# Methods in Learning Graph Representations of Programs for Malware Detection

Pascal U Lek Hou

April 21, 2026



**Abstract**

Malicious software poses a pervasive threat in the domain of cybersecurity. Malware authors systematically develop sophisticated evasion techniques such as code obfuscation and abstraction, to bypass defence mechanisms. Conventional signature-based detection have no mechanism for identifying zero-day attacks or code abstractions, conversely, traditional machine learning methods, leveraging static and dynamic heuristics, are increasingly hindered by concept-drift and act as black-boxes with no mechanism for identifying malicious patterns for analysis. To overcome the limitations of previous methods, graph representation learning (GRL) has emerged as a robust alternative, able to capture both a global view, and fine-grained information about a program. This review attempts to dissect the most relevant research papers and identify key challenges faced, and shortcomings present in the overall space of GRL malware detection. Though this review is particularly focused on GNN-based GRL pipelines, many of the challenges are also relevant for shallow embedding-based pipelines, and are thus discussed in some detail to highlight advantages of GNN-based pipelines. Namely, their end-to-end optimisation of learned representations, allow patterns to be learned directly through the loss function of the model. Further, shallow embeddings have no direct mechanism to implement explainable AI evaluators such as GNNExplainer.

This review walks through the entire GRL pipeline for identifying and analysing malware, from binary disassembly, code transformation and graph pre-processing, methods of graph representation in learned vector embeddings, and finally a brief discussion of classification techniques and explainability. The final main part of this review discusses challenges in the domain and potential avenues for future work. Further, the TESSERACT framework is invoked to discuss challenges with spatio-temporal bias and concept drift, where the vast majority of papers surveyed, did not meet the constraints laid out by the framework. As graph-based classifiers remain preeminent, they are increasingly susceptible to sophisticated problem-space adversarial attacks, malicious actors find creative ways of perturbing underlying graph structures while preserving executable functionality, thereby demanding the need for evaluation of model robustness in future classifier designs.

## 1 Introduction

Despite being heavily researched and having numerous sophisticated countermeasures and defence methods, malware continues to hold a large portion of attention in the field of cybersecurity. This can be attributed to the evolving nature of malware, designed by their authors to evade detection, often for prolonged periods of time, malware authors frequently employ various techniques to obfuscate, splice, or abstract code such that malware classifiers identify them as benign or goodware.

To that end, this has continually motivated researchers to innovate new techniques for identifying malware. Many modern methods of malware classification employ machine learning models which are trained on abstract representations of programs to capture the fundamental features of their underlying objectives. A commonly used technique is graph representation learning (GRL) which may represent programs as flow graphs from the direction of all function executions in the program (Function Call Graphs), or more granularly representing all possible paths that a program may traverse during its execution (Control Flow Graphs), though many other representation methods exist. A more advanced method of representation is to form a hierarchical graph representation, where an upper-layer graph may

be composed as a function call graph, and each node which represents each function of the program is attributed with an underlying control flow graph. This may allow for a more nuanced classification strategy to be learned by the model, as it offers both a global view of the program, embedded with more fine-grained execution patterns.

Machine learning models are not able to learn directly from graph structures however. Some further processing is required to transform a graph structure into vector embeddings which are representations in feature-space that summarise information about the graph, which can be used to train and inference a machine learning model, and finally passed to a classifier.

Many challenges are faced by security researchers when tackling malware. The adversarial dynamic between malware authors and researchers naturally leads to performance degradation of techniques over time, known in machine learning as concept drift, requiring new methods to compete against evolving newer forms of malware. Further, programs can have very complex structures which need to be represented by lots of features, requiring expensive computations, which further is not useful when an adversary is able to hide malicious portions of a malware within a sea of obfuscatory data.

One approach to tackling large and complex graphs is to prune the graphs to simpler forms, which ideally retain most of their critical information. This can be done by removing the graph's nodes and/or edges, known as graph sparsification, or reconstructing a new, smaller graph from the ground up while summarising the critical information, known as graph condensation, or by clustering nodes and edges and representing them as higher-level graphs, known as graph coarsening. The motivation for pruning is two-fold; to reduce the computational complexity of training and inference, and by removing excess information not relevant to the classification task, this might help increase the model's ability to successfully identify malware.

### 1.1 Structure of the Review

This review continues from Section 2 with important background information on previous methods of malware detection, discussing their motivations, limitations and a high level explanation of each detection method. Then follows a discussion on what core features are present in all types of malware, which are important for classification, and can be extracted through machine learning or identifiable for better pre-processing. For ease of reference, some background is then provided for definitions and types of graphs in context of program representation for GRL.

Section 3 walks through the steps required for transforming a program and its binary into a graph representation in preparation for training or inference on a machine learning model. Essentially, most research workflows make use of publicly available reverse engineering tools to first disassemble a program, and then construct a graph from the disassembled program. Then, we discuss methods of graph representation learning for malware detection, briefly outlining some modern methods which utilise the generalised pipeline, and then discuss the differences between a shallow embedding based-pipeline and the GNN-based pipeline. We further discuss graph classifiers and explainable AI, and their impact in the field of malware analysis.

Then, Section 4 focuses on challenges faced by researchers, and discusses shortcomings of existing methodologies according to the Tesseract framework [18]. The section concludes with a review on existing literature of the adversarial landscape in graph-based malware detection.

Finally we conclude with Section 5, where we discuss directions for future work, and a critical analysis of the field of malware detection through graph representation learning.

## 2 Background

Malware detection has undergone many phases in the history of the field. Traditionally, detection was done through signature matching which compares the digital fingerprint of suspected programs 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 [24]. The consequence of this form of detection is that any new malware which has never been encountered before (i.e, zero-day attacks) will have a new signature which has never been documented. And secondly, malware authors may cleverly modify pieces of their programs in ways that do not affect the program's semantics and still execute malicious content.

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 them identifiable among goodware. These heuristics can be separated into two categories; **Static features** - which are obtained by analysing the program's files, these features include permissions, embedded API calls, the manifest file's activity and broadcast receivers. And **dynamic features** - which are retrieved from executing the application in a configured environment, these features include system calls, network activity, battery and CPU usage, file operations. More advanced heuristic based detectors analyse both static and dynamic features, this can be done by combining the two in a single detection cycle which have showed can improve accuracy, as opposed to using the features separately [2, 14].

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. This naturally means that those defining the rules base their findings on patterns and similarities which can become outdated very soon in an adversarial space. Yunmar et al. [26] identify that many studies rely on outdated datasets that are not representative of modern malware. To keep up with the evolving threat landscape, instead of using human defined rules, research on leveraging machine learning and deep learning classifiers have proved to achieve very high accuracy. Though suffering the same problem with datasets becoming outdated, training new models on new datasets becomes a matter of computational resource, instead of human capital.

The popularity of machine-learning methods dwarfed knowledge-based heuristic detection due to their adaptability, which did not have to rely on static knowledge. Supervised learning algorithms such as support vector machines (SVMs), random-forest (RFs) and k-nearest neighbors (k-NNs) have been implemented [6, 1], motivated by promisingly high accuracy in results. A surprising method with promising results is to use Convolutional Neural Networks (CNNs) on grayscale images of binary files which, in theory, allow the model to detect patterns of malicious behaviour[17].

## 2.1 Feature Selection for Machine Learning

We have briefly mentioned static and dynamic features, as retrieved through static and dynamic analysis of the program's files and execution behaviour, 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. The information about a program is known as its features, and selecting which features are relevant for classification is known as feature selection (also known as feature engineering).

There are some challenges as to what features you want to include, as it may vary for each model what is applicable and relevant. For example, Taheri et al. [23], proposed a two-layer system that first analyses static features (permissions and intent) to first classify whether a program has malicious intent, and then passes the program to the second layer which analyses its dynamic features (API calls and network activity). The hybrid combination of static and dynamic analysis allowed the model to combine the strengths of both types of features. Broadly, the goal of feature selection is to select a set of features to maximise the model's ability to identify specific behaviours which correlate to different malware types.

### 2.1.1 Malware Features

Listed below are some features commonly extracted and used for malware classification.
**Static Features**

- **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 signify their capacity of exploitation, discrepancies between their stated purpose and requested permissions may reveal more about their malicious intent (e.g, a calculator app requesting for location data).
- **Intent** - is the basic communication mechanism used to exchange the inter- and intra-app messages. An intent conveys the intention of the app to perform some action. It specifies the label for a component, its category and action to be performed [10]. Malware often leverages implicit intents

to intercept system events to trigger background payloads without user intervention, so detection methods may leverage this to identify applications that listen for system broadcasts.

**Dynamic Features**

- **System Calls** - are how a program requests a service from the operating system kernel. Malware often exhibits characteristic patterns of system call usage, such as attempting to access restricted resources or perform unauthorised actions. Analysing the frequency and time stamps of which resources were accessed through system calls, analysts can uncover malicious behaviours and classify malware families according to the similarity of their feature vectors.[4]
- **Network Activity** - Communication artifacts generated by an application, including DNS features, TCP and UDP requests, packet origins and destinations, can be collected and examined. The motivation is that malware often need to interact with Command and Control (C2) servers or participate in botnets, analysing network activity and identifying unusual communication can reveal their malicious intent.[7]

**Hybrid Features**

- **API Calls** - Similar to system calls, API calls provide more semantic context into the functionality and behaviour of the executable, as APIs themselves have specific functionalities and their invocation can imply the behaviour of a program. API calls can be statically analysed by examining the Import Address Table (IAT), or dynamically by hooking functions during execution.
- **Entropy** - Describes a measure of randomness or uncertainty present in the file's contents. High entropy within a file will indicate the presence of encrypted or compressed data. As malware authors frequently use custom encryption methods to hide malicious payloads from static scanners, high entropy (within certain sections, e.g, `.text`) is one indicator of a program's obfuscation.
- **Graphs** - represent the structural and functional topology of a program, and are a popular representation for modern malware detectors, we will discuss this in greater detail in the following section.

## 2.2 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. There are many variations of graphs which may be used for program representation, below are some broadly defined common representations.

- **Attributed Graphs** - We may additionally want to attach information to elements of a graph, to create an attributed graph. Node attributes are mapped through a mapping function $F_n$ to assign a feature vector of $d_n$ elements, to nodes. In context, for program graphs, these attributes may represent stored variables such as strings, or assembly opcodes, or even sub-graphs of the functions within the program. Similarly, edge attributes are mapped are mapped through a mapping function $F_e$ to assign a feature vector of $d_e$ elements, to edges which may represent timestamps or data transmission sizes.
- **Hierarchical Graphs** - A more specialized form of attributed graphs, also known as Graphs of Graphs, Hierarchical graphs (HGs) involve 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 methodology of representing programs, one common way is to have a inter-function call graph of the decompiled binary and have an intra-control flow graph, as an attributed subgraph of each function.
- **Heterogeneous Graphs** - can model complex systems that involve multiple types of entities and relationships as nodes and edges. In context, malware often require large numbers of interactive entities independent of the binary file itself, thus HGs are able to capture these higher-level semantic relationships.

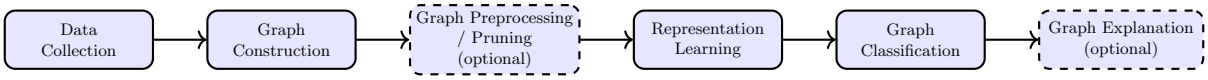


Figure 1: Generalised Graph Representation Learning pipeline.

#### 2.2.1 Program Graphs

A program can be represented in various ways in graph structure form. Below are some common methods of program graph representation.

- **Function Call Graphs (FCGs)** - are a method of representing the inter-procedural relationship between the set of 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 semantics but may lack fine-grained intra-procedural information.

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

- **Control Flow Graphs (CFGs)** - Compared to FCGs, CFGs are able to more granularly represent a program or function, as a set of possible paths to be taken throughout the program's execution. CFGs offer highly precise information about a function's semantics but can lack global context and may be weak at identifying code fragments that are operationally the same but syntactically different.

  Like FCGs, CFGs can be constructed statically by simply traversing the binary and identifying blocks and branch targets and connecting them according to the possible transfer of control through sequential execution, conditional branches, loops and indirect jumps. Dynamic analysis may also be used to obtain more accurate runtime control transfers, particularly if obfuscation or indirect branching is present in the binary, though similarly suffers limited execution coverage as FCGs.

- **API Call Graphs** - In the case of Android malware, a prevalent approach is to statically extract the API call sequences from the application and represent them using a FCG. This representation is particularly 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 not be able to represent non-API logic and can be sensitive to code obfuscation.

## 3 Methods for Learning Graph Representations of Programs

Graph representations of malware mark a significant paradigm shift from previous ML methods, despite their success in achieving high accuracies. By modelling malware as a graph, we may train a graph neural network (GNN) to learn structures and correlations which may have been unidentifiable by previous methods. This is known as graph representation learning (GRL), their success is in part due to their resilience against obfuscation techniques, and ability to retain their accuracy against evolving malware, known as concept drift. Further, the broader growth in academic interest for explainable AI (XAI) has given GRL an edge in becoming the leading malware detection method. Since most ML models serve as black-boxes, only being able to give predictions, they provide no explanation of how they came to their conclusion. GNNs have been researched as strong candidates for XAI due to the success of GNNExplainer [25], which provides interpretable explanations for predictions by taking a trained GNN and its predictions, and returning an explanation in the form of a small subgraph of the input graph, along with a small subset of node features that are most influential for the predictions.

This section walks through the general methodology and preprocessing of program binaries into graph representations such that they can be processed by ML models, and then discusses current GRL methodologies that employ hierarchical and heterogenous graphs as they represent the current state-of-the-art in malware detection. The overall GRL pipeline can be summarised as in Figure 1.

Table 1: Comparison of reverse-engineering tools for graph-based code analysis.

| Tool | Mode | Targets | Notes |
|---|---|---|---|
| Androguard | static (and limited dynamic) | Android APK/DEX | Android-specific disassembly; supports basic decompilation, and call-graph generation. |
| SootUp | static | JVM bytecode and APK related workflows | A Jimple-based static analysis framework, API call graphs are derived by restricting the call graph to library or framework calls. |
| Ghidra | static and dynamic | Native binaries, firmware | A Reverse Engineering platform; allows graph viewing, decompilation, scripting, and debugging. |
| IDA Pro | static and dynamic | Native/mobile binaries | A commercial disassembler, debugger and decompiler, supports cross-reference and import analysis. |
| radare2 | static and dynamic | Binaries, dumps, processes | CLI-based framework with graphing, scripting and debugging support. |
| angr | static and symbolic dynamic | Native binaries | Python-based analyser, allows symbolic execution of programs. Support for semantic analysis is available. |

## 3.1 Transforming code into a graph

Most studies follow a broadly similar pipeline in preparing source code for graph representation learning. Code is first extracted from the binary, in static analysis, the binary is typically disassembled into assembly code or decompiled into a higher-level language. Dynamic analysis instead would focus on the runtime-derived sequence of API calls or interactions between system entities. The extracted code or behavioral information is then converted by a graph constructor into a graph representation that should ideally preserve the program's semantics. In most cases, the two stages can be done simultaneously with reverse-engineering tools such as those listed in Table 1, which includes their operating modes, target binaries and ideal usage. The tools listed are all able to support construction of FCGs, CFGs, and API call graphs (but with additional scripting required for Ghidra and angr).

## 3.2 Graph Pruning

Once an initial graph is made of a program, it can likely be far too large to efficiently inference and train on directly. "Pruning" is require to reduce the graph into an efficiently computable state, and by removing excess information that is largely not relevant to the classification task, a GNN model may be able to focus better on malicious identifiers and achieve higher accuracy.

Broadly, there are three main categories of graph pruning:

- **Graph Sparsification** - reduces the number of edges in a graph while maintaining the same overall information. For program graphs, this means removing redundant data-flow, control-flow, or dependency edges that do not contribute to the learning objective.
- **Graph Condensation** - constructs a smaller graph that summarizes the behaviour of the original graph. Rather than pruning edges or nodes, condensation generates a compact surrogate representation that retains critical information for training and inference. This is useful for reducing prohibitively large graphs for efficient computation.
- **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 the same instructions as blocks, or generalizing instructions with similar semantics as the same block. Coarsening preserves a global structure, and may improve scalability.

Shokouhinejad et al. [22] identified three criteria for a pruning technique to be considered for malware detection:

1. The method must apply for graph classification, since the majority of malware detection methods revolve around this task.

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

3. The method is explainable, as mentioned previously, explainability is highly important in the field of malware research, as they allow practitioners to identify and validate the model's ability.

However, criteria 2 mainly applies to pruning techniques which are adapted from other graph classifier tasks and is not a hard rule for developing a graph pruning method. As a graph may degrade its usefulness if too much information is lost in pruning, some knowledge of malware heuristics and labelling on graphs may help ensuring that the pruned graph retains its critical information better. For example, Liu et al. [13] proposed SeGDroid which used a novel function call graph pruning method that employs sparsification, and relies heavily on labelling sensitive nodes. They achieve higher accuracy (from a baseline of 0.9757 to 0.9807) and found significant decrease in training time of around 40% (no exact figure given).

Other graph pruning methods have also similarly demonstrated increase in detection accuracy. Mohammadian et al. [16], construct CFGs and FCGs from binaries and apply a dedicated graph-reduction stage before passing the reduced graphs to a GCN classifier. Their reduction stage compares several pruning strategies, including component pruning, $k$-core, walk index sparsification, their proposed leaf prune, and using no pruning strategies (baseline). The authors found leaf prune to provide the highest accuracy of 0.954 (to a baseline of 0.945), however did not report the baseline time taken, thus no reduction in computation can be determined. However, Shokouhinejad et al. [21] demonstrated their proposed node centric pruning method to improve accuracy from the baseline of 0.85 to 0.89, and run time from 81.12 to 75.16 seconds.

To my knowledge, graph pruning is still an under-explored concept in the field of malware graph representation learning, there is currently questionable evidence for pruning techniques being able to increase accuracy in any meaningful way, though this is likely change from more extended research into the question. Though, the evidence for graph pruning being able to at least retain critical information is likely true from the above examples showing they do not degrade model performance.

## 3.3 Learning Graph Representations

After the pre-processing of a program transforming it into a graph representation, it is then possible to learn graph classifications of a labelled dataset of program graphs. Graph representation learning (GRL) is about transforming graphs from a simple data structure, into a form that can support predictive modelling. In malware detection, GRL is typically used for extracting patterns and "explanations" as subgraphs from labelled program graphs so that samples with similar behaviour or structural properties occupy similar regions of feature space. The main point is for malicious and benign samples to become more separable.

### 3.3.1 Graph Embeddings

Graph embeddings are a mapping of graph-structured data into a low-dimensional vector space such that important properties of the original graph are preserved. The embedded object may be a node, edge, subgraph, or the entire graph, and the goal is to encode structural relations, contexts, attributes, and semantics in a form that machine learning models can process efficiently. In modern graph representation learning, embeddings are typically produced either by shallow methods such as random-walk-based models, or by graph neural networks. These learned vector embeddings serve as compact representations for downstream tasks such as classification, clustering, and explanation.

Classically, GRL was done using shallow embeddings, which typically use random walk-based methods including Node2Vec [8], DeepWalk [19], Word2Vec [15], etc. One example is API2Vec proposed by Cui et al. [5]. They use a hierarchical inter-temporal process graph (TPG) with intra-temporal API graph (TAG), and use a heuristic random walk to generate behavioural paths intended to capture fine-grained intra- and inter-process malware behavior, while avoiding the API interleaving problem in multi-process programs. API2Vec passes these generated paths to pretrain a Doc2Vec model to learn path embeddings. The program's final representation is obtained by taking the average of a 64-dimensional embedding of all extracted paths from that sequence, and finally passing it to a malware classifier. This classifier is a standard ML model (such as $k$-NN, SVM, RF) trained on these embeddings.

A GNN-based GRL pipeline iteratively aggregates information from local neighborhoods through message passing and then, when needed, pools node representations into a graph-level embedding. For example, Ling et al. [11] proposed MalGraph, which used a hierarchical graph of an inter-FCG of $G_e$ and intra-CFGs $G_{f_l}$ as a representation of each program $e$. At each $G_{f_l} \in G_e$, GraphSAGE (A GNN-based model for learning node embeddings) is applied to obtain a hidden embedding $h_{l,i}^{(t_1)}$ of each block node of the CFG. Then, a max-pooling function is used to aggregate the embeddings as

$$h_{G_{f_l}} = \textbf{max-pooling}\{h_i^{(T_1)}\}_{i=1}^{i=|G_{f_l}|} \in \mathbb{R}^{d_1}$$

The graph is again passed through a GraphSAGE model over the FCG with attributed local embeddings $\{\dots h_{G_{f_l}} \dots\}$ to produce hidden embeddings $\{\dots z_k^{(t_2)} \dots\}$. These embeddings are max-pooled again to produce a single embedding for the program $e$ as

$$z_{G_e} = \textbf{max-pooling}\{z_k^{(T_e)}\}_{k=1}^{k=n} \in \mathbb{R}^{d_2}$$

Which is then passed to a prediction layer which consist of three multi-layer perceptrons and a sigmoid function to obtain the final prediction probabilities. The methodology of MalGraph is made explicitly clear to use the simplest, and straightforward pooling function for efficient and effective results. Their results prove to be promising and serve as an important initial attempt at the full GRL malware detection pipeline.

Guo et al. [9] proposed a more complicated, transformer-based framework MalHAPGNN, by representing programs first as FCGs (of initial graph form $G_i = (V_i, E_i, H_i)$) where function nodes $v \in V_i$ are assigned an initial feature vector $h \in H_i$, obtained through a BERT-based function embedding method. This is done so that the input graph consists of structural and semantic information from the graph relationships and node attributes respectively. The model applies a graph attention layer (GAT) to update each node representation by aggregating information from neighbors through a LeakyRELU function and applying an initial softmax layer

$$\alpha_{ij} = \text{softmax}_j(\text{LeakyReLU}(e_{i,j}^\top [Wh_i \,\|\, Wh_j]))$$

allowing for important functions to contribute more strongly to the new embeddings, and then averaging the output vectors to improve the model's generalization ability

$$\mathbf{h}_i' = \sigma(\frac{1}{K}\sum_{k=1}^{K}\sum_{j\in\mathcal{N}_i}\alpha_{ij}^{(k)}W^{(k)}\mathbf{h}_j)$$

Then a hierarchical pooling stage ranks nodes according to a "node-importance" score, and retains only the top-ranked nodes to form a smaller induced subgraph.

$$\text{NIV} = \sigma(\sum_{p=1}^{P}\omega_p\,\text{NIV}_p)$$

This pooling stage may disconnect important parts of the graphs, therefore the model additionally reconstructs sparse connections among the retained nodes. These steps are repeated over multiple layers to produce progressively coarsened graphs, and the final graph embedding of the program is obtained by summing graph-level vectors $r$ from each level $k$ as

$$r_i = \sum_{k=1}^{K} r_i^{(k)}$$

obtaining a multi-scale representation that can be passed to a classifier.

The general differences in the pipelines for shallow embedding-based GRL and GNN-based GRL can be summarised in Figure 2. Essentially, for shallow embedding pipelines, embedding features are usually learned before downstream classification, often with a separate unsupervised objective and do not naturally capture graph attributes. In GNN pipelines, representations are typically learned end-to-end through message passing and at the graph-level readout therefore attributes are directly learned. Further, GNN pipelines typically learn embeddings that minimize classification error, therefore the embeddings are influenced by the loss function, whereas in shallow embedding pipelines, the objective is learned separately.

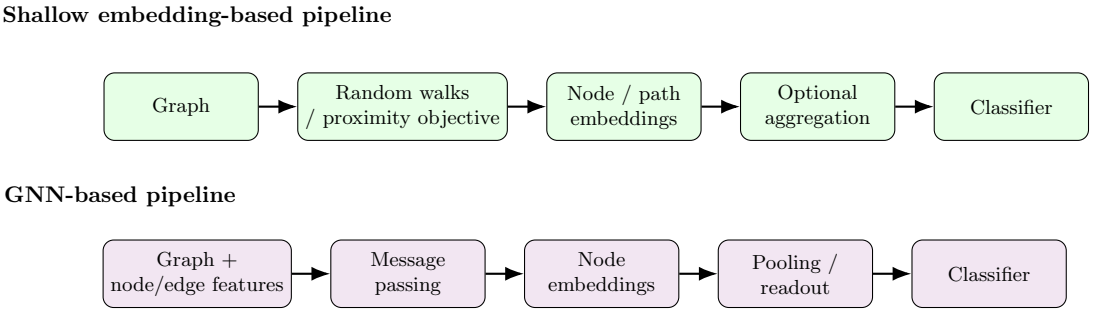


Figure 2: Generalised comparison between shallow embedding-based and GNN-based graph learning pipelines.

### 3.4 Graph Classifiers and Explainable AI

After a graph is learned and transformed, typically as a learned vector representation as embeddings in a higher-dimensional feature space. The final stage is to perform classification on the learned program graph embedding. For shallow based-embedding pipelines, this is typically done by first extracting a fixed graph embedding, then passing it to a conventional classifier such as $k$-NN, SVM or Random Forest. For GNN-based pipelines, classification is more commonly integrated into the model itself, where graph-level embeddings produced through message passing and pooling are passed to a prediction head. For example, MalGraph [11] utilised three multi-layer perceptrons followed by a final sigmoid function to obtain the final prediction probabilities.

$$p_e = \mathbf{sigmoid}(\mathbf{MLP}_{\theta 3}(\mathbf{MLP}_{\theta 2}(\mathbf{MLP}_{\theta 1}(z_{G_e}))))$$

The key difference is that for shallow-based, representation learning and classification are separated, whilst GNN-based are usually optimised jointly. This end-to-end optimisation is often advantageous for accuracy as the classification objective directly shapes the learned representation.

However, this results in a lack of transparency of the model as it becomes difficult to identify weights and embeddings in a way that humans can understand. This black-boxing of the model introduces the need for explainability, since malware analysts are required to do more than just identify malware, they must also analyse and understand their behaviours and structure. This problem was directly tackled by Ying et al. [25] in their proposed GNNExplainer, which is a post-hoc explainability method for GNNs that, given a trained GNN and one of its predictions, identifies a subgraph and subset of node features that were most influential for that prediction. This is particularly useful in malware analysis in identifying malicious behaviour such as in function call relationships or sensitive API usage, and enables explainable AI (XAI) to bridge the gap between malware detection and interpretable malware analysis.

It is important to note that XAI can only explain the reasoning of a trained model, and does not necessarily provide the true semantic cause of maliciousness in the program itself. Identified subgraphs may simply be correlations learned from bias in datasets, and there is little research on evaluating the quality of explanations. Therefore, in its current form, XAI should primarily be treated as a diagnostic and validation tool, rather than a complete solution to the black-box problem.

## 4 Challenges in GRL

Despite promising capabilities and high accuracy, GRL malware detectors face several challenges that hinder the development of fully resilient defence systems. Further, as the field of malware detection is vast and can be quite complex, there are many components within the generalised GRL structure that can be individually worked on by researchers, and each component can be evaluated on multiple aspects, making direct comparisons between methodologies a challenge. It may not simply be enough to evaluate a methodology based on the reported detection accuracy, as there can be other factors that play a part in how well a model fits its stated objective. Figure 3 provides an overview of challenges present in the current state of GRL-based malware detection, further discussed in this section.

One significant challenge is the absence of a unified benchmarking system, this deficiency is largely attributable to the continuously evolving nature of malware, causing static datasets and benchmarks to rapidly lose their relevance over time. While many proposed GRL models report high performance

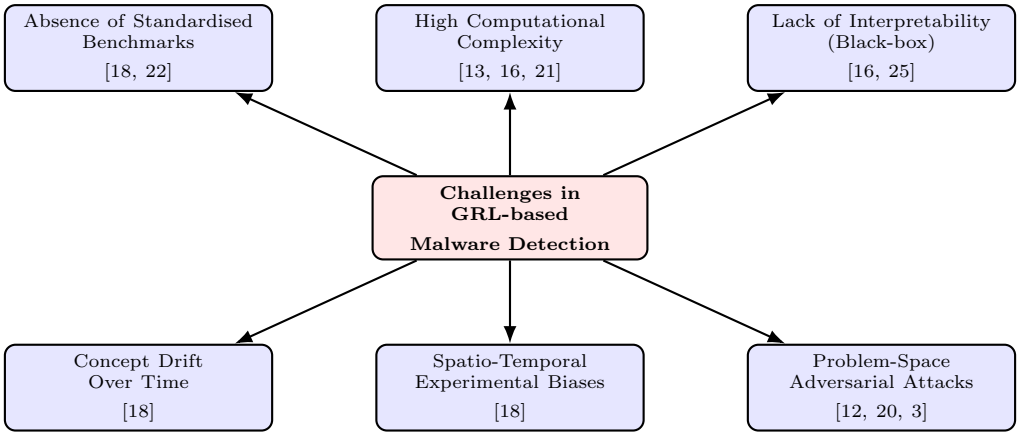


Figure 3: Overview of primary challenges in graph representation learning (GRL) for malware detection.

metrics, they are seldom operating on the same sets of data, researchers often supplement baseline datasets with their own collected data, thus making empirical comparisons between studies difficult. Further, as training and inference of different models require different input data, namely different graph structures, the input data has to be regenerated for each compared methodology.

Computational complexity is a challenge previously mentioned, and directly tackled by graph pruning techniques. However, as an under-explored field of study, is not well addressed in the current literature. Adversaries might seek to exploit this fact either by exploiting weak pruning methods that remove too much crucial information, or they may conceal malicious code within an exceedingly large program file, knowing that current models have difficulty identifying important information among such large target programs.

As mentioned in the previous section, the 'black-boxing' of deep learning architectures, which include GNNs, have hindered interpretability of results, and have introduced a need for explainability. Additionally, though robust explainability techniques such as GNNExplainer are able to provide insight into a GNN model's rationale for predictions, there is no guarantee of an interpretable explanation in that an explanation may simply be a model's learned correlation of bias in datasets. Further, just as a researcher may be able to derive correlations in malware programs, so can a malware author reverse engineer new obfuscation techniques to evade detection.

Therefore, as graph-based detectors become more solidified as the leading detection framework, they naturally become targets of evasion techniques. The adversarial dynamic between malware authors and security researchers eventually leads to degradation in performance of detection techniques over time. This is a phenomenon in machine learning known as concept drift, where the statistical properties of the data analysed are time dependent, in this case, the characteristics and structures of malware in a given period of time, will change due to an adversarial dynamic, or from just general trends in the malware community. Pendlebury et al. [18] proposed the TESSERACT framework to evaluate malware classifiers under strict temporal constraints. Meaning that, by ensuring testing data is strictly post-date of training data, researchers can gain a more accurate reflection of a model's true capability to detect future, unseen threats in real-world scenarios. The authors also found that spatial bias introduced in the testing phase can skew a model's detection rate, especially if the ratio of malicious and benign software is unrealistic (the authors suggest 10% as a realistic ratio), results will likely not correspond to real world settings as malware is much less frequently encountered than goodware. The TESSERACT framework introduced several key contributions to the malware space, including spatio-temporal constraints for experiment design (**C1**: no training on future data, **C2**: goodware and malware time windows must align, **C3**: realistic class-ratios for testing), a proposed area under time (AUT) metric to summarize robustness under time decay, and a procedure for tuning imbalanced malware settings.

In practice, few papers indicate practicing any of the three constraints laid out by TESSERACT, and to my knowledge, no graph-based detector has yet been proposed, which follows all three constraints. In the literature reviewed thus far, API2Vec [5] comes the closest with **C1** and **C2**, but fails to provide a realistic malware to goodware split (2,782 and 1,309 samples respectively) in their testing, and thus does not satisfy constraint **C3**. Likely future work can involve validating past models according to these constraints laid out by TESSERACT, and further evaluation of each model's robustness using an AUT graph.

### 4.1 Adversarial attacks

Expanding on adversarial techniques, evasion still poses a difficult challenge for detectors, GRL-based techniques can inherit broader vulnerabilities of machine learning systems that are susceptible to adversarial manipulation. In this setting, attackers may be able to induce changes in the underlying graph structure, such that detectors output a benign classification, while the malicious semantics remain in the program [3].

This problem is not unique to graph-based techniques but differ somewhat in the perturbation techniques common for image classification. For an image, a valid attack should not perturb the image in any way that is visibly noticeable to a human. However, for a program, perturbation includes modifying function call invocations, modifying jump calls, adding and removing functions or opcodes, all for the purpose of perturbing the graph representation, and shifting the graph embedding. Pierazzi et al. [20] formalise this through the notion of problem-space attacks where an attack is carried out on the real program object, rather than on its extracted feature vector. Since changing a graph in feature-space is only meaningful if one can realise the corresponding change in the executable while satisfying domain-specific constraints, where the key issue is ensuring that the underlying malware still remains functional.

Recent work has been done on addressing this gap between feature and problem space perturbations. Ling et al. [12] proposed MalGuise, which is a black-box adversarial framework for evading learning-based Windows malware detectors through fine-grained manipulation of a malware's derived CFG. MalGuise relies on a semantic preserving transformation called 'call-based redividing', which alters both CFG nodes and edges while reconstructing a valid PE file. The results reported have shown to have real world evasion capabilities, able to evade current anti-virus software with significant attack success rate (ASR), where MalGuise achieved $> 30\%$ ASR in 4 of the 5 evaluated anti-viruses.

Overall, the literature on adversarial attacks suggests that GRL-based detection should not be assessed solely on accuracy, nor solely on its ability to retain capabilities under spatio-temporal constraints, but should also be tested on robustness under adversarial conditions.

## 5 Conclusion

The recent paradigm shift towards graph representation learning has provided security researchers a strong framework for identifying complex and evolving malware. Through the modelling of programs in graph structure, machine learning models, particularly graph neural networks are able to learn intricate representations of programs that are fundamentally more resilient to code obfuscation and concept drift, compared to traditional heuristics or shallow-embedding methods. However, despite achieving promisingly high detection rates, a critical analysis of the current state-of-the-art reveals several foundational deficiencies and under-explored components that hinder fully resilient defense systems.

A fundamental challenge in the field is reducing the computational complexity of inferencing on exceedingly large graphs. Advanced methodologies frequently utilise sophisticated graph structures, such as hierarchical graphs to capture both global context and fine-grained execution patterns. Consequently, pruning techniques are essential to reduce this prohibitively large data into an efficiently computable state. However, graph pruning remains an under-explored domain within malware detection, contrary to the recommendation by Shokouhinejad et al. [22], researchers may benefit from tailoring graph pruning techniques to identify node and edge labels to prioritise pre-labelled important information. Further research should be done to identify empirical differences in either approach, one concern is that adversaries may be able to exploit such tailored pruning techniques to obfuscate malicious code, future work must focus on validating that pruning strategies are robust against code obfuscations and adversarial perturbations, while simultaneously ensuring efficient computations.

Furthermore, the inherent black-box nature of deep learning architectures are a significant obstacle for malware analysts requiring interpretable insights to malicious behaviour. Explainable AI tools, such as GNNExplainer [25] bridge the gap between detection and interpretability by highlighting influential subgraphs and features, however, do not necessarily identify the true semantic cause of a program's maliciousness. For now, GNNExplainer serves as a powerful diagnostic tool, but is not a complete solution to the black-box problem.

The lack of a unified standardised benchmarking system hinders researchers in making empirical comparisons of their models. This is attributed to the continually evolving nature of malware, where static datasets quickly lose relevance in this space, and researchers frequently using custom datasets. Though not a complete solution, researchers may be able to more directly compare results if future work follows strict adherence to the TESSERACT framework. In particular to the three constraints; **C1**: no

training on future data, **C2**: goodware and malware time windows must align, **C3**: realistic class-ratios for testing. Additionally, evaluating future models using an Area Under Time metric will be necessary to accurately summarise robustness of models, against concept-drift.

Finally, as graph-based classifiers continue to be the leading detection framework, they naturally invite the development of targeted evasion techniques. Adversaries increasingly become more capable at executing problem-space attacks, where the underlying graph structure is perturbed, such that its representation in feature space becomes misidentified by classifiers. Frameworks such as MalGuise [12] demonstrate high attack success rates against commercial anti-viruses deploying graph-based detectors, by modifying function invocations or adding opcodes, while simultaneously maintaining the malicious semantics of the malware. Future work in graph representation learning should be done in evaluating models, not solely on their detection accuracy or temporal resistance, but to include a measure of resilience against problem-space adversarial attacks, such that methodologies evaluated can be determined to be secure and reliable malware detection systems.

# References


[1] F. Alhaidari, N. A. Shaib, M. Alsafi, H. Alharbi, M. Alawami, R. Aljindan, A.-u. Rahman, and R. Zagrouba. Zevigilante: Detecting zero-day malware using machine learning and sandboxing analysis techniques. *Computational Intelligence and Neuroscience*, 2022(1):1615528, 2022. URL https://doi.org/10.1155/2022/1615528.

[2] 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.

[3] 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.

[4] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 15–26. Association for Computing Machinery, 2011. URL https://doi.org/10.1145/2046614.2046619.

[5] L. Cui, J. Cui, Y. Ji, Z. Hao, L. Li, and Z. Ding. Api2vec: Learning representations of api sequences for malware detection. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, page 261–273. Association for Computing Machinery, New York, NY, USA, 2023. ISBN 9798400702211. URL https://doi.org/10.1145/3597926.3598054.

[6] F. C. C. Garcia and F. P. M. II. Random forest for malware classification. *CoRR*, abs/1609.07770, 2016. URL http://arxiv.org/abs/1609.07770.

[7] S. Garg, S. K. Peddoju, and A. K. Sarje. Network-based detection of android malicious apps. *International Journal of Information Security*, 16(4):385–400, 2017. URL https://link.springer.com/article/10.1007/s10207-016-0343-z.

[8] A. Grover and J. Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 855–864. Association for Computing Machinery, New York, NY, USA, 2016. ISBN 9781450342322. URL https://doi.org/10.1145/2939672.2939754.

[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] F. Idrees, M. Rajarajan, M. Conti, T. M. Chen, and Y. Rahulamathavan. Pindroid: A novel android malware detection system using ensemble learning methods. *Computers & Security*, 68: 36–46, 2017. ISSN 0167-4048. URL https://www.sciencedirect.com/science/article/pii/S0167404817300640.

[11] 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/abstract/document/9796786.

[12] 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.

[13] 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.

[14] 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.

[15] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013. URL https://doi.org/10.48550/arXiv.1301.3781.

[16] 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.

[17] 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.

[18] 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.

[19] B. Perozzi, R. Al-Rfou, and S. Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710. Association for Computing Machinery, 2014. URL https://doi.org/10.1145/2623330.2623732.

[20] F. Pierazzi, F. Pendlebury, J. Cortellazzi, and L. Cavallaro. Intriguing properties of adversarial ml attacks in the problem space. In *2020 IEEE symposium on security and privacy (SP)*, pages 1332–1349. IEEE, 2020. URL https://ieeexplore.ieee.org/abstract/document/9152781.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.