Evaluating Behavior Graph Reduction Strategies for Machine Learning-Based Malware Detection

Samy Bettaieb*, Serena Lucca*, Charles-Henry Bertrand Van Ouytsel, Etienne Rivière
UCLouvain, Louvain-la-Neuve, Belgium
Emails: {samy.bettaieb, serena.lucca, charles-henry.bertrand, etienne.riviere}@uclouvain.be

***Abstract*—Graph-based representations of program behavior are a powerful foundation for machine learning-based malware detection. However, the large size and complexity of these behavior graphs pose scalability challenges. This paper presents a systematic evaluation of five graph reduction strategies—covering both coarsening and sparsification—designed to simplify graphs while preserving meaningful behavioral features. Using dynamic analysis data from Windows PE32 binaries, we analyze the impact of these reductions on computational efficiency, detection performance, and model robustness against adversarial mimicry attacks. Our results show that several strategies substantially reduce graph size and extraction time without significant accuracy loss. We also find that coarsening based on API calls' action maintains stronger robustness to adversarial manipulation.**

***Index Terms*—Malware Analysis, Dynamic Analysis, Behavior Graphs, Graph Neural Networks, Graph Reduction Strategies**

I. Introduction

Cybersecurity threats evolved rapidly in recent years, with increasingly sophisticated malware targeting both individuals and organizations [1]. This calls for robust and efficient malware detection mechanisms that can accurately identify malicious software samples based solely on their binary code. Static analysis, which consists of examining the executable binary without executing it, has been widely used for this purpose (e.g., YARA rules [2]). This relatively simple approach is scalable to large binary collections. However, static analysis is also vulnerable to evasion techniques such as syntactic obfuscation, packing, encryption, or polymorphism [3]. To overcome these limitations, dynamic analysis offers an alternative by observing the behavior of executable binary samples during execution (e.g., by monitoring system or API calls).

Machine learning (ML) techniques have become central in modern malware detection, offering the ability to generalize beyond known threats and recognize novel variants, thus addressing the principal limitation of signature-based methods [4]. The performance of ML approaches depends not only on the quality of training data but also on the representativeness of the extracted features. In particular, raw feature vectors derived from behavioral information (e.g., API calls, network interactions, or file system accesses) often lack the capacity to capture the structural and semantic relationships between events, which would characterize malware behavior [5].

Graph-based feature representations emerged as a promising approach for modeling the relationships between behavioral events and addressing these limitations. From such representations, ML models can learn to identify behavioral patterns and improve detection performance [6]. In particular, execution traces composed of API calls can be used to create a *behavior graph* [7]–[9], in which nodes represent API or system calls and edges represent dependencies between these calls.

However, building and using behavior graphs comes at a cost. Some programs can execute thousands of API calls. Depending on the graph-building strategy employed, we may end up with extremely large graphs, making detection prohibitively costly in terms of both memory and computational time. Yet, it remains unclear whether this graph complexity is necessary to achieve good detection performance. Graph reduction techniques precisely aim to reduce the size of such graphs while preserving semantic information.

Reduction methods for graph representations have been extensively studied for *general* graph learning tasks. In a recent survey, Hashemi et al. [10] distinguish two types of graph reduction strategies that preserve graph properties. Graph *sparsification* consists of selecting a subset of existing nodes and edges from the graph. Graph *coarsening* builds super-nodes and super-edges that map to multiple original nodes and edges. Reducing the size of feature representation graphs inherently implements a tradeoff with the accuracy, representativeness, and robustness of the resulting model(s).

While several approaches to ML-based malware detection using dynamic analysis proposed ad-hoc solutions to graph reduction [8], [11], [12], the different contexts, datasets, and models make it hard to compare how they implement this tradeoff. There is also a lack of analysis of the robustness of graph reduction methods, e.g., against standard adversarial strategies such as the mimicry attack [13].

Contributions. In this paper, we conduct the first systematic evaluation of the impact of various graph reduction strategies on dynamic behavior-based malware detection. We focus on Graph Neural Network (GNN) classifiers. We describe and evaluate five graph reduction strategies that aim to reduce graph complexity while preserving relevant behavioral patterns. Some of these strategies apply sparsification by restricting graphs to a fixed number of distinct calls or a selected subset of API calls relevant for malware detection. Others apply graph coarsening by grouping related calls that share the same name or belong to the same higher-level category.

Our experimental pipeline begins with the dynamic execu-

*These authors contributed equally to this work.

tion of samples using the Cuckoo sandbox on a representative dataset of Windows PE32 binaries. We extract API call traces and convert them into behavior graphs, where reduction strategies are applied during the graph construction phase. We use the resulting graphs to train and evaluate a GNN classifier. To comprehensively assess each reduction method, we evaluate not only classification accuracy on both IID (Independent and Identically Distributed) and OOD (Out-of-Distribution) datasets, but also the associated resource costs in terms of memory usage and processing time. Additionally, we conduct a robustness analysis against a mimicry attack to understand how these reduction techniques affect the models' robustness. This methodology allows us to explore the trade-offs between graph simplification, detection effectiveness, and robustness, leading us to answer the following research questions (RQ):

- **RQ1**: How does graph reduction impact the time and resource consumption of graph extraction?
- **RQ2**: How do graph reduction strategies affect detection performance on IID and OOD datasets?
- **RQ3**: How do graph reduction strategies impact models' robustness to adversarial (mimicry) attacks?

Our results show that the graph reduction strategy that groups API calls based on their action yields the smallest graphs, while achieving the best detection performance, not only on IID and OOD samples, but also under adversarial mimicry attacks, demonstrating better generalization and robustness. We provide open-source access to our code [1] and datasets [2] to support experimental reproducibility.

Outline. The paper is organized as follows: in Section II, we introduce the necessary concepts on which this paper is based, namely, behavior graphs, GNNs, dynamic adversarial attacks, and related work on graph reduction. In Section III we define our experimental methodology, including the different graph reduction methods and mimicry attack. We present the experiments and discuss the results in Section IV, followed by a review of related work in Section V. We conclude in Section VI.

II. Background

This section presents the main concepts used in this paper: malware detection with machine learning, graph neural networks and associated notations, and adversarial attacks.

A. Malware Analysis and Detection

The most common and efficient way to analyze a binary file involves examining its static characteristics, such as file metadata, headers, imported libraries and functions, strings, opcodes, bytes, section characteristics, and other relevant details. This data can be used for rule-based detection or fed into Machine Learning (ML) algorithms such as Gradient Boosted Decision Trees [14]. More complex feature representations can be extracted from binary files and used in the same way: Control Flow Graphs (CFG), Function Call Graphs (FCG), or other decompiled code features. While feature extraction from static analysis is often efficient, it can frequently be evaded through obfuscation: malware authors can encrypt strings, modify control flow, pack the code, or insert benign code to fool detectors based on static features.

Dynamic analysis comes as a solution to the challenges posed by static analysis: it extracts information about a binary by executing it. This execution is typically performed in a virtual environment (e.g., Cuckoo [15]) and extracts information such as runtime behavior (API or system calls) or interactions with the environment (file operations, registry access, process activity, network activity). This information can be structured in various representations (e.g., feature vectors, graphs) that are then used by detection methods.

In this paper, we focus on a graph-based representation extracted through dynamic analysis, known as the behavior graph. This graph representation consists of nodes that represent API calls and edges that represent data flows or dependencies between these API calls. First introduced by Christodorescu *et al.* [16], this representation can be constructed through dynamic taint analysis [11], [17], parsing of execution traces, or symbolic analysis [7], [18], [19]. For our study, we consider a typical machine learning-based malware detection workflow as illustrated in Figure 1. First, each sample is executed in a controlled environment for a fixed duration, during which its execution trace is recorded. From these traces, behavior graphs are extracted. Once a sufficient number of labeled graphs (malware or cleanware) is assembled to form a dataset, these can be used to train ML models for malware detection. After model training, the same dynamic analysis and graph extraction steps are applied to new binaries before classification by the trained model, to infer whether they exhibit malicious behavior.

Several stages of the pipeline in Figure 1 are time and resource-intensive. At step 1, the dynamic analysis is performed for each sample. While the execution in the sandbox lasts for a short duration (e.g., two minutes as suggested by prior research [20]), creating the execution report usually takes a similar time, i.e., an additional 1 to 3 minutes. Step 2 extracts the graphs from execution traces. Long execution traces (e.g., more than 10,000 API calls) result in large graphs that consume gigabytes of memory space and take hundreds of seconds to extract. Using numerous large behavior graphs also results in significant time and memory consumption for the ML model's training phase in step 3. The high memory consumption can be mitigated by training the model in batches; however, the large amount of time required to train the model is more challenging to overcome. This can be problematic, for example, if we wish to retrain the model frequently to keep it robust against evolving threats. Finally, the consumption of time and resources is critical for the inference phase, where large volumes of files–potentially thousands per day–must be processed efficiently.

[1] https://doi.org/10.5281/zenodo.17352103

[2] https://www.kaggle.com/datasets/sigehmireglbjevf/api-call-graphs-reduction-strategies

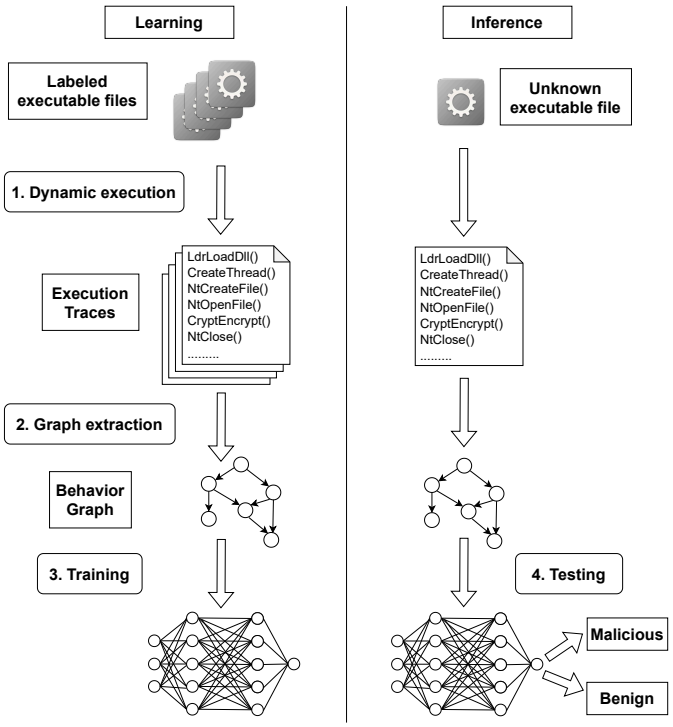


Fig. 1. Machine Learning pipeline to detect malware with behavior graphs.

This phase comprises the dynamic execution and the extraction of the graph for unlabeled executable (step 1 and 2) as well as the testing of the trained model on the graphs (step 4).

B. Learning on Graphs

Graph Neural Networks (GNNs) are machine learning models designed to operate directly on graph-structured data. Inspired by Convolutional Neural Networks (CNNs) for grid-like inputs, GNNs extend learning to non-Euclidean domains [21], making them well-suited for tasks involving irregular and interconnected structures, such as graphs and relational data. In malware analysis, GNNs are relevant to analyze graphs issued from static analysis, such as CFGs and FCGs, or from dynamic analysis, such as behavior graphs.

Notation. We define a graph $G = (V, E)$, where V is a set of nodes and $E = \{(u, v, t) | u, v \in V, t \in T\}$ is a set of directed edges from couple of nodes u and v, with T being the set of possible edge types. Each node $v \in V$ is associated with a node attribute x_v.

A Graph Isomorphism Network (GIN) is a GNN architecture [22] that has been widely use for malware detection [8], [23], [24]. This architecture achieves a high discriminative power compared to other GNNs, as it is inspired by Weisfeiler-Lehman (WL) graph isomorphism test [22]. This expressiveness makes it suitable for malware detection, where even minor differences in behavioral patterns may carry significant meaning.

For these reasons, we use a GIN variant in this study, noted GINE [25]. This version takes into account edge attributes for learning, in order to distinguish between graphs that have different edge types, such as $u \xrightarrow{1} v$ and $u \xrightarrow{2} v$.

C. Attacking Behavior-Based Malware Detection

Evaluating the robustness of malware detection models is crucial due to the efforts of malware authors to evade detection. The most common approach to evaluate this robustness is to create adversarial examples (i.e., deliberately crafted inputs designed to cause misclassification by the model). While these attacks have been extensively studied for static analysis-based malware detection [26], attacks targeting behavior-based features are more challenging to design and therefore receive less attention. Altering the semantic attributes of malware without affecting its functionality is inherently complex, especially in the absence of access to the source code.

Despite these challenges, several works have studied attacks targeting behavior-based methods. Using dynamic binary instrumentation to intercept the system calls, Banescu *et al.* [27] show that replacing sequences of system calls with functionally equivalent alternatives can degrade the performance of behavior-based malware detection approaches. Similarly, Ming *et al.* [28] apply replacement attacks directly on source code and target models based on SCDGs. Expanding on API-based attacks, Rosenberg *et al.* [29] propose GADGET, a tool using Import Address Table hooking to insert benign calls in the execution trace in a specific order and position, such that the resulting API sequence is classified as benign by the targeted model. Similarly, Fadadu *et al.* [13] present a mimicry attack that creates samples evading detection from malware classifiers based on API call sequences. Finally, Tsingenopoulos *et al.* introduce AutoRobust [30], a framework to attack models based on sandbox execution summaries. They apply functionality-preserving transformations in the feature space and provide a proof of concept demonstrating how these modifications can be reflected in a real binary.

III. Methodology

We are now ready to describe our methodology for systematically studying the impact of behavior graph reduction strategies. We target the standard class of 32-bit Windows (PE32) malware. We define several strategies with varying degrees of graph reduction, and point out related work that adopted these strategies or close variants in an ad-hoc manner.

We execute PE32 samples with the Cuckoo sandbox and extract execution traces (§III-A). We detail the process of creating a full, original behavior graph (§III-B). We explain and formalize different reduction methods (§III-C), and detail how to generate the associated reduced edge set (§III-D). Finally, we present a generic mimicry attack, which we use to assess the robustness of the resulting ML models (§III-E).

A. Execution Trace Extraction

We use Cuckoo Sandbox [15] to execute both benign and malicious samples in a controlled virtual environment and extract execution traces. For the setup, we use a host machine

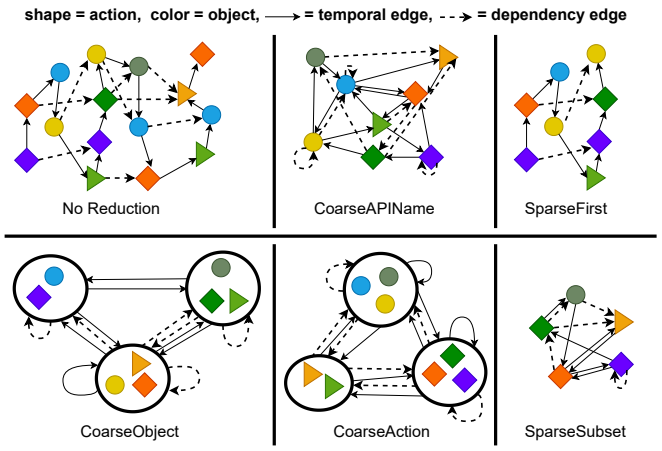


Fig. 2. Illustration of different reduction strategies, from the original behavior graph labeled "No Reduction".

running Ubuntu 18.04.6 LTS and a Windows 7 64-bit guest system. Each sample is executed in the guest environment for a duration of two minutes [31] while Cuckoo monitors and records its behavior. Execution traces are extracted from Cuckoo's behavioral analysis reports. We use version 2.0.7 rather than version 3 [32] as the latter does not provide behavioral reports at the time we conduct this work.

B. Original Behavior Graph Construction

When constructing the original graph, we assume that each node corresponds to a single API call in the execution trace. We create two types of edges, as commonly found in related work: temporal and dependency edges. Temporal edges connect two calls that appear consecutively in the execution trace. Dependency edges between two calls indicate that an argument or the return value of the first call is found in the arguments of the second call. We implement several constraints on these edges to avoid creating false dependencies. First, we do not consider arguments whose values are 0, 1, or -1 for the creation of edges based on integers. Moreover, for matching string arguments, we create a link if the two strings are identical or if they have at least 12 characters in common, with one being a substring of the other. These two constraints are inspired by the work of Christodorescu *et al.* [16]. Lastly, we create edges between two nodes only if the API calls they represent were called from the same process. We thus ensure that the graphs representing different processes are disjoint.

C. Graph Reduction Strategies

We now introduce and formally define the graph reduction strategies that form the core material of our analysis. These strategies, illustrated by Figure 2, implement different trade-offs among three critical factors: the size of the reduced graph, the computational cost of processing it, and the classification performance of the resulting model. Our definitions are based on the graph formalism introduced in Section II-B.

Grouping calls by function name. Grouping several API calls into one node based on their name is the most straightforward way to reduce the size of behavior graphs [8], [9], [33]. Taking inspiration from the work of Cui *et al.* [33], we group API calls into the same node if they have the same API function name and originate from the same process. Given the nodes $v \in V$ representing API calls with attributes x_v (the function name and the PID), we define a reduction operation S_{func} that merges nodes with the same function name and same PID. Let $P : V \to P(V)$ be a partition function that groups nodes v_i, v_j if $x_i = x_j$. Replace each group $C \in P(V)$ with a super-node v_C whose attributes x_C are the inherited function name and PID. We refer to this technique as **CoarseAPI**.

Grouping calls by category. This strategy groups API calls from the same category to the same node. It is derived from the work of Nikolopoulos *et al.* [11]. We create two types of graphs based on two different category mappings as proposed by Li *et al.* [34], respectively, a mapping from API calls to actions and a mapping from API calls to manipulated objects. For example, API function `CreateThread` maps to action category `Create` and object category `Thread`. A detailed description of our mappings is available in our reproducible experimental artifact [35]. Formally, we define a category mapping function $M : \mathit{FunctionName} \to \mathit{Category}$. For each node $v \in V$, we determine its category $c_v = M(x_v)$. We partition nodes into groups C_i where $C_i = \{v \in V | M(x_v) = c_i\}$. We replace each group C_i with a super-node v_{C_i} labeled with c_i. The strategy based on action mapping is referred to as **CoarseAction**, while the object-based one is **CoarseObject**

Keeping only the first *k* calls. This strategy reduces the size of a graph by considering only the first k API calls and ignoring consecutive duplicates (i.e., repeated back-to-back calls to the same API). This method is adapted from that of Li *et al.* [34]. We first define a total order on the nodes V based on the chronological order of the execution trace. The set of nodes that is considered corresponds to the first $k = 1000$ API calls in the ordering excluding consecutive repetitions of the same call: $V_k = \{v_1, v_2, \ldots, v_k\}$ where $x_{v_i} \neq x_{v_{i-1}}$ for $i \in \{2, k\}$. We refer to this technique as **SparseFirst**.

Keeping only a subset of API calls. This strategy, inspired by the work of Kolbitsch *et al.* [36], reduces a graph based on a subset of API functions that are considered instrumental to malicious behavior. We use the same list of suspicious API calls as Cui *et al.* [12], which is derived from MalAPI [37]. The complete list is available in our reproducible experimental artifact [35]. Formally, let $S \subseteq \mathit{FunctionNames}$ be the set of suspicious API functions. The new set of nodes is created by filtering the nodes: $V_R = \{v \in V | x_v \in S\}$. We refer to this technique as **SparseSubset**. As this alone does not significantly reduce the graph size, we also aggregate calls if they share the same API name and PID, as in the **CoarseAPI** strategy.

D. Edge Construction after Reduction of Nodes

We detailed how nodes were merged or pruned from the original graph by various strategies. We now explain how the

set of edges for the resulting reduced graph is constructed to preserve as much semantic information as possible. To convert edges from the original graph to those of the reduced graph, we define the original graph as $G = (V, E)$, where

$$E = \{(u, v, t) \mid u, v \in V,\ t \in \{0, 1\}\}$$

with t denoting the edge type (0 for temporal, 1 for dependency). Let $g : V \to V'$ be a mapping function associating each original node $v \in V$ with a reduced node $g(v) \in V'$, noting that g may be undefined for some nodes in V.

The reduced edge set is then given by:

$$E' = \{(g(u), g(v), t) \mid (u, v, t) \in E,\ g(u) \text{ and } g(v) \text{ defined}\}$$

This transformation may produce self-loops when $g(u) = g(v)$.

Specific care is required when reconstructing edges for the graph reduced with the **SparseSubset** strategy. Temporal edges must be reconstructed to preserve the temporal flow of execution, despite the removal of non-suspicious nodes (API calls). Suppose two suspicious API calls were connected in the original trace through a contiguous sequence of removed (non-suspicious) API calls. In that case, we add a temporal edge directly between these two calls in the reduced graph. This way, the simplified graph retains the temporal relationships implied by the original trace, even though intermediate nodes were removed. Formally, the set of temporal edges for this graph reduction method becomes $E'_T = \{(v_i, v_j) | x_{v_i}, x_{v_j} \in S, i < j, \forall k \in]i, j[, x_{v_k} \notin S\}$ where S contains the name of all the API functions that are considered as interesting and thus kept for the creation of the graph.

E. Robustness against Adversarial Attack

We evaluate the robustness of malware detection models against adversarial attacks designed to induce misclassification. Our approach uses the *mimicry* attack.

Threat Model. We focus on evasion attacks targeting the misclassification of malware as benign. The attacker can only access the output of our models (i.e., we consider an opaque-box attack scenario). Moreover, the attack is executed within the feature space, i.e., directly on the behavior graphs.

Graph Alteration. Our approach consists of combining the behavior graph of a clean software with that of a malware, as illustrated in Figure 3. The behavior graphs of the clean software and that of the malware are disjoint as they originate from distinct processes. We deliberately avoid establishing connections between these processes so that the attack can be more easily applied in the problem space and thus be more realistic. This alteration in the feature space can be translated to the problem space by adding an API call to the malware's code to execute a cleanware process on the host (e.g., with *ShellExecute* or *CreateProcessA*). This modification can be easily implemented in the source code. Still, it is much harder to apply directly in an executable binary, which is why we operate in the feature space rather than the problem space.

We generate adversarial samples by extending malware behavior graphs from the test set with cleanware graphs—also

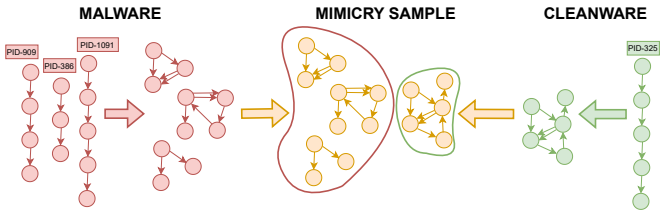


Fig. 3. Creation of a mimicry sample.

from the test set—using only samples that were correctly classified by the model trained on the original graphs.

Evaluating Attack Success. The adversarial samples (labeled as malicious) are tested against the models trained with each reduction method to measure the Attack Success Rate (ASR).

$$\text{ASR} = \frac{\#\text{misclassified adversarial samples}}{\#\text{adversarial samples}} \quad (1)$$

Moreover, we compute the Clean Node ratio (CNR), which represents the proportion of nodes in the graph of the adversarial sample that originate from the *injected* cleanware.

$$\text{CNR} = 1 - \frac{\#\text{nodes in the original malware graph}}{\#\text{nodes in the adversarial malware graph}} \quad (2)$$

We examine the mean of the CNR across all adversarial samples to determine if there is a correlation between this value and the ASR.

IV. Experiments

We are now ready to experimentally evaluate the strengths and weaknesses of each reduction strategy and answer (**A**) to the three research questions (**RQ**) identified in our introduction. For **RQ1**, we evaluate the time required to create and reduce graphs, as well as their size (number of nodes and edges). These graphs are then used to train ML malware-detection models. For **RQ2**, we determine which reduced graphs yield the best performance in terms of balanced accuracy, F1-score, recall, and precision, when tested on test data similarly distributed to the training data (IID) or on out-of-distribution (OOD) data. Finally, for **RQ3**, we study the impact of the mimicry attack presented in Section III-E, to determine the models that perform best in an adversarial setting and the corresponding original reduction strategies.

A. Datasets

We use two sources of malware and two sources of cleanware to form two separate datasets. Our primary dataset consists of malware from the BODMAS repository [38] and cleanware from the Chocolatey package manager [39]. From this dataset, we use 5,000 samples from each class for training the models. The rest of the samples (7,519 malware and 1,644 cleanware) are combined to form the TestIID testing set (Independent and Identically Distributed). A second dataset, TestOOD, is used in full as a testing set to assess the robustness of our model against out-of-distribution data. It contains 816 malware samples from the MalwareBazaar

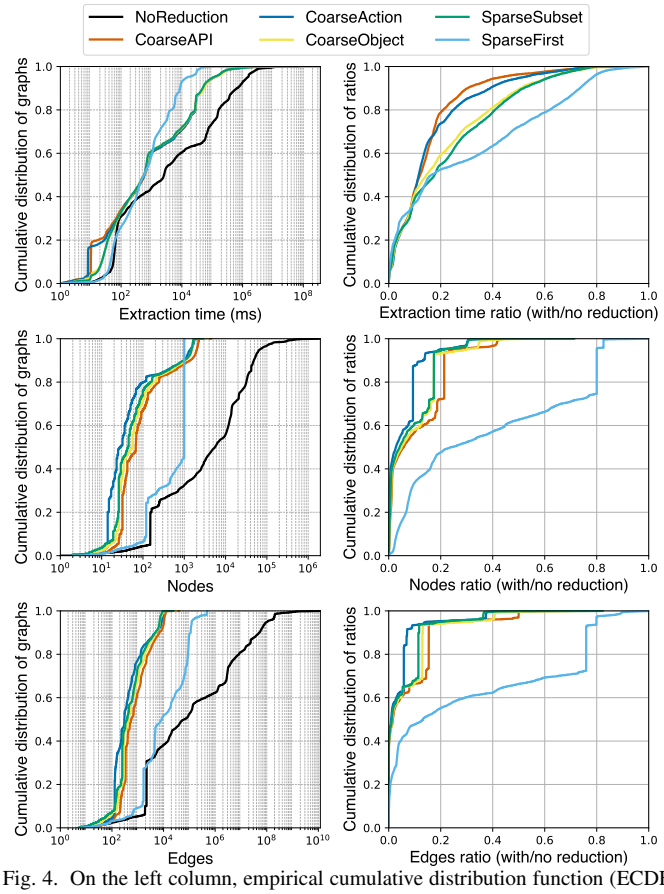


Fig. 4. On the left column, empirical cumulative distribution function (ECDF) for the extraction time, number of nodes, and number of edges generated by the five reduction strategies. On the right column, ECDF of the ratios of these values relative to their corresponding sample when using no reduction.

database [40] and 857 cleanware samples from the Portable Freeware dataset [41].

B. Resource Consumption of Graph Extraction

Figure 4 visually presents the time and space consumption of the graph extraction process for each reduction strategy, compared to an extraction process without any reduction. Space consumption, in Bytes, can be estimated as $4 + 12 \times \#nodes + 4 + 8 \times \#edges$.

The top subplots present the empirical cumulative distribution function (ECDF) of the extraction times (on the left) and the ECDF of the relative reduction in extraction time with and without graph reduction (on the right). It shows that **SparseFirst** is the most time-efficient strategy, extracting nearly all graphs in under 10 seconds. The extraction with other reduction strategies can take tens or even hundreds of seconds, due to the complexity of extracting graphs from long execution traces. We also imposed a timeout for overly long extractions due to the high computational cost of generating unreduced graphs. Among the samples that completed within the time limit, 30.2% took over one minute, and 9.5% exceeded ten minutes, underscoring the impracticality of graph creation without any reduction strategy. On the right, we see that, no matter the reduction method used, we always extract the graph faster with graph reduction than without it (ratio is always $<$ 1). **SparseFirst** appears to offer less gain in extraction time compared to the other reduction methods, but this is due to the absence of data for the longer extractions without reduction, where **SparseFirst** can provide the most significant reduction.

The middle subplots show the ECDF of the number of nodes (on the left) and of the relative reduction in node count (on the right). When using graph reduction, most graphs contain fewer than 1,000 nodes, and only a small portion exceeds 5,000. Without graph reduction, over half of the graphs have more than 5,000 nodes, with some reaching hundreds of thousands. On the right, we see that the node count reduction is especially pronounced for all reduction methods except **SparseFirst**, for which we find lower gains because of the smaller execution traces ($<$ 1,000 API calls). Still, **SparseFirst** offers a decent reduction in node count for larger traces, which constitute a significant portion of the dataset. Other methods typically reduce the node count by a factor of 5 in over 90% of cases.

The bottom subplots focus on the number of edges (on the left) and the relative reduction in edge count (on the right). With graph reduction, most graphs contain fewer than 10,000 edges, and in all cases, fewer than 50,000. **SparseFirst** tends to produce larger graphs than other strategies: about half of the graphs have more than 10,000 edges, and roughly 20% exceed 100,000 edges. In contrast, graphs extracted without graph reduction are significantly larger; half of the dataset has more than 100,000 edges, and 20% of it exceeds 10 million edges, making them impractical for many GNN algorithms. As for node count, all graph reduction methods substantially reduce the edge count, especially for larger original graphs.

A1. All strategies significantly lower the graph size and extraction time. **SparseFirst** offers the fastest graph construction, while **CoarseAction** achieves the highest compression, reducing graph size by a factor of 10.

C. Evaluation of the Machine Learning Models

We now evaluate the performance of the GINE models trained on the datasets resulting from the application of the different graph reduction strategies.

Hyperparameter Search. We use Optuna [42], an open-source hyperparatemer optimization framework, to select the hyperameters for the models. We performed a grid search over an arbitrary search space to tune the number of layers, batch size, learning rate and hidden dimensions of the model to optimize its performance, using 3-fold cross-validation.

Evaluation. Detection results for the TestIID and TestOOD test sets are presented in Table I.

For TestIID, all reduction strategies exhibit high detection performance. Yet, even if results are close, we observe that **CoarseAction** yields the highest balanced accuracy and F1-score, and the lowest deviations.

In contrast, for TestOOD, we observe a significant performance drop with all strategies. Yet, **CoarseAction** again

TABLE I
EVALUATION SCORES ON IID AND OOD SAMPLES. MEAN$^{\pm std}$ OVER 10 RUNS ON THE TEST SET.

Dataset	Reduction Method	Balanced Accuracy (%)	F1-score (%)	Recall (%)	Precision (%)
TestIID	**CoarseAPI**	$97.08^{\pm 1.06}$	$97.49^{\pm 1.20}$	$95.37^{\pm 2.32}$	$99.72^{\pm 0.06}$
	CoarseAction	$\mathbf{98.41}^{\pm 0.42}$	$\mathbf{98.91}^{\pm 0.52}$	$\mathbf{98.14}^{\pm 1.08}$	$99.71^{\pm 0.10}$
	CoarseObject	$95.10^{\pm 4.85}$	$94.94^{\pm 5.79}$	$91.01^{\pm 9.86}$	$\mathbf{99.81}^{\pm 0.06}$
	SparseSubset	$97.94^{\pm 0.63}$	$98.51^{\pm 0.70}$	$97.40^{\pm 1.41}$	$99.66^{\pm 0.15}$
	SparseFirst	$97.65^{\pm 2.41}$	$98.12^{\pm 2.83}$	$96.74^{\pm 5.21}$	$99.68^{\pm 0.21}$
TestOOD	**CoarseAPI**	$56.29^{\pm 2.85}$	$24.12^{\pm 9.54}$	$14.30^{\pm 6.26}$	$88.92^{\pm 5.11}$
	CoarseAction	$\mathbf{71.69}^{\pm 2.89}$	$\mathbf{60.91}^{\pm 5.90}$	$\mathbf{44.63}^{\pm 6.24}$	$\mathbf{97.23}^{\pm 1.38}$
	CoarseObject	$56.46^{\pm 4.59}$	$22.74^{\pm 13.55}$	$13.59^{\pm 9.48}$	$94.74^{\pm 3.70}$
	SparseSubset	$67.63^{\pm 3.63}$	$53.21^{\pm 7.95}$	$37.44^{\pm 7.27}$	$94.10^{\pm 2.93}$
	SparseFirst	$54.80^{\pm 4.60}$	$23.50^{\pm 14.93}$	$15.26^{\pm 12.08}$	$69.79^{\pm 11.27}$

TABLE II
ATTACK SUCCESS RATE OF THE MIMICRY ATTACK AND MEAN$^{\pm std}$ OF THE CLEAN NODE RATIO FOR THE ADVERSARIAL GRAPHS

Reduction method	ASR (%)	CNR-d (%)	CNR-e (%)
CoarseAPI	29.3	$17.4^{\pm 15.2}$	$55.9^{\pm 17.5}$
CoarseAction	6.8	$22.7^{\pm 18.5}$	$67.1^{\pm 17.1}$
CoarseObject	54.2	$11.4^{\pm 12.8}$	$43.3^{\pm 20.3}$
SparseFirst	10.5	$5.6^{\pm 13.5}$	$57.7^{\pm 21.9}$
SparseSubset	22.6	$20.5^{\pm 18.1}$	$59.4^{\pm 15.6}$

outperforms other methods by achieving the highest scores and the lowest deviations. This suggests that grouping API calls by action improves generalization and stability in capturing behavioral patterns. However, reductions like **CoarseObject** and **SparseFirst** perform poorly in this out-of-distribution setting, particularly in terms of recall, suggesting that the resulting models may oversimplify or discard critical behavior patterns. Rather than a strict trade-off, the results point to differences in generalization: while most simplification strategies preserve performance in familiar settings, **CoarseAction** should retain reasonable accuracy when faced with evolving malware.

A2. Using **CoarseAction**, we consistently achieve the best detection performance across both IID and OOD settings, indicating a stronger ability to preserve relevant behavioral patterns after graph reduction. While differences between reduction strategies are minor in-distribution, they become more pronounced out-of-distribution, where **CoarseAction** demonstrates superior generalization.

D. Robustness against Adversarial Mimicry Attack

We finally discuss the robustness of the models resulting from the different reduction strategies to the mimicry adversarial attack. We compare the Attack Success Rate (ASR) to the Clean Node ratio (CNR). We report the mean CNR separately for adversarial examples that were correctly detected as malware (CNR-d) and for those that successfully evaded detection (CNR-e). This provides valuable insight into how much benign behavior is needed to mislead the model.

Table II presents the impact of the mimicry attack across models built after applying different graph reduction strategies. The Attack Success Rate (ASR) is significantly lower for **CoarseAction** (6.8%) compared to all other strategies, indicating greater robustness.

The CNR is higher among successful attacks than among failed ones (CNR-d<CNR-e), confirming that the inclusion of clean behavior helps leading to evasion. Interestingly, **CoarseAction** exhibits the highest CNR among failed attacks (22.7%), suggesting it captures behavioral context more effectively and is more resilient to the addition of misleading calls.

Despite having the lowest CNR in successful attack (43.3%), **CoarseObject** is the most vulnerable (ASR of 54.2%), showing that even a CNR $< 50\%$ can be enough to evade the model.

A3. Among all reduction strategies, **CoarseAction** offers the strongest robustness to mimicry attacks, with the lowest evasion rate. Differences in attack success are more pronounced than in standard detection, revealing the impact of reduction strategies on robustness.

V. RELATED WORK

Several works have proposed feature or graph reduction techniques to enhance the scalability and efficiency of malware detection pipelines. In static analysis, dimensionality reduction is often applied at the feature level to improve detection performance. Common approaches include feature selection, embedding (e.g., API2Vec [33]) or hashing tricks as used in the EMBER dataset [43], which maps a high-dimensional sparse feature set (e.g., DLL library names, section names) into a fixed-length representation more suitable for machine learning. Beyond malware detection, graph reduction has also been explored in vulnerability detection [44], [45], where compact graph representations improve both model efficiency and accuracy. Recent studies also suggest that the robustness of GNNs under graph reduction varies with the nature of the adversarial threat [46], including in scenarios involving backdoor attacks [47]. In comparison to prior works, we provide a systematic evaluation of multiple graph reduction strategies for behavior graphs, assessing their impact on detection performance, resource consumption, and robustness against adversarial mimicry attacks.

VI. CONCLUSION AND FUTURE WORK

This paper presented the first systematic evaluation of graph reduction strategies for dynamic behavior-based malware detection using Graph Neural Networks. By analyzing five reduction strategies based on sparsification or coarsening, we demonstrated that reduction can significantly lower graph size and extraction cost, with limited impact on detection performance. Among these strategies, **CoarseAction**–which groups API calls by semantic action–consistently achieved the best trade-off between compression, detection accuracy, and generalization. It also exhibited the strongest robustness against mimicry attacks, suggesting that semantic grouping helps preserve behavioral structure even under adversarial conditions. In our future work,

we intend to explore additional graph reduction strategies, such as graph condensation or the k-core decomposition. We also plan to investigate other adversarial attacks such as API call injection or replacement, and eventually transfer these attacks in the problem space.

ACKNOWLEDGMENTS

This research is supported by the Walloon region's CyberExcellence program (grant #2110186).

REFERENCES

[1] ENISA. (2024) Enisa threat landscape report. [Online]. Available: https://www.enisa.europa.eu/publications/enisa-threat-landscape-2024

[2] VirusTotal. (2019) Yara: The pattern matching swiss knife. [Online]. Available: http://virustotal.github.io/yara/

[3] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection." IEEE, 2007.

[4] N. Z. Gorment, A. Selamat, L. K. Cheng, and O. Krejcar, "Machine learning algorithm for malware detection: Taxonomy, current challenges, and future directions," *IEEE Access*, vol. 11, 2023.

[5] M. R. Smith, N. T. Johnson, J. B. Ingram, A. J. Carbajal, B. I. Haus, E. Domschot, R. Ramyaa, C. C. Lamb, S. J. Verzi, and W. P. Kegelmeyer, "Mind the gap: On bridging the semantic gap between machine learning and malware analysis," ser. AISec, 2020.

[6] S. D. Nikolopoulos and I. Polenakis, "Behavior-based detection and classification of malicious software utilizing structural characteristics of group sequence graphs," *Journal of Computer Virology and Hacking Techniques*, 2022.

[7] C.-H. Bertrand Van Ouytsel and A. Legay, "Malware analysis with symbolic execution and graph kernel," ser. NordSec, 2022.

[8] C. Li, Z. Cheng, H. Zhu, L. Wang, Q. Lv, Y. Wang, N. Li, and D. Sun, "DMalNet: Dynamic malware analysis based on API feature engineering and graph learning," *Computers & Security*, vol. 122, 2022.

[9] P. Feng, L. Gai, L. Yang, Q. Wang, T. Li, N. Xi, and J. Ma, "DawnGNN: Documentation augmented windows malware detection using graph neural network," *Computers & Security*, 2024.

[10] M. Hashemi, S. Gong, J. Ni, W. Fan, B. A. Prakash, and W. Jin, "A comprehensive survey on graph reduction: sparsification, coarsening, and condensation," ser. IJCAI, 2024.

[11] S. D. Nikolopoulos and I. Polenakis, "A graph-based model for malware detection and classification using system-call groups," *Journal of Computer Virology and Hacking Techniques*, 2017.

[12] L. Cui, J. Yin, J. Cui, Y. Ji, P. Liu, Z. Hao, and X. Yun, "Api2Vec++: Boosting API sequence representation for malware detection and classification," *IEEE Transactions on Software Engineering*, 2024.

[13] F. Fadadu, A. Handa, N. Kumar, and S. K. Shukla, "Evading api call sequence based malware classifiers," ser. ICICS, 2020.

[14] D. Ucci, L. Aniello, and R. Baldoni, "Survey of machine learning techniques for malware analysis," *Computers & Security*, vol. 81, 2019.

[15] Cuckoo, "Cuckoo sandbox," https://cuckoo.cert.ee/, 2024.

[16] M. Christodorescu, S. Jha, and C. Kruegel, "Mining specifications of malicious behavior," ser. ESEC/FSE, 2007.

[17] Y. Ding, X. Xia, S. Chen, and Y. Li, "A malware detection method based on family behavior graph," *Computers & Security*, vol. 73, 2018.

[18] S. Sebastio, E. Baranov, F. Biondi, O. Decourbe, T. Given-Wilson, A. Legay, C. Puodzius, and J. Quilbeuf, "Optimizing symbolic execution for malware behavior classification," *Computers & Security*, vol. 93, 2020.

[19] N. B. Said, F. Biondi, V. Bontchev, O. Decourbe, T. Given-Wilson, A. Legay, and J. Quilbeuf, "Detection of mirai by syntactic and behavioral analysis," ser. ISSRE. IEEE, 2018.

[20] A. Küchler, A. Mantovani, Y. Han, L. Bilge, and D. Balzarotti, "Does every second count? Time-based evolution of malware behavior in sandboxes," ser. NDSS, 2021.

[21] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst, "Geometric deep learning: Going beyond euclidean data," *IEEE Signal Processing Magazine*, vol. 34, no. 4, Jul. 2017.

[22] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" *arXiv preprint arXiv:1810.00826*, 2018.

[23] B. Wu, Y. Xu, and F. Zou, "Malware classification by learning semantic and structural features of control flow graphs," ser. TrustCom. IEEE, 2021.

[24] V. Malhotra, K. Potika, and M. Stamp, "A comparison of graph neural networks for malware classification," *Computer Virology and Hacking Techniques*, 2024.

[25] W. Hu, B. Liu, J. Gomes, M. Zitnik, P. Liang, V. Pande, and J. Leskovec, "Strategies for pre-training graph neural networks," *arXiv preprint arXiv:1905.12265*, 2019.

[26] B. Kolosnjaji, A. Demontis, B. Biggio, D. Maiorca, G. Giacinto, C. Eckert, and F. Roli, "Adversarial malware binaries: Evading deep learning for malware detection in executables," ser. EUSIPCO. IEEE, 2018.

[27] S. Banescu, T. Wuchner, A. Salem, M. Guggenmos, M. Ochoa, and A. Pretschner, "A framework for empirical evaluation of malware detection resilience against behavior obfuscation," ser. MALWARE. IEEE, 2015.

[28] J. Ming, Z. Xin, P. Lan, D. Wu, P. Liu, and B. Mao, "Impeding behavior-based malware analysis via replacement attacks to malware specifications," *Journal of Computer Virology and Hacking Techniques*, vol. 13, 2017.

[29] I. Rosenberg, A. Shabtai, L. Rokach, and Y. Elovici, "Generic black-box end-to-end attack against state of the art API call based malware classifiers," 2018.

[30] I. Tsingenopoulos, J. Cortellazzi, B. Bošanskỳ, S. Aonzo, D. Preuveneers, W. Joosen, F. Pierazzi, and L. Cavallaro, "How to train your antivirus: RL-based hardening through the problem space," ser. RAID, 2024.

[31] H. Aghakhani, F. Gritti, F. Mecca, M. Lindorfer, S. Ortolani, D. Balzarotti, G. Vigna, and C. Kruegel, "When malware is packin'heat; limits of machine learning classifiers based on static analysis features," ser. NDSS, 2020.

[32] CERT-EE, "Cuckoo3 - malware analysis tool," https://github.com/cert-ee/cuckoo3.

[33] L. Cui, J. Cui, Y. Ji, Z. Hao, L. Li, and Z. Ding, "Api2Vec: Learning representations of API sequences for malware detection," ser. ISSTA, 2023.

[34] C. Li, Q. Lv, N. Li, Y. Wang, D. Sun, and Y. Qiao, "A novel deep framework for dynamic malware detection based on API sequence intrinsic features," *Computers & Security*, vol. 116, 2022.

[35] S. Bettaieb and S. Lucca, "Reproducibility material: Behavior graph reduction strategies for machine learning-based malware detection," https://github.com/CloudLargeScale-UCLouvain/malware-graph-reduction.

[36] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X.-y. Zhou, X. Wang *et al.*, "Effective and efficient malware detection at the end host." 2009.

[37] mr.d0x, *MalAPI.io*, accessed: 03-06-2025. [Online]. Available: https://malapi.io/

[38] L. Yang, A. Ciptadi, I. Laziuk, A. Ahmadzadeh, and G. Wang, "BODMAS: An open dataset for learning based temporal analysis of PE malware," ser. SPW, 2021.

[39] Chocolatey, *Chocolatey:The Package Manager for Windows*, accessed: 11-02-2025. [Online]. Available: https://chocolatey.org/

[40] abuse.ch, *MalwareBazaar*, accessed: 11-02-2025. [Online]. Available: https://bazaar.abuse.ch/

[41] C. Sébastien, "Portable freeware dataset." [Online]. Available: https://doi.org/10.5281/zenodo.2528209

[42] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A next-generation hyperparameter optimization framework," 2019.

[43] H. S. Anderson and P. Roth, "Ember: an open dataset for training static pe malware machine learning models," *arXiv preprint arXiv:1804.04637*, 2018.

[44] Y. Luo, W. Xu, and D. Xu, "Compact abstract graphs for detecting code vulnerability with GNN models," ser. ACSAC, 2022.

[45] X.-C. Wen, Y. Chen, C. Gao, H. Zhang, J. M. Zhang, and Q. Liao, "Vulnerability detection with graph simplification and enhanced graph representation learning," ser. ICSE, 2023.

[46] K. Wu, K.-H. Chow, W. Wei, and L. Yu, "Understanding the impact of graph reduction on adversarial robustness in graph neural networks," *arXiv preprint arXiv:2412.05883*, 2024.

[47] Y. Zhu, M. Mandulak, K. Wu, G. Slota, Y. Jeon, K.-H. Chow, and L. Yu, "On the robustness of graph reduction against GNN backdoor," ser. AiSec, 2024.