitive function call graph (sFCG) pruning and Leaf Pruning (LP) against an unpruned FCG baseline, conducted across both GCN and GraphSAGE encoder architectures.
- An evaluation of the scalability-performance trade-offs introduced by pruning and measuring graph reduction, training-time reduction, malware detection performance, and temporal performance.
- An analysis of sensitivity of pruning to original FCG size, evaluating the effect of pruning as FCGs become larger, using a novel size-stratified metric to summarise consistency of performance across graph-size ranges.
- An evaluation of how graph pruning affects the faithfulness of GNN explanations, measured using fidelity metrics.

Chapter 2

Background

Malware detection has undergone many phases in the history of the field. Traditionally, detection was done through the use of signature matching, whereby digital fingerprints of suspected programs were compared with signatures of already known malware. If a program's signature matches the signature of a known malware, that program is identified as a sample of that malware [33]. The consequence of this form of detection is that any new malware which has never been encountered before (i.e, zero-day samples) will have a new signature which has never been documented. Second, malware authors may modify parts of their programs without affecting the program semantics while still executing malicious content, thereby circumventing detection.

2.1 Heuristic-Based Detection

This led researchers to reason that the patterns of behaviour (heuristics) of malware would be more effective against zero-days and obfuscation techniques. This means applying human-defined rules or indicators for classification, rather than exact signatures. The reasoning for this is that malware should typically behave in certain ways that make it identifiable among goodware. Heuristics can be identified within two broad sets of features: **Static-features**, which can be obtained by analysing program files, and include permissions, embedded API calls, the manifest file's activity and broadcast receivers; and **dynamic-features**, which can be retrieved by executing the application in a configured environment, and include system calls, network activity, battery and CPU usage, and file operations. More advanced heuristic based detectors analyse both static and dynamic features. Combining the two in a single detection cycle has shown to improve accuracy compared with using the feature sets separately

[3, 23].

Knowledge-based heuristics require expert knowledge of what features are prevalent in malware, in order to define rules for what programs or files are likely to be malware by observing patterns and similarities. This naturally means that those rules can become outdated very soon in an adversarial space. Yunmar et al. [37] identify that many studies rely on outdated datasets that are not representative of modern malware. Therefore, to keep up with the evolving threat landscape, instead of using human-defined rules, researchers have begun leveraging machine learning and deep learning classifiers for malware detection tasks and have proved to be effective in achieving high performance metrics. Although these models suffer from the same problem of datasets becoming quickly outdated, training new models on new datasets becomes primarily a matter of computational resources, rather than human capital.

2.2 Feature Selection and Representation

The adoption of machine learning detection methods thus requires a different view of which features should be important, and marks a shift from heuristic-based approaches, towards more statistical analysis of malware behaviours. Information about a program is represented through its features. We have briefly mentioned static and dynamic features, retrieved through static and dynamic analysis of the program's files and execution behaviour, resepectively. Here we discuss in more detail what these features are. In general, to train a machine learning model to classify programs, a dataset should have clearly defined sets of information that the model should learn identifiers and patterns from.

Selecting which features remain relevant is known as feature selection, it is an important task for many machine learning classification systems and can be as challenging as designing the model architecture. Android malware detection is no different from such tasks. For example, Taheri et al. [32] propose a multi-layered system, that first uses static permissions and intents to determine a program's malicious intent, then passes the program to a second layer to analyse its dynamic API calls and network activity, allowing the model to combine the strengths of both types of features. Broadly, feature selection involves identifying a set of features that enables a model to identify behaviours correlated with their malware classification.

2.2.1 Malware Features

Listed here are features commonly extracted and used for malware classification.

- **Opcodes** – are operation codes that represent machine language instructions, extracted through disassembly of a program's binary. Analysis of a program's opcodes can reveal sequences and frequency distributions which match with identifiable malicious operations.

- **Permissions** – are a high-level static feature most prevalent for Android malware, essentially representing explicit requests made by an application to access data and functionalities (i.e, camera, contacts, location, etc.). Permissions alone do not reveal much about an application's malicious behaviour but indicate its capacity to access sensitive functionality. Discrepancies between their stated purpose and requested permissions may reveal more about their malicious intent (e.g, a calculator app requesting for location data).

- **API Calls** – provide semantic context about an application's functionality and behaviour as individual Android library APIs expose specific operations. Through static analysis, API usage can be identified from method-invocation instructions and method references in the application's DEX bytecode. Malware often exhibit distinct patterns of API calls that can be distinguishable from other forms of malware and benign software, often leveraged by malware classifiers.

- **Centrality** – is a feature of graphs borrowed from social network analysis which can be applied to the study of malware [34]. The idea is leveraging the fact that benignware and malware have different distributions of centrality measures of sensitive functions and can therefore be used to separate classifications.

- **Graph Features** – often exhibit other measures which can be used to separate classifications of malware, they can represent a top-level view of structural or functional topology of a program. These features can be sensitive node counts, edge-to-node ratios, shortest path length to sensitive nodes, etc.

2.3 Graphs

A graph can be formally defined as $G = (V, E)$ where $V = \{v_1, \ldots, v_N\}$ is a set of $N = |V|$ nodes, and $E = \{e_1, \ldots, e_M\}$ is a set of $M = |E|$ edges which may be directed

or undirected (bi-directional), depending on the implementation. Many graph variants may be used for program representation, this dissertation utilises attributed graphs (AGs), which encode additional features in nodes and/or edges. Node attributes are mapped through a mapping function F_n to assign a feature vector of d_n elements to nodes. These attributes may represent stored variables such as strings, or assembly opcodes, or even subgraphs of the functions within the program. Similarly, edge attributes are mapped through a mapping function F_e that assigns a feature vector of d_e elements to edges; these features may represent timestamps or data transmission sizes. A more specialized form of attributed graph is the hierarchical graph (HG), also known as graph of graphs, which involves embedding one graph representation $G_i = (V_i, E_i)$ within the feature vector attributes of another graph $G_o = (V_o, E_o)$. HGs are a recently explored method for representing programs, commonly by embedding an intra-procedural control flow graph of a function within the nodes of an inter-function call graph.

2.3.1 Function Call Graphs (FCGs)

FCGs are a method of representing inter-procedural relationships among the functions within a program, where the nodes are the program's functions and the edges are the directed function calls. FCGs offer a global view of the program's structure but may lack fine-grained intra-procedural information.

FCGs can be constructed statically, simply by traversing all function bodies and adding edges when a function call is encountered. Dynamic analysis can also be utilised to record the program's actual call relationships during runtime; however, this may result in low coverage of malicious calls and can be more expensive for large-scale datasets.

In the case of Android malware, we can additionally use the API call sequences from the application and represent them using a FCG. This representation can be useful for identifying high-level behavioural patterns including file accesses, network communication, cryptographic operations or permission requests, which are highly relevant for malware detection. However, API call graphs are unable to represent non-API logic, so they can only convey partial information about a program, and therefore can be combined with the program's FCG to represent the program in a more complete way.

2.4 Learning from Program Representations

The adaptability of machine-learning methods has led to their widespread adoption in the space of malware research, and a wide variety of novel detection schemes have been proposed and researched extensively. Many of these schemes implement vastly different and often ingenious methods of representation used to encode programs into a format on which a machine-learning model can perform inference. One notable method is to represent binary files as greyscale images and pass these images through a convolutional neural network (CNN) to learn visual patterns associated with malware classes [25]. Various models construct graph representation through different methods for encoding information about each function and require additional processing to retrieve features of each function and encode them in a way that can be interpreted by a machine learning model.

However, not all representation methods are suitable for inference with different models. FCGs for example cannot normally be processed by a traditional CNN, and are therefore typically passed to graph neural networks.

2.5 Graph Neural Networks

Graph Neural Networks (GNNs) are a family of neural network architectures designed to directly operate on graph-structured data. GNNs are able to learn from objects represented as nodes, and the relationships between them represented as edges.

This makes them particularly suitable for learning from program graph representations such as FCGs. Most modern GNN architectures operate through a process known as message passing. Each node is associated with a feature vector that condenses its attributes into a format that can be processed during GNN message passing. At each layer, a node receives information from neighbouring nodes, aggregates these messages, and combines the result with its existing representation to produce an updated embedding. This process can be generalised for node v, at layer k as

$$m_v^{(k)} = \text{AGGREGATE}^{(k)}\left(h_u^{(k-1)} : u \in \mathcal{N}(v)\right).$$

The node then updates its embedding by combining this message with its representa-

tion from the previous layer $h_u^{(k-1)}$ to obtain $h_v^{(k)}$, conceptually

$$h_v^{(k)} = \text{UPDATE}^{(k)}(h_v^{(k-1)}, m_v^{(k)}).$$

Each GNN architecture may use a different design and implement AGGREGATE and UPDATE differently, but are all generally unified by the core concept of message passing.

2.6 Graph Pruning

FCGs can have exceedingly many nodes and edges, such that accurate inference becomes difficult, and their size can substantially increase computational load when training a model. Graph Pruning is the process of reducing the graph into an efficient representation of what should ideally be the structural core of the program that retains important semantic information crucial to its classification. Further, the act of reducing excess information that is not relevant to the classification task may potentially focus the model on malicious identifiers such that it can achieve better performance results.

Broadly, there are three main categories of graph pruning:

- **Graph Sparsification** – reduces the number of edges in a graph, whilst retaining critical information. For program graphs, this means removing unimportant data- or control-flows and edges that do not contribute to the learning objective.
- **Graph Condensation** – constructs a smaller graph that summarises the behaviour of the original graph. Rather than pruning edges or nodes, condensation generates a more compact representation that retains critical information for training and inference.
- **Graph Coarsening** – merges groups of nodes and edges into higher-level 'supernodes' and 'superedges', producing a hierarchical abstraction of the original graph. For a program graph, this may mean aggregating related instructions into blocks, or grouping instructions with similar semantics into the same block, therefore preserving the global structure.

Shokouhinejad et al. [31] identified three criteria for a pruning technique to be considered for malware detection:

1. The method must be applicable to graph classification, as most malware detection methods revolve around this task.

2. The method must not rely on node labelling, allowing more flexible use cases as many malware datasets do not include labels.

3. The method is explainable, as explainability allows practitioners or researchers to identify and validate malware classification.

Criterion 2 generally makes sense for most malware classification tasks because malware authors can name functions arbitrarily and bypass detectors using graph pruning methods that discard nodes based on labels. However, for Android malware classification, this rule does not apply external API calls because their naming conventions fall outside of the malware author's control.

Though there are many methodologies in the domain which utilise graph pruning in some form or other, the concept itself is rarely studied, and graph pruning methods are rarely evaluated against one another in a direct, fair comparison. As such, graph pruning remains an under-explored concept in malware detection.

2.7 GNNExplainer

Research in malware analysis requires not only the detection of malware samples; domain experts also frequently require the evidence and motivation for why a malware classification is given to a sample. To that end, a practical graph representation learning model for malware detection requires an explainability module which can identify malicious subgraphs which motivate the classification.

GNNExplainer is one such graph-explainability module which specifies an explanation as a rich subgraph of the entire graph the GNN was trained on, such that the subgraph maximizes the mutual information with the GNN's prediction [36].

2.7.1 Fidelity

To determine the suitability of the pruning models for graph explanations, the quality and sensitivity of the subgraph explanations must be evaluated in an objective fashion. The immediate idea for objective evaluation is to compare generated explanations with the ground-truth subgraphs which exhibit malicious behaviour. However, due to the lack of such a dataset, we must evaluate the models on proxy measures of explainability

performance. Fidelity is one such measure which allows us to experimentally verify the importance of an explanation in context with the model that which produced it. Though it should be mentioned that fidelity is a measurement of 'faithfulness', and is more a proxy to explanation correctness, rather than a direct measure of it.

Chapter 3

Related Work

Graph Representation Learning (GRL) is a fundamental shift from traditional malware detection and analysis, and represents a move towards applying advanced machine learning methods. Its recent surge in popularity can be partially attributed to the success and achievements of works applying GRL techniques in other specialised domains, and to the ease of adapting existing frameworks for explainable artificial intelligence (XAI). In general, GRL is about transforming graphs from a simple data structure, into a form that can support predictive modelling. For malware detection, this means extracting patterns of malicious behaviour from labelled program graphs so that samples with similar behaviour or structural properties occupy similar regions of feature space. The main point is making malicious and benign samples more separable, thus identifiable.

This chapter explores recent work in the field, their contributions, and further outlines some of the difficulties faced when performing graph-based machine learning on program graphs. We necessarily include works that do not use graph pruning per se, but explore problems which must be addressed when developing a GRL-based malware detector, and especially when such problems are related to model scalability.

3.1 Related Graph Representation Strategies

Defined in Chapter 2, Function Call Graphs (FCGs) are graph representations of the program's functions as nodes, and edges represent invocation. Many GRL studies feature the use of FCGs as they can be easily constructed and offer substantial representation power when coupled with semantically rich feature embeddings.

3.1.1 Control Flow Graphs as Node Embeddings

Control Flow Graphs (CFGs) are one such method of enriching FCG nodes with semantic information as they capture intra-procedural blocks as nodes and execution jumps as directed edges and can thus represent fine-grained detail of a function's logic. Malgraph [18] leverages both CFGs and FCGs to construct a hierarchical graph structure of an inter-FCG representation G_e, comprised of intra-CFGs G_{f_l}, to represent each program e. At each $G_{f_l} \in G_e$, a GraphSAGE encoder is applied to obtain a hidden embedding $h_{l,i}^{(t_1)}$ of each block node of the CFG, with a max-pooling function applied to aggregate nodes as $h_{G_{f_l}} = \textbf{max-pooling}\{h_{l,i}^{(T_1)}\}_{i=1}^{i=|G_{f_l}|} \in \mathbb{R}^{d_1}$. With each FCG node enriched with embeddings $\{\dots h_{G_{f_l}} \dots\}$, another GraphSAGE encoder is applied to produce $\{\dots z_k^{(t_2)} \dots\}$, which are max-pooled again to produce a single embedding for program e as $z_{G_e} = \textbf{max-pooling}\{z_k^{(T_e)}\}_{k=1}^{k=n} \in \mathbb{R}^{d_2}$.

3.1.2 NLP-Style Embeddings

FCG node enrichment can also be done using NLP techniques applied to Dalvik opcode. According to Norouzian et al. [26], Dalvik opcodes follow a power-law distribution similar to words in the English language, and thus suggest that word-embedding techniques may be used to encode semantic representations with sufficient accuracy. SeGDroid [21] makes use of this fact in their implementation of two Word2Vec encoders:

- **opcode2vec** – is a model trained on a corpus of opcode sequences extracted from the training dataset. It uses the Skip-gram architecture to learn a vector representation for each opcode. During training, a fixed-size sliding window moves across each opcode sequence to generate the training samples. The model is trained by updating the opcode embeddings so that, given a target opcode, it may predict the opcodes that occur within its surrounding context.

- **api2vec** – is another model trained on a corresponding corpus of API packages. Similar to opcode2vec, Word2Vec learns the feature vector for each word in every package and the API package's vector representation is used to represent the external API. According to the researchers, the high cohesion in software design suggests that the APIs in the same package may have similar purposes and their feature vectors should be close.

SeGDroid concatenates both features for each FCG node and scales them using social-network-style centrality scores.

Encoding Out-of-Vocabulary Word Embeddings

A notable issue with Word2vec is that it cannot generate a representation for out-of-vocabulary (OOV) words, as they have never been encountered before, impacting any use case which would necessarily encounter rare words. MalHAPGNN [9] solved this issue by developing a BERT-based function embedding (BBFE) method to extract contextual instruction semantics using a pre-trained Transformer architecture, that attempts to preserve behavioural meaning even when function identifiers are stripped or randomized.

3.1.3 Heterogeneous Information Networks

Though homogeneous graph representation methods have shown to have substantial representation power, heterogeneous information networks (HINs) are able to capture multiple layers of information relationship networks and thus may be able to model more complex behaviours. HinDroid [13] formalizes APK analysis through the use of a multi-relational graph incorporating four entity types: Applications, APIs, Code Blocks, and App Packages, and encodes the relationship between entity types as meta-paths with specific behavioural semantics. Then computes commute-entity matrices over meta-paths for a multi-kernel SVM classification.

Since then, Hawk [12] has improved over HinDroid's performance by developing a seven-entity typed system and introducing meta-graphs alongside meta-paths, comprising the higher-order meta-structure of the represented application. This allows Hawk to capture more complex behavioural relationships, and they further develop a GNN architecture suitable to learn over the HIN representation called Meta-Structure Guided Graph Attention Networks (MSGAT). They showed that GNN architectures can capture higher-order relationships among entities in the HIN.

3.2 Related Problems in Graph Representation Learning

3.2.1 The Oversmoothing Problem

Another frequently encountered issue when utilising GNNs is the oversmoothing problem, defined as the phenomenon in which graph-node representations converge toward uniform vectors due to stacking deep message-passing layers in the graph encoder.

Lo et al. [22] posit that malware FCGs contain structural information about its malicious behaviour and is limited by the information dilution induced by the message passing layers of the GNN architecture. The authors thus proposed jumping-knowledge as a method of architecting multi-layered GNN encoders to address oversmoothing and used a purely structural representation of FCGs encoded only with their centrality scores to avoid domain-specific semantic information influencing the classification.

3.2.2 Signal Dilution

Malicious functionality often occupies a small fraction of the application's whole codebase. As such, global graph pooling operations risk diluting malicious code with a large volume of benign functions. Various detection methods employ different techniques to tackle this exact problem, one of the most commonly used techniques is to use social-network-style centrality.

As first identified by MalScan [34], centrality allows us to measure the 'importance' of a function in the whole FCG and reflects the structural properties of the graph. Their preliminary study showed that malware and benign graphs, exhibited disparate behaviours in the distribution of sensitive API invocations.

MsDroid [11] separately tackled signal dilution by implementing a novel 'snippet-based' detection framework, by sequestering parts of the FCG into behavioural subgraph sets and segmenting whole program graphs into localized subgraphs to train a semi-supervised snippet-level classifier. This allowed MsDroid to employ a semi-supervised learning approach which leverages the fact that malicious subgraphs appear at least once in malware, but never in benignware. This form of learning can be identified as a form of multiple-instance learning (MIL) which is a form of weakly supervised learning where labels are provided for groups of examples (known as a bag), rather than for each individual example (known as an instance). MIL is well suited to

the task of malware classification but remains relatively underexplored in this domain.

3.3 Graph Reduction and Scalability

Previously identified in Section 2.6, formulating a graph representation for very large datasets must take into consideration the computational cost of training a model at scale. As such, for a detection pipeline to be able to handle the load of a large-scale dataset, it must be able to reduce the set of information down to only the critical parts relevant to the classification.

Graph pruning is one technique used to help models scale their detection pipeline, however other methods have shown promising results. Notably, for similar reasons to the motivations for pruning, many methods tend to address signal dilution and scalability at the same time, as often tackling either problem goes hand-in-hand with tackling the other. MalScan [34] for instance constructs an FCG, but does not use it in its representation, instead uses a 426-dimensional vector of 426 representative sensitive APIs, each encoded with their centrality measure calculated on the constructed FCG, and performs inference on that vector.

MsDroid [11] uses the same snippet mechanism as described above to reduce the size of the graph required to perform inference on, and thus reduces the computational workload. This can be described as a kind of sparsification technique, although the fact that the method uses pruning is incidental to the model's overall architecture and would not be generalisable to other detection pipelines.

SeGDroid [21] uses a sensitive API list to create a subgraph of the FCG which reduces the total amount of computation required for inference on the whole FCG. Mohammadian et al. [24] previously explored various graph pruning techniques and their suitability in identifying Windows PE malware, and demonstrated their method of 'Leaf-Pruning' achieves high performance metrics while reducing graph nodes by over 97% achieving very high computational speed.

Shokouhinejad et al. [30] proposed a novel 'Node-Centric' pruning technique (NCP) consisting of a two-step graph reduction framework intended to retain topological information. The model identifies critical "nexus nodes" based on reachability and prunes unconnected "sparse nodes", then evaluates "connector nodes" through their Jaccard similarity to nexus nodes and removes redundant nodes below a certain threshold. The authors evaluate this pruning method against a small (50 benign and 50 malware sam-

ples) Windows PE CFG dataset and a protein enzyme-classification dataset achieving an increased accuracy over the baseline unpruned run however provides little reduction in training time (81.12s to 75.16s).

3.4 Summary

Existing graph-based malware detectors demonstrate that semantic node representations and graph sparsification can improve efficiency and, to some extent, performance, but comparisons are made difficult because of their heterogeneous designs, differences in dataset selection, malware domains, model architectures and evaluation methods.

Prior work has focused on methods for enriching graph nodes to increase semantic information, localising substructures to enhance structural representation, and adapting GNN architectures to improve message passing behaviours. This dissertation attempts to build upon these foundations to create a methodology which capture these motivations, and to then isolate the effect of structurally different pruning techniques and evaluate their effectiveness.

Unlike prior studies, we adopt a holistic approach which unifies the evaluation of comparable graph pruning methods and benchmarks them against a suite of metrics assessing scalability, predictive performance, graph size sensitivity, and explainability, especially under spatially and temporally realistic conditions (discussed in detail in Section 5.1.1).

Chapter 4

Proposed Methodology

This chapter presents our proposed methodology designed to measure the effects of function call graph pruning by isolating their effects and evaluate them against two GNN architectures: GCN and GraphSAGE, producing six model configurations in total. The use of two GNN architectures allows the effects of pruning be examined independently within each architecture while also assessing whether observed behaviour is consistent across different message-passing models. Consequently, observed differences between configurations can be evaluated principally with respect to the changes introduced by pruning, or by the interaction between pruning and the GNN model.

We first define the pruning strategies and their underlying assumptions, then present the proposed node representation and resulting attributed FCG construction pipeline. We then summarise the GNN classification and explanation framework used across all graphs.

4.1 Graph Representations

The overall design for this work's model takes inspiration from SeGDroid [21] and implements key features proposed by them. However, our design differs in all stages of the pipeline (depicted in Figure 4.1), particularly because of the different classification strategy demanded by evaluating under a more recent (and therefore more evolved malware) dataset.

4.1.1 SeGDroid

To illustrate the reasons why our approach differs methodologically, we first describe the model proposed by SeGDroid.

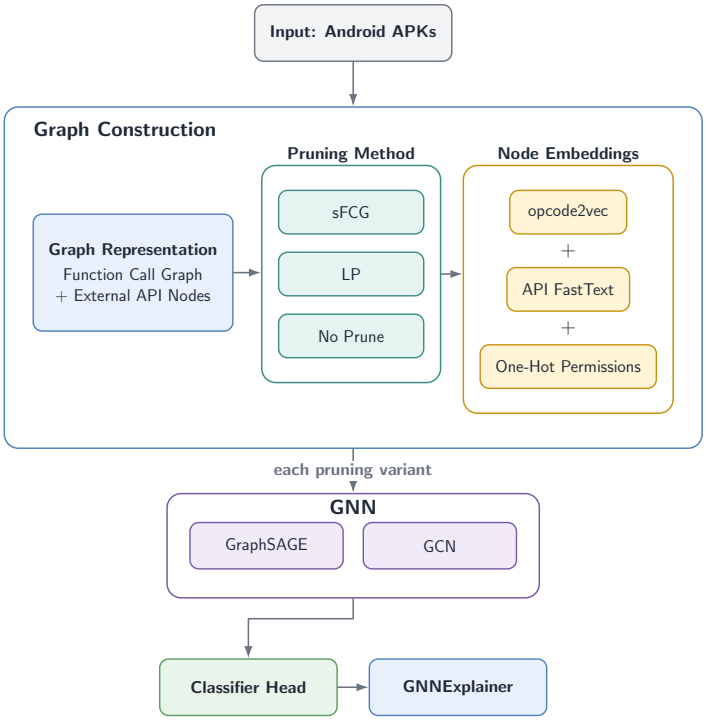


Figure 4.1: The proposed evaluation pipeline. Each APK is converted into an FCG and processed under one of three graph configurations: unpruned, leaf prune, or sFCG prune. All configurations use the same opcode, API, and permission node representation before being independently evaluated using GCN and GraphSAGE classifiers; GNNExplainer is subsequently applied to analyse explanation faithfulness.

SeGDroid uses FCGs to represent both the system APIs or self-defined functions. They use a two-tiered system of embedding node features. First, by using a concatenation of two word2vec features, namely; opcode2vec and api2vec. Then, by using a social networking concept known as centrality (in their case Degree-Centrality) to scale the embedding weights such that the model focuses on highly connected nodes. This can justified as according to Malscan [34], there is evidence of a correlation between highly connected sensitive API nodes and malware classified programs.

Sensitive Function Call Graphs

SeGDroid uses the same set of sensitive APIs as MalScan generated by another work PScout [4], and propose the generation of a sensitive FCG (we use sFCG for brevity for the rest of this report) as a method of pruning. The overall method is described in Algorithm 1, and outlined below:

(**1**) lines 1–3. First, search the graph for sensitive API nodes to obtain the set of all sensitive nodes in the graph S_{sv}.

Algorithm 1: SeGDroid Sensitive FCG pruning algorithm

Input : an FCG $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, sensitive API set S_{api}
Output: a sensitive FCG $\mathcal{SG} = (\mathcal{V}, \mathcal{E})$

```
1 for v_i ∈ V do
2     if v_i ∈ S_api then
3         S_sv ← S_sv ∪ {v_i}
4 for sv_i ∈ S_sv do
5     addAncestor(sv_i, S_sav, G)
6 for sav_i ∈ S_sav do
7     addDescendant(sav_i, S_sdv, G)
8 SG ← obtainPrunedGraph(G, S_sdv, S_sav)
```

(**2**) lines 4–5. Search the neighbouring nodes invoking any $v_i \in S_{sv}$ in the backward direction (of a directed edge) until the entry point, obtaining the ancestor node set S_{sav} of S_{sv}.

(**3**) lines 6–7. Continue a search for descendant nodes until reaching the leaf nodes in the forward direction, obtaining the descendant node set S_{sdv}.

(**4**) line 8. Remove all nodes and edges not in S_{sdv}, to obtain the pruned sensitive FCG.

Once the sensitive FCG is generated, each node can be populated using the two-tiered embedding process. Then a GNN classifier can be trained on a labelled dataset of generated sensitive FCG graphs, and passed to their graph explainer module (for brevity, we will not discuss their implementation here as this work uses a different architecture).

4.1.2 Graph Pruning Strategies and Baseline

This project will evaluate SeGDroid's sFCG method alongside a simple but effective state-of-the-art method in Leaf Prune, against the full FCG (no prune) baseline. We use the same cutoff as SeGDroid for both pruning methods to prevent pruning of graphs under 8,000 nodes. Consequently, our pruned configurations are conditional.

Leaf Prune

Leaf Prune (LP) is a method proposed by Mohammadian et al. [24]. The idea is to remove all nodes which are identified as leaf nodes V_{leaf}, and all edges associated with

them E_{leaf}. A leaf node $v \in V_{\text{leaf}}$ can be identified by whether they satisfy the following property: $\text{degree}(v) \leq 1$.

The leaf pruned graph $G' = (V', E')$ is then identified as the subset of $G = (V, E)$ such that

$$G' = G \backslash V_{\text{leaf}} \backslash E_{\text{leaf}}$$

The authors identify LP as a lightweight method of removing nodes that have minimal impact on the graph structure. LP was originally developed for the purpose of malware detection for statically generated FCG/CFGs decompiled from Windows PE/x86 binaries. Thus, we evaluate LP's transferability across domains and graph generation methods.

Comparison of Graph Pruning Methods

The two chosen pruning strategies were identified as state-of-the-art methods in malware detection; however, they approached reducing the graph with very different strategies. sFCG relies on traversing through the graph from identified sensitive nodes to select retained nodes and edges to obtain its pruned graph, while LP uses a single heuristic applied to every node to obtain its pruned graph.

These two pruning methods analysed in this work, represent vastly different approaches to pruning, sFCG makes a qualitative assumption that we can reliably expect core malicious behaviour to occur in the causal chain of sensitive API calls. Whereas LP makes a quantitative assumption that core malicious behaviour will have multiple invocations such that we can ignore leaf nodes. Alongside the two graph pruning methods, they will be compared against the baseline full FCG with no pruning applied.

4.1.3 Different Embedding Strategies

Our model makes use of SeGDroid's NLP-style semantic embeddings, but departs from SeGDroid by including an additional permissions embedding (modelled after a component of MsDroid [11]), and discarding the use of centrality-based scaling, opting to let the model learn relevant nodes through permission invocations. Further, we use FastText [7] as our model for learning representations for API vectors instead of word2vec, though we retain the same opcode2vec implementation as SeGDroid. To the best of our knowledge, this is the first work to use FastText to encode Android API packages for any task.

MsDroid Permissions

The primary motivation for discarding centrality was prior testing of the SeGDroid model yielding poor results in a small set of testing graphs. And further motivated by the fact that centrality scaling resulted in very small vector values which would yield poor results when applying gated-attention pooling. This is explained by the fact that centrality scaling would result in many vectors with small values resulting in a vast increase of zero vectors. In preliminary testing, we found that removing centrality did not impact the model's performance. This is further explained by the fact that MalScan's motivation for using centrality is that malware often features highly centralised behaviour for sensitive API invocations, however, testing for this behaviour in our dataset (of more recent years) yielded no such results (see Appendix Figure A.2), likely due to malware evolution, disguising their presence amongst benign software.

Therefore, to substitute this method of 'focusing' on malicious features, we opted to model the permissions embedding from MsDroid's implementation of node-level permissions as a one-hot encoding of security-relevant permissions. This method uses prior work studying Android permissions in Axplorer [5], allowing us to map API nodes with the specific permission checks they require.

FastText API Embeddings

We use FastText as a word embedding model for API package naming sequences, due to its particular suitability to the task. API packages that do very similar tasks are generally also named very similarly. However, some packages are seldom encountered, and very likely will not feature in the model's training corpus, for models such as word2vec, this means that such packages will be ignored and will feature all zero vectors. FastText solves this by representing words as character-level n-grams allowing it to construct meaningful embeddings for package names not in its dictionary based on their constituent subword patterns and their similarity to names encountered during training.

For example, take the package word `telephone`, FastText represents the word

through its character n-grams, suppose $n \in \{3 \text{ to } 6\}$, then its character n-grams are[1]:

$$\begin{aligned} G_3 &= \{\texttt{<te}, \quad \texttt{tel}, \quad \ldots, \quad \texttt{ne>}\} \\ G_4 &= \{\texttt{<tel}, \quad \texttt{tele}, \quad \ldots, \quad \texttt{one>}\} \\ G_5 &= \{\texttt{<tele}, \quad \texttt{telep}, \quad \ldots, \quad \texttt{hone>}\} \\ G_6 &= \{\texttt{<telep}, \quad \texttt{teleph}, \quad \ldots, \quad \texttt{phone>}\} \end{aligned}$$

Its subwords are therefore $G_w = G_3 \cup G_4 \cup G_5 \cup G_6$. As FastText also uses the complete word itself to contribute to a vector, $\mathbf{q}_w$ is the whole word vector representation. FastText then constructs the representation of `telephone` as a combination of embeddings from every n-gram of G_w, where $\mathbf{z}_g \in \mathbb{R}^d$ is a learned embedding of n-gram $g \in G_w$.

$$\mathbf{v}_w = \frac{1}{|G_w| + 1} \left(\mathbf{q}_w + \sum_{g \in G_w} \mathbf{z}_g \right).$$

The scoring function of the word with context c is then

$$s(w, c) = \sum_{g \in G_w} \mathbf{z}_g^\top \mathbf{v}_c = \mathbf{v}_w \mathbf{v}_c$$

Regarding, out-of-vocabulary (OOV) words, for example `telephones`, its embeddings can still be constructed, though its whole-word representation does not exist, it is instead constructed from its constituent character n-grams. Because $G_{\texttt{telephone}}$ and $G_{\texttt{telephones}}$ are similar, their set of n-grams will have a large overlap, therefore its embedding is represented as

$$\mathbf{v}_{\texttt{telephones}} = \frac{1}{|G_{\texttt{telephones}}|} \left(\underbrace{\mathbf{z}_{\texttt{tel}} + \mathbf{z}_{\texttt{tele}} + \mathbf{z}_{\texttt{phon}} + \mathbf{z}_{\texttt{phone}} + \cdots}_{\text{buckets meaningfully updated during training}} + \underbrace{\mathbf{z}_{\texttt{phones}}}_{\text{possibly never updated}} \right).$$

If the n-gram `phones` was not encountered during training, its corresponding hash bucket may nevertheless have been updated by another n-gram that hashed to the same location. If no training n-gram updated that bucket, its vector remains at its initial value. Despite such unseen subwords, the large number of trained subword representations shared between `telephone` and `telephones` allows FastText to construct a somewhat accurate embedding for the OOV word.

[1]FastText adds angle brackets "<" and ">" to the word, so `telephone` is actually parsed as `<telephone>`.

Algorithm 2: Node Feature Generation

Input: Graph G, API embedding model M_{API}, opcode embedding model M_{opcode}, permission mapping $\mathcal{P}$
Output: Node feature matrix X

foreach *node* $v \in V(G)$ **do**
  $\mathbf{e}_v^{\text{API}} \leftarrow \mathbf{0}^{64}$
  $\mathbf{e}_v^{\text{opcode}} \leftarrow \mathbf{0}^{64}$
  $\mathbf{p}_v \leftarrow \mathbf{0}^{247}$
  if *v is an external API* **then**
    $T_{\text{API}} \leftarrow \text{APITokenize}(v)$
    $\mathbf{e}_v^{\text{API}} \leftarrow \text{GenerateEmbeddingAPI}(T_{\text{API}}, M_{\text{API}})$
    $\mathbf{p}_v \leftarrow \text{GetNodePermissions}(v, \mathcal{P})$
  else
    $T_{\text{opcode}} \leftarrow \text{OpcodeTokenize}(v)$
    $\mathbf{e}_v^{\text{opcode}} \leftarrow \text{GenerateEmbeddingOpcode}(T_{\text{opcode}}, M_{\text{opcode}})$
  $X[v] \leftarrow \mathbf{e}_v^{\text{API}} \,\|\, \mathbf{e}_v^{\text{opcode}} \,\|\, \mathbf{p}_v$
return X

4.1.4 Proposed Embedding Strategy

Our node embeddings use 375 dimension vectors which consist of **64** opcode2vec embeddings, **64** FastText API embeddings, **247** one-hot encoded permissions embeddings.

$$\mathbf{x}_v = \left[\underbrace{\mathbf{x}_v^{\text{API}}}_{64} \,\middle\|\, \underbrace{\mathbf{x}_v^{\text{opcode}}}_{64} \,\middle\|\, \underbrace{\mathbf{x}_v^{\text{permission}}}_{247} \right] \in \mathbb{R}^{375}$$

Since there is a trade-off between semantic expressibility and the size and quality of the node features, which will inevitably impact both training time and metric performance, the dimensionality of opcode and API embeddings was chosen to reduce the computational load whilst retaining a sufficient amount of expressibility. We train both Word2Vec and FastText models on a corpus of opcodes and APIs extracted from a subset of 2,000 APKs from the training dataset over 10 epochs with a fixed sliding window of 5.

Node embeddings are generated using the Feature Generation method as outlined in Algorithm 2, our graph representation is thus generated using the FCG construction method in Algorithm 3.

Algorithm 2 can be generalised as processing each node as two types:

- **External API nodes** - are represented using embeddings derived from their API tokens, and if they are associated with permissions as identified by GetN-

Algorithm 3: FCG Construction and Feature Generation

Input: Set of APKs $\mathcal{A}$, pruning_technique
Output: Attributed function-call graph for each APK

foreach *APK* $a \in \mathcal{A}$ **do**
  $G_{\text{FCG}} \leftarrow \textsc{GetCallGraph}(a)$
  $\mathcal{P} \leftarrow \textsc{LookupPermissions}(a)$
  $\mathcal{S} \leftarrow \emptyset$
  foreach *node* $v \in V(G_{\text{FCG}})$ **do**
    if v *is an external API* ***and*** $v \in$ SensitiveAPIList **then**
      $\mathcal{S} \leftarrow \mathcal{S} \cup \{v\}$
  $G \leftarrow \textsc{Prune}(G_{\text{FCG}}, \mathcal{S}, \text{pruning\_technique})$
  $X \leftarrow \textsc{BuildNodeFeatures}(G, M_{\text{API}}, M_{\text{opcode}}, \mathcal{P})$
  $X \leftarrow \textsc{NormaliseEmbeddingBlocks}(X)$
  $\textsc{WriteMetadata}(G)$
  $\textsc{Save}(G, X)$

odePermissions$(v, \mathcal{P})$, then those permissions are additionally one-hot encoded to the embeddings.

- **Internal Function nodes** - are represented using embeddings derived from their opcode sequences.

These components are concatenated to form the final feature vector for each node.

Algorithm 3 constructs the attributed call graph by first using AndroGuard to extract the full call graph G_{FCG} from the Android APK file, identifying sensitive API nodes $\mathcal{S}$, then applying the chosen pruning method and generates feature vectors through BuildNodeFeatures as detailed in Algorithm 2 for the (un)pruned graph G. These features are normalised with L2 normalisation and stored together with the graph structure and metadata.

4.2 GNN Architecture

For our modelling framework, we opted to use two foundational GNN architectures: **GraphSAGE** and **Graph Convolutional Network (GCN)**. Both models process the same form of graph-structured inputs and use a common high-level dimensional configuration, readout strategy, and downstream binary classification head. The overall GNN architecture is summarised in Figure 4.2.

For both models, the encoder dimensions are fixed at $375 \rightarrow 128 \rightarrow 64 \rightarrow 64$, giving a final node embedding dimension of $d_h = 64$. For a graph $G = (V, E)$, the

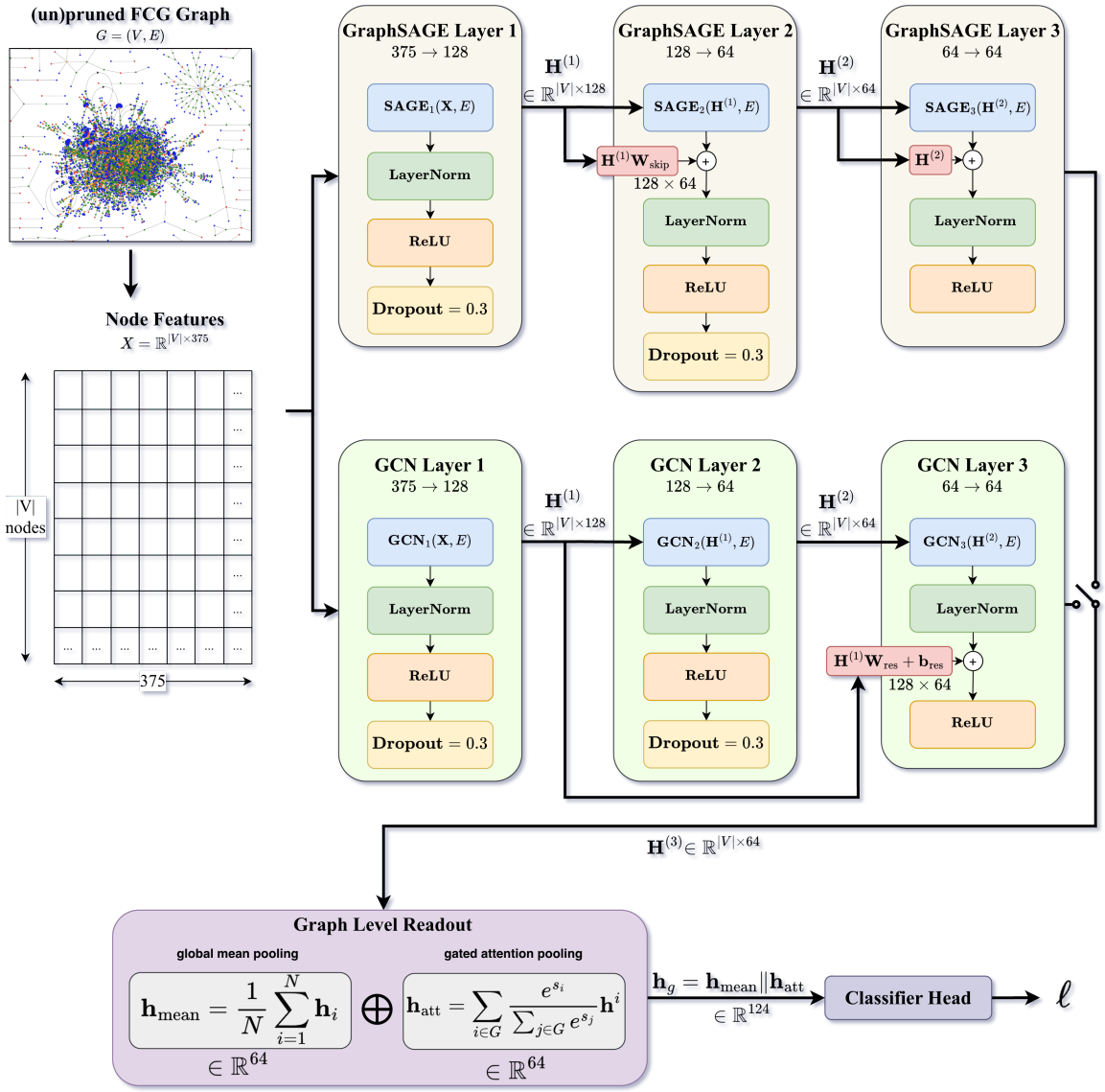


Figure 4.2: GNN classification architecture used for all pruning configurations. The 375-dimensional node features are processed by either a GraphSAGE or GCN encoder network, producing 64-dimensional node representations. Global mean and gated-attention pooling are concatenated before the binary classifier head.

node-feature matrix is therefore $\mathbf{X} \in \mathbb{R}^{|V|\times 375}$ and each encoder learns the mapping $f_\theta : (\mathbf{X}, E) \longrightarrow \mathbf{H}^{(3)} \in \mathbb{R}^{|V|\times 64}$, where $\mathbf{H}^{(3)}$ contains the learned node representations to be passed to the graph pooling functions outlined in Section 4.2.3.

4.2.1 GraphSAGE Encoder

Our GraphSAGE model consists of three mean-aggregating GraphSAGE layers with feature dimensions $375 \to 128 \to 64 \to 64$, respectively:

- $\mathbf{H}^{(1)} = \textbf{Dropout}\left(\textbf{ReLU}\left(\textbf{LayerNorm}\left(\textbf{SAGE}_1(\mathbf{X}, E)\right)\right)\right) \in \mathbb{R}^{|V|\times 128}$,
- $\mathbf{H}^{(2)} = \textbf{Dropout}\left(\textbf{ReLU}\left(\textbf{LayerNorm}\left(\textbf{SAGE}_2(\mathbf{H}^{(1)}, E) + \mathbf{H}^{(1)}\mathbf{W}_{\text{skip}}\right)\right)\right)$ $\in \mathbb{R}^{|V|\times 64}, \mathbf{W}_{\text{skip}} \in \mathbb{R}^{128\times 64}$,
- $\mathbf{H}^{(3)} = \textbf{ReLU}\left(\textbf{LayerNorm}\left(\textbf{SAGE}_3(\mathbf{H}^{(2)}, E) + \mathbf{H}^{(2)}\right)\right) \in \mathbb{R}^{|V|\times 64}$.

Where $\mathbf{W}_{\text{skip}}$ is a learned residual projection. No dropout is applied after the final layer, and $\mathbf{H}^{(3)} \in \mathbb{R}^{|V|\times 64}$ is passed directly to the graph-level readout.

4.2.2 GCN Encoder

Our GCN encoder similarly consists of three graph convolutional layers with feature dimensions $375 \to 128 \to 64 \to 64$, respectively.

- $\mathbf{H}^{(1)} = \textbf{Dropout}\left(\textbf{ReLU}\left(\textbf{LayerNorm}\left(\textbf{GCN}_1(\mathbf{X}, E)\right)\right)\right) \in \mathbb{R}^{|V|\times 128}$,
- $\mathbf{H}^{(2)} = \textbf{Dropout}\left(\textbf{ReLU}\left(\textbf{LayerNorm}\left(\textbf{GCN}_2(\mathbf{H}^{(1)}, E)\right)\right)\right) \in \mathbb{R}^{|V|\times 64}$,
- $\mathbf{H}^{(3)} = \textbf{ReLU}\left(\textbf{LayerNorm}\left(\textbf{GCN}_3(\mathbf{H}^{(2)}, E)\right) + \mathbf{H}^{(1)}\mathbf{W}_{\text{res}} + \mathbf{b}_{\text{res}}\right)$ $\in \mathbb{R}^{|V|\times 64}, \mathbf{W}_{\text{res}} \in \mathbb{R}^{128\times 64}, \mathbf{b}_{\text{res}} \in \mathbb{R}^{64}$

where $\mathbf{H}^{(1)}\mathbf{W}_{\text{res}} + \mathbf{b}_{\text{res}}$ is a learned residual projection. No dropout is applied after the final layer, and $\mathbf{H}^{(3)}$ is passed directly to the graph-level readout.

4.2.3 Graph-Level Readout

Following the final GNN layer, each node is represented as a 64-dimensional embedding. This is done in two components, first using **global mean pooling** to average the final node embeddings through

$$\mathbf{h}_{\text{mean}} = \frac{1}{N}\sum_{i=1}^{N}\mathbf{h}_i$$

where N is the number of nodes in the graph and $\mathbf{h}_i \in \mathbb{R}^{64}$ is the output node embedding from the three-layers of GNN message passing, of node i.

In parallel, motivated by the classifying task's suitability to Multiple Instance Learning (MIL), the model applies **gated-attention pooling** [14] which learns the relative importance of each node. For each node embedding $\mathbf{v}_i = \tanh(\mathbf{W}_v\mathbf{h}_i + \mathbf{b}_v)$ and $\mathbf{g}_i = \sigma(\mathbf{W}_g\mathbf{h}_i + \mathbf{b}_g)$, where σ denotes the sigmoid function, these representations are combined element-wise as $\mathbf{q}_i = \mathbf{v}_i \odot \mathbf{g}_i$ and projected to a scalar attention score $s_i = \mathbf{w}_a^T\mathbf{q}_i + b_a$. These scores are normalised using softmax

$$\mathbf{h}_{\text{att}} = \sum_{i \in G} \frac{e^{s_i}}{\sum_{j \in G} e^{s_j}} \mathbf{h}^i$$

The mean-pooled and attention-pooled representations are then concatenated:

$$\mathbf{h}_g = \mathbf{h}_{\text{mean}} \| \mathbf{h}_{\text{att}}$$

resulting in a fixed graph representation $\mathbf{h}_g \in \mathbb{R}^{128}$.

Classifier Head

Our classifier head receives the 128-dimensional vector $\mathbf{h}_g$, and performs layer normalisation, then a linear layer reduces it to a 64-dimensional hidden vector. A LeakyReLU with negative slope 0.1 is applied, followed by dropout:

$$\tilde{\mathbf{z}} = (\mathbf{Dropout}\,(\mathbf{LeakyReLU}\,(\mathbf{Linear}_{64}\,(\mathbf{LayerNorm}\,(\mathbf{h}_g)))))$$

The final classification is generated by a 64-dimension linear layer reduction to a single binary-classification logit: $\ell = \mathbf{Linear}_1\,(\tilde{\mathbf{z}})$

4.2.4 Training Objective

All models use a standard weighted binary cross-entropy loss with malware as the positive class and class weighting defined as

$$w_+ = \frac{N_{\text{benign}}}{N_{\text{malware}}}$$

No weighted sampler is used. Therefore, we can define our loss function as

$$\mathcal{L}_{\text{BCE}} = -\frac{1}{B}\sum_{i=1}^{B}\left[w_{+}y_i \log \sigma(\ell_i) + (1-y_i)\log(1-\sigma(\ell_i))\right],$$

where B is the batch size, ℓ_i is the logit for graph i, $y_i \in \{0,1\}$ is the ground-truth label, and σ is the sigmoid function.

Chapter 5

Evaluation

This chapter details our process of dataset selection and experimental setup, and further outlines various evaluation methods used for this project, finally we present various results and findings of experimentation.

5.1 Dataset Preparation

In the Android ecosystem, there exist many different types of malware and goodware alike, and the task of preparing a dataset such that they represent the landscape of applications can be overlooked and can skew classifiers into achieving very high metric scores that are unrepresentative of their true classification abilities.

5.1.1 Tesseract

Tesseract [27] is a framework and methodology for evaluating malware classifiers under realistic spatial and temporal conditions. They identify the sources of biases, that arise from an improper dataset collection, can be traced to the following:

- **Temporal bias** – Introduced when training and testing samples have unrealistic splits in relation to time, this can result in the model learning from future knowledge about the testing objects.
- **Spatial bias** – Refers to the unrealistic splitting of benign to malware ratio which is not a true representation of the ecosystem.

As such, Tesseract proposes three rules which prevent such biases from leaking into model evaluations.

C1. Temporal training consistency – All objects in the training set must strictly be temporally precedent to the objects in the testing set, such that:

$$time(s_i) < time(s_j), \forall s_i \in Tr, \forall s_j \in Ts$$

where s_i is an object in the training set Tr, and s_j is an object in the testing set Ts.

C2. Temporal benign/malware windows consistency – For every testing slot of size Δ, all test objects must be from the same time window:

$$t_i^{min} \leq time(s_k) \leq t_i^{max}, \forall s_k \in \text{slot}[t_i, t_i + \Delta)$$

where $t_i^{min} = \min_k \text{time}(s_k)$ and $t_i^{max} = \max_k \text{time}(s_k)$.

C3. Realistic malware-to-benign ratio in testing – Let φ be the average percentage of malware in the training data, and δ be the average percentage of malware in the testing data. Let $\hat{\sigma}$ be the estimated percentage of malware in the wild. For a realistic evaluation, it must be reasoned that δ should be as close as possible to $\hat{\sigma}$:

$$\delta \approx \hat{\sigma}$$

The researchers note that if $\hat{\sigma}$ is overestimated, the results are positively inflated, and that although δ cannot be changed, φ may be tuned.

5.1.2 Hypercube

To evaluate the model in accordance with the above rules, the dataset of malware samples must have a reliable date label such that testing samples are not inferenced on with training samples with a later date. This dissertation makes use of the Hypercube dataset [8] which uses Google Play upload dates (`gp_date`) corresponding to samples allowing for temporally consistent evaluations.

We use a subset of Hypercube comprising 26,994 samples from the months of 2023-01 to 2023-9 to train our model, and use 9,735 samples from 2023-10 to 2023-12 for validation. For the final evaluation, our testing set comprises 19,800 samples from the months of 2024-01 to 2024-12. For the year 2023, samples were chosen at random and as such have more monthly variation than the samples present in the 2024 set.

2023				2024			
Month	**Split**	**Benign**	**Malware**	**Month**	**Split**	**Benign**	**Malware**
2023-01	train	2,725	300	2024-01	test	1,500	150
2023-02	train	2,765	280	2024-02	test	1,500	150
2023-03	train	2,809	306	2024-03	test	1,500	150
2023-04	train	2,731	276	2024-04	test	1,500	150
2023-05	train	2,712	243	2024-05	test	1,500	150
2023-06	train	2,666	266	2024-06	test	1,500	150
2023-07	train	2,657	284	2024-07	test	1,500	150
2023-08	train	2,737	272	2024-08	test	1,500	150
2023-09	train	2,693	272	2024-09	test	1,500	150
2023-10	val	3,219	364	2024-10	test	1,500	150
2023-11	val	3,059	311	2024-11	test	1,500	150
2023-12	val	2,646	136	2024-12	test	1,500	150

Table 5.1: Benign and malware sample counts by GP month and dataset split.

The 2024 samples from Hypercube originally had a noisier distribution across months. Therefore, for each testing month to strictly adhere to C3, a flat 10 : 1 benign-to-malware ratio is used (a slight conservative figure to TESSERACT's $\hat{\sigma} \approx 10\%$), this has the added benefit of ensuring no overrepresentation of any months when evaluating with the *Area Under Time* (AUT) metric.

5.1.3 Dataset Sourcing and Generation Details

For procurement of the dataset's Android apk files, we use Androzoo [1] as a source, and preprocess each application to generate their corresponding FCGs with the Python reverse engineering tool Androguard. Generation of NLP-style node embeddings makes use of the Python Gensim [29] library.

One phenomenon encountered in preprocessing is that a handful of apks will fail the preprocessing stage due to containing malformed opcodes and thus prevented Androguard from generating the program's FCG entirely. To keep the preprocessing stage as simple as possible, we opted to remove those samples from the training and validation dataset, however we replaced those affected in the testing set with random samples which matched their classification and belonged to the same month in the `gp_date` column. Note that the impact on the malware ratio of the training set φ is negligible with regard to rule C3.

The preprocessing phase used a Python script to generate a zipped file comprising 1,000 `.npz` formatted FCGs locally (fewer if any graphs failed preprocessing), then

uploaded the batch of files onto a Google Cloud Storage bucket. Once uploaded, each batch is deleted from local storage, and begins processing the next batch.

5.2 Experimental Setup

All model training was conducted on an NVIDIA L4 GPU, with 8 vCPUs and 32GB RAM, hosted on the Google Cloud Platform.

All training runs were initialised with the same hyperparameters: Epochs = 50, batch size = 16, learning rate = 0.0005, weight decay = 0.00005, dropout = 0.3, training uses shuffled batches with no over/under-sampling. We use a random seed (42) to initialise runs, however this does not guarantee reproducibility across repeated runs.

Each training run is initialised in a Python virtual environment with the following library versions: `Python 3.11.11`, `PyTorch 2.12.1`, `PyTorch Geometric 2.8.0`, `NumPy 2.4.4`, `scikit-learn 1.9.0`[1], `tqdm 4.68.4`.

5.2.1 Additional Information on Model Selection

The model selected for evaluation on the test set is the one which achieves the highest malware F1 score on the validation set, set at a classification threshold of 0.5. From preliminary testing, evaluation of models optimised for lower validation loss favour greater malware recall at the expense of precision, resulting in high false-positive rates, unsuitable for realistic deployment scenarios, and results in overall lower F1 scores. We do not perform any threshold optimisation on the validation set to preserve comparability between models.

5.3 Pruning Scalability and Reduction Metrics

Because both pruning methods are taken from different malware detection tasks, both methods will behave and perform differently to previously recorded reduction metrics. We may evaluate their scalability impact by measuring how much of the original graph, each pruning method retains, along with measuring each run's training time.

Table 5.2 presents the average node and edge reduction ratios after applying the pruning method to the unpruned FCG. Table 5.3 shows the training time of each

[1]The scikit-learn library is used only to report performance metrics.

Pruning Method	Node Retainment Ratio		Edge Retainment Ratio	
	Benign	Malware	Benign	Malware
LP	0.6460	0.6660	0.8811	0.8872
sFCG	0.3452	0.4188	0.4099	0.4871

Table 5.2: Average FCG node and edge retainment ratios for the pruning methods.

Run	Epoch Time (s)
GCN – unpruned	1,771
GraphSAGE – unpruned	1,675
GCN – LP	1,319
GraphSAGE – LP	1,113
GCN – sFCG	586
GraphSAGE – sFCG	574

Table 5.3: Average training time per epoch for GCN and GraphSAGE models on the training set of 26,994 graphs calculated from each epoch's `tqdm` progress bar.

model's run averaged over all epochs.

5.3.1 Analysis

Malware graphs processed under sFCG pruning exhibit a noticeable increase in both node and edge retainment compared to benign graphs, with malware retaining 7.36% more nodes and 7.72% more edges than benign graphs. whereas there is only a slight difference for graphs processed under LP. Since sFCG constructs its retained subgraph around sensitive API calls, this result is consistent with malware samples containing greater network centrality around sensitive APIs [34].

Notably, sFCG offers drastic reduction in training time as illustrated in Table 5.3 with almost 3 times training time reduction, whereas LP reduces training time by about 1.4 times. This is a significant difference to the reduction ratio reported for windows PE files in Mohammadian et al. [24] which showed a reduction of nodes by 97.71% (43.7 times reduced). This may signal to the heterogeneity between Windows PE files and Android DEXs, and therefore LP's ability to capture malicious substructures will be altered between the two types of graphs.

Overall, with regard to **RQ1**, we can observe that both pruning strategies improve computational scalability relative to the complete FCG representation, but to different degrees of effectiveness. LP provides a more conservative reduction in training time, whilst sFCG offers a stronger and more practical reduction for large-scale training.

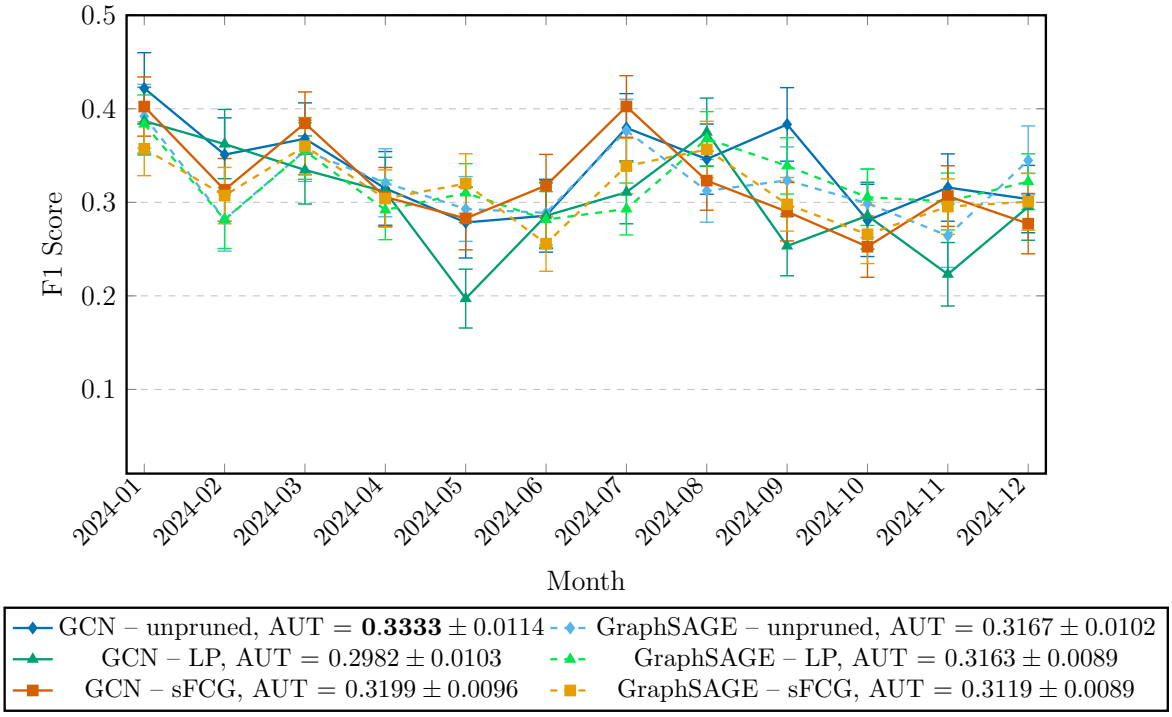


Figure 5.1: Monthly malware F1 scores for GCN and GraphSAGE models under different graph-pruning configurations. Error bars denote a bootstrap derived standard error of the monthly F1 score. AUT values are reported as AUT $\pm$ $SE_{bootstrap}$. Higher AUT indicates better overall temporal performance.

5.4 Performance Metrics

The results reported here are based on a single experimental run for each configuration due to computational and time constraints. Notably, the chosen epoch for each experimental run occurred near the final epoch (see Appendix Table A.3), therefore we cannot guarantee convergence of model training.

Consequently, the observed differences in the experimented models are interpreted descriptively rather than as statistical differences between models. Note that the bootstrap derived standard error rate for each month quantify the sampling uncertainty on each model and are not estimates of training run variance. Nevertheless, the experiments do suggest that pruning approaches can induce a performance difference when compared to the unpruned baselines.

Metric Definitions

We primarily use malware F1 score to judge model performance however we include the Area Under Time (AUT) metric introduced in TESSERACT [27] as a supplementary

Run	F1	AUT	FPR	Recall	Precision
GCN – unpruned	**0.3370**	**0.3333**	**0.0376**	0.2789	**0.4258**
GCN – LP	0.3014	0.2982	0.0543	0.2739	0.3351
GCN – sFCG	0.3226	0.3199	0.0744	0.3356	0.3107
GraphSAGE – unpruned	0.3221	0.3167	0.0598	0.3067	0.3391
GraphSAGE – LP	0.3198	0.3163	0.0962	**0.3733**	0.2797
GraphSAGE – sFCG	0.3141	0.3119	0.0965	0.3661	0.2750

Table 5.4: Performance metrics for GCN and GraphSAGE models under different graph-pruning configurations.

measure of temporal robustness defined as

$$\mathrm{AUT}(\mathrm{F1}, N) = \frac{1}{N-1} \sum_{k=1}^{N-1} \frac{\mathrm{F1}(x_{k+1}) + \mathrm{F1}(x_k)}{2}.$$

We additionally present the false-positive rate (FPR), recall and precision of the malware class for each run.

5.4.1 Analysis

Across the six configurations, we can observe that GCN – unpruned achieves the highest F1 score of 0.3370 and AUT of 0.3333. Separating the two GNN models, we can similarly see that GraphSAGE – unpruned achieves the highest F1 score of 0.3221 and AUT of 0.3167 amongst the GraphSAGE runs. Therefore, on the primary performance measures, pruning does not by itself improve predictive performance over the unpruned FCG baseline.

Observing individual classification metrics, it can be presumed that pruning induces a change in the precision-recall characteristics rather than uniformly reducing its malware classifying ability. For instance, in GCN – sFCG, the model produces substantially more false positives, reducing precision to its unpruned variant from 0.4258 to 0.3107. However, improves recall from 0.2789 to 0.3356 thereby only suffering a minor loss in F1 (0.3370 to 0.3226). We can note similar occurrences for both GraphSAGE pruned variants obtaining higher recall, but lower precision. For the GCN – LP variant, both recall and precision decrease from $(0.2789, 0.4258)$ to $(0.2739, 0.3351)$, resulting in an absolute decrease in F1 of 3.56%. Though, due to the lack of reproducible runs, we cannot conclusively interpret this alone as evidence for LP inducing performance decay in the GCN.

From Figure 5.1, we can observe a slight decay in F1 performance over time, how-

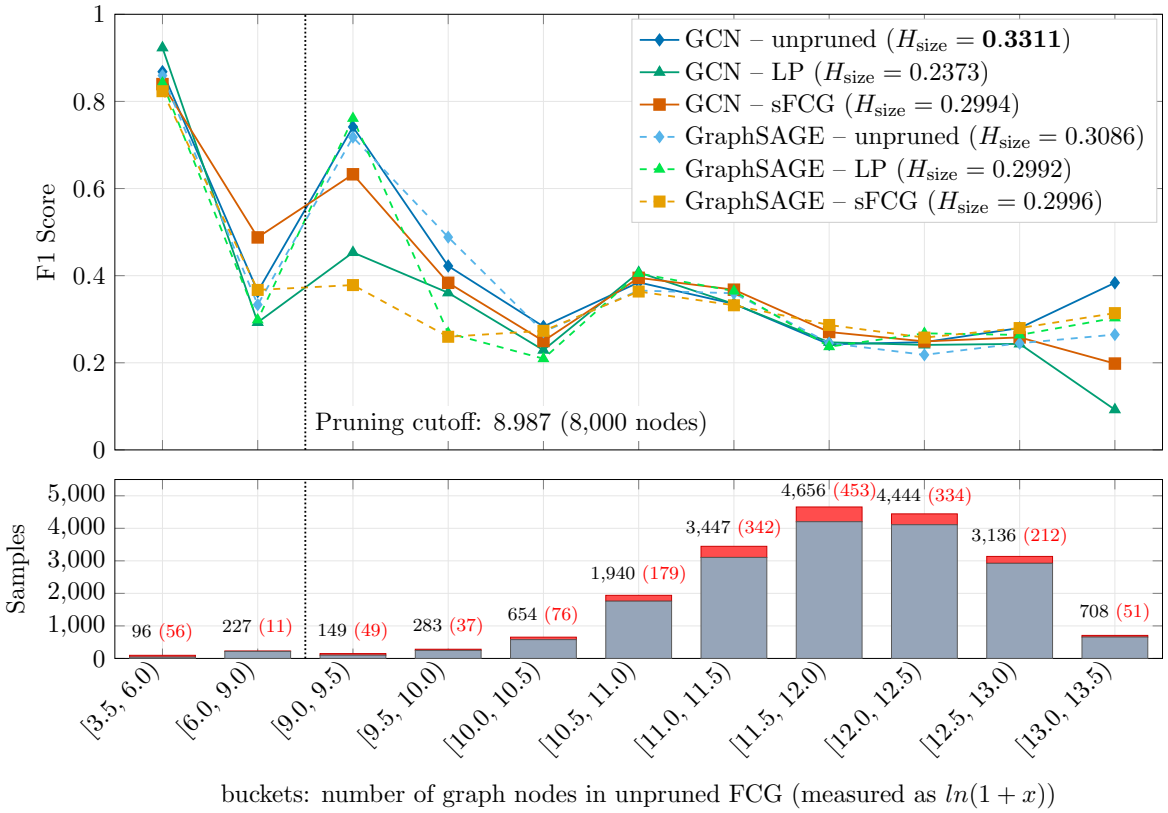


Figure 5.2: F1 performance across variable sizes of unpruned graphs (measured in the scale $ln(1 + x)$ of x number of nodes in the sample's unpruned FCG). The bottom graph indicates the number of samples that exist in that bucket along with the number of malware samples in red. H_{size} denotes the size-stratified harmonic F1 score across post-cutoff buckets.

ever the 12-month window is generally insufficient in providing evidence for concept drift, thus AUT is used as a supplementary metric of performance over the temporal evaluation period, rather than measuring temporal decay.

With respect to **RQ2**, we show no evidence that pruning provides an overall improvement in classification performance under either architecture. Its primary effect is instead altering the classifier's precision-recall trade-off, to favour higher malware recall at the expense of lower precision and higher false-positive rates. Given the computational reduction results of the previous section, it is then more reasonable to evaluate pruning as an efficiency and performance trade-off.

5.5 Evaluation Across Varying Sized Graphs

This section outlines the variability of performance of each run, across varying sizes of the original unpruned FCG.

To evaluate this, we first retrieve the original sizes of each FCG and denote groups of graphs as having similar size according to their $\ln(1+x)$ node count, and place them in bucket intervals of 0.5. This allows us to use log scales to effectively capture the fact that larger graphs occupy a larger space of values than small graphs. For example, the bucket $[9.0, 9.5)$ comprises graphs with node counts between $8,103 < x < 13,360$ whilst the bucket $[13.0, 13.5)$ comprises graphs with node counts between $442,413 < x < 729,416$.

We calculated the F1 for each bucket and plot the results in Figure 5.2, which additionally shows the distribution of samples in the testing dataset across buckets. This visualisation reveals a disparity between comparative dataset representation in terms of unpruned graph size, and the model's performance among those graphs. Overall, there is a downward trend in performance as the detected graphs grow larger, though this is an expected result in the case of malware detection, as larger graphs will contain significantly noisier features and thus negatively impact classification ability.

5.5.1 Size-Stratified Harmonic F1 score (H_{size})

We interpret these results by aggregating the F1 scores of each bucket using the harmonic mean, providing a conservative summary of performance. As this value is strongly reduced by poor performance in any individual bucket, it therefore rewards configurations which maintain comparatively strong F1 across the full range of graph sizes. We can calculate the harmonic mean F1 score across each bucket as:

$$H_{\text{size}} = \begin{cases} 0, & \text{if } \exists\, i \in \{1, \ldots, K\} \text{ such that } \mathrm{F1}(i) = 0, \\ \dfrac{K}{\sum_{i=1}^{K} \frac{1}{\mathrm{F1}(i)}}, & \text{otherwise.} \end{cases}$$

As this does not appear in any of our test runs, we can ignore the edge case of $\exists i \in \{1, \ldots, K\}$ `s.t.` $\mathrm{F1}(i) = 0$. Then, as each bucket contributes equally to H_{size}, irrespective of its sample count, our measurements of H_{size} in interpreting classification performance across graph sizes may only partially capture this effect for the edge buckets with fewer samples to inference. Therefore, this metric is not used as a complete evaluation of performance, rather as a diagnostic tool to indicate models with poor size-sensitivity.

5.5.2 Analysis

For Figure 5.2, H_{size} is calculated using only the buckets past the pruning cutoff, due to pruning not being enabled for such small graphs. For GCN models, the unpruned configuration achieves the highest size-stratified harmonic F1 of $H_{\text{size}} = 0.3311$, followed by sFCG at 0.2994 and LP at 0.2373. The comparatively low LP score reflects the persistently weak F1 performance across most buckets and especially apparent at the largest $[13.0, 13.5)$ bucket exhibiting a decline in performance relative to the unpruned configuration. For GraphSAGE models, each run produces similarly performing H_{size} scores suggesting GraphSAGE is less sensitive to structural changes introduced by pruning as the original graph size increases.

Regarding **RQ3**, FCGs of smaller size between buckets $[9.0, 9.5)$ and $[10.0, 10.5)$ seem to be negatively affected by pruning, as they are generally outperformed by the unpruned models, whilst the performance difference narrows amongst more commonly encountered graph sizes. This supports the hypothesis that FCGs of smaller size lose more critical information due to pruning than FCGs of larger size.

The signal dilution problem (as outlined in Section 3.2.2) might suggest that larger graphs will benefit performance-wise from graph pruning due to the reduction of irrelevant noisy information. From our findings, there is little evidence to support this hypothesis, though it may be noted that for very large graphs, GraphSAGE models only exhibit this behaviour to a negligible degree, but are surpassed by the GCN – unpruned model's performance.

5.6 Evaluation of Fidelity Metrics

This section explores how each measured graph pruning method may induce change to the faithfulness of GNN explanations that each model outputs. We evaluate faithfulness based on necessity and sufficiency of subgraph explanations in increasing subgraph sizes of each explanation's top-K edges.

To provide a controlled comparison of explanation behaviours, we evaluate the fidelity measurements on the intersection of malware samples correctly classified by all six evaluation runs. This produces a common set of 254 malware graphs, ensuring that differences in fidelity are not attributable to different configurations of incorrect predictions. We ignore benign graphs in general because analysing the behaviour of benign software is typically not relevant in malware detection.

This restriction may introduce a selection bias towards strongly identifiable malware samples, and consequently we can only evaluate the faithfulness for such malware, rather than the complete testing set of malware.

5.6.1 Fidelity Definitions

Before continuing, we first define what we mean by the term fidelity. Fidelity in the case of graph explanations comes in two forms:

- **Fidelity+** Denoting **Necessity**, is measured by the difference in classification confidence incurred when removing the generated subgraph explanation edges from the complete graph (probability-level), and whether the model's prediction changes from malware to benign (label-level).
- **Fidelity−** Denoting **Sufficiency**, is measured by the difference in classification confidence gained when retaining only the generated subgraph explanation edges of the complete graph (probability-level), and whether the model's prediction remains the same as malware (label-level).

GraphFramEx [2] presents a **characterisation score** which incorporates the equal-weight harmonic mean of necessity and sufficiency into one metric, rewarding methods that satisfy both properties:

$$\text{charact} = \frac{2(\text{fid}_+)(1 - \text{fid}_-)}{(\text{fid}_+) + (1 - \text{fid}_-)}$$

5.6.2 Analysis of Necessity

The strongest effect of pruning is observed for GCN – sFCG, where removing the Top-K explanation edges from the original graph changes the predicted label for 16.5% of samples at $K = 50$, increasing to 85% at $K = 1,000$. By comparison, GCN – unpruned reaches only 8.7% necessity at $K = 1,000$, while GCN – LP remains close to zero across all budgets. This signifies that GNNExplainer identifies substantially more decision-critical edges when GCN is trained on sFCG representations. The result is also identifiable for GraphSAGE – sFCG albeit the magnitude is substantially weaker than for GCN and plateaus beyond $K = 250$.

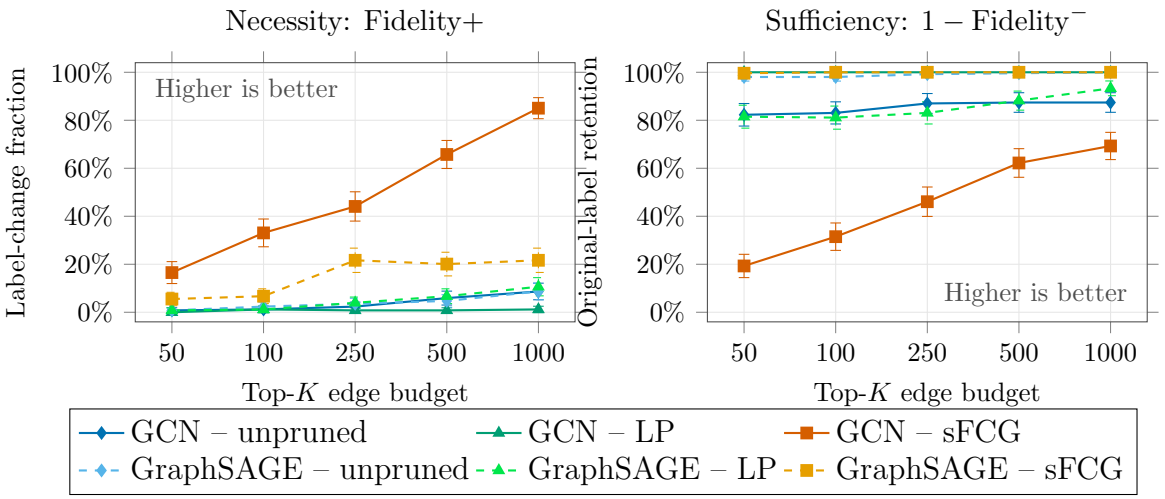


Figure 5.3: Label-level fidelity of Top-K edge explanations on correctly predicted malware graphs ($n = 254$). Error bars show mean ± 1.96 SE.

5.6.3 Analysis of Sufficiency

From our results, there is an inverse correlative behaviour between models having poor necessity and high sufficiency. GCN – LP for example exhibits both extremes, having nearly no necessity in the resulting classification, and being entirely sufficient to the classification. This combination suggests redundancy in the predictive structure whereby the selected explanation is sufficient to preserve the malware prediction, but alternative graph structures are apparently able to support the same prediction when the explanation edges are removed. By constrast, GCN – sFCG exhibits the same correlation in reverse, having very high necessity, and comparatively low sufficiency. Though as K grows, GCN – sFCG becomes increasingly both necessary and sufficient.

5.6.4 Analysis of Characterisation Score

Another interesting result is that both unpruned runs produce very similarly low characterisation scores throughout all Top-K edge budgets, however each pruned variant manifests vastly different behaviours across the different GNN models. sFCG in particular strongly improves characterisation upon its unpruned counterpart, especially pronounced in the GCN – sFCG run achieving 76.4% at a budget of $K = 1,000$, and GraphSAGE – sFCG achieving a score of 35.6%, at budget $K = 1,000$, improving from both GCN and GraphSAGE – unpruned baselines of 15.8% and 15.9% respectively. LP, on the other hand has diverging fidelity behaviour, with GraphSAGE – LP increasing characterisation to 19.1%, but GCN – LP vastly reduces it to 2.3%.

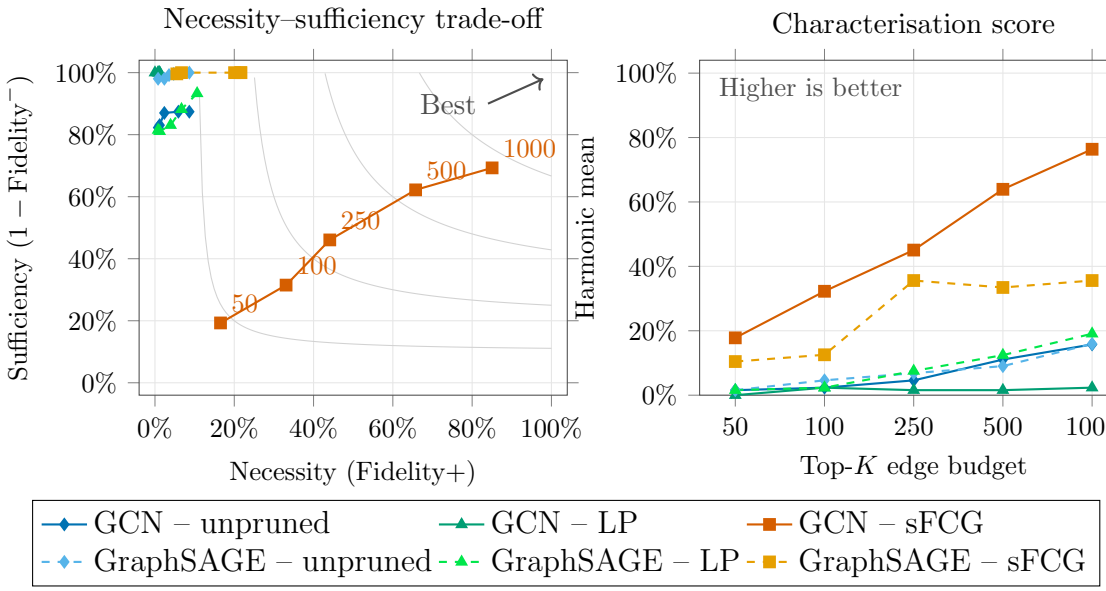


Figure 5.4: Overall quality of the Top-K explanations. The left panel shows the necessity–sufficiency trade-off; faint curves indicate constant characterisation-score contours. The right panel shows the characterisation score, defined as the equal-weight harmonic mean of Fidelity+ and $1 - \text{Fidelity}^-$.

Method	$K = 50$	$K = 100$	$K = 250$	$K = 500$	$K = 1,000$
GCN unpruned	1.6	2.3	4.6	11.1	15.8
GCN LP	0.0	2.3	1.6	1.6	2.3
GCN sFCG	**17.8**	**32.3**	**45.1**	**63.9**	**76.4**
GraphSAGE unpruned	1.6	4.6	6.8	9.0	15.9
GraphSAGE LP	1.6	2.3	7.5	12.4	19.1
GraphSAGE sFCG	10.4	12.5	35.6	33.4	35.6

Table 5.5: Characterisation score, in percent. The score is the equal-weight harmonic mean of Fidelity+ and $1 - \text{Fidelity}^-$; higher is better.

5.6.5 Analysis of Probability-Level Fidelity

We can again derive a similar analysis of results in examining the probability-level fidelity metrics as the label-level analysis. Where in GCN-sFCG, removing the top-K explanation edges produces increasingly large reductions in malware confidence (the necessity panel of Figure 5.5), reaching a mean probability change of 0.442 at $K = 1,000$. The corresponding changes for the unpruned and LP variants remain close to zero, indicating edges identified as explanations exert little influence on the resulting prediction.

For three of the models, GraphSAGE – unpruned, GraphSAGE – sFCG, and GCN – LP, we can observe that restricting the graph to the explanation edges (Figure 5.5: (sufficiency)) increases malware confidence, relative to having the complete graph.

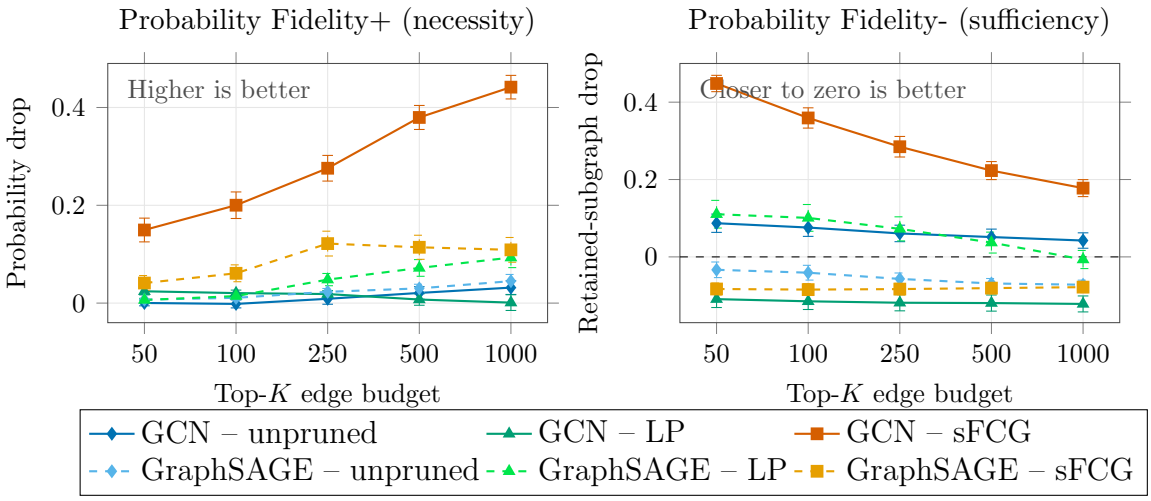


Figure 5.5: Confidence-level fidelity of Top-K edge explanations. Probability fidelity is a signed change in malware confidence. A negative Fidelity- value means that the retained subgraph receives higher malware confidence than the original graph. Error bars are mean ± 1.96 SE.

This appears to indicate that removing excess structure produces a stronger malware classification for these models, and that the removed structure exerts a dilutive effect on the malware signal. This is not necessarily evidence of explanation faithfulness as it also suggests that the model derives its classification through other pathways, likely not involving message passing but through learned heuristics or artefacts encoded in non-interacting node features.

Overall Analysis

Concerning **RQ4**, these results clearly demonstrate that graph pruning can substantially alter the faithfulness of GNN explanations, though the effect depends strongly on both pruning strategy and GNN architecture. GCN – sFCG produces the clearest improvement, where explanation subgraphs become increasingly both necessary and sufficient as the Top-K budget increases. Whilst GCN – LP provides weak necessity and suspiciously high sufficiency, suggesting weak faithfulness to the explanation as reflected in its corresponding characterisation score. These findings suggest that domain-informed pruning reduces redundancy in the structures supporting a model's prediction and therefore produces explanations with greater predictive capacity. However, it is important to note that GNNExplainer does not guarantee explanations which correspond to ground-truth malicious functions, therefore, fidelity is not necessarily a measure of explanation accuracy.

Chapter 6

Discussion

The research undertaken in this dissertation examines whether graph pruning can enable a scalable framework for GNN-based Android malware detectors. We explore four aspects of capabilities related to graph pruning in tackling our proposed research questions: scalability, performance, sensitivity to graph size, and explainability.

Our results show that pruning has significantly different effects across these objectives. We found that SeGDroid's sFCG substantially reduces graph size and GNN training time, but neither sFCG nor Leaf Pruning was found to improve their overall malware F1 score over the corresponding unpruned baselines. We show that classification performance among GNN classifiers is variable amongst different sized graphs, but are not variable amongst pruning methodology. Further, we show that pruning considerably alters the behaviour of graph explanations generated under GNNExplainer, with the strongest effect observed under the GCN – sFCG variant. These findings suggest that graph pruning should be primarily considered as a trade-off between computational cost, predictive performance, and explanation behaviour.

6.1 Scalability-Performance Trade-off

We find through sFCG, that domain-informed pruning can make otherwise costly GNN training substantially more feasible under constrained computational resources. Although, this is paired with an observable decrease in classification performance, particularly affecting the precision-recall trade-off by increasing malware recall, but at the expense of precision and higher false-positive rates. This potentially indicates that pruning may remove some structural information which contributes to the overall classification.

This trade-off between high computational speedup and small reduction in F1 may be acceptable in certain scenarios where resources are constrained, although the higher false-positive rate may eventually constrain resources if inaccurate classifications are costly.

6.2 Effect on Varied Graph Sizes

In Section 3.2.2 we argued that global graph pooling operations risk diluting malicious code with a large volume of benign functions. We attempt to address this by incorporating into our graph readout a MIL-style gated-attention pooling function to reduce these effects (detailed in Section 4.2.3). Therefore, we suspect that our unpruned results in Section 5.5 are partially influenced by this, and consequently, their differences in performance for large graphs are less pronounced. Though, we additionally acknowledge that due to the small sample sizes on the far end buckets, the analysis here may be rendered statistically insignificant. We can still note the variance in differences between GNN architectures in a qualitative analysis, where GCN runs appear to be more sensitive to graph representation than their GraphSAGE counterparts.

This motivates further investigation into how each pruned FCG is received by each message-passing method, and in turn how they interact with the graph pooling readout functions.

6.3 Explanation Faithfulness

We identify in Section 5.6 that sFCG changes GCN explanation fidelity whereby its characterisation score improves upon the unpruned baseline drastically, particularly due to its pronounced ability to capture necessary explanations.

One interpretation is that as sFCG concentrates the graph around sensitive API related call structures, the GNN is better able to identify malicious substructures crucial to its explanation. The result supports the hypothesis, briefly mentioned in Section 2.6, that graph pruning can 'focus' the model on malicious identifiers, however, does not necessarily result in better classification performance. As previously stated, fidelity does not establish that these identified explanations correspond to ground-truth malicious functionality, only that they are faithful to the GNN's classification.

We can note that GCN – LP, on the other hand has very high sufficiency, but virtually no necessity. This means that the classifier's prediction arises through alternative

structures that survive the removed explanation edges, potentially arising from

- strong node-level features,
- a weak dependence on edge interactions,
- some effect on the GNN due to the masking procedure.

Because explanation faithfulness is not determined solely by the pruning strategy, but emerges from an interaction between the graph representation and the downstream message-passing architecture, further investigation is required to determine which of these explain this behaviour.

6.4 Limitations and Threats to Validity

This dissertation was conducted under limited computational resources, along with limited amounts of time to conduct full training, validation and testing runs for different model variants. Therefore, under these conditions we used only a singular run per configuration. The differences between models cannot be cleanly separated from stochastic variation in parameter initialisation. Additionally, each model was restricted to a 50-epoch budget, and as all configurations achieved their best validation F1 score after 40 epochs, it is doubtful that convergence was achieved within this budget. Moreover, the size of the training dataset with 26,994 samples likely does not represent the full breadth of applications that comprise the Android ecosystem at the chosen temporal frame. However,Our primary limitation is the lack of range in our evaluated pruning strategies and GNN architectures, limited due to computational constraints.

Since these models were trained and evaluated over just a two-year time span, our findings may not generalise to later time periods, as we cannot guarantee that the models learn stable features. Further, our specific model may perform differently to different graph-generation and embedding pipelines.

Our evaluation makes use of metrics which may introduce bias in themselves. For instance, H_{size} should ideally be used for when there are sufficient sample counts in the extreme ends of bucket sizes, nevertheless, we use it to exemplify the variability of these extremes. Fidelity is another measure which does not evaluate accuracy of explanations, but of faithfulness to the model's classification.

Although we attempt to address the issue of over-smoothing in utilising skip-connections within both model's architectures, we do not evaluate them, nor do we

validate their effectiveness apart from preliminary testing results indicating their performance benefits. This may have somewhat limited the GraphSAGE model's capabilities.

6.5 Future Work

Future evaluations should repeat each architecture-pruning configuration across multiple random seeds, and report the mean variance of each measured metric. This would allow us to determine whether our observed differences are statistically relevant. Furthermore, future work can include incorporating evaluation across multiple time frames such that we can determine whether each pruning technique is generalisable, and not the result of temporal luck [8], or to train the model under a temporally invariant framework to improve stable representation learning [38].

We do not conduct any ablation studies; as such, future work may separately optimise each pruning method with each architecture, initialising with different hyperparameters to achieve better performance. This may extend to varying the $8,000$-node cutoff to different numbers, as from Figure 5.2, we can see each pruning method performs observably worse among smaller buckets post-cutoff than the unpruned counterparts, suggesting a higher cutoff could potentially improve performance, albeit at the expense of an increase in computational load.

Though we evaluate model explanations with fidelity, due to time constraints, we do not analyse specific instances of malware samples, which may be worth studying to understand what features and structures are identified by GNNExplainer to be malicious.

This dissertation focused on only two pruning strategies, despite other general-purpose pruning methods also existing, we identified sFCG and LP to be the strongest candidates for practical use due to their performance in their respective studies. As such, future work should also evaluate other pruning methods which may include: Node Centric Pruning [30], Walk Index Sparsification [28], Comp Prune [24], and potentially other techniques identified in [31], to verify our hypothesis.

Chapter 7

Conclusion

In this dissertation, we explore the various trade-offs, improvements, and weaknesses that graph pruning imposes on function call graph representations. We evaluate two state-of-the-art sparsification-based pruning methods: sensitive function call graph pruning (sFCG) and leaf pruning (LP) under GCN and GraphSAGE architectures using a spatio-temporal aware Android dataset, evaluating scalability, predictive performance, graph-size sensitivity, and explanation faithfulness.

We find sFCG is able to significantly reduce average epoch training time from $1,771$ to 586 seconds for GCN, and $1,675$ to 574 seconds for GraphSAGE, while neither pruning method improved upon its corresponding unpruned baselines on overall F1 score.

We find no convincing evidence for pruning becoming more beneficial for predictions as the original graph size grows. However, we do find pruning has a significant effect on explanation faithfulness, with its most pronounced effects measured on GCN – sFCG ($K = 1,000$), where the characterisation score reaches 76.4% versus 15.8% for GCN – unpruned.

Therefore, we conclude that graph pruning should not be treated as a lossless preprocessing step. Rather, our results indicate that whilst pruning alleviates computational burden, it may do so at the expense of predictive performance, and inducing structural changes upon the FCG changing the artefacts that GNNs learn upon. As such, the practical value of pruning lies in a trade-off between scalability, detection performance, and explanation faithfulness.

Bibliography

[1] K. Allix, T. F. Bissyande, J. Klein, and Y. Le Traon. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM, New York, NY, USA, 2016. URL http://doi.acm.org/10.1145/2901739.2903508.

[2] K. Amara, Z. Ying, Z. Zhang, Z. Han, Y. Zhao, Y. Shan, U. Brandes, S. Schemm, and C. Zhang. Graphframex: Towards systematic evaluation of explainability methods for graph neural networks. In B. Rieck and R. Pascanu, editors, *Proceedings of the First Learning on Graphs Conference*, volume 198 of *Proceedings of Machine Learning Research*. PMLR, 2022. URL https://proceedings.mlr.press/v198/amara22a.html.

[3] A. Arora and S. K. Peddoju. Ntpdroid: a hybrid android malware detector using network traffic and system permissions. In *2018 17th IEEE international conference on trust, security and privacy in computing and communications/12th IEEE international conference on big data science and engineering (TrustCom/BigDataSE)*, pages 808–813. IEEE, 2018. URL https://ieeexplore.ieee.org/document/8455983.

[4] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228. ACM, 2012. URL https://dl.acm.org/doi/pdf/10.1145/2382196.2382222.

[5] M. Backes, S. Bugiel, E. Derr, P. McDaniel, D. Octeau, and S. Weisgerber. On demystifying the android application framework:{Re-Visiting} android permission specification analysis. In *25th USENIX security symposium (USENIX security 16)*, pages 1101–1118. USENIX, 2016. URL https://www.usenix.org/system/files/conference/usenixsecurity16/sec16_paper_backes-android.pdf.

[6] T. Bilot, N. El Madhoun, K. Al Agha, and A. Zouaoui. A survey on malware detection with graph representation learning. *ACM Comput. Surv.*, 56(11), June 2024. ISSN 0360-0300. URL https://doi.org/10.1145/3664649.

[7] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146, 2017. ISSN 2307-387X. URL https://arxiv.org/pdf/1607.04606.

[8] T. Chow, M. D'Onghia, L. Linhardt, Z. Kan, D. Arp, L. Cavallaro, and F. Pierazzi. Beyond the TESSERACT: Trustworthy Dataset Curation for Sound Evaluations of Android Malware Classifiers. In *IEEE Conference on Secure and Trustworthy Machine Learning (SaTML)*. IEEE, 2026. URL https://arxiv.org/pdf/2506.23814.

[9] W. Guo, W. Du, X. Yang, J. Xue, Y. Wang, W. Han, and J. Hu. Malhapgnn: An enhanced call graph-based malware detection framework using hierarchical attention pooling graph neural network. *Sensors*, 25(2), 2025. ISSN 1424-8220. URL https://www.mdpi.com/1424-8220/25/2/374.

[10] W. Hamilton, Z. Ying, and J. Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30, 2017. URL https://dl.acm.org/doi/pdf/10.5555/3294771.3294869.

[11] Y. He, Y. Liu, L. Wu, Z. Yang, K. Ren, and Z. Qin. Msdroid: Identifying malicious snippets for android malware detection. *IEEE Transactions on Dependable and Secure Computing*, 20(3):2025–2039, 2022. URL https://ieeexplore.ieee.org/document/9762803.

[12] Y. Hei, R. Yang, H. Peng, L. Wang, X. Xu, J. Liu, H. Liu, J. Xu, and L. Sun. Hawk: Rapid android malware detection through heterogeneous graph attention networks. *IEEE Transactions on Neural Networks and Learning Systems*, 35(4): 4703–4717, 2021. URL https://ieeexplore.ieee.org/document/9524453.

[13] S. Hou, Y. Ye, Y. Song, and M. Abdulhayoglu. Hindroid: An intelligent android malware detection system based on structured heterogeneous information network. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1507–1515. ACM, 2017. URL https://dl.acm.org/doi/pdf/10.1145/3097983.3098026.

[14] M. Ilse, J. Tomczak, and M. Welling. Attention-based deep multiple instance learning. In *International conference on machine learning*, pages 2127–2136. PMLR, 2018. URL https://proceedings.mlr.press/v80/ilse18a/ilse18a.pdf.

[15] R. Jordaney, K. Sharad, S. K. Dash, Z. Wang, D. Papini, I. Nouretdinov, and L. Cavallaro. Transcend: Detecting concept drift in malware classification models. In *26th USENIX security symposium (USENIX security 17)*, pages 625–642. Usenix, 2017. URL https://www.usenix.org/system/files/conference/usenixsecurity17/sec17-jordaney.pdf.

[16] A. Kantchelian, S. Afroz, L. Huang, A. C. Islam, B. Miller, M. C. Tschantz, R. Greenstadt, A. D. Joseph, and J. D. Tygar. Approaches to adversarial drift. In *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, pages 99–110. ACM, 2013. URL https://dl.acm.org/doi/pdf/10.1145/2517312.2517320.

[17] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016. URL https://arxiv.org/pdf/1609.02907.

[18] X. Ling, L. Wu, W. Deng, Z. Qu, J. Zhang, S. Zhang, T. Ma, B. Wang, C. Wu, and S. Ji. Malgraph: Hierarchical graph neural networks for robust windows malware detection. In *Ieee infocom 2022-ieee conference on computer communications*, pages 1998–2007. IEEE, 2022. URL https://ieeexplore.ieee.org/document/9796786.

[19] X. Ling, Z. Wu, B. Wang, W. Deng, J. Wu, S. Ji, T. Luo, and Y. Wu. A wolf in sheep's clothing: Practical black-box adversarial attacks for evading learning-based windows malware detection in the wild. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 7393–7410. USENIX Association, Philadelphia, PA, Aug. 2024. ISBN 978-1-939133-44-1. URL https://www.usenix.org/conference/usenixsecurity24/presentation/ling.

[20] K. Liu, S. Xu, G. Xu, M. Zhang, D. Sun, and H. Liu. A review of android malware detection approaches based on machine learning. *IEEE access*, 8:124579–124607, 2020. URL https://ieeexplore.ieee.org/document/9130686.

[21] Z. Liu, R. Wang, N. Japkowicz, H. M. Gomes, B. Peng, and W. Zhang. Segdroid: An android malware detection method based on sensitive function call graph learning. *Expert systems with applications*, 235:121125, 2024. URL https://doi.org/10.1016/j.eswa.2023.121125.

[22] W. W. Lo, S. Layeghy, M. Sarhan, M. Gallagher, and M. Portmann. Graph neural network-based android malware classification with jumping knowledge. In *2022 IEEE Conference on Dependable and Secure Computing (DSC)*, pages 1–9. IEEE, 2022. URL https://ieeexplore.ieee.org/document/9888878.

[23] F. Martinelli, F. Mercaldo, and A. Saracino. Bridemaid: An hybrid tool for accurate detection of android malware. In *Proceedings of the 2017 ACM on Asia conference on computer and communications security*, pages 899–901. Association for Computing Machinery, 2017. URL https://doi.org/10.1145/3052973.3055156.

[24] H. Mohammadian, G. Higgins, S. Ansong, R. Razavi-Far, and A. A. Ghorbani. Explainable malware detection through integrated graph reduction and learning techniques. *Big Data Research*, 41:100555, 2025. ISSN 2214-5796. URL https://www.sciencedirect.com/science/article/pii/S2214579625000504.

[25] L. Nataraj, V. Yegneswaran, P. Porras, and J. Zhang. A comparative assessment of malware classification using binary texture analysis and dynamic analysis. In *Proceedings of the 4th ACM Workshop on Security and Artificial Intelligence*, page 21–30. Association for Computing Machinery, New York, NY, USA, 2011. ISBN 9781450310031. URL https://doi.org/10.1145/2046684.2046689.

[26] M. R. Norouzian, P. Xu, C. Eckert, and A. Zarras. Hybroid: Toward android malware detection and categorization with program code and network traffic. In J. K. Liu, S. Katsikas, W. Meng, W. Susilo, and R. Intan, editors, *Information Security*. Springer International Publishing, 2021. URL https://link.springer.com/chapter/10.1007/978-3-030-91356-4_14.

[27] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro. {TESSERACT}: Eliminating experimental bias in malware classification across space and time. In *28th USENIX security symposium (USENIX Security 19)*, pages 729–746. USENIX Association, 2019. URL https://www.usenix.org/conference/usenixsecurity19/presentation/pendlebury.

[28] N. Razin, T. Verbin, and N. Cohen. On the ability of graph neural networks to model interactions between vertices. *Advances in Neural Information Processing Systems*, 36:26501–26545, 2023. URL https://arxiv.org/pdf/2211.16494.

[29] R. Řehůřek and P. Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50. ELRA, Valletta, Malta, 2010. URL https://www.researchgate.net/publication/255820377.

[30] H. Shokouhinejad, R. Razavi-Far, G. Higgins, and A. A. Ghorbani. Node-centric pruning: a novel graph reduction approach. *Machine Learning and Knowledge Extraction*, 6(4):2722–2737, 2024. URL https://www.mdpi.com/2504-4990/6/4/130.

[31] H. Shokouhinejad, R. Razavi-Far, H. Mohammadian, M. Rabbani, S. Ansong, G. Higgins, and A. A. Ghorbani. Recent advances in malware detection: Graph learning and explainability. *arXiv preprint arXiv:2502.10556*, 2025. URL https://doi.org/10.48550/arXiv.2502.10556.

[32] L. Taheri, A. F. A. Kadir, and A. H. Lashkari. Extensible android malware detection and family classification using network-flows and api-calls. In *2019 International Carnahan Conference on Security Technology (ICCST)*, pages 1–8. IEEE, 2019. URL https://ieeexplore.ieee.org/document/8888430.

[33] K. Tam, A. Feizollah, N. B. Anuar, R. Salleh, and L. Cavallaro. The evolution of android malware and android analysis techniques. *ACM Computing Surveys (CSUR)*, 49(4):1–41, 2017. URL https://doi.org/10.1145/3017427.

[34] Y. Wu, X. Li, D. Zou, W. Yang, X. Zhang, and H. Jin. Malscan: Fast market-wide mobile malware scanning by social-network centrality analysis. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 139–150. IEEE, 2019. URL https://ieeexplore.ieee.org/document/8952382/.

[35] L. Yang, W. Guo, Q. Hao, A. Ciptadi, A. Ahmadzadeh, X. Xing, and G. Wang. {CADE}: Detecting and explaining concept drift samples for security applications. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2327–2344. Usenix, 2021. URL https://www.usenix.org/conference/usenixsecurity21/presentation/yang-limin.

[36] R. Ying, D. Bourgeois, J. You, M. Zitnik, and J. Leskovec. Gnnexplainer: Generating explanations for graph neural networks. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper_files/paper/2019/file/d80b7040b773199015de6d3b4293c8ff-Paper.pdf.

[37] R. A. Yunmar, S. S. Kusumawardani, F. Mohsen, et al. Hybrid android malware detection: a review of heuristic-based approach. *IEEE Access*, 12:41255–41286, 2024. URL https://ieeexplore.ieee.org/document/10473005.

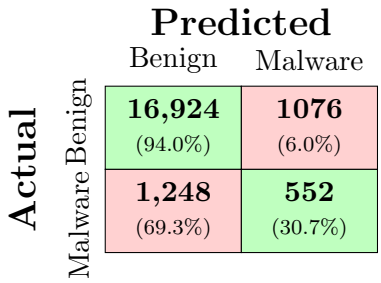
[38] X. Zheng, S. Yang, E. C.-H. Ngai, S. Jana, and L. Cavallaro. Tif: Learning temporal invariance in android malware detectors. *IEEE Transactions on Software Engineering*, 2026. URL https://ieeexplore.ieee.org/document/11541244.

Appendix A

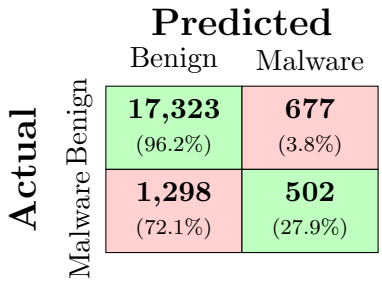
Supplementary Tables and Figures

Table A.1: F1 scores across graph-size buckets for GCN and GraphSAGE under unpruned, leaf-pruned (LP), and sFCG-pruned configurations. H_{size} denotes the size-stratified harmonic F1 score across post-cutoff buckets.

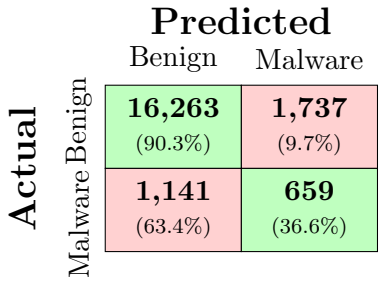
Bucket	**GCN**			**GraphSAGE**		
$\ln(1+x)$ **range**	**Unpruned**	**LP**	**sFCG**	**Unpruned**	**LP**	**sFCG**
$[3.5, 6.0)$	0.8682	0.9231	0.8397	0.8594	0.8421	0.8235
$[6.0, 9.0)$	0.3636	0.2927	0.4878	0.3333	0.2985	0.3673
$[9.0, 9.5)$	0.7414	0.4533	0.6327	0.7179	0.7611	0.3784
$[9.5, 10.0)$	0.4222	0.3607	0.3836	0.4884	0.2680	0.2597
$[10.0, 10.5)$	0.2833	0.2290	0.2500	0.2712	0.2094	0.2734
$[10.5, 11.0)$	0.3846	0.4074	0.3950	0.3658	0.4051	0.3636
$[11.0, 11.5)$	0.3352	0.3351	0.3678	0.3596	0.3646	0.3320
$[11.5, 12.0)$	0.2424	0.2470	0.2706	0.2452	0.2369	0.2868
$[12.0, 12.5)$	0.2473	0.2410	0.2491	0.2182	0.2676	0.2578
$[12.5, 13.0)$	0.2798	0.2434	0.2583	0.2449	0.2639	0.2797
$[13.0, 13.5)$	0.3836	0.0923	0.1982	0.2647	0.3030	0.3140
H_{size}	**0.3311**	0.2373	0.2994	0.3086	0.2992	0.2996



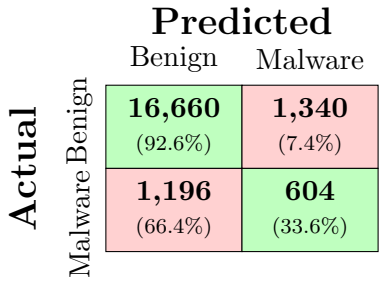
(a) **GraphSAGE (unpruned)**
Accuracy = 0.8826, F1 = 0.3221



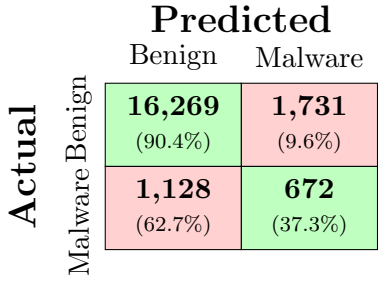
(b) **GCN (unpruned)**
Accuracy = 0.9003, F1 = 0.3370



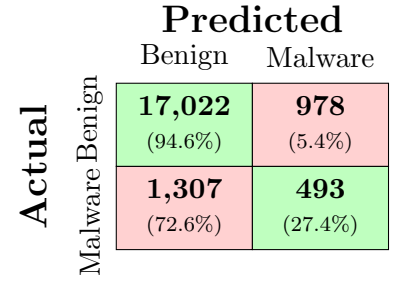
(c) **GraphSAGE (sFCG)**
Accuracy = 0.8556, F1 = 0.3141



(d) **GCN (sFCG)**
Accuracy = 0.8719, F1 = 0.3226



(e) **GraphSAGE (LP)**
Accuracy = 0.8546, F1 = 0.3198



(f) **GCN (LP)**
Accuracy = 0.8846, F1 = 0.3014

Figure A.1: Confusion matrices for each test run. Rows denote actual classes and columns denote predicted classes. Percentages are normalised by actual class.

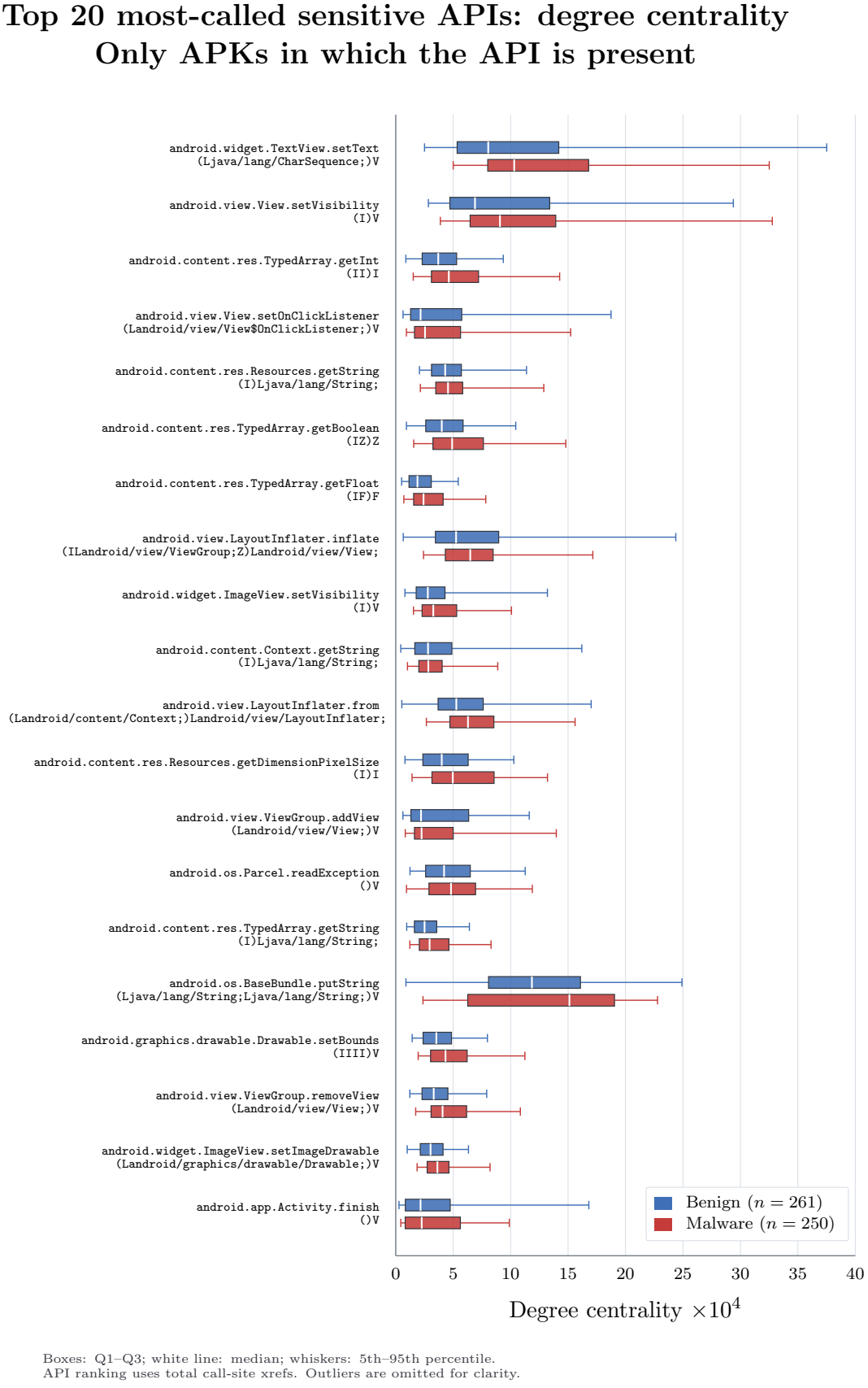


Figure A.2: Degree-centrality distributions for the 20 most-called sensitive APIs in a subset of 511 Android apks in the training set.

Table A.2: Distribution of samples across graph-size buckets. (60 benign samples are > 13.5 and excluded from the figure as no malware samples are found beyond $[13.0, 13.5)$.)

Bucket $\ln(1+x)$ **range**	**Total samples**	**Malware samples**	**Benign samples**
$[3.5, 6.0)$	96	56	40
$[6.0, 9.0)$	227	11	216
$[9.0, 9.5)$	149	49	100
$[9.5, 10.0)$	283	37	246
$[10.0, 10.5)$	654	76	578
$[10.5, 11.0)$	1,940	179	1,761
$[11.0, 11.5)$	3,447	342	3,105
$[11.5, 12.0)$	4,656	453	4,203
$[12.0, 12.5)$	4,444	334	4,110
$[12.5, 13.0)$	3,136	212	2,924
$[13.0, 13.5)$	708	51	657
Total	**19,740**	**1,800**	**17,940**

Table A.3: Epoch at which each model configuration achieved its highest F1 score.

Model	**Epoch with highest F1**
GCN – Unpruned	41
GCN – LP	41
GCN – sFCG	41
GraphSAGE – Unpruned	45
GraphSAGE – LP	45
GraphSAGE – sFCG	50

Table A.4: Label-level Fidelity+ (necessity), in percent. Entries are mean $\pm 1.96\,\mathrm{SE}$ over $n = 254$ graphs; higher is better.

Method	$K = 50$	$K = 100$	$K = 250$	$K = 500$	$K = 1,000$
GCN unpruned	0.8 ± 1.1	1.2 ± 1.3	2.4 ± 1.9	5.9 ± 2.9	8.7 ± 3.5
GCN LP	0.0 ± 0.0	1.2 ± 1.3	0.8 ± 1.1	0.8 ± 1.1	1.2 ± 1.3
GCN sFCG	16.5 ± 4.6	33.1 ± 5.8	44.1 ± 6.1	65.7 ± 5.8	85.0 ± 4.4
GraphSAGE unpruned	0.8 ± 1.1	2.4 ± 1.9	3.5 ± 2.3	4.7 ± 2.6	8.7 ± 3.5
GraphSAGE LP	0.8 ± 1.1	1.2 ± 1.3	3.9 ± 2.4	6.7 ± 3.1	10.6 ± 3.8
GraphSAGE sFCG	5.5 ± 2.8	6.7 ± 3.1	21.7 ± 5.1	20.1 ± 4.9	21.7 ± 5.1

Table A.5: Label-level sufficiency $1 - \text{Fidelity}^{-}$, in percent. Entries are mean $\pm 1.96\,\text{SE}$ over $n = 254$ graphs; higher is better.

Method	$K = 50$	$K = 100$	$K = 250$	$K = 500$	$K = 1,000$
GCN unpruned	82.3 ± 4.7	83.1 ± 4.6	87.0 ± 4.1	87.4 ± 4.1	87.4 ± 4.1
GCN LP	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0
GCN sFCG	19.3 ± 4.9	31.5 ± 5.7	46.1 ± 6.1	62.2 ± 6.0	69.3 ± 5.7
GraphSAGE unpruned	98.0 ± 1.7	98.0 ± 1.7	99.2 ± 1.1	99.6 ± 0.8	100.0 ± 0.0
GraphSAGE LP	81.5 ± 4.8	81.1 ± 4.8	83.1 ± 4.6	88.2 ± 4.0	93.3 ± 3.1
GraphSAGE sFCG	99.6 ± 0.8	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0

Table A.6: Probability Fidelity+ (necessity). Entries are signed probability changes, mean $\pm 1.96\,\text{SE}$; higher is better.

Method	$K = 50$	$K = 100$	$K = 250$	$K = 500$	$K = 1,000$
GCN unpruned	0.000 ± 0.006	-0.002 ± 0.008	0.009 ± 0.011	0.021 ± 0.013	0.032 ± 0.014
GCN LP	0.024 ± 0.007	0.020 ± 0.006	0.018 ± 0.007	0.007 ± 0.012	0.001 ± 0.016
GCN sFCG	0.149 ± 0.024	0.200 ± 0.027	0.276 ± 0.026	0.380 ± 0.025	0.442 ± 0.024
GraphSAGE unpruned	0.007 ± 0.003	0.011 ± 0.004	0.023 ± 0.007	0.030 ± 0.008	0.045 ± 0.014
GraphSAGE LP	0.006 ± 0.002	0.014 ± 0.004	0.048 ± 0.013	0.072 ± 0.017	0.093 ± 0.021
GraphSAGE sFCG	0.041 ± 0.015	0.061 ± 0.017	0.122 ± 0.025	0.114 ± 0.025	0.109 ± 0.025

Table A.7: Probability Fidelity- (sufficiency). Entries are signed probability changes, mean $\pm 1.96\,\text{SE}$. Values closer to zero indicate better confidence-level sufficiency; negative values mean the retained subgraph has higher malware confidence than the original graph.

Method	$K = 50$	$K = 100$	$K = 250$	$K = 500$	$K = 1,000$
GCN unpruned	0.087 ± 0.024	0.076 ± 0.023	0.061 ± 0.021	0.051 ± 0.021	0.042 ± 0.020
GCN LP	-0.109 ± 0.022	-0.115 ± 0.021	-0.119 ± 0.021	-0.119 ± 0.021	-0.122 ± 0.021
GCN sFCG	0.448 ± 0.021	0.359 ± 0.026	0.285 ± 0.027	0.223 ± 0.023	0.178 ± 0.022
GraphSAGE unpruned	-0.033 ± 0.020	-0.041 ± 0.019	-0.057 ± 0.015	-0.069 ± 0.013	-0.072 ± 0.012
GraphSAGE LP	0.110 ± 0.036	0.101 ± 0.035	0.072 ± 0.031	0.037 ± 0.027	-0.007 ± 0.023
GraphSAGE sFCG	-0.083 ± 0.015	-0.085 ± 0.015	-0.083 ± 0.015	-0.081 ± 0.015	-0.078 ± 0.014