UCLouvain

École polytechnique de Louvain

From Exploration to Classification: Harnessing Symbolic Execution and Graph Neural Networks in Malware Analysis

Author: **Samy Bettaieb**
Supervisor: **Axel Legay**
Readers: **Charles-Henry Bertrand Van Ouytsel, Charles Pecheur**
Academic year 2022–2023
Master [120] in Computer Science

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my supervisor, Prof. Axel Legay for his guidance and encouragement during both this year and the previous year when he helped me identify a research subject.

I am also grateful to Charles-Henry Bertrand Van Ouytsel for his valuable assistance in helping me choosing a subject that aligns with my interests and academic goals. His insightful suggestions and advice have been invaluable in shaping the direction of my research.

Furthermore, I would like to extend my thanks to Charles-Henry Bertrand Van Ouytsel, Christophe Crochet and Serena Lucca for their continuous support throughout the year. Their dedication, responsiveness, and willingness to assist, even with the most trivial of questions, have been greatly appreciated. Whether it was through formal interactions or chance encounters in the corridors, their presence and assistance have made a significant difference in this journey.

Moreover, I would like to extend my deepest appreciation to the readers and members of the jury, Charles-Henry Bertrand Van Ouytsel and Prof. Charles Pecheur for taking the time to review this work.

On a more personal note, I am deeply grateful to my family for their unwavering support during this challenging year. Their love, encouragement, and understanding helping me overcome personal obstacles and navigate through the ups and downs of this last year. Without their sacrifices and belief in me, I would not have been able to pursue and complete these studies. I am deeply thankful for their presence in my life.

Lastly, I would like to thank my friends and the members of the "Salade de fruits" for their love and support during the last five years. The bonds of friendship forged during this time have made a great positive impact on this adventure.

Abstract

The proliferation of cybercrime and the escalating threat of malware attacks necessitate more effective and efficient analysis techniques. Traditional methods, such as static and dynamic analysis, have limitations that hinder their effectiveness against sophisticated and evasive malware. Symbolic execution, a promising approach, overcomes some of these limitations but faces scalability challenges.

This thesis explores the enhancement of SEMA, a toolchain using Symbolic Execution for Malware Analysis, with two main contribution: (1) developing novel exploration strategies to enhance symbolic execution, and (2) incorporating graph neural networks (GNNs) to improve the classification accuracy of malware samples. Additionally, this work ventures into the realm of GNN explainability, aiming to shed light on the decision-making process of GNN models.

To address the scalability issues of symbolic execution, in particular the path explosion problem, this study proposes seven new exploration strategies that aim to enhance the symbolic execution process by selecting paths based on assigned weights, aiming to improve code coverage within a limited timeframe. Additionally, graph neural networks (GNNs) are integrated as new classifiers into the toolchain, leveraging the structural properties of malware represented as graphs, in particular System Call Dependency Graphs (SCDGs), for accurate and efficient classification and detection. Specifically, a Graph Isomorphism Network (GIN) and a Graph Isomorphism Network with JumpingKnowledge (GINJK) are implemented. Furthermore, a GNN explainability module is developed to provide insights into the decision-making process of GNN models.

The experimental results demonstrate that GINJK, combined with an exploration strategy that focuses on the nature of the system calls and on exploring states leading to new assembly instruction addresses, outperforms other classifiers combined with different exploration strategies. The thesis concludes by highlighting the potential of future research directions. Overall, this work contributes to the advancement of the SEMA Toolchain, enabling more effective and accurate malware analysis.

Contents

Chapter 1

Introduction

In today's digital era, the proliferation of cybercrime and the escalating threat of malware attacks have become major concerns. Statistics reveal a significant increase in monetary damages caused by reported cybercrime. For instance, between 2001 and 2022, the annual loss of complaints referred to the Internet Crime Complaint Center (IC3) increased from 6.9 billion U.S. dollars to a staggering 10.3 billion U.S. dollars [40].

Malware attacks cause extended recovery periods, averaging 50 days, and significant disruptions. Small businesses, comprising 50% of incidents, are particularly vulnerable with 73% lacking proper cybersecurity measures.[9].

Fortunately, the financial repercussions and potential damages caused by cybercrimes have spurred extensive research and interest in the field of cybersecurity. Traditional malware analysis techniques, such as static analysis and dynamic analysis, have played a crucial role in understanding and mitigating malware threats. However, these methods have inherent limitations that hinder their effectiveness in combating sophisticated and evasive malware.
Symbolic execution has emerged as a promising approach that overcomes some of these limitations by systematically exploring all possible program paths. Nonetheless, symbolic execution is not without challenges, as it suffers from path explosion problems, limiting its scalability and applicability in real-world scenarios.

Machine learning has revolutionized various domains, and its potential in malware analysis is increasingly recognized. By leveraging machine learning algorithms, researchers have made significant strides in classifying and detecting malware. Graph neural networks (GNNs) have received particular attention in the field of machine learning, enabling the analysis of graph-structured data. The explainability of machine learning models has also gained relevance, providing a better understanding of the decision-making process and increasing trust in the results.

The SEMA toolchain [22] [23], a powerful framework for malware analysis, combines the strengths of symbolic execution and machine learning. It performs symbolic execution directly on binaries to generate System Call Dependency Graphs (SCDGs). Then it applies machine learning techniques on this graph structured data in order to detect if a binary is a malware or cleanware, or to classify malware in families.

This thesis focuses on extending the capabilities of the SEMA toolchain by developing novel exploration strategies and incorporating a graph neural network to enhance the symbolic execution process and improve the classification accuracy of malware samples. At the end of this work, we show that the chosen exploration strategy influences greatly the machine learning classifier, meaning that it must be chosen with care. We also reveal that we can improve the performance of malware classification with graph neural networks combined to symbolic execution, if used the right exploration strategy.

Contribution In this study, we make two key contributions to the SEMA Toolchain. First, we enhance the exploration strategies of the symbolic executor to face the path explosion problem and improve code coverage within a limited timeframe. We implement 7 new exploration strategies. These strategies choose the path to follow based on assigned weights. Second, we integrate graph neural networks (GNNs) as new classifiers into the toolchain. By leveraging the structural properties of malware represented as graphs, these classifiers can enable an accurate and efficient malware classification and detection. We implement two new classifiers, a Graph Isomorphism Network (GIN) and a Graph Isomorphism Network with JumpingKnowledge (GINJK) . Still in the context of graph neural networks, we implement a first version for a GNN explainability module that aims to help understanding the reasons behind the decision of a GNN model.

We show that GINJK combined with a new exploration strategy that focuses on interesting system calls and exploring new addresses, yields to the best performance among the new and already existing classifiers, with the new and already existing exploration strategies.

Organization The remainder of this thesis is organized as follows. In Chapter 2, we provide background information and a review of literature. We will discuss topics such as the different malware analysis techniques and introduce symbolic execution in Section 2.1. A survey of existing techniques to optimize state management during symbolic execution in Section 2.2. We will present graph related concepts and introduce graph neural networks in Section 2.3. We will finish this chapter by giving an overview of the SEMA Toolchain in Section 2.4. In Chapter 3, we

describe our implementations and design choices. We start by the search strategies in Section 3.1 and follow with the graph neural networks in Section 3.2. In chapter 4, we present our experimental results. In chapter 5 we discuss the future research directions and possible contributions to the SEMA Toolchain. In Chapter 6, we summarize our contribution and conclude our thesis.

Chapter 2

Background knowledge and literature review

This chapter aims to provide a comprehensive review of the state of the art and the relevant background knowledge for the subsequent chapters.

We will start by describing the different malware analysis techniques and the challenges they face in Section 2.1 along with the framework we use to analyse the binaries: angr [39]. Then, we will focus on symbolic execution, and the different path scheduling approaches to face the path explosion problem, in Section 2.2. We will follow by outlining some motivations and limitations behind machine learning for malware classification in Section 2.3. We will list common graph-related notations and definitions, and focus on Graph Neural Networks, in Section 2.3.2. We will conclude by presenting the SEMA toolchain (Symbolic Execution for Malware Analysis) [22][23], which is used in our case to implement some of the techniques mentioned above, in Section 2.4.

2.1 Malware analysis techniques

Two of the most common techniques for analyzing binaries are static analysis and dynamic analysis. In this section, we will introduce both of them along with symbolic execution.

Static analysis Static analysis is a technique used to analyze software source code or compiled binary without executing it. It helps detect bugs, security vulnerabilities, and other issues early in the development process. Static analysis tools analyze code at the source level, allowing them to identify potential issues that may be missed in manual code review or testing.

In the context of malware analysis, static analysis aims to identify malicious or suspicious behaviors in a program. This includes identifying malicious functions, detecting strings indicating malicious behavior (e.g., command and control server addresses), and detecting encryption or obfuscation techniques commonly used by malware. Static analysis tools, like Yara [45], can detect malware by directly matching patterns in binary files. However, this approach is susceptible to evasion techniques, as creating a variant of the malware can easily bypass such detection. For example, if we have a malware that prints `"I'm evil"` and we try to detect this by matching this pattern, we could easily bypass it by splitting the string and concatenate it before printing. Other tools such as Ghidra [30] or IDA allow decompiling the code, which consists in transforming the binary code into a high-level language easily understandable by humans.
Unfortunately, we can easily modify the syntactic signature of a malware without affecting the behavior of the malware using binary transformation or obfuscation techniques [60] [29]. These techniques include but are not limited to: Code obfuscation, Packing, Polymorphism, and Metamorphism.

Dynamic analysis In contrast to static analysis, dynamic analysis is a technique used to analyze software behavior by running the program in a controlled environment such as a sandbox or a virtual machine. In the context of malware analysis, the goal is to observe the behavior of a program during its execution to identify malicious or suspicious actions, such as network communication with command and control servers, file system modifications, or attempts to evade detection.
Dynamic analysis offers the advantage of capturing the actual program behavior, despite the presence of encryption or obfuscation techniques. For example, by monitoring network traffic, we can identify IP addresses or domains contacted by the malware, aiding in tracking and blocking it. Behavioral properties and signatures can be created based on these interactions.
Nevertheless, this type of analysis has limitations. Some malware may use anti-analysis techniques, such as detecting the presence of a virtual machine or sandbox and adapting their behavior accordingly. Since dynamic analysis will follow only one execution path, the analyst will not be able to see malicious behavior if the malware adapts and acts harmlessly. Furthermore, this analysis requires the resources to monitor and emulate the environment for running malware, since we need to replicate the real world as much as possible while preventing its spread. While dynamic analysis provides valuable insights, it should be used alongside other techniques such as static analysis and threat intelligence for a comprehensive understanding of malware behavior.

Symbolic execution Symbolic execution is a technique that uses symbolic values to explore all possible execution paths. In contrast to dynamic analysis, which observes only one execution using concrete values, symbolic execution will explore all execution paths until a certain target is reached (e.g. the end of the program). *Inputs are represented as symbolic values, and an execution path is represented as a logical formula (i.e., first-order logic constraints over symbolic values) called path predicate* [12]. To know if an execution is possible, we need to check that the path predicate is *satisfiable*. Symbolic execution rely on a SAT or a SMT (satisfiability modulo theories) solver to find concrete values that satisfy the path predicate.

Symbolic analysis allows understanding the program's behaviors faster since we easily (in theory) find the input that triggers a particular path. It can also guarantee some properties of the target program, for example the unsatisfiablity of a certain path or the unreachability of a particular state. Moreover, since it constructs the concrete input to reach a target state, it can overcome the limitations of black box fuzzing, where we try different inputs and wait to see what happens [16]. Since symbolic analysis reasons on how the input triggers certain program behaviors, it offers a better semantic insight than static or dynamic analysis which are based on replayability.

However, in practical applications, the scalability of symbolic execution is limited by a few factors. One prominent limitation is the computational burden of constraint solving when dealing with complex predicates. This process entails considerable resources and hinders the effectiveness and practicality of symbolic execution. In addition, another important challenge encountered is the problem of state and path explosions. This phenomenon refers to the exponential growth in the potential program state and execution path states.

To analyze various possible paths, a symbolic executor maintains a worklist that consists of both satisfiable and unchecked paths. Unchecked paths have been explored but are not yet checked for satisfiability, so they can be satisfiable or unsatisfiable. The symbolic executor selects a path from the worklist based on a chosen exploration strategy, for example breadth-first search (BFS) or depth-first search (DFS), and then extends it by one basic block to generate its successor(s). In the course of exploration, when the symbolic executor encounters a condition or a path that depends on a symbolic value, it branches it into two paths: one in which the condition is satisfied, and another in which the condition is violated (i.e., the negated condition is satisfied). Since the number of forks grows exponentially and the symbolic execution keeps track of all possible execution paths, it requires important computing resources. This path explosion problem becomes even worse if we analyze shared libraries [12]. To mitigate this path explosion problem and maximize code coverage, we can use some domain-specific optimizations or path exploration heuristics to guide the symbolic explorer during the search. Another

approach is to study path pruning techniques in order to reduce the search space and keep interesting execution paths for a deeper exploration.
We will review various strategies to mitigate this problem in Section 2.2.

The first part of our work during this master thesis is to implement new exploration strategies to counter the path explosion problem.

angr framework To perform symbolic execution, we use the angr framework [39]. angr is an open-source and well-maintained Python framework. It has a wide range of interesting features that make it suitable for binary analysis. Particularly, it has built-in support for analyzing binaries in different formats (ELF, PE, etc.), and provides a high-level API for interacting with them.
From a technical standpoint, by default, angr uses the VEX intermediate representation to perform symbolic execution, which allows it to support a wide range of architectures and platforms. It also has support for several popular solvers, including Z3, which is a SMT solver widely used in the symbolic execution community.

2.2 Symbolic execution state management

Optimizing the symbolic execution state manager can be achieved through various approaches. One such approach is hybrid execution, which combines both concrete and symbolic inputs to reduce the storage of symbolic values and increase code coverage. Concolic execution (Dynamic Symbolic Execution) [38] [18] is an example of this approach in which the program is executed dynamically with a concrete state and a symbolic state that are maintained simultaneously. Symcretic Symbolic Execution [15] is another hybrid technique that combines backward symbolic execution (BSE) and concrete execution to target specific instructions.
Another approach to optimizing symbolic execution is program summarization. This technique involves using additional constraints or simplifications to reduce the generated states.

The first part of our thesis work falls under the category of path scheduling approaches, which can be further classified into two subcategories: path pruning and path prioritization techniques. Path pruning techniques aim to reduce the search space, while path prioritization techniques guide the exploration towards interesting paths. In Section 2.2.1 and Section 2.2.2, we will provide a review of the existing approaches in the literature for both these subcategories respectively. This review might be used as a reference for future work.

2.2.1 Path pruning techniques

Many notable efforts have been made to face the state and path explosion problem. Path pruning techniques seek to reduce the size of the search space by detecting and eliminating (early enough) uninteresting paths.

Pre-conditioned symbolic execution used in Automatic Exploit Generation (AEG) [3], is an effective technique. It involves passing a predefined constraint Π_{prec} along with the program being tested. This constraint allows the execution to only explore program branches that meet the Π_{prec} condition. As a result, the search space is effectively reduced, as steps for unsatisfied branches are pruned away.

Under-Constrained symbolic execution [34] on the other hand, addresses this state explosion problem by starting the analysis and checking directly individual functions instead of the whole program. This reduces the length and number of the explored paths, resulting in cost savings by eliminating the need to traverse the path from the `main` function to the analyzed functions. Moreover, libraries and OS kernel codes without a main function can be checked easily and carefully with this method. Under-Constrained symbolic execution is implemented in UC-KLEE, a scalable framework for checking C/C++ systems code.

Post-Conditioned symbolic execution [58] [57] is another way to manage the search space. This method aims to eliminate redundant path suffixes encountered during symbolic execution. Each program location l is associated with a postcondition that summarizes the path suffixes that were explored from that location l. As new path suffixes are discovered during the testing process, they are added to the postcondition as a quantifier-free first-order logic constraint. During the generation of new test inputs, the method checks if the current path condition is included in the postcondition. If yes, the execution of the remaining path is skipped.

In contrast to preconditioned symbolic execution, postconditioned symbolic execution calculates constraints dynamically during execution instead of predefining them beforehand. As explained by Qiuping Yi et al. [58], the goal of preconditioned symbolic execution is to eliminate paths that do not satisty the predefined conditions. If this precondition is `false` then no path is explored. It shows that the predefined constraint needs to be chosen with care; otherwise, it may not be effective. Since postconditioned symbolic execution computes postconditions dynamically during the execution, it results in path coverage equivalent to standard symbolic execution. The computed postconditions eliminate only the redundant paths, hence increasing the efficiency of the search. Qiuping Yi et al. [58] have implemented their solution

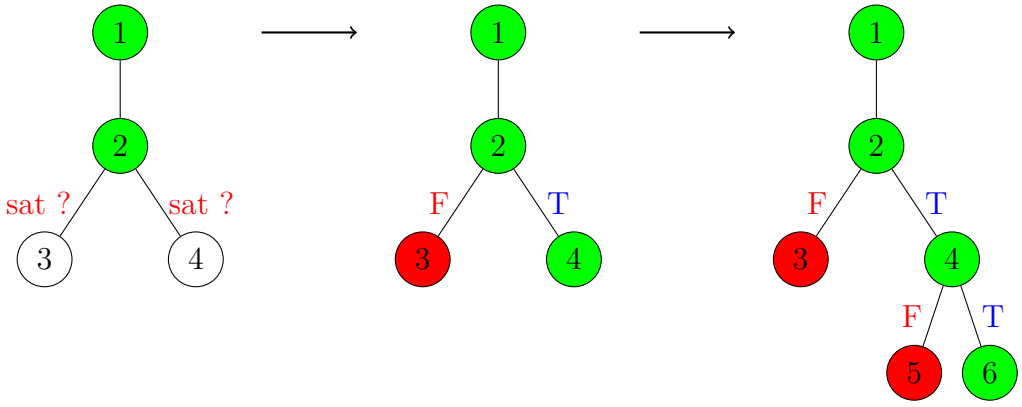

Figure 2.1: *CheckAll* strategy

based on the KLEE symbolic virtual machine [10], which is another framework that performs symbolic execution.

Dynamic path pruning [12] is another state of the art method. In this case, the goal is to minimize the computation time by optimizing the number of checks for the satisfiability of paths, since it is one of the consuming tasks in the symbolic execution computation. As explained in Section 2.1, during symbolic execution, the executor maintains a worklist of satisfiable and unchecked paths. When it extends a path from the worklist to generate its successor(s) and encounters a new path constraint, it has the choice to either check the satisfiability of the path or skip the check and continue the exploration. As a natural consequence of the path explosion problem, we need an important amount of memory to maintain the worklist. One way to reduce the memory overhead is to check the satisfiability of paths immediately when encountering a new condition using the SMT solver. This is called the *CheckAll* strategy, it is illustrated by the Figure 2.1. The figures 2.1, 2.2 and 2.3 are inspired from the presentation of F. Gritti [16]. Nodes represent states. A green node means that the path to this node has been checked and is satisfiable. A red node means that the path to this node has been checked and is unsatisfiable. A white node means that the path to the node is explored, or the state is added to the worklist, but the satisfiability is not checked yet.

If this checking strategy reduces the memory overhead, it doesn't scale that much since solving the predicates is very time consuming and might even result in a timeout when encountering a complex predicate. The symbolic executor can go further in exploration if it skips solving these complex predicates, as the new constructed predicate from the subsequent paths might be easier to solve.

The opposite strategy is the *CheckNothing* strategy, where the executor only solves the predicates at the end of the exploration, as illustrated in the Figure 2.2.

This strategy reduces the time used to check the satisfiability of paths but doesn't solve the memory overhead since it keeps multiple useless states in memory until the check.

Ying-Shen Chen et al. [12] explain that these two static path pruning strategies are suboptimal since they are independent from the path predicate at runtime. The number of (un)satisfiable paths varies from one program to another: the more satisfiable paths we have, the better a *CheckNothing* strategy will perform; while the *CheckAll* strategy will be more suitable if we have a lot of unsatisfiable paths. For some programs, none of them will perform well, for example a program with *a lot of unsatisfiable paths at the beginning and a lot of satisfiable paths with complex predicates at the end* [12].

For this reason, Ying-Shen Chen et al. [12] propose a solution called Dynamic path pruning (DPP). The goal of this *CheckDynamic* strategy is to find the optimal checking rate. They model the path pruning problem (PPP) as an optimization problem and derive the optimal checking rate. DPP aims to compute a probability of checking the path so that the expected total exploration time is minimized. This checking rate is computed based on the observation of previously checked paths, and this rate is dynamically adjusted per binary, per layer or per path during the execution. The work of Ying-Shen Chen et al. [12] extends the angr framework and focuses on optimization per layer, as it can be easily coupled with the BFS search strategy. The checking rate of the next layer is calculated based on the observed unsatisfiable paths at the previous layer such that the exploration time is minimized. Thus, we only check satisfiablity of a path when we deem it relevant. This is illustrated in the Figure 2.3.

2.2.2 Path prioritization techniques

In contrast to path pruning, path prioritization techniques doesn't aim to reduce the search space by eliminating uninteresting states. The goal is to guide the exploration towards the states we deem most interesting first. What we define as interesting states varies from one method to another and from one use case to another (e.g. bug hunting vs malware detection). Usually, we rely on search strategies; they might be uninformed of the nature of the problem (e.g. BFS or DFS) or informed strategies based on heuristics.

Random Path Selection and Coverage-Optimized Search are two search heuristics used in conjunction in KLEE [10] for path scheduling. These heuristics are interleaved to guide the exploration of paths during symbolic execution. Random Path Selection keeps records in a binary tree the program path followed by all active states. In this tree, the leaves represent the current states while the internal

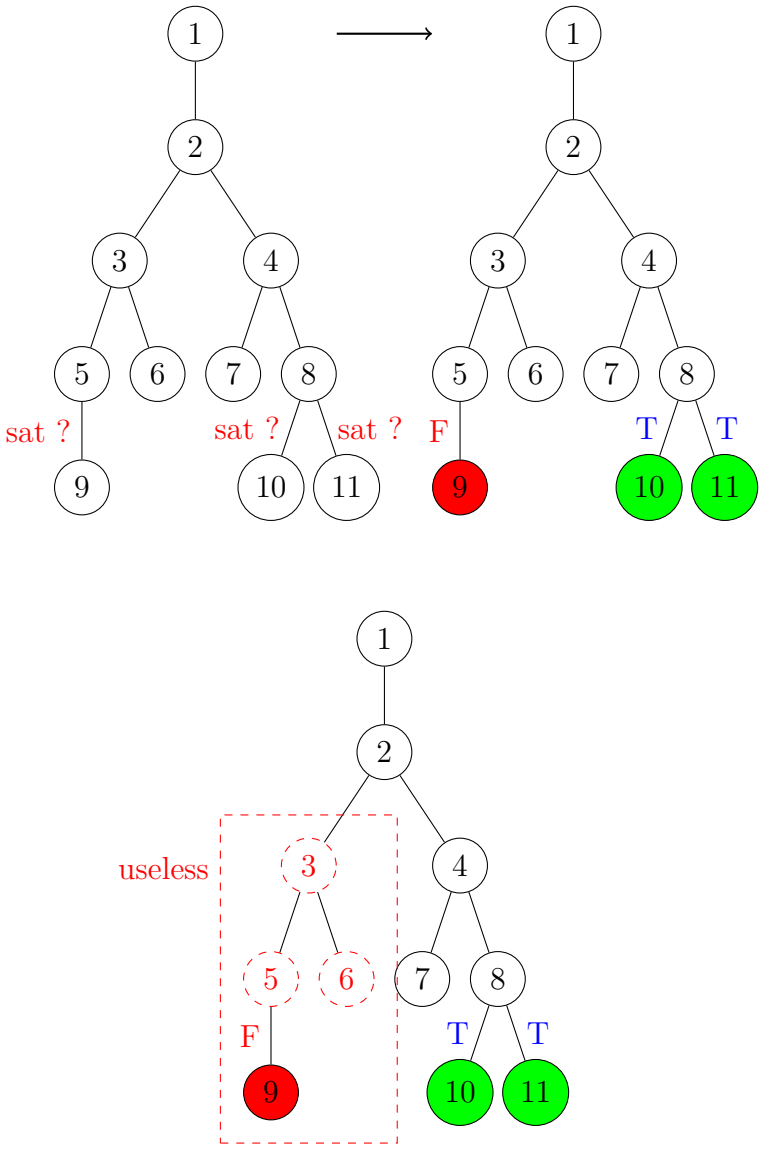

Figure 2.2: *CheckNothing* strategy

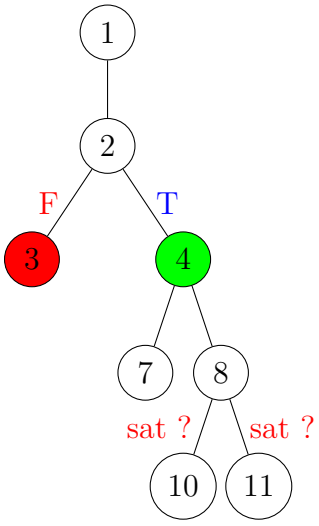

Figure 2.3: *CheckDynamic* strategy

nodes represent the locations where execution was forked. As implied by the name, states are chosen by randomly selecting the path to follow when meeting a branch point while traversing the binary tree from its root. This entails that each subtree has an equal probability of being selected, regardless of its size. First, this search heuristic favors states high in the branch tree, which tend to have fewer constraints on their symbolic inputs and are therefore more likely to reach uncovered code. Second, it helps to prevent the problem of "fork bombing," which occurs when a program rapidly creates new states, such as in a tight loop with a symbolic condition. It is noted in [10] that simply selecting a state at random has neither property.

Coverage-Optimized Search aims to choose states that are more likely to cover new code in the immediate future based on heuristics that compute a weight for each state, and then randomly picks a state based on these weights. For example, we might consider the minimum distance to an uncovered instruction and the call stack of the state as implemented in [10]. KLEE uses each strategy in a round-robin fashion to improve overall effectiveness and to protect against cases where an individual strategy gets stuck. To ensure that a state which frequently executes expensive instructions does not dominate execution time, KLEE runs each state for a "time slice" defined by both a maximum number of instructions and a maximum amount of time. Overall, these strategies have strengths and limitations, and the use of both can improve the effectiveness of KLEE in exploring possible paths in a program.

Directed Symbolic Execution is studied in the work of Kin-Keung Ma et al. [26]. They study how to reach a specific target line in a program. Their work introduces two directed search strategies : Shortest-distance symbolic execution (SDSE) and Call-chain-backward symbolic execution (CCBSE). SDSE uses a distance metric computed in an interprocedural control-flow graph (ICFG) to guide the symbolic execution towards a particular target line. The search chooses the path with the shortest distance to the target line in the ICFG, when deciding which path to continue executing. This approach is inspired by a heuristic used in the coverage-based search strategy from KLEE but is modified to focus on a single target line instead of all uncovered lines. CCBSE starts at the target line and works backward until it finds a feasible path from the start of the program, using forward symbolic execution as a subroutine. The CCBSE begins forward symbolic execution from the start of the function containing the target line, looking for paths to the target line. It then finds the possible callers of the function and runs forward symbolic execution from the start of each caller, searching for paths that call the function. The process continues backward up the call chain until a feasible path from the start of the program to the target line is found or the process times out.

Chopped symbolic execution is another prioritization technique proposed by David Trabish et al. [41]. The idea is to pre-define parts of the program to skip during symbolic execution and come back later if needed. The skipped code is executed in a lazy fashion. Completely discarding some parts will most likely result in inaccurate results, but executing some parts lazily maintains the soundess guarantees of symbolic execution, except for skipped functions that do not terminate, which could be considered a bug. The idea is to explore only the feasible paths while discarding the paths that are not relevant to the analysis.

Additionally, several efforts relying on machine learning have been made.

SyML [35] aims to guide symbolic execution toward vulnerable states through pattern learning. The first step is to extract features that characterize execution paths, functions, basic blocks, and connected components. These features are used to discover and learn patterns of vulnerable paths through supervised machine learning techniques. This proposed prioritization technique does not require any explicit prior knowledge about vulnerability patterns. The key intuition is that programs with similar bugs tend to have similarly dysfunctional execution states, which can be characterized by specific features such as sequences of API calls, memory dereferences, complex functions, or loop behaviors. This can be extended to malware: malware of the same family tend to have the same malicious states, which can be characterized by specific features.

LEARCH proposed by Jingxuan He et al. [20] is a learning-based state selection strategy. It is a machine learning regression model that directly estimates the contribution of each state towards maximizing coverage within a time budget. During training, LEARCH leverages the knowledge of existing heuristics, which means that it can benefit from any new expert-designed heuristics.

2.3 Machine learning for malware classification

2.3.1 Background and Motivation

In recent years, machine learning (ML) has emerged as a promising approach for malware classification, which aims to automatically identify and categorize malware based on their behavioral and/or structural features. Unlike traditional signature-based methods that rely on known patterns of malicious code, machine learning-based approaches can generalize to previously unseen malware families and variants, and can thus provide a more proactive and adaptive defense against evolving threats.

In this section, we discuss the motivations for using ML for malware classification, and highlight the limitations of traditional methods. We also describe different possible formats to represent a program using graph structures.

As discussed in Section 2.1, traditional methods for malware classification, such as signature-based detection and static analysis have limitations in detecting new and unknown malware variants, as these variants might be a slightly modified (e.g. obfuscated) version of a previous malware. Machine learning provides an alternative solution that has the potential to overcome these limitations and improve the accuracy and efficiency of malware classification. By using machine learning algorithms to learn from large datasets of labeled malware samples, it is possible to detect and classify unknown malware variants with a higher degree of accuracy than traditional methods [21].

However, ML-based classification systems face the challenge of striking a balance between: 1- Focusing on discriminative malware features that can achieve high classification accuracy; 2- Ensuring that the features are generalizable to different operational environments. Features that are overly specific to a particular malware variant may not be effective in detecting other types of malware that exist in different contexts. On the contrary, if the features are too generic, they may lack the necessary discriminative power to accurately classify malware [54].

To strike a balance between effectiveness and generality, a potential solution is to identify a malware representation (input to the classifier) that captures and maintains the structural and logical characteristics of the malware, while also being extractable from diverse formats of malware code. Graph representations

such as Control Flow Graphs (CFGs), Data Flow Graphs (DFGs), or System Call Dependency Graphs (SCDGs) are potential candidates that can effectively express the structural and logical information of malware and can be extracted from diverse malware code formats.

A CFG consists of nodes representing basic blocks of code and edges representing the control flow between these blocks. Each basic block represents a sequence of instructions that are executed sequentially without any branches or jumps. The edges indicate the possible transitions between these basic blocks, such as conditional branches, loops, or function calls.
A DFG represents data dependencies between a number of operations. Nodes are operations or computations, such as arithmetic operations or variable assignments, and edges represent the flow of data between these operations. The edges indicate the dependencies between the operations, showing how data is generated, modified, and propagated throughout the program.
A System Call Dependency Graph (SCDG) is a type of graph representation that captures the dependencies between system calls in a program. The vertices are represented and labeled by the system calls and the edges represent any information flow between the calls [6] (e.g, sharing an argument). These graphs can be constructed using different strategies. SCDGs will be described in more details in Section 2.4.

The power of graph structures comes from the fact that it is a general representation of data with a logical structure, and we can represent diverse types of data with graphs, even text or images [4].

Several machine learning classifiers based on graph structures have already been developed for malware classification. These classifiers include graph neural networks applied on CFGs [55] [48] and function call graphs [52] [11], similarity models based on siamese networks [11], and Support Vector Machines (SVM) with graph kernels to classify SCDGs [6].

In our study, we implement a graph neural network model to classify system call dependency graphs, as described in Section 2.4.

2.3.2 Graph neural networks

In this section, we will introduce graph neural networks, list commonly used notations, outline graph-related concepts and present the chosen architecture for our graph neural network.

Graph notations. We define a graph G as a pair (V, E) with V is a set of vertices and E a set of edges such that $\{\{u, v\} \subseteq V | u \neq v\}$. Let G be a graph with the set of edges and vertices given by $E(G)$ and $V(G)$, respectively. We

also consider labeled graphs where a label function $l : V(G) \to \Sigma$ assigns a label from Σ to each vertex of G. We use $l(v)$ to denote the label of vertex v. A graph $G' = (V', E')$ is a subgraph of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. Note that this is the same definitions as in [6].

Graph neural networks (GNN). In order to define graph neural networks, we will note $X_v \in R^{N \times F}$ the node feature vectors for $v \in V$ with N the number of nodes in the graph and F the number of features for each node. We also define a representation vector $h_r \in \{G, V\}$ that represents a node or an entire graph, depending on the nature of the task of the GNN. The main classification tasks of a GNN include but are not limited to node classification, graph classification and link prediction. The first two are more relevant to us since our goal is to perform graph classification:

1. Node classification: Each node $v \in V$ has an associated label y_v, and the goal is to learn a representation vector h_v of v such that the label y_v can be predicted as $y_v = f(h_v)$.

2. Graph classification: Given a set of graphs $G_1, \ldots, G_N \subseteq G$ and their labels $y_1, \ldots, y_N \subseteq Y$, we aim to learn a representation vector h_G that helps predict the label of an entire graph, $y_G = g(h_G)$.

Note that these are the same definitions as in [50].

The learning strategy of a GNN. The representation vector for a node, denoted as h_v, or for the entire graph, denoted as h_G is learned based on the structure of the graph (usually represented as an adjacency matrix) and the node feature vectors X_v.
The learning strategy of the GNN uses the *aggregate*, *combine*, and *readout* functions and is illustrated in Figure 2.4 taken from [11]). The learning relies on an aggregation scheme where the *aggregation* and *combine* functions are used to iteratively update the representation of a node by aggregating representations of its neighbors. Through k iterations of aggregation, the node's representation captures the structural information within its k-hop network neighborhood. Formally, the k-layer of a GNN is defined as follows [50]:

$$a_v^{(k)} = aggregate^{(k)}(\{h_u^{(k-1)} : u \in \mathcal{N}(v)\}) \tag{2.1}$$

$$h_v^{(k)} = combine^{(k)}(\{h_v^{(k-1)}, a_v^{(k)}\}) \tag{2.2}$$

$h_v^{(k-1)}$ is the feature vector of node v at the k-th iteration/layer. $\mathcal{N}(v)$ is the set of nodes adjacent to v and $h_v^{(0)} = X_v$.

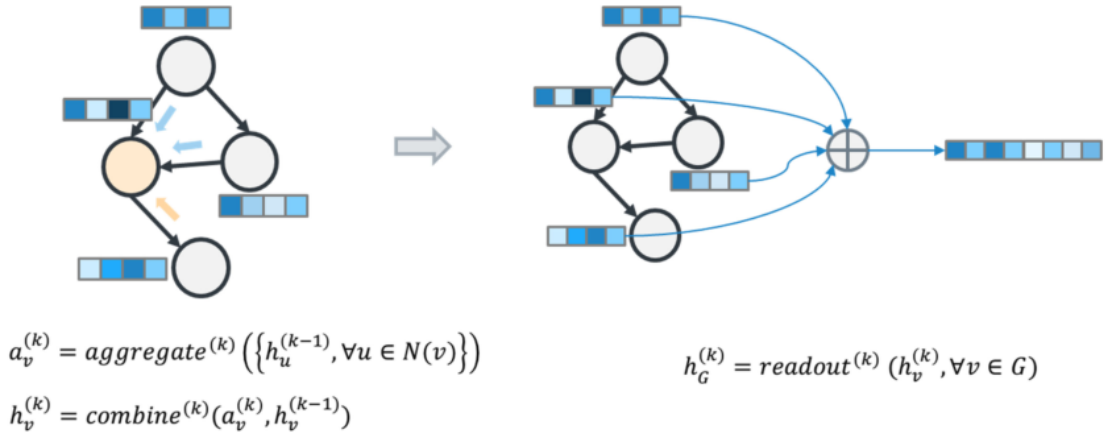

Figure 2.4: GNN learning strategy, from [11]

Usually, the different GNN architectures differs in the aggregation mechanism.

In the context of node classification, the representation $h_v^{(K)}$ obtained after the final iteration K, is employed for prediction. For graph classification, we add the use of the *readout* function that aggregate node features of the final layer in order to output a representation of the whole graph h_G, defiend as in [50]:

$$h_G = readout(\{h_v^{(K)} | v \in G\}) \tag{2.3}$$

The *readout* function can be a basic permutation invariant function (e.g. summation), or a more advanced graph-level pooling function.

Graph Isomorphism Network (GIN). In our work, we decided to use the GIN architecture for our GNN. It has been demonstrated in [50] to be a GNN as powerful as the Weisfeiler-Lehman (WL) graph isomorphism test [47], a test known to distinguish a broad class of graphs. It shows the expressive power of the GIN architecture in capturing different graph structures.

The GIN updates the node representation vector as follows:

$$h_v^k = MLP^{(k)}((1 + \epsilon^{(k)}).h_v^{(k-1)} + \Sigma_{u \in \mathcal{N}(v)} h_u^{(k-1)}) \tag{2.4}$$

The graph-level readout in GIN aims to capture increasingly refined and global node representations as iterations progress. While a sufficient number of iterations is crucial for strong discriminative power, earlier iteration features may exhibit better generalization. GIN incorporates an architecture inspired by Jumping Knowledge Networks (JK-Nets) [51] where graph representations from all iterations

are concatenated to form a comprehensive representation, rather than using a specific equation. It can be formalized with:

$$h_G = concat(readout(\{h_v^{(K)} | v \in G\}) | k = 0, 1, ..., K) \tag{2.5}$$

GNN explainability. The integration of graph structure and feature information in models gives rise to intricate systems, and the explanation of predictions generated by graph neural networks (GNNs) has become a topic of growing interest within the research community. This interest can be attributed to the rising popularity of GNN models and the desire to gain insights into the decision-making processes of these models. The work in [61] contains a survey of current methods of GNN explainability. The study in [2] proposes an evaluation framework for GNN explainability, analyzing different explaining algorithms, referred to as Explainers. In our work we implemented a first version of an explainability module, based on the GNNExplainer algorithm [59]. GNNExplainer is a method designed to help us understand how Graph Neural Networks (GNNs) make predictions in tasks involving graphs. It aims to identify a compact subgraph structure and a subset of node features that play a crucial role in the GNN's prediction. By maximizing the mutual information between the GNN's prediction and the distribution of possible subgraph structures, GNNExplainer generates consistent and concise explanations for a range of instances.

For constructing our GNN model, this study utilizes PyTorch Geometric [33], an extension library for PyTorch that specializes in geometric deep learning. PyTorch Geometric (PyG) offers a range of implemented state-of-the-art methods, enabling the efficient development of Graph Neural Networks and their explainability.

2.4 SEMA - Symbolic Execution for Malware Analysis

The SEMA Toolchain [22] [23] is where symbolic execution and machine learning are brought together for malware detection and classification.
The complete workflow of the toolchain is illustrated in Figure 2.5.

We start by performing symbolic execution on the binary using the angr framework, in order to extract execution traces that contain all the API calls that are encountered during symbolic execution. Those traces are combined to build a System Call Dependency Graph (SCDG). The SCDG is then used to detect or classify the malware using machine learning or graph mining approaches.

In the rest of this section we describe the different steps involved in the pipeline of SEMA Toolchain and its important components.

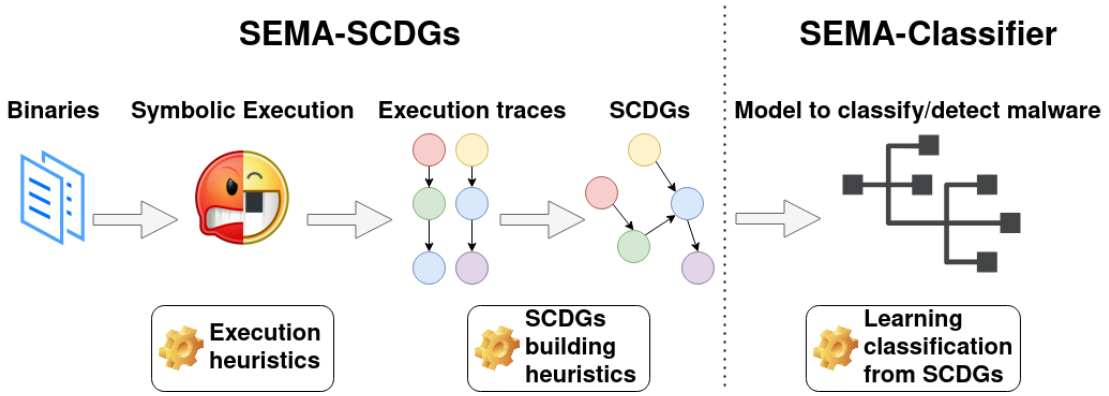

Figure 2.5: SEMA Toolchain workflow, from [22]

Binaries SEMA Toolchain performs symbolic execution directly on binaries, this is how we find most malware in the wild. The first step is to collect and label a list of binaries from different malware families. We can easily obtain such malware binaries using sources such as MalwareBazaar [1]. However, not all binaries are equally suitable for our analysis.

In order to construct the System Call Dependency Graph (SCDG), we rely on the system calls that occur during the execution of the malware. If a malware sample were to replace all the system calls with their corresponding source code, it would significantly limit the quality of the SCDG and our analysis. Additionally, some malware use techniques that complicate their analysis to SEMA Toolchain.

Obfuscation poses challenges not only for static analysis but also for symbolic execution. Some obfuscation techniques are intentionally designed to interfere with symbolic execution, often resulting in state or path explosion [46].

Symbolic execution The next step in the pipeline is to perform symbolic execution on the selected binaries in order to extract system calls and create execution traces. This is where the toolchain extends the angr framework.

The goal is to find a maximum of possible (satisfiable) executions for a given binary. Since we face the state and path explosion problem, the toolchain extends angr in order to optimize the exploration. Previous work [7] [37] [13] showed how optimization at the SMT solving level could impact performance. In this work we focus on how search strategies could improve the exploration.

The toolchain already includes four different search strategies: Breadth-First Search (BFS), Depth-First Search (DFS), a Custom Breadth-First Search (CBFS), and a Custom Depth-First Search (CDFS). These strategies guide the symbolic execution process in different ways to explore the program's execution paths.

BFS (Breadth-First Search) explores the paths in a breadth-first manner, starting from the initial program state and systematically exploring all feasible paths at each depth level before moving on to deeper levels. This strategy ensures that all paths at a certain depth level are covered before proceeding to the next level. Previous work using the toolchain [36] implements this strategy. This approach results in a significant increase in the number of states generated and a higher memory usage.

DFS (Depth-First Search) explores the paths in a depth-first manner, prioritizing deeper paths before exploring shallower ones. It starts from the initial program state and explores as far as possible along each branch before backtracking. This strategy often reaches deeper program states quickly but may miss certain paths that require exploring other branches. Since we have limited resources, if we have a large binary with a set of forks and deep paths, we might miss multiple paths.

CBFS and CDFS aim to reduce the limitations of BFS and DFS respectively. These custom versions will prioritize states that allow exploring new assembly instruction addresses of the program (new states). Both algorithms are presented in Figure 2.6.

CBFS-Strategy (Algorithm 1) starts by selecting a set of states, denoted as L, for exploration and adds them to the list R (line 4). It then iterates through the remaining available states and checks if a state leads to an unexplored part of the code or has a shorter path of execution (line 6). If such a state is found, it is added to R while removing a state with lower priority (e.g., a state that does not lead to a new instruction or has a longer depth). Once all states have been processed, R is returned to allow angr to perform a new execution step on the updated set of states. The CBFS-Strategy complements the BFS-Strategy, providing an optimized approach for state exploration.

CDFS-Strategy (Algorithm 2), differs from the CBFS-Strategy in the condition used to select the successor state at Line 6.

Regarding the satisfiability check strategy, the toolchain currently uses the *CheckAll* strategy.

At the end of the symbolic analysis (timeout or exploration of all paths), the feasible paths found represent the execution traces (as in Figure 2.5) and will be used in the next step to create the SCDG.

In order to conserve memory during symbolic execution, SEMA organizes states into separate stashes based on their priorities, instead of exploring all available states concurrently [43]. This allows us to manage memory usage more efficiently by selectively focusing on specific subsets of states at any given time. The most relevant stashes for our work are the following:

- Active (default in ANGR): This stash contains states that are currently being explored. To optimize memory usage, an option has been implemented to

Algorithm 1 CBFS exploration

```
1: Inputs: A set of states: S
2: Limit of states: L
3: Outputs: A set of L states: R
4: R ← S[: L]
5: for state s ∈ S do
6:     if new(s.next_ip)|{∃state ∈ R|s.depth < state.depth & !new(state.next_ip)}
       then
7:         Remove state from R
8:         Add s to R
9: Return R
```

Algorithm 2 CDFS exploration

```
1: Inputs: A set of states: S
2: Limit of states: L
3: Outputs: A set of L states: R
4: R ← S[: L]
5: for state s ∈ S do
6:     if new(s.next_ip)|{∃state ∈ R|s.depth > state.depth & !new(state.next_ip)}
       then
7:         Remove state from R
8:         Add s to R
9: Return R
```

Figure 2.6: CBFS and CDFS algorithms, from [6]

limit the number of states in this stash, referred to as *simul_state*. If the limit is exceeded, some states are transferred to the Pause stash.

- Pause: This stash serves as a temporary storage for states that are put on hold until space becomes available in the Active stash. The size of the space in the Pause stash is a configurable parameter in the toolchain. If new states emerge and there is no available space in the Pause stash, some states are dropped to accommodate the new arrivals.

The first goal of our work is to evaluate the different search strategies and implement new ones.

System Call Dependency Graph Based on the execution traces obtained in the previous step, we extract the encountered system calls along with their arguments and addresses and use them to create the SCDG by merging and linking calls between the different traces.

Each system call is represented as a vertex in the graph. The edges in the graph represent information flow between the system calls, such as sharing arguments or dependencies on the output of other system calls as illustrated in Figure 2.7.

CFGs might indicate the transitions between blocks of instructions but they don't necessarily reflect the concrete program's behavior: we could split basic blocks and add multiple jump instructions easily without necessarily changing the behavior of the program while obtaining a totally different graph. On the other hand, the SCDG provides a representation of how system calls interact and

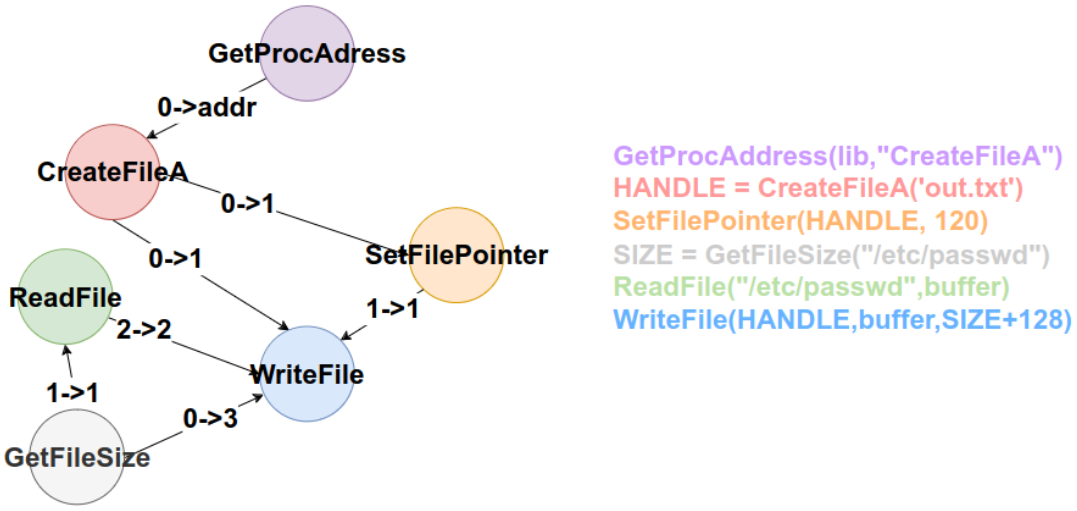

Figure 2.7: Example of a SCDG built with the three-edges strategy, from [6]

exchange information during program execution, it gives insights on the actual program's behavior.

Different merging strategies can be applied based on how we define dependencies between calls. Previous work on the toolchain [6] describes and evaluates 5 different building strategies.

One possible strategy is called the three-edges strategy. The first type of edge, (if used alone it's known as a "one-edge strategy") connects two calls if they share an argument with an identical value. This occurs, for instance, when two calls have the same file handler.
The second type of edge connects two calls that have arguments with the same symbolic value. An example would be a symbolic file size returned by a call and passed as an argument to another call along with another value.
The third type of edge links two calls if an argument of the first call corresponds to the calling address of the second call. This situation commonly arises in the dynamic loading of a library.
To provide further information, each edge is labeled with the index of the argument in both calls, with the return value of a call assigned index 0. This labeling scheme helps to identify and differentiate the various arguments within the linked calls.

The resulting graphs are used as inputs for the classifiers.

In situations where a binary does not produce an adequate number of edges, the creation of the System Call Dependency Graph (SCDG) is halted. This choice is made because it would be ineffective to construct a dependency graph that fails to

accurately represent the dependencies among system calls. This situation highlights the fact that certain malware employ techniques that are currently not supported or comprehensible by SEMA, serving as an example of such cases. This can serve as a motivation for future contributions and improvements in order to enhance the capabilities of SEMA and address these challenges.

SEMA-Classifer Now that the SCDGs are built, the final step is to feed them to the classifier for training.

We have two classification tasks: malware detection and malware family classification. The first one consist in determining if a binary is a malware or a cleanware, while the second one consist in determining to which malware family the binary belongs (assuming it is a malware).

Prior to this work, SEMA already implements three classification modules. The first technique leverages graph mining and utilizes the *gspan* algorithm [56]. This algorithm, based on graph isomorphism, can extract a common subgraph from a collection of SCDGs belonging to the same malware family. This subgraph serves as the family's signature and can be employed in the detection process.

The second approach is based on graph kernels [6], which are functions designed to measure the similarity between pairs of graphs. Unlike isomorphism-based techniques, graph kernels enable the detection of similarities that go beyond structural equivalence [8] [53]. These functions can be utilized in Support Vector Machine (SVM) algorithms.

The final technique employs an autoencoder to compute vectors that represent each SCDG. These vector representations are then fed into deep neural network algorithms, as described in [14], to enable effective classification and detection.

Additionally, SEMA introduces a novel federated learning module for symbolic malware analysis [5], inspired by similar efforts in the field of Android malware analysis [17]. In this federated learning approach, each client trains its own classifier using its individual malware dataset. Instead of sharing the entire dataset, only the parameters of the trainer model are exchanged among the clients, employing homomorphic encryption techniques [28]. SEMA's aim is to encourage open-source contributions, including new strategies and learning algorithms, while ensuring the confidentiality of clients' malware datasets. The addition of a federated learning module might encourage this cooperation and reduce the reservation of different possible collaborators.

The second goal of our work is to add a new classification module based on graph neural networks.

Chapter 3

Implementation

In this chapter we will present the exploration strategies that have been investigated, in Section 3.1, and provide detailed information on the implemented classifiers, in Section 3.2.

3.1 Exploration strategies

Inspired by the Random Path Selection and Coverage-Optimized Search in KLEE [10], our study evaluates the impact of randomness and the attribution of weights in an exploration strategy. We explore three distinct categories of strategies: stochastic search, which involves purely random selection; custom stochastic search, which combines random selection with weighted prioritization of states; and deterministic weighted path selection based on assigned state weights.

During exploration, angr relies on a `step` function to take one step of exploration. Using angr's API, we need to define our own `step` function to implement our own exploration technique.

We keep the same structure for the `step` functions of the different search strategies. The only difference is how we pick the states to explore.

Stochastic search The starting point of our study, which serves as the foundation for rest of our exploration strategies, is the stochastic search, where we select the paths to explore based on weights assigned randomly to the states. This exploration strategy is referred to as **STOCH**.

We start by presenting the `step` function. The structure is illustrated in Algorithm 3. Concretely, the explorer will continue to fill the *active* stash with the encountered new states and explore them until: 1- No new states are found while the *pause* stash is not empty. 2- The *active* stash size reaches the limit of

simul_state (number of states that can be explored simultaneously). 3- There is some space left in the *active* stash and the *pause* stash is not empty.

In the first scenario, we move states randomly from the *pause* stash to the *active* stash. We ensure that enough states are moved to completely fill the *active* stash. If there are fewer states in the *pause* stash than needed, we move all available states from the *pause* stash to the *active* stash. The states are picked using the `weighted_pick` function (line 3), which is defined differently for each exploration strategy.
In the second and third scenarios, we move the states currently in the *active* stash to the *pause* stash, then we repeat the first scenario: pick states from the *pause* stash according to weights and place them in the *active* stash.

You'll notice that we choose and filter states based on their address, not the complete state. This done to prevent choosing multiple paths leading to the same state address.

Algorithm 3 Step function for STOCH

Input: simgr: angr's state manager
Output: simgr: angr's state manager

1: **if** active is empty **then**
2: **if** pause is not empty **then**
3: chosen = weighted_pick(pause, max_simul_state);
4: move(from=pause,to=active,filter=s | s.addr ∈ addresses of chosen);
5: **if** (len(active) > max_simul_state) or (len(active) < max_simul_state and pause is not empty) **then**
6: move(from=active,to=pause,filter=True);
7: chosen = weighted_pick(pause, max_simul_state);
8: move(from=pause,to=active,filter=s | s.addr ∈ addresses of chosen);

The implementation of the `weighted_pick` function for **STOCH** strategy is illustrated in Algorithm 4. The function takes a list of states as input along with the number n of states it has to choose. For each state address from the input states, we assign a random score that we call affinity (line 5), this is the weight of the state address. In line 10 we normalize these weights so that the new value $\in [0; 1]$. These values represent the probability for the corresponding state address to be chosen. The next step is to pick addresses according to the (random) probabilities (line 14). For this, use `numpy.random.choice` [31] that supports making choices according to probabilities. Since we can have multiple paths, and thus multiple states, that lead to the same state address, we still need to randomly pick states with the corresponding picked addresses. We identify all candidate states and select randomly between them, using uniform probabilities.

We could argue that there is no need to incorporate weights in state selection with this method, but we view this strategy as more of an experimental approach. It serves as a groundwork for subsequent exploration strategies with heuristics.

Algorithm 4 Weighted Pick for STOCH

Input: states: List of states to pick from; n: Number of states to pick
Output: picked: List of selected states

if len(*states*) = 1 **then**
    picked = *states*
    return *picked*
for each *state* in *states* **do**
    $affinity[state.address]$ = random(); ▷ Assign a random score to *state* if it is a new *state* address
Initialize empty lists: *weights*, *population*;
for each *state* in *states* **do**
    Add *state.address* to *population*;
    Add $affinity[state.address]$ to *weights*;
total_weight = sum(*weights*)
norm_weights = $[\frac{w}{total\_weight} \ \forall \ w$ in *weights*]; ▷ Normalize the weights
if n > len(*population*) **then**
    Set n to len(*population*)
picked_addr = random.choice(*population*, p=*norm_weights*, size=n, replace=False); ▷ Pick n addresses
    Initialize an empty list: *picked_states*;
for each *state* in *states* **do**
    if *state.address* is in *picked_addr* and *state* is not in *picked_states* **then**
        Add state to *picked_states*;
picked = random.choice(*picked_states*, size=n); ▷ Pick n states randomly from *picked_states*
return *picked*

Custom stochastic search Using the previous strategy as foundation, we create a custom stochastic search strategy. We keep the same idea: we choose state randomly based on probabilities, but this time the probabilities are not random. The weights are defined using an evaluation function.

We have designed an evaluation function that assigns higher importance to execution traces containing a larger number of interesting system calls, as well as traces with a greater overall number of system calls and those that lead to new

state addresses. We have implemented three versions of this evaluation function, which will be discussed in detail in the following Section 3.1.1.

We consider interesting system calls to be those frequently encountered in malware [27]. Our reasoning is based on the assumption that if a trace has previously shown suspicious system calls, it is more likely to expose or reveal additional malicious behavior in the binary. Furthermore, since our classification is built upon System Call Dependency Graphs (SCDGs), it is pertinent to consider the characteristics of system calls in our analysis.

The previous algorithms shown will undergo a few changes. First, since the score is computed based on the execution trace history, we no longer filter states based on their addresses. Instead, we utilize the unique *id* of each state for selection and filtering purposes. This can be seen in the new *step* function in Algorithm 5. The only differences from the previous version are the lines 4 and 8. Second, for the `weighted_pick` function, instead of using a random number generator, we now compute scores for the states using the evaluation function. We also identify states using their unique id. The new version is illustrated in Algorithm 6, the modifications are in the lines 5, 9 and 14.

Algorithm 5 Step function for CSTOCH* and WSELECT*

Input: simgr: angr's state manager
Output: simgr: angr's state manager

1: **if** active is empty **then**
2: **if** pause is not empty **then**
3: chosen = weighted_pick(pause, max_simul_state);
4: move(from=pause,to=active,filter=s | s.id ∈ ids of chosen);
5: **if** (len(active) > max_simul_state) or (len(active) < max_simul_state and pause is not empty) **then**
6: move(from=active,to=pause,filter=True);
7: chosen = weighted_pick(pause, max_simul_state);
8: move(from=pause,to=active,filter=s | s.id ∈ ids of chosen);

Under this category, we implemented three exploration strategies **CSTOCH1**, **CSTOCH2** and **CSTOCH3**. They differ in how `eval_func(.)` is defined.

Deterministic weighted selection The final category is similar to the custom stochastic search. We assign and compute weights based on an evaluation function. However, instead of picking the states randomly with weights, simply select the states with the best scores without randomness. We still identify states using their unique *id*.

Algorithm 6 Weighted Pick for CTOCH*

Input: states: List of states to pick from; n: Number of states to pick
Output: picked: List of selected states

if len(*states*) = 1 **then**
    picked = *states*
    return *picked*
for each *state* in *states* **do**
    $affinity[state.id]$ = eval_func(*state*); ▷ Assign a new score to *state*
Initialize empty lists: *weights*, *population*;
for each *state* in *states* **do**
    Add *state* to *population*;
    Add $affinity[state.id]$ to *weights*;
total_weight = sum(*weights*)
norm_weights = $[\frac{w}{total\_weight} \; \forall \; w$ in *weights*]; ▷ Normalize the weights
if n > len(*population*) **then**
    Set n to len(*population*)
picked = random.choice(*population*, p=*norm_weights*, size=n, replace=False); ▷ Pick n states randomly from *population*
return *picked*

The *step* function is the same as for the stochastic search, shown in Algorithm 5. However, the *weight_pick* is adapted as in Algorithm 7. We compute the scores for each state, then we sort theses states according to their scores in a descending way. Finally, all we have to do is to return the n first elements from that sorted list.

In this category, we implemented three exploration strategies **WSELECT1**, **WSELECT2** and **WSELECT3**. They differ in how the evaluation function `eval_func(.)` is defined. The evaluation function is defined in the Section 3.1.1.

3.1.1 Evaluation function

CSTOCH1 - WSELECT1 This first version will only take into account the number of interesting system calls, as well as the overall number of system calls in the trace leading to a particular state. Formally, the score can be expressed as follows:

$$\text{score} = 2 \times n^* + n \tag{3.1}$$

Where n^* denotes the number of interesting system calls and n the number of the other system calls encountered.

Algorithm 7 Weighted Pick for WSELECT*

Input: states: List of states to pick from; n: Number of states to pick
Output: picked: List of selected states

if len(*states*) = 1 **then**
    picked = *states*
    return *picked*
for each *state* in *states* **do**
    $affinity[state.id]$ = eval_func(*state*); ▷ Assign a new score to *state*
population = *states*;
if n > len(*population*) **then**
    Set n to len(*population*)
sorted_list = sort(*population*, key=lambda x: $affinity[x.id]$], reverse=True)
picked = *sorted_list*[:n]; ▷ Pick n best states
return *picked*

CSTOCH2 - WSELECT2 The second version of the evaluation function considers the count of interesting system calls, the total number of system calls in the trace leading to a specific state, and whether the state has a new address. The score can be formulated as follows:

$$\text{score} = 2 \times n^* + n + n\_new\_addr \tag{3.2}$$

Where n^* denotes the number of interesting system calls, n the number of the other system calls encountered and n_new_addr which is equal to 1 if the state has a new address, 0 otherwise.

CSTOCH3 - WSELECT3 The third version, is the same as the second, but we add 2 points to the score if the state has a new address. Formally:

$$\text{score} = 2 \times n^* + n + 2 \times n\_new\_addr \tag{3.3}$$

Where n^* denotes the number of interesting system calls, n the number of the other system calls encountered and n_new_addr which is equal to 1 if the state has a new address, 0 otherwise.

Furthermore, we have also implemented a variation of these three versions where we consider only the unique system calls (represented as a set instead of a list). However, during initial experiments, we observed that this approach did not significantly improve performance and resulted in a lower overall number of system calls discovered during exploration. Therefore, we decided not to pursue these variations further.

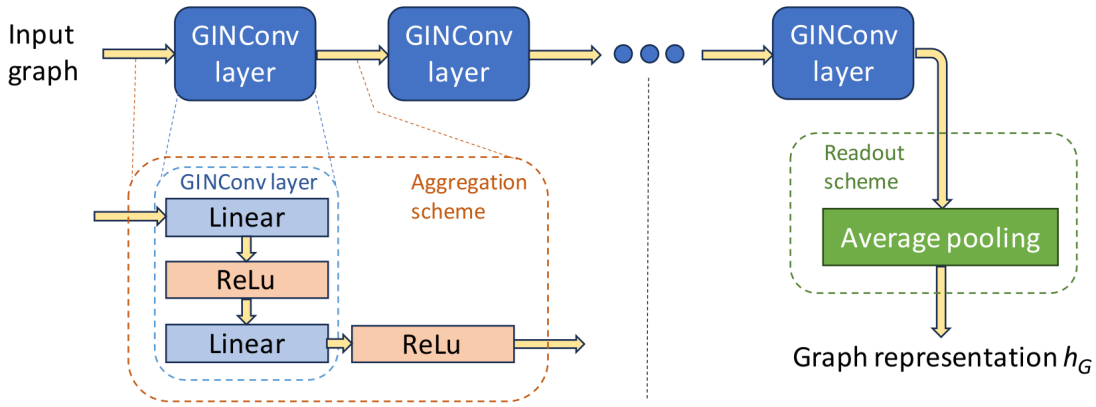

Figure 3.1: GIN structure

3.2 Graph neural network models

In this section, we will give more insight on the structure of our GIN models. Our work is inspired by the models implemented in [48].

Graph Isomorphism Network (GIN) For our first model we take the equation 2.4 and set ϵ to 0, we refer to it as GIN. The MLP structure we used is illustrated in the Figure 3.1. One GINConv layer, applies a sequence of linear transformations followed by ReLU activation functions. After the last iteration, we use an average pooling to obtain a graph-level representation. This is the *readout* function (equation 2.3).

Once we have the graph representation h_G, it is passed through an output layer to produce logits, which are converted to a probability distribution using the log softmax function.

Graph Isomorphism Network with Jumping Knowledge (GINJK) Our second model is a GIN model with JumpingKnowledge, we refer to it as GINJK. We keep the same structure as for the first GIN model, but rather than relying only on node representation from the last iteration ($h_v^{(K)}$), we utilize the JumpingKnowledge (JK) layer. As explained in Section 2.3.2, this layer concatenates node representation from all iterations (equation 2.5), allowing us to capture information from multiple stages of the model's execution. This is illustrated in the Figure 3.2. We use an average pooling function on the concatenated vectors to produce the graph-level representation h_G.

We follow the same prediction approach as in GIN. The graph representation

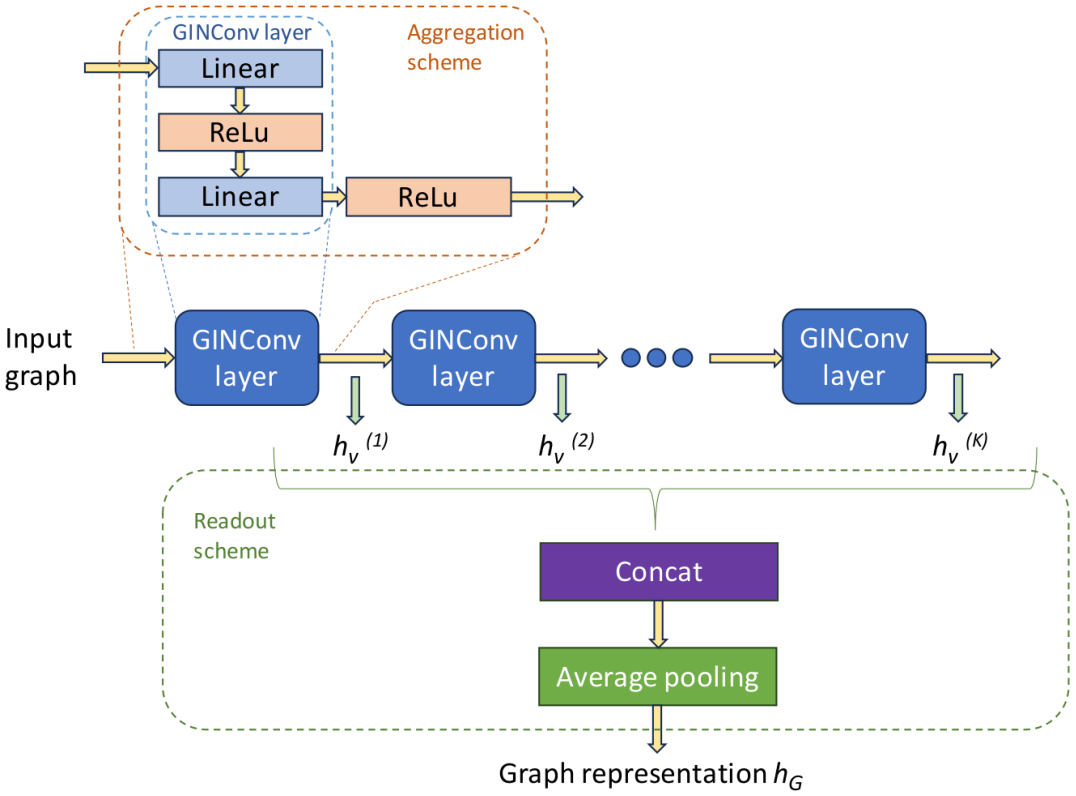

Figure 3.2: GIN with JumpingKnowledge structure

h_G is passed through a linear output layer to generate logits, which are then transformed into a probability distribution using the log softmax function.

Chapter 4

Experimental results and analysis

In this chapter we will explain our evaluation setup and present our results.
We can evaluate the effectiveness of our exploration strategies by addressing the following key questions:

1. Characteristics of the generated System Call Dependency Graphs (SCDGs):
    - What are the distinct features observed in the generated SCDGs?
    - Does the generated SCDGs uncover previously unknown behaviors in the binary?
    - To what extent does the generated SCDGs improve code coverage?
2. Impact of the generated SCDGs on classification:
    - How does the generated SCDGs impact the classification?
    - What insights or advantages do SCDGs provide for accurate classification?

Addressing these questions allows us to assess the quality and effectiveness of a System Call Dependency Graph.

We start by evaluating exploration strategies at the code coverage level, answering the first set of questions in Section 4.1. Then we evaluate our classifiers with the different exploration strategies, addressing the second set of questions, in Sections 4.2. Finally, we study a small example illustrating how we can use the GNN explainability module to interpret a GNN's prediction, in Section 4.3.

4.1 Exploration strategies evaluation

4.1.1 Experimental setup

Knowing in advance all the possible execution paths a binary can take is not an easy task. This is partially due to the limitation of static analysis. As a result, instead of computing the accurate code coverage to evaluate our exploration strategies, we approximate it by studying different metrics easier to obtain. The principal metrics are:

- `tot_syscalls`: This is the number of all system calls found during exploration.
- `tot_uniq_syscalls`: This is the number of distinct system calls found.
- `nb_instruction`: This is the number of instruction symbolically executed during exploration.
- The number of edges in the generated SCDGs.

We had 2 virtual machines at our disposal for measurements. Their configuration is as in Table 4.1.

Except for the time experiment, in all experiments to create SCDGs we used a timeout of ten minutes for the symbolic execution. We allowed a maximum of 5 states to be explored at the same time (`max_simul_states`=5).

We generated SCDGs for 583 malware samples divided into 15 families, as specified in Table 4.2. We used a set of the malware samples used in [6]. These samples are originated from Cisco and MalwareBazaar [1]. Each family should at least have 32 samples. Families with less than that, are instances where the construction of the SCDG have been aborted or failed. This can occur when a binary fails to generate a sufficient number of edges in the graph. Such instances serve as examples that highlight the challenges faced in accurately capturing the dependencies among system calls. Moreover, we note that even if we built SCDGs for 583 different samples, not all search strategies reached that number of generated SCDGs. Table 4.3 shows the percentage of the samples for which we generated SCDGs out of the 583 samples with constructed SCDGs for each exploration strategy.

4.1.2 Code coverage approximation

When analyzing the results, we saw that all the methods showed equivalent performances on malware families nitol and wabot. We decided to discard them in our comparisons since they do no add a particular value to our analysis. We can

Table 4.1: Platforms configuration for experiments

	Machine 1	Machine 2
Operating System	Ubuntu 18.04.3 LTS	Ubuntu 22.04.2 LTS
Kernel	Linux 4.15.0-142-generic	Linux 5.15.0-25-generic
Architecture	x86-64	x86-64
CPU(s)	8	4
Thread(s) per core	1	1
CPU model name	Intel Core Processor (Haswell, no TSX, IBRS)	Intel Core Processor (Haswell, no TSX, IBRS)
CPU MHz	1995.312	1995.278
RAM	24GB	12GB

Table 4.2: Families and Number of Samples

Family	**# samples**	**Family**	**# samples**
bancteian	42	sfone	32
ircbot	36	lamer	52
sillyp2p	42	RedLineStealer	32
sytro	42	RemcosRAT	43
simbot	42	Sodinokibi	3
FeakerStealer	41	delf	44
nitol	50	gandcrab	42
wabot	40		

Table 4.3: Percentage of graphs generated per method, out of the 583 constructed graphs

Method	Percentage
CSTOCH1	96.74%
CSTOCH2	96.57%
CSTOCH3	96.57%
STOCH	96.05%
WSELECT1	94.34%
WSELECT2	98.11%
WSELECT3	96.05%
CDFS	94.17%
CBFS	97.77%

however conclude that on shorter programs or with enough time, most methods can converge to equivalent results.

To counter the bias due to randomness, the following measurements are based on averages over 5 iterations for the methods that involve randomness: STOCH and CSTOCH1-2-3.

The experiments of this section have been made on Machine 1.

Total number of system calls To begin our analysis, we compare the total number of system calls identified by the various methods for the 15 selected families. The findings are depicted in Figures 4.1 and 4.2. We present the average count of encountered system calls within each family's samples. Due to variations in scale among the families, we plot normalized values for better visualization.

We find that CBFS is usually the strategy that finds the most system calls and beats CDFS (with a few exceptions). We explain this result with the behavior of the classical BFS algorithm. For a given amount of time, a DFS will go deeper in the tree while the BFS will explore the tree in "width". In the case of a symbolic execution tree where each fork is an execution path, we can say that the CBFS will see the beginning of multiple different executions without necessarily seeing the end or going much deeper in the tree.

Seeing a lot of different beginning of execution paths increases the number of all system calls encountered. We can have multiple execution traces that follow the same pattern in the beginning. It might also find one (or more) loop(s) that is (are) not found by the CDFS strategy, this increases the counter of function calls if it is inside the loop.

We can also observe that the methods involving randomization perform better than the deterministic methods. This can be due to the fact that randomization might lead to a path you wouldn't normally take.

Number of distinct system calls Now, if we look at the number of distinct system calls found, depicted in Figure 4.3 and 4.4, we have slightly different results.

Despite finding the most of overall system calls, CBFS has generally the least diversity in the calls it finds. This is explained by the behavior that make it explore the beginning of multiple different traces at the expense of going deeper. This actually shows that the different paths have less variety at the beginning.

CDFS, on the other hand, goes deeper in the execution tree but sees less different traces than CBFS. Exploring in depth make it possible to see more of one particular behavior of the binary, thus finding a more diverse set of system calls. Since most of the traces have similar beginnings, we could argue that we don't actually need to see all traces to learn the most of the structure of the program.

We observe that methods involving randomization find also less diversity in the

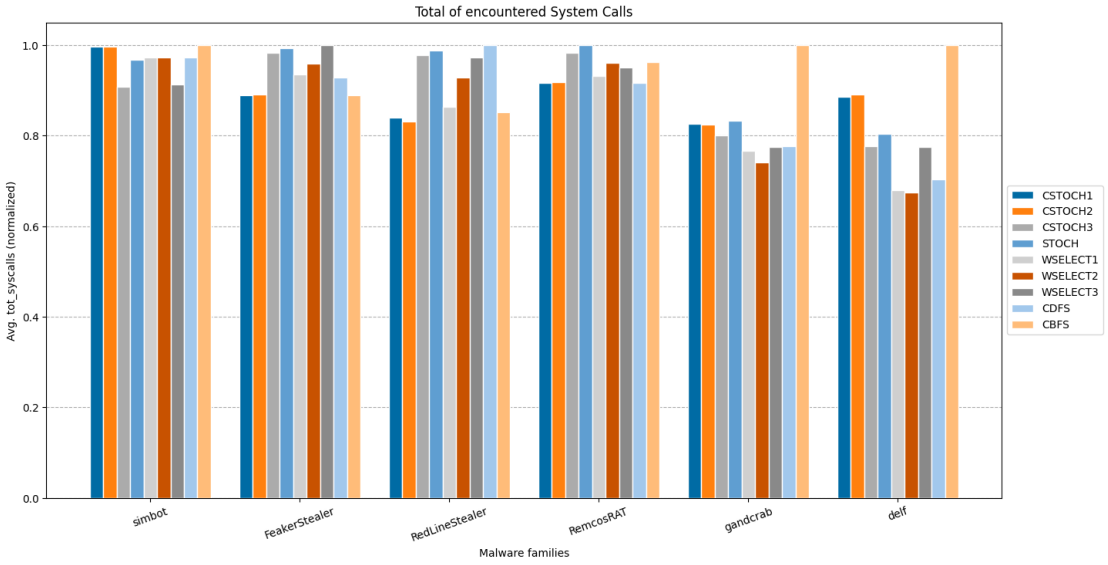

Figure 4.1: Number of system calls, part 1

set of system calls found than the methods **WSELECT***. Actually, **WSELECT*** methods seem to be on par with CDFS in most of the families, particularly WSELECT3. We also note that accross random methods, CSTOCH3 is the one that finds the most diversity in the system calls. WSELECT3 and CSTOCH3 are the methods that give more importance to the states that lead to new state addresses. This heuristic guides the exploration toward new state addresses and thus new system calls.

Number of instructions To help us get a sense of the code coverage, we report the number of new instructions symbolically executed during the exploration. The results are illustrated in Figures 4.5 and 4.6.

As expected, the CBFS method exhibits a lower count of newly executed instructions due to its shallower exploration compared to the other methods. This aligns with the reasons previously discussed, where CBFS explores the program to a shallower extent. In contrast, search heuristics such as CSTOCH2-3 and WSELECT2-3, which actively consider new state addresses, encounter a greater number of instructions than strategies that does not take into account unexplored state addresses.

Number of edges Since edges in SCDGs express dependencies, we decided to compare the number of edges found during exploration according to the different

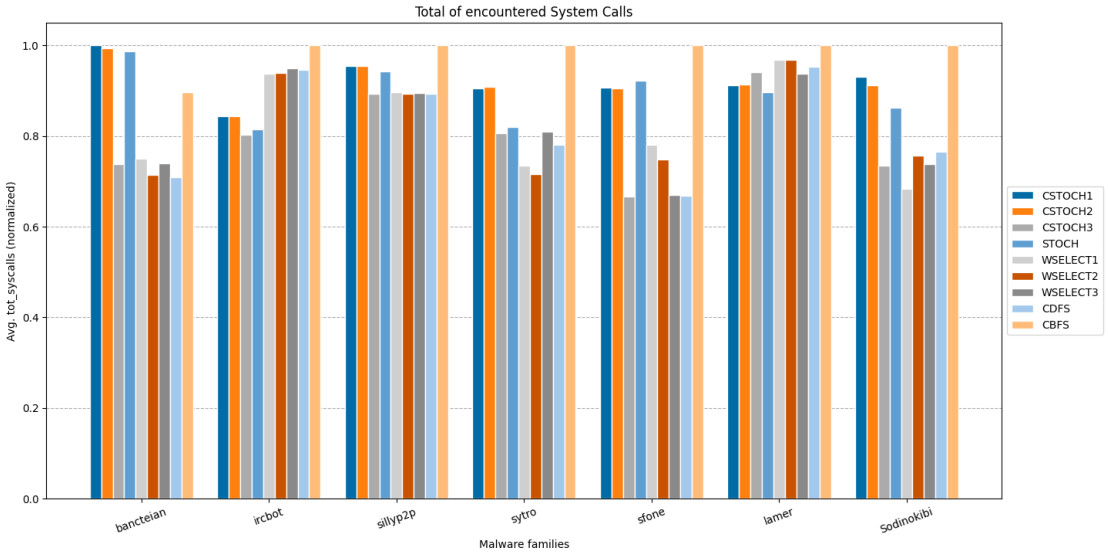

Figure 4.2: Number of system calls, part 2

strategies. Figures 4.7 and 4.8 report the average number of edges for each family, with each exploration method, normalized as previously. What we notice is that CDFS looks the most stable, along with WSELECT3. In contrast, with a few exceptions, CBFS leads to the creation of fewer edges most of the time. What we deduce from that is that for most families, most of the dependencies are found when going deeper in the exploration, instead of having a shallow exploration that explores different traces. If we make a link with our previous results, we can establish a connection that emphasizes the importance of the diversity of system calls in expressing dependencies. Rather than aiming for a higher count of system calls overall, it is preferable to prioritize diversity. In other words, it is more valuable to have a smaller set of distinct system calls that cover a wider range of behaviors, as opposed to a larger set that includes redundant entries.

Overall analysis After having an overview on the results of these metrics, we notice that according the chosen metrics, CDFS and CBFS are usually opposites: they are rarely both the highest or the lowest at the same time.

We observe that the new exploration strategies **CSTOCH*** and **WSELECT*** will rarely perform worse that CDFS and CBFS at the same time. They are usually between the two or higher than both.

CSTOCH* and **WSELECT*** strategies both guide the exploration towards finding system calls. We see that the less we give room for randomness for example by adding weight on states that lead to new addresses, the closer we get to the

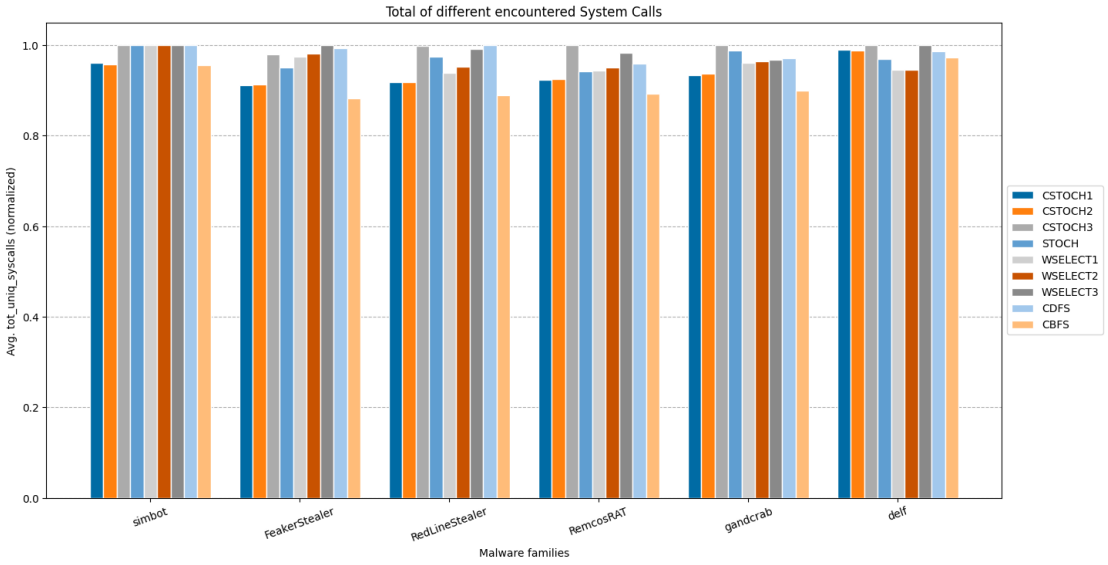

Figure 4.3: Number of distinct system calls, part 1

CDFS results, which means we go deeper in the exploration and have better sense of an execution's full behavior.

We notice that randomness increases the number of overall system calls while still giving a higher diversity than CBFS. This shows that it allows exploring new states, without necessarily staying at the surface of the exploration. It might also allow getting out of a loop that keeps executing the same system call.

What we conclude is that our new heuristics can be considered as a compromise between the strategy that goes deep with the risk of missing different behaviors (CDFS) and the strategy that tries to explore all possible behaviors with the risk of staying at the surface and not seeing the end of a behavior (CBFS).

4.1.3 The impact of randomness

As mentioned earlier, the experiments methods involving randomness were done five times.

In this section we study the impact of randomness by analyzing the standard deviation between the results of these five iterations.

Table 4.4 reports the standard deviation of the mean values across five iterations for each family, grouping all random strategies STOCH and CSTOCH1-2-3.

In general, there is an important variation for the overall system calls found and the number of instructions. However, the number of unique system calls is far more stable.

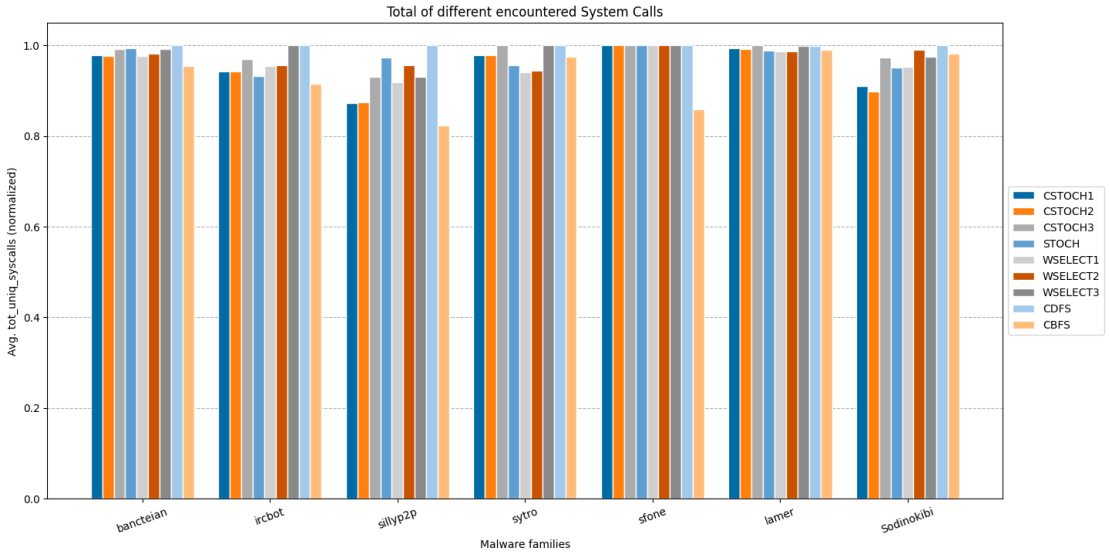

Figure 4.4: Number of distinct system calls, part 2

Our intuition is that randomness influences the path taken, it allows for example to leave a loop or to stay in it, that is what changes between iterations. But we don't necessarily need to do several iterations to find the majority of distinct system calls.

Our analysis may be influenced by the inherent biases of the heuristics employed, as they tend to guide the exploration towards a specific set of system calls. Additionally, another source of bias may come from the similarity in the types of system calls present within the samples. Furthermore, since there is a finite number of distinct system calls in a program, exploring the symbolic execution tree with the current heuristics allows finding most of them in the given time space, creating a considerable overlap between the different run instances.

4.1.4 The impact of timeout

In this section, we evaluate how the timeout value influence the results of the previous section. The experiments are made on the Machine 2.

Our experiments consist in running the symbolic analysis with a timeout of one minute (instead of ten minutes). We wanted to use a five minute timeout, but the memory load on the virtual machine was too much (Machine 2 has less memory that Machine 1, which was already running other experiments).

Figures illustrating the numbers of overall system call, distinct system calls and instructions encoutered can be found in Section A.1.1 of the appendix.

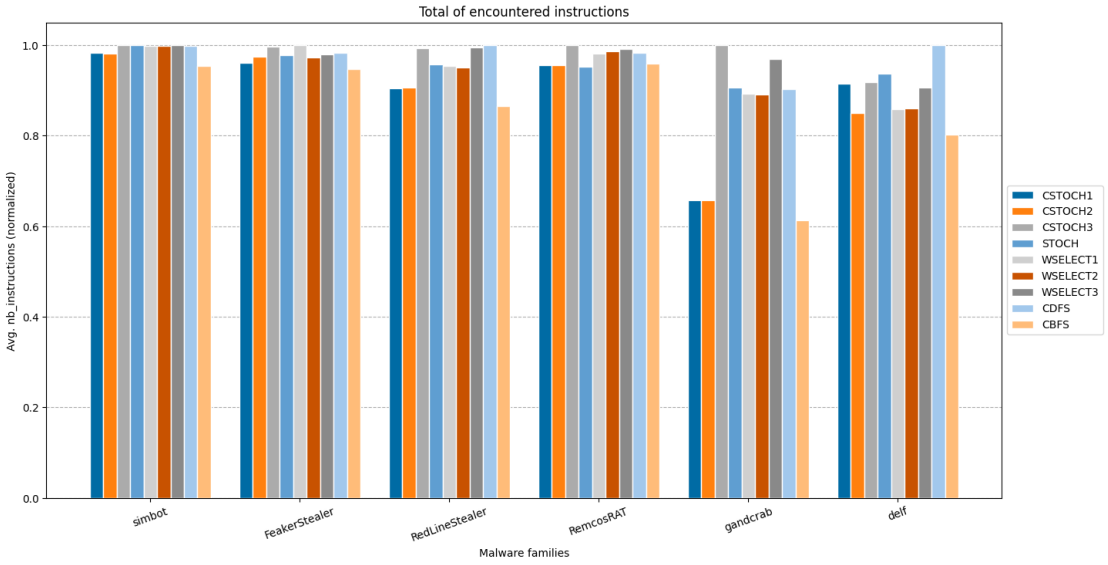

Figure 4.5: Number of instructions encountered, part 1

What we find is that the general tendency stays the same:

- CBFS is good at finding a lot of system calls but they are less diversified. It executes less instructions than CDFS.
- CDFS is find less system calls that CBFS but they are more diversified. Generally, it is the one that executes the most instructions.
- **CSTOCH*** and **WSELECT*** strategies tend to be a good compromise between CDFS and CBFS.

What we notice, however, is that the gaps between the results are less important. This shows that for a shorter exploration time, the code coverage might be equivalent between the different exploration strategies.

Of course, seeing the same numbers does not mean that the same paths are taken and the same system calls are found with the different exploration strategies.

The quality of an SCDG is mostly defined by its contribution to better classification, since this is the final goal of SEMA's pipeline. In the next Section, we will evaluate the search strategies with the classifiers.

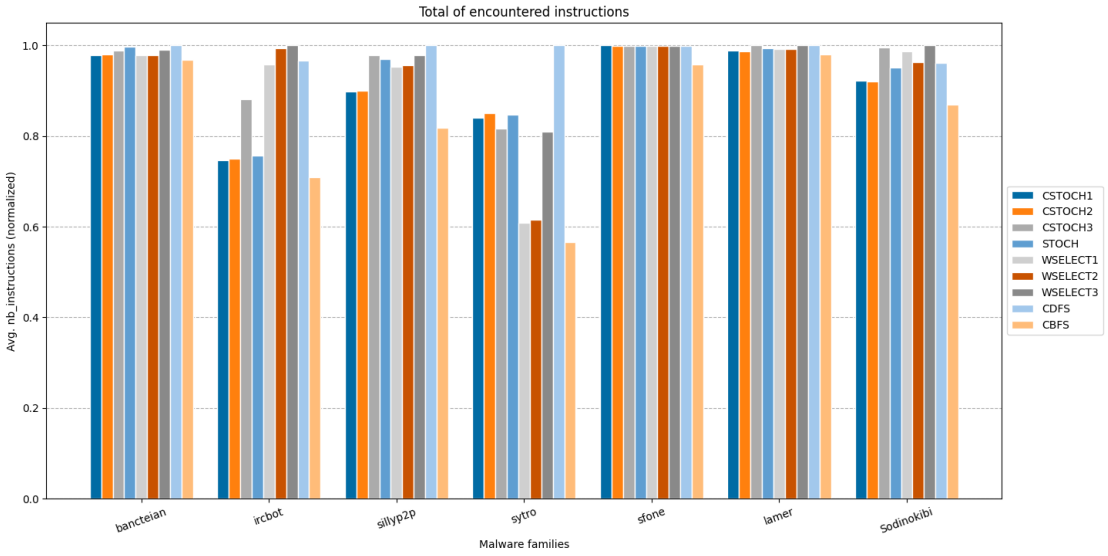

Figure 4.6: Number of instructions encountered, part 2

4.2 Graph neural networks evaluation

In this section we evaluate the performance of our graph neural networks on the SCDGs produced by the different exploration strategies. We use the SCDGs as input to the classifiers and we evaluate the results of the classification.

4.2.1 Experimental setup

The experiments are done on the Machine 1. We use the version `2.0.1+cu117` of `pytorch`.

While training and evaluating the graph neural networks models, the following steps were followed to ensure a robust and reliable model selection:

- Family Selection: Malware families with at least 36 samples for each method were chosen for evaluation, keeping 10 families. This criterion ensures an adequate representation of families in the dataset. The chosen malware families are: bancteian, ircbot, sillyp2p, sytro, simbot, RemcosRAT, delf, nitol, gandcrab and wabot.

- Data Splitting: For each selected family, 5 different splits were created, resulting in 5 pairs of train-test sets. Each split consisted of 26 samples for training and 10 samples for testing. Overlaps between the samples of the pairs are possible.

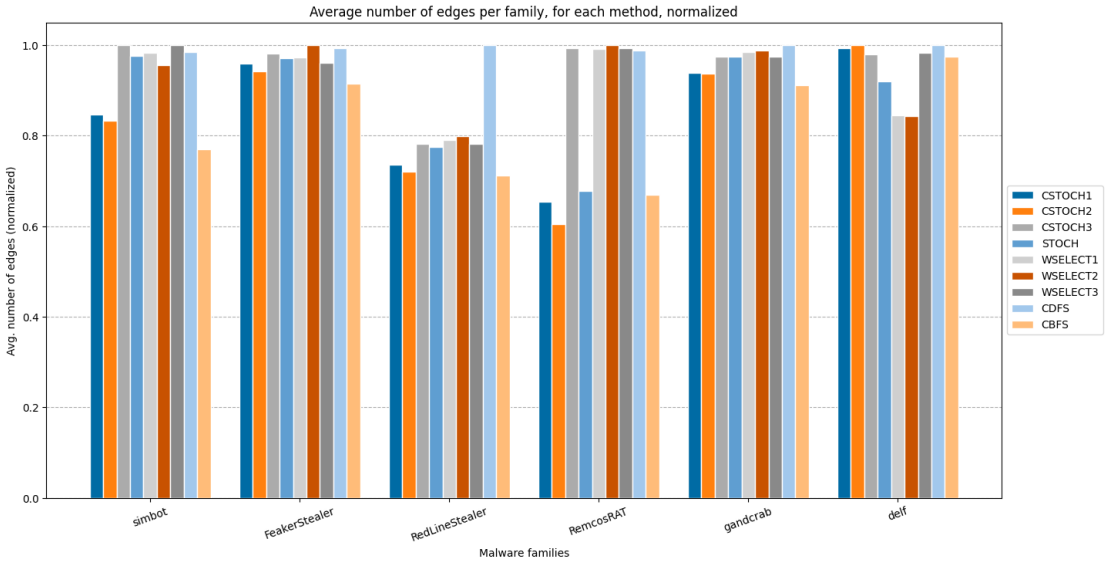

Figure 4.7: Average number of edges per family, for each method, normalized, part 1

- Train-Validation Split: During the training phase, the 26 samples were further divided into a training set and a validation set. The training set comprised 60% of the 26 samples, while the validation set accounted for the remaining 40%. This split allowed for model training and performance assessment during training.

- Model Training: Due to the small amount of data, we use a batch size of 1. The model was trained over 350 epochs using the Adam optimizer, CrossEntropyLoss criterion, and a CosineAnnealingLR scheduler for the learning rate [24]. This combination of optimization techniques helps in efficient parameter updates and loss minimization.

- Evaluation during Training: At each epoch, the model's performance was evaluated on the validation set. This step allowed for monitoring the model's progress and identifying the epoch with the highest balanced accuracy.

- Best Model Selection: In case of a tie in balanced accuracy, the model state with the lowest validation loss was chosen as the best model. This additional criterion ensures the selection of the model that exhibits better generalization and robustness.

By following this evaluation setup, the aim was to identify the best-performing

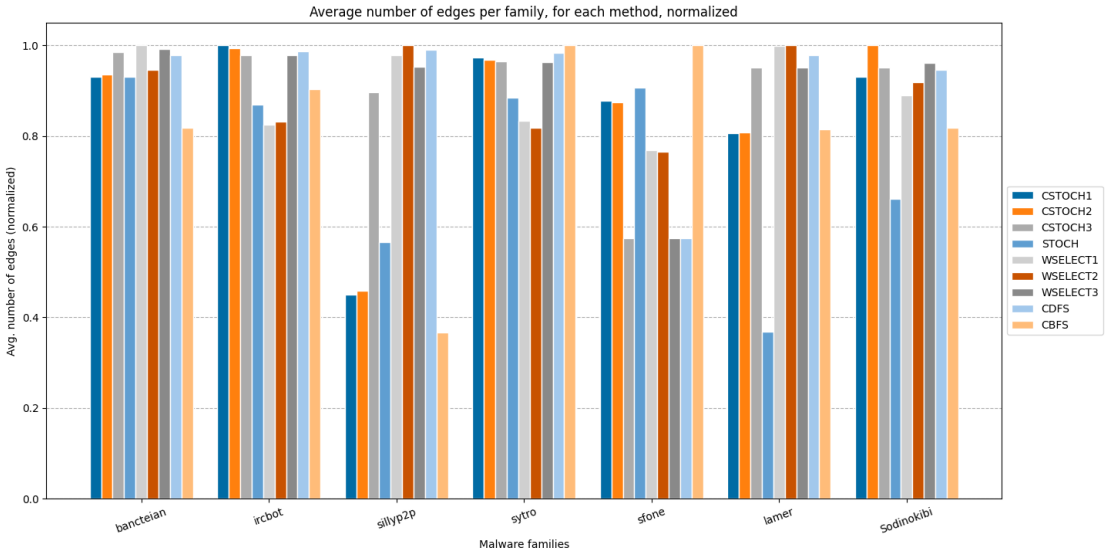

Figure 4.8: Average number of edges per family, for each method, normalized, part 2

model for malware classification, considering both the balanced accuracy and loss metrics.

In our experimental setup, we trained and tested two graph neural network models, GIN and GINJK, using different numbers of layers: 2, 3, 4, and 5. We also included the SVM classifier based on a graph kernel from SEMA [6] for comparison. The graph kernel model is using the 1-dimensional Weisfeiler-Lehman kernel, we will refer to this model as WL.

To ensure reliable results, we performed four evaluations for each model on each train-test dataset pair. The reported metrics are the average values obtained from these evaluations.

4.2.2 Results

The table 4.5 presents the average balanced accuracy of the different models, namely WL, GIN, and GINJK, on datasets generated with different exploration strategies. The graph neural networks models were evaluated using varying numbers of layers: 2, 3, 4, and 5.

The results indicate that for most exploration strategies, GINJK outperforms the other models in terms of balanced accuracy. Notably, GINJK with 4 layers combined with WSELECT3 achieves the highest accuracy across all strategies. In the Figure 4.9 we can see that GINJK is more consistent than GIN with a smaller

Table 4.4: Standard deviation of results, using STOCH and CSTOCH1-2-3 strategies

Family	`tot_uniq_syscalls`	`tot_syscalls`	`nb_instructions`
FeakerStealer	0.81	188.48	2427.23
RedLineStealer	0.39	152.60	39.89
RemcosRAT	0.25	23.68	2009.63
Sodinokibi	1.02	175.54	82.70
bancteian	0.15	65.24	19.42
delf	0.08	10.02	516.36
gandcrab	2.48	6675.93	5978.08
ircbot	0.61	353.36	744.32
lamer	0.14	119.84	18.14
nitol	1.63	124.75	12344.13
sfone	0.00	43.36	0.68
sillyp2p	0.36	34.65	32.53
simbot	0.07	673.54	1.95
sytro	0.25	5.55	2697.74
wabot	2.18	36.59	201.97

standard deviation across the different data splits and the multiple evaluations over these splits. The figure illustrates the comparison between both classifiers with 4 layers, but we find the same tendency with other numbers of layers (see Appendix Section A.2.1). This shows that taking into account the representation vectors of the previous layers has positive impact on the performance of the GNN. In contrast, WL consistently achieves lower accuracy compared to GIN and GINJK except with the CDFS exploration strategy. We do recognize, however, that the WL works pretty well with CDFS, it is a good and stable performance.

Our previous results showed that CDFS usually finds the most of the distinct system calls, still, it finds an important number of edges and executed instructions. WSELECT3 and CSTOCH3 on the other hand, even if they are close to CDFS, they usually find less distinct system calls and more overall system calls but still are close to CDFS in terms of edges and number of executed instructions. We conclude from this experience that the graph kernel based classifier WL will prefer highly connected graphs with more diversity in the nodes. As for the graph neural networks, they will prefer highly connected graphs with more focus on the nature of the system calls, with a little more redundancy. We note that the connectivity of the graph and the number of instructions found are important due to the low performance of CBFS.

Among the exploration strategies, WSELECT3 consistently yields high accuracy across all models, followed by CSTOCH3. These strategies guide the exploration

towards interesting system calls while also assigning importance to states that lead to unexplored addresses. We can conclude that these search heuristics show promise in improving the classification performance. We might achieve even better performance if we refine these strategies.

Search strategies that rely on randomness or do not prioritize exploring new paths, such as STOCH and CSTOCH1, exhibit relatively lower performance. This outcome aligns with the expectation that the classifiers require graphs that provide comprehensive insights into the program's behavior for better accuracy. The exploration strategy needs to find an important number of edges and execute an important number of instructions.

It is also interesting to note that while the number of layers has an impact on the performance of GIN and GINJK, the improvement is not necessarily monotonic. For example, GINJK with 4 layers achieves higher accuracy than with 5 layers in most cases.

Table 4.5: Average balanced accuracy of WL, GIN and GINJK on datasets generated with different exploration strategie

Model	WL	GIN				GINJK			
#layers / St.	-	2	3	4	5	2	3	4	5
CBFS	81.206	84.968	84.275	83.300	83.524	**85.531**	84.324	84.058	83.350
CDFS	**90.978**	88.996	87.111	86.339	85.867	89.680	88.661	88.089	87.806
CSTOCH1	74.361	77.889	**80.607**	79.569	76.756	80.467	78.696	78.160	77.118
CSTOCH2	77.756	79.698	79.822	82.361	79.189	81.716	81.111	**82.367**	79.039
CSTOCH3	85.244	90.998	**92.263**	90.747	89.793	90.822	91.611	91.653	91.089
STOCH	72.622	77.981	78.983	79.068	76.133	**79.300**	79.264	77.394	74.938
WSELECT1	89.200	88.698	89.640	88.922	89.311	88.844	89.822	**89.939**	89.761
WSELECT2	80.578	89.022	88.582	88.856	87.611	87.924	88.678	89.444	**89.500**
WSELECT3	85.628	92.483	91.319	91.818	88.349	92.592	92.569	**93.438**	92.221

It is important to acknowledge that the analysis conducted in this study was based on a limited amount of data, which may have impacted the scope of our results and evaluation. Generating graphs for all methods was time consuming and we were limited in terms of available time. Further evaluation on a larger and more diverse dataset is encouraged to gain a more comprehensive understanding. By expanding the scope of evaluation and focusing on the most interesting strategies, we can enhance the robustness and reliability of the findings.

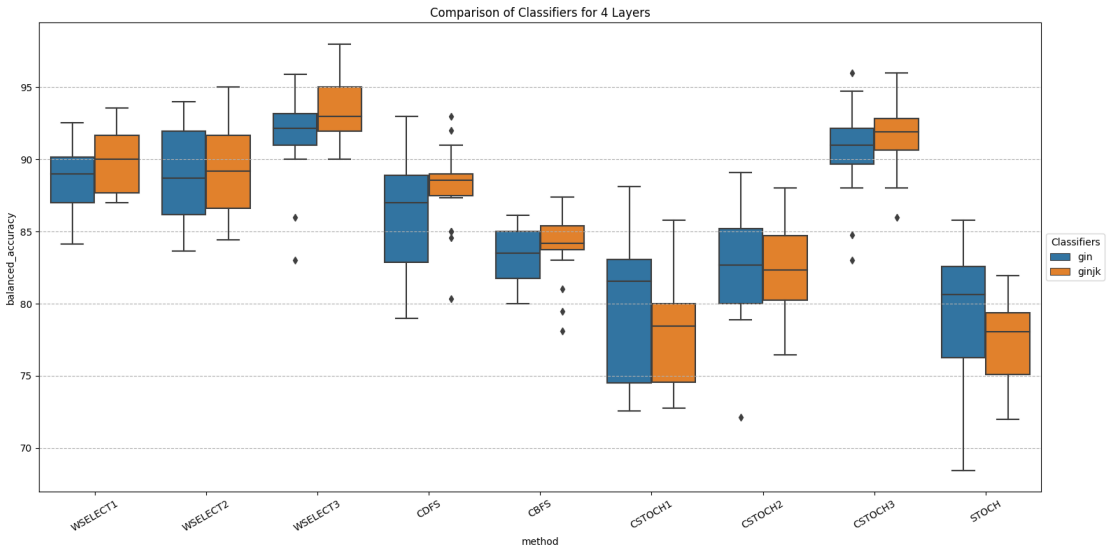

Figure 4.9: Performance of GIN and GINJK with 4 layers

4.3 GNN explainability

We implemented an explainability module with an explainer based on the GNNExplainer method [59]. We focus our explanation on the *phenomenon*, which means that want to explain why a *certain decision has been returned for a particular input* [2].

This explainer will detect a subgraph where the edge or node features have the greatest impact on the model's decision. We use it to identify the influential edges (dependencies) that contribute to the model's prediction.

In this section we will study a small example to illustrate how one can use this new module and interpret a prediction.

We use the GNN explainability framework supported by PyTorch geometric [32]. It offers a visualization function that shows the identified subgraph. It is called an explanation graph. The opacity of one edge corresponds to the edge importance. We slightly modified this visualization function to show the names of the system calls linked by the edges. Currently, the labels on the nodes of an explanation graph have the following form: *idx <n>; <syscall>*. *syscall* is the name of the system call, while n is the index (row or column depending on the direction of the link) in the adjacency matrix that represents the graph.

In our example, we use this module to compare predictions on graphs generated by CDFS and WSELECT3 and classified with our GIN model with 2 layers, trained on dataset of SCDGs created with the corresponding exploration strategy. We set

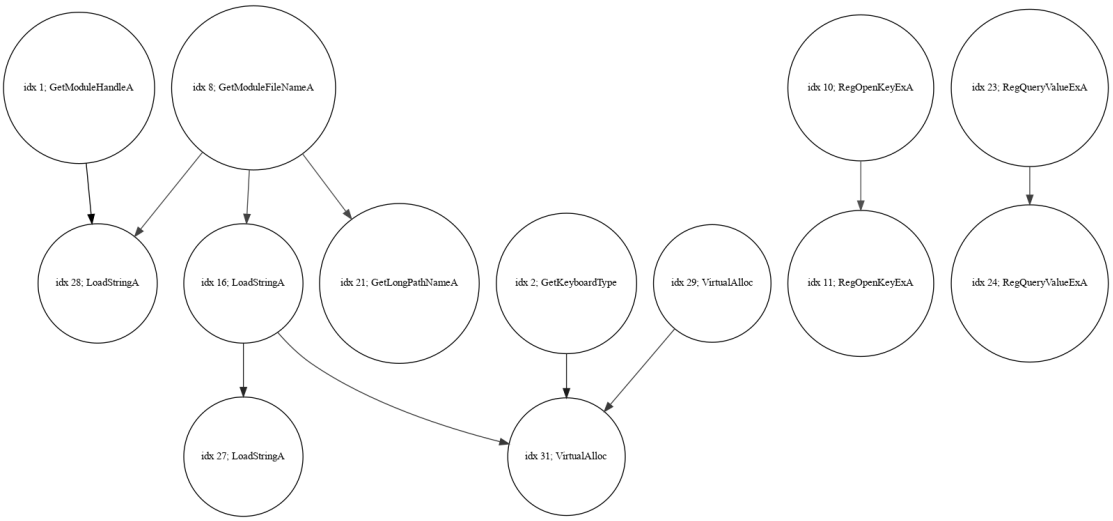

Figure 4.10: Explanation graph with the 10 most important edges to the classifier. SCDG generated by CDFS; RemcosRAT sample classified as delf sample

a parameter to show only the 10 most influential edges.

The GIN model with 2 layers performs better predictions on SCDGs generated with the WSELECT3 strategy than with the CDFS strategy. Looking closer, we saw that RemcosRAT is the worst classified family when using CDFS to create SCDGs. We have chosen one delf sample and one RemcosRAT sample for this analysis.

We run our explainer and take a look to the explanation graph (Fig. 4.10) for one RemcosRAT sample that was classified as a delf sample. The SCDG of this sample was created based on the CDFS exploration strategy. Since the edge opacity corresponds to edge importance, it is difficult with the resulting graph to narrow the set of most important edges with only 10 edges. There is no big difference between the opacity of the edges.

The next step is to look at the explanation graph (Fig 4.11) of one delf sample that was classified correctly as a delf sample. The SCDG of this sample was also created based on the CDFS exploration strategy. With this explanation graph we see that edges like: `GetModuleFileNameA` → `LoadStringA`, or `LoadStringA` → `LoadStringA` and `LoadStringA` → `VirtualAlloc` might influence the model to classify a sample in the delf family. If we look back to the previous the explanation graph for the RemcosRAT sample classified as delf, we can see the presence of these same links.

Now let us take a look to the explanation graph (Fig. 4.12) for the same RemcosRAT sample, but this time the SCDG was generated with WSELECT3

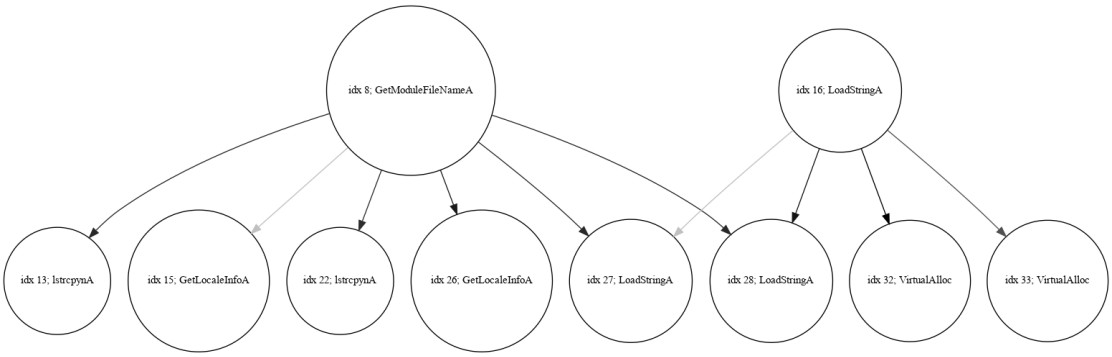

Figure 4.11: Explanation graph with the 10 most important edges to the classifier. SCDG generated by CDFS; delf sample classified as delf sample

search strategy. The classifier was also trained on samples created by the WSELECT3 method. This RemcosRAT sample was correctly classified in the right family. In this case we have a different set of important of edges. In particular, we no longer see the edge `GetModuleFileNameA` → `LoadStringA` in the top 10 edges. Instead, we see the following suite of dependencies: `GetLongPathNameA` → `RegQueryValueExA` → `RegCloseKey`, which probably means that the program manipulates some Windows registry.

Let's reproduce the operation with the delf sample. When the SCDG is created based on the WSELECT3 exploration strategy, this sample is still correctly classified. However, the explanation graph (Fig 4.13) shows a different set of important edges. We see that the `LoadStringA` → `LoadStringA` and `LoadStringA` → `VirtualAlloc` edges are less important. Instead, we see two new edges `GetLongPathNameA` → `RegQueryValueExA` and `FindFirstFileA` → `RegQueryValueExA`. This proves that the exploration strategy can have a deeper influence on the decision of the classifier. Also that our search heuristics based on the nature of the system calls might help to create SCDGs with more meaningful dependencies.

This small study shows that working with interpretable machine learning and such explainability frameworks can help analysts detect the presence of biases in the model. Even if the delf sample was correctly classified in both cases, it wasn't done for the same reasons. In the case of the SCDG created based on CDFS, the edges `LoadStringA` → `LoadStringA` and `LoadStringA` → `VirtualAlloc` seemed important for the delf family, which led a RemcosRAT sample to be classified as such.

In the context of SEMA Toolchain, it helps uncover areas that could be refined when building the SCDG. For example, when looking at the different explanation graphs for different samples, we saw that some edges like `VirtualAlloc` → `VirtualAlloc`

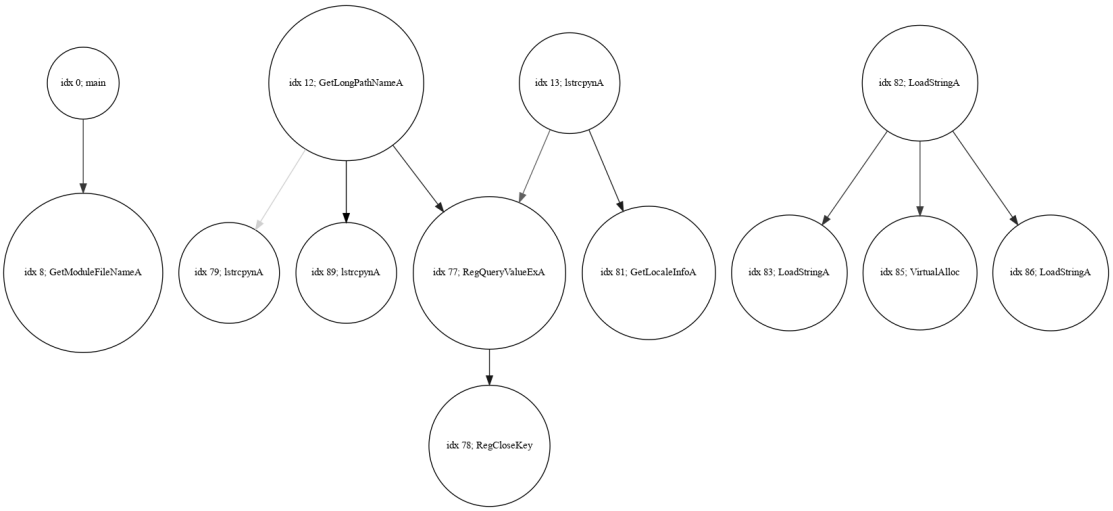

Figure 4.12: Explanation graph with the 10 most important edges to the classifier. SCDG generated by WSELECT3; RemcosRAT sample classified as RemcosRAT sample

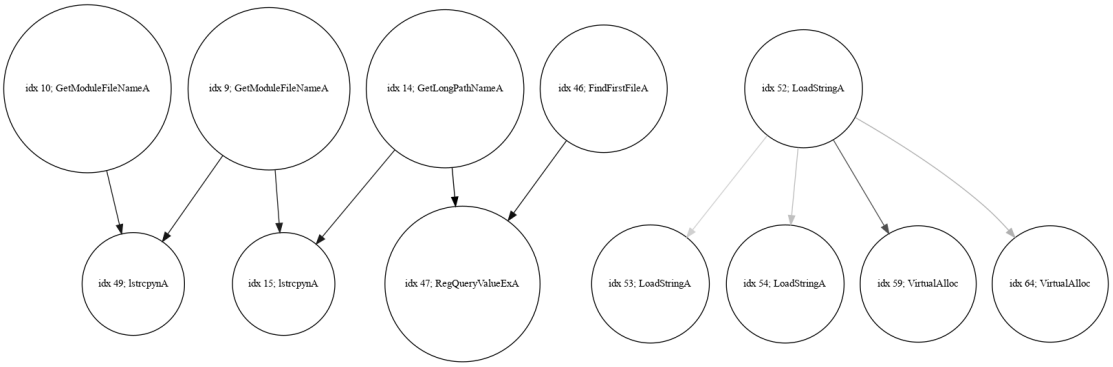

Figure 4.13: Explanation graph with the 10 most important edges to the classifier. SCDG generated by WSELECT3; delf sample classified as delf sample

can influence the decision of the model without necessarily providing information on the specific behavior of a malware family. We can refine the construction of the SCDGs by discarding dependencies that are frequently present in malware samples but do not provide concrete information about malware behavior. We can also address this aspect at the search strategy level. For example, we demonstrated that a search heuristic like WSELECT3, which is based on the nature of the system calls, avoided giving importance to the edge `VirtualAlloc` → `VirtualAlloc` when classifying the RemcosRAT sample, even though the `VirtualAlloc` call is included in the list of interesting system calls. This is in contrast to the CDFS strategy, which guides deeper exploration while prioritizing only new states.

However, it is important to note that the work in the GNN explainability research field is still in progress and that the accuracy of the different explainers still needs to be improved [2]. Nevertheless, interpretable machine learning can help us have a better understanding of our models. It is still important to be aware of the possible biases present in our models when working with machine learning tools.

Chapter 5

Future work

In this chapter, we discuss the potential future directions and opportunities that lie ahead, paving the way for continued progress.

Malware-cleanware detection One avenue to explore is the evaluation of the search strategies in terms of detection, specifically distinguishing between malware and cleanware, rather than solely focusing on classification between different families. This approach would provide valuable insights into the effectiveness and robustness of the methods in real-world scenarios.

We have reasons to believe that the search strategy might influence the detection. Since the search strategies guide the exploration towards the suspicious system calls, they might create a discriminating bias that increases false positives. On the other hand, might also help detecting malware that are very good at hiding their malicious behavior. We think it is an important experiment and a natural progression for our work.

While this evaluation was not conducted in the current study, it is important to highlight the reasons behind this decision. Factors such as the availability of cleanware samples, computational resources, and time constraints might have influenced the scope of the evaluation. By openly discussing these limitations, we can provide a transparent account of the study and emphasize the opportunities for future researchers to delve deeper into this aspect of evaluation.

Dynamic path pruning Our second suggestion for future work, is to work on reducing the computation workload due to checking the satisfiablity of predicates. The aim of the first part of our work was to optimize the performance of the symbolic executor, improving code coverage for a giving time. However, while doing our experiments, we were directly faced with the performance limitations of symbolic execution when running on huge files. A couple times, the virtual machines on which were running the experiments crashed due to insufficient RAM

to perform computation. This shows that we still need to work on optimization for the symbolic executor. We suggest implementing the Dynamic path pruning strategy [12] described in Section 2.2.1. This method can be easily coupled with per-layer strategies such as CBFS, hopefully allowing it to go deeper in exploration.

Improvements to existing strategies Our third suggestion is to slightly modify existing search strategies. Currently, CDFS and CBFS perform respectively the well known DFS and BFS search strategies but by prioritizing unknown state addresses. We could keep these strategies but add on top of it a prioritization of particular system calls. Unlike what is currently done with **CSTOCH*** or **WSELECT*** methods, we expect the exploration to be a bit more structured if it followed a DFS or BFS like exploration behavior. We might lose the trade-off offered by **CSTOCH*** or **WSELECT*** methods, by it might allow the exploration tree of CDFS to be wider and the exploration tree of CBFS to be deeper. Alternatively, we could simply modify **CSTOCH*** or **WSELECT*** to consider the minimum distance to an uncovered instruction in the evaluation function.

The next possible modification on existing strategies is to simply modify the list of interesting system calls [27]. We could use a more elaborate way of constructing and using this list. For example, in the scope of malware classification, we could use different lists based on the different families used for training. Further more, instead of giving the same importance to all system calls of this list, we could set a different weight to each call.

Graph neural networks architecture In our study we decided to implement a GIN architecture since it has been proven to be a powerful graph neural network architecture [50]. But there exist a lot of different graph neural network architectures [49], each one with its own advantages and drawbacks. Vignac et al. [44] present a few guidelines on how to choose and build a GNN architecture. Their study leads to the conclusion that most of the times simple architectures and structures perform better. However Valsesia et al. [42] critique their work by pointing a bias in the chosen dataset for their evaluation and proposes another approach in building the GNN layers. Instead of stacking layers to build deeper networks, they suggest to "randomly wire them" to obtain a richer representation of the graph. As a future extension to our work we can imagine implementing and evaluating a randomly wired GIN (or another architecture) as described in [42].

Federated graph neural networks An interesting possibility is the development of a federated graph neural network module within the SEMA toolchain, drawing inspiration from the research conducted in [19] and the existing federated learning module already implemented in SEMA [5].

Exploring this research avenue in the future presents a compelling opportunity to both enhance the capabilities of the SEMA toolchain and make significant contributions to the field of federated learning for malware analysis. By incorporating graph-related tasks into the framework of federated learning, we can push the boundaries of collaborative learning on graph-structured data while addressing critical challenges related to data privacy and limited resources. A federated learning approach can also compensate to the lack of available data for each party.

GNN explainability In this study, we have provided an initial implementation of an explainer for the GNN model. However, it is important to note that there is potential for further analysis and exploration in this area. Future investigations could involve comparing various strategies and classifiers, utilizing different explainer algorithms, and evaluating their accuracy specifically for the SEMA Toolchain. The scope of our analysis was limited by time constraints, but we encourage future research to dive deeper into this aspect and refine the findings.

Adversarial Examples Another interesting direction would be to establish adversarial attacks on graph neural networks [25] and evaluate the robustness of our models.

An evaluation framework During our experiment, we encountered a notable challenge related to the evaluation process. The generation of a sufficient number of SCDGs proved to be time-consuming, and we acknowledge that a larger dataset would have provided a more comprehensive evaluation. Additionally, assessing the quality of the SCDGs remains a complex task, as we lack clear guidelines on which types of SCDGs yield the best classifier accuracy and the conditions under which they are most effective.

As a potential future contribution, we propose an extension to the SEMA Toolchain that focuses specifically on benchmarking SEMA's performance. This extension would involve the automated and periodic downloading of malware samples, allowing for the generation of corresponding SCDGs using various methods, thus providing readily available data for evaluation purposes. Furthermore, we recognize the valuable insight proposed by Christophe Crochet, which involves establishing an objective function for SEMA. The idea is to define one or more metrics to assess the quality of a SCDG. These metrics would then serve as the basis for developing new and improved exploration strategies within the toolchain. Such an enhancement has the potential to greatly impact the effectiveness and efficiency of the SEMA Toolchain. The GNN explainability module can help as a starter point for evaluating and understanding the quality of SCDGs.

Chapter 6

Contribution and conclusion

In this master thesis, we showed that it is possible to improve code coverage in symbolic execution and face the path explosion problem by implementing new search strategies in SEMA toolchain. However, the path explosion problem remains an ongoing challenge that requires further investigation. Our new search strategies represent a trade-off between the behavior of the CDFS that goes to deep into the exploration process, but runs the risk of overlooking diverse behaviors and CBFS that attempts to explore all potential behaviors, with the risk of only scratching the surface and not fully capturing the entirety of a behavior.
We showed that we don't necessarily need to explore all execution paths to get a sense of a program behavior.

Our findings highlight the influence of randomness in guiding the symbolic executor towards new paths and how it helps escaping from unproductive states, such as endless loops. However, this comes at the trade-off of not consistently selecting the optimal path or making the most advantageous decisions.

We have also laid the groundwork for explainability of graph neural networks in SEMA Toolchain, opening up opportunities for future research and advancements in this area.

We revealed that the exploration strategies can influence the decision of the classifier, and thus we should think of which strategy to use when generating SCDGs for a classifier. With the GNN explainer, we've seen that our strategies focusing on the nature of system calls influence the classifier differently and can avoid possible biases.
We were able to show the potential of graph neural networks for malware analysis, working with system call dependency graphs. Using the GIN model with Jumping-

Knowledge and taking into account the representation vectors of previous layers demonstrated a positive impact on the classification.

We have integrated seven new search strategies to the SEMA toolchain: STOCH, CSTOCH1-2-3 and WSELECT1-2-3. We also added two new classifiers: GIN and GINJK. They can be directly used following the guidelines described in the tool paper of SEMA [23]. Moreover, we implemented a first GNN explainability module in the toolchain. We provide more details on running SEMA Toolchain with these methods in the Appendix section A.3.

Our study highlighted challenges and possible extensions for our work. Looking ahead, further research and contributions are encouraged to address the identified challenges and explore new avenues for refinement and enhancement.

Bibliography

[1] abuse.ch. *MalwareBazaar by abuse.ch, fighting malware and botnets.* https://bazaar.abuse.ch/.

[2] Kenza Amara et al. *GraphFramEx: Towards Systematic Evaluation of Explainability Methods for Graph Neural Networks.* 2022. arXiv: 2206.09677 [cs.LG].

[3] Thanassis Avgerinos et al. "Automatic Exploit Generation". In: *Commun. ACM* 57.2 (Feb. 2014), pp. 74–84. ISSN: 0001-0782. DOI: 10.1145/2560217.2560219. URL: https://doi.org/10.1145/2560217.2560219.

[4] Benjamin Sanchez-Lengeling, Emily Reif, Adam Pearce, Alexander B. Wiltschko. *A Gentle Introduction to Graph Neural Networks.* https://distill.pub/2021/gnn-intro/.

[5] Charles-Henry Bertrand Van Ouytsel, Khanh Huu The Dam, and Axel Legay. "Symbolic Analysis Meets Federated Learning to Enhance Malware Identifier". In: *Proceedings of the 17th International Conference on Availability, Reliability and Security.* ARES '22. Vienna, Austria: Association for Computing Machinery, 2022. ISBN: 9781450396707. DOI: 10.1145/3538969.3538996. URL: https://doi.org/10.1145/3538969.3538996.

[6] Charles-Henry Bertrand Van Ouytsel and Axel Legay. "Malware Analysis with Symbolic Execution and Graph Kernel". In: *NordSec'22: The 27th Nordic Conference on Secure IT Systems.* 2022.

[7] Fabrizio Biondi et al. "Effectiveness of Synthesis in Concolic Deobfuscation". working paper or preprint. Dec. 2015. URL: https://inria.hal.science/hal-01241356.

[8] Guillaume Bonfante, Matthieu Kaczmarek, and Jean-Yves Marion. "Architecture of a Morphological Malware Detector". In: *Journal in Computer Virology* 5.3 (2009), pp. 263–270. DOI: 10.1007/s11416-008-0102-4. URL: https://inria.hal.science/inria-00330022.

[9] Branka Vuleta. *44 Worrying Malware Statistics to Take Seriously in 2023.* https://legaljobs.io/blog/malware-statistics/.

[10] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs." In: *OSDI*. Vol. 8. 2008, pp. 209–224.

[11] Yu-Hung Chen, Jiann-Liang Chen, and Ren-Feng Deng. "Similarity-Based Malware Classification Using Graph Neural Networks". In: *Applied Sciences* 12.21 (2022). ISSN: 2076-3417. DOI: 10.3390/app122110837. URL: https://www.mdpi.com/2076-3417/12/21/10837.

[12] Ying-Shen Chen et al. "Dynamic Path Pruning in Symbolic Execution". In: *2018 IEEE Conference on Dependable and Secure Computing (DSC)* (2018), pp. 1–8.

[13] Zhenbang Chen et al. "Synthesize solving strategy for symbolic execution". In: *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2021, pp. 348–360.

[14] Khanh-Huu-The Dam, Thomas Given-Wilson, and Axel Legay. "Unsupervised behavioural mining and clustering for malware family identification". In: *Proceedings of the 36th Annual ACM Symposium on Applied Computing* (2021).

[15] Peter Dinges and Gul Agha. "Targeted Test Input Generation Using Symbolic-Concrete Backward Execution". In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. ASE '14. Vasteras, Sweden: Association for Computing Machinery, 2014, pp. 31–36. ISBN: 9781450330138. DOI: 10.1145/2642937.2642951. URL: https://doi.org/10.1145/2642937.2642951.

[16] Fabio Gritti. *Symbolic Execution For Bug Hunting in Binaries*. https://docs.google.com/presentation/d/1E3uE-4mYpenw0s40rtMbIdxj3fJgC79aHCeiIlJSY5Y/.

[17] Rafa Galvez, Veelasha Moonsamy, and Claudia Díaz. "Less is More: A privacy-respecting Android malware classifier using Federated Learning". In: *CoRR* abs/2007.08319 (2020). arXiv: 2007.08319. URL: https://arxiv.org/abs/2007.08319.

[18] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: Directed Automated Random Testing". In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '05. Chicago, IL, USA: Association for Computing Machinery, 2005, pp. 213–223. ISBN: 1595930566. DOI: 10.1145/1065010.1065036. URL: https://doi.org/10.1145/1065010.1065036.

[19] Chaoyang He et al. *FedGraphNN: A Federated Learning System and Benchmark for Graph Neural Networks*. 2021. arXiv: 2104.07145 [cs.LG].

[20] Jingxuan He et al. “Learning to explore paths for symbolic execution”. In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2021, pp. 2526–2540.

[21] J. Zico Kolter and Marcus A. Maloof. “Learning to Detect and Classify Malicious Executables in the Wild”. In: *J. Mach. Learn. Res.* 7 (Dec. 2006), pp. 2721–2744. ISSN: 1532-4435.

[22] Bertrand Van Ouytsel, C.H.; Crochet, C.; Dam, K.H.T.; Legay, A. *SEMA*. `https://github.com/csvl/SEMA-ToolChain`.

[23] Bertrand Van Ouytsel, Charles-Henry ; Crochet, Christophe ; Legay, Axel. “Tool paper - SEMA: Symbolic Execution toolchain for Malware Analysis”. In: *CRiSIS '22: Proceedings of the 17th International Conference on Risks and Security of Internet and Systems*. 2022.

[24] Leonie Monigatti. *A Visual Guide to Learning Rate Schedulers in PyTorch*. `https://towardsdatascience.com/a-visual-guide-to-learning-rate-schedulers-in-pytorch-24bbb262c863`.

[25] Jiaqi Ma, Junwei Deng, and Qiaozhu Mei. “Adversarial Attack on Graph Neural Networks as An Influence Maximization Problem”. In: *Proceedings of the Fifteenth ACM International Conference on Web Search and Data Mining*. WSDM '22. Virtual Event, AZ, USA: Association for Computing Machinery, 2022, pp. 675–685. ISBN: 9781450391320. DOI: `10.1145/3488560.3498497`. URL: `https://doi.org/10.1145/3488560.3498497`.

[26] Kin-Keung Ma et al. “Directed symbolic execution”. In: *Static Analysis: 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings 18*. Springer. 2011, pp. 95–111.

[27] Marco Ramilli. *Windows System Calls For Hunters*. `https://marcoramilli.com/2022/08/23/windows-system-calls-for-hunters/`.

[28] Microsoft. *SEAL*. `https://github.com/Microsoft/SEAL`.

[29] Andreas Moser, Christopher Kruegel, and Engin Kirda. “Limits of Static Analysis for Malware Detection”. In: *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. 2007, pp. 421–430. DOI: `10.1109/ACSAC.2007.21`.

[30] National Security Agency. *Ghidra*. `https://github.com/NationalSecurityAgency/ghidra`.

[31] Numpy. *numpy.random.choice*. `https://numpy.org/doc/stable/reference/random/generated/numpy.random.choice.html`.

[32] PyG. *GNN Explainability*. `https://pytorch-geometric.readthedocs.io/en/latest/tutorial/explain.html`.

[33] PyG. *PyTorch Geometric*. https://pytorch-geometric.readthedocs.io/en/latest/.

[34] David A Ramos and Dawson Engler. "Under-constrained symbolic execution: Correctness checking for real code". In: *24th {USENIX} Security Symposium ({USENIX} Security 15)*. 2015, pp. 49–64.

[35] Nicola Ruaro et al. "SyML: Guiding symbolic execution toward vulnerable states through pattern learning". In: *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses*. 2021, pp. 456–468.

[36] Najah Ben Said et al. "Detection of Mirai by Syntactic and Behavioral Analysis". In: *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)* (2018), pp. 224–235.

[37] Stefano Sebastio et al. "Optimizing symbolic execution for malware behavior classification". In: *Comput. Secur.* 93 (2020), p. 101775.

[38] Koushik Sen, Darko Marinov, and Gul Agha. "CUTE: A Concolic Unit Testing Engine for C". In: *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ESEC/FSE-13. Lisbon, Portugal: Association for Computing Machinery, 2005, pp. 263–272. ISBN: 1595930140. DOI: 10.1145/1081706.1081750. URL: https://doi.org/10.1145/1081706.1081750.

[39] Yan Shoshitaishvili et al. "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis". In: (2016).

[40] Statista. *Amount of monetary damage caused by reported cyber crime to the IC3 from 2001 to 2022*. https://www.statista.com/statistics/267132/total-damage-caused-by-by-cyber-crime-in-the-us/.

[41] David Trabish et al. "Chopped symbolic execution". In: *Proceedings of the 40th International Conference on Software Engineering*. 2018, pp. 350–360.

[42] Diego Valsesia. "DON'T STACK LAYERS IN GRAPH NEURAL NETWORKS, WIRE THEM RANDOMLY". In: 2021.

[43] Charles-Henry Bertrand Van Ouytsel and Axel Legay. "Detection and classification of malware based on symbolic execution and machine learning methods". In: (2021).

[44] Clément Vignac, Guillermo Ortiz-Jiménez, and Pascal Frossard. "On The Choice of Graph Neural Network Architectures". In: *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2020, pp. 8489–8493. DOI: 10.1109/ICASSP40776.2020.9054357.

[45] VirusTotal. *Yara*. http://virustotal.github.io/yara/.

[46] Zhi Wang et al. "Linear Obfuscation to Combat Symbolic Execution". In: *Proceedings of the 16th European Conference on Research in Computer Security*. ESORICS'11. Leuven, Belgium: Springer-Verlag, 2011, pp. 210–226. ISBN: 9783642238215.

[47] Boris Weisfeiler and A. A. Lehman. "A Reduction of a Graph to a Canonical Form and an Algebra Arising During This Reduction". In: *Nauchno-Technicheskaya Informatsia* Ser. 2.N9 (1968), pp. 12–16.

[48] Bolun Wu, Yuanhang Xu, and Futai Zou. "Malware Classification by Learning Semantic and Structural Features of Control Flow Graphs". In: *2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. 2021, pp. 540–547. DOI: 10.1109/TrustCom53373.2021.00084.

[49] Zonghan Wu et al. "A Comprehensive Survey on Graph Neural Networks". In: *IEEE Transactions on Neural Networks and Learning Systems* 32.1 (Jan. 2021), pp. 4–24. DOI: 10.1109/tnnls.2020.2978386. URL: https://doi.org/10.1109%2Ftnnls.2020.2978386.

[50] Keyulu Xu et al. "How Powerful are Graph Neural Networks?" In: *CoRR* abs/1810.00826 (2018). arXiv: 1810.00826. URL: http://arxiv.org/abs/1810.00826.

[51] Keyulu Xu et al. *Representation Learning on Graphs with Jumping Knowledge Networks*. 2018. arXiv: 1806.03536 [cs.LG].

[52] Peng Xu, Claudia Eckert, and Apostolis Zarras. "Detecting and Categorizing Android Malware with Graph Neural Networks". In: *Proceedings of the 36th Annual ACM Symposium on Applied Computing*. SAC '21. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 409–412. ISBN: 9781450381048. DOI: 10.1145/3412841.3442080. URL: https://doi.org/10.1145/3412841.3442080.

[53] Xiaojun Xu et al. "Neural Network-based Graph Embedding for CrossPlatform Binary Code Similarity Detection". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Oct. 2017. DOI: 10.1145/3133956.3134018. URL: https://doi.org/10.1145%5C%2F3133956.3134018.

[54] Guanhua Yan, Nathan Brown, and Deguang Kong. "Exploring Discriminatory Features for Automated Malware Classification". In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Ed. by Konrad Rieck, Patrick Stewin, and Jean-Pierre Seifert. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 41–61. ISBN: 978-3-642-39235-1.

[55] Jiaqi Yan, Guanhua Yan, and Dong Jin. "Classifying Malware Represented as Control Flow Graphs using Deep Graph Convolutional Neural Network". In: *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2019, pp. 52–63. DOI: 10.1109/DSN.2019.00020.

[56] Xifeng Yan and Jiawei Han. "gSpan: graph-based substructure pattern mining". In: *2002 IEEE International Conference on Data Mining, 2002. Proceedings.* 2002, pp. 721–724. DOI: 10.1109/ICDM.2002.1184038.

[57] Qiuping Yi et al. "Eliminating path redundancy via postconditioned symbolic execution". In: *IEEE Transactions on Software Engineering* 44.1 (2017), pp. 25–43.

[58] Qiuping Yi et al. "Postconditioned symbolic execution". In: *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE. 2015, pp. 1–10.

[59] Rex Ying et al. *GNNExplainer: Generating Explanations for Graph Neural Networks*. 2019. arXiv: 1903.03894 [cs.LG].

[60] You, Ilsun and Yim, Kangbin. "Malware obfuscation techniques: A brief survey". In: *2010 International conference on broadband, wireless computing, communication and applications*. IEEE. 2010, pp. 297–300.

[61] Hao Yuan et al. *Explainability in Graph Neural Networks: A Taxonomic Survey*. 2022. arXiv: 2012.15445 [cs.LG].

Appendices

Appendix A

Appendix

A.1 Exploration strategies evaluation

A.1.1 The impact of timeout

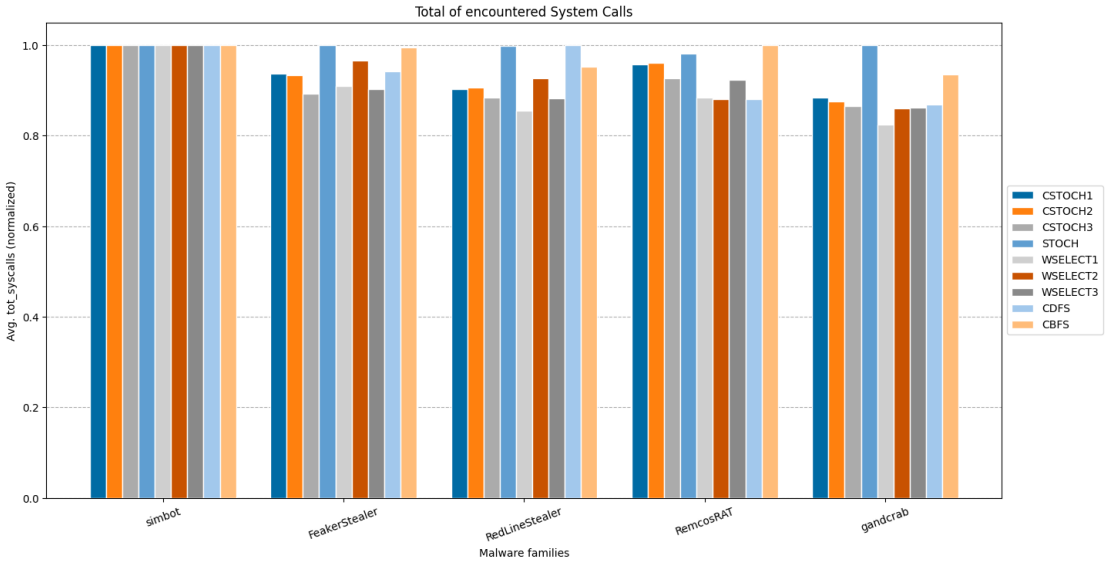

Figure A.1: Number of system calls, timeout of 1min, part 1

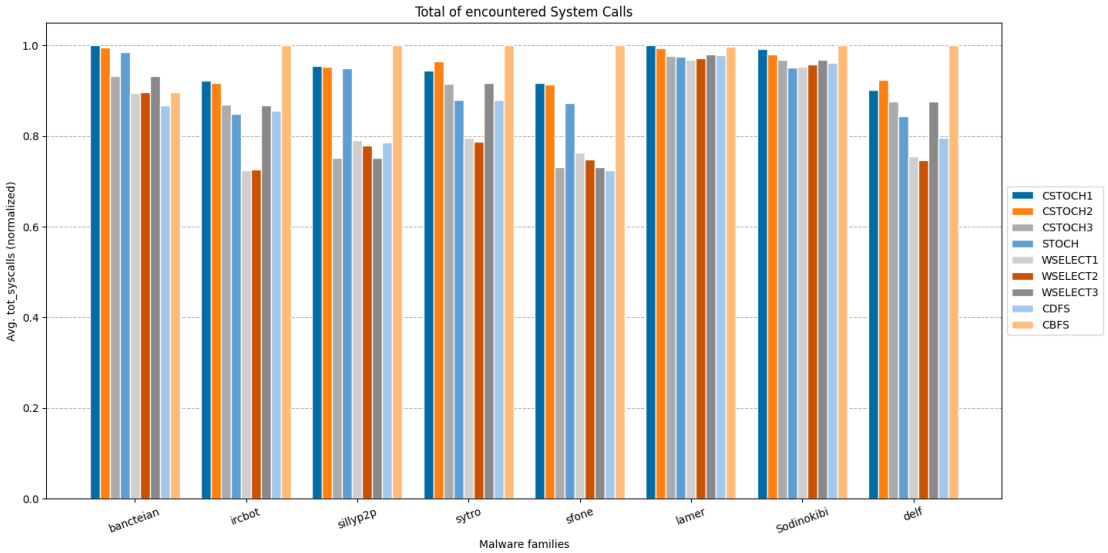

Figure A.2: Number of system calls, timeout of 1min, part 2

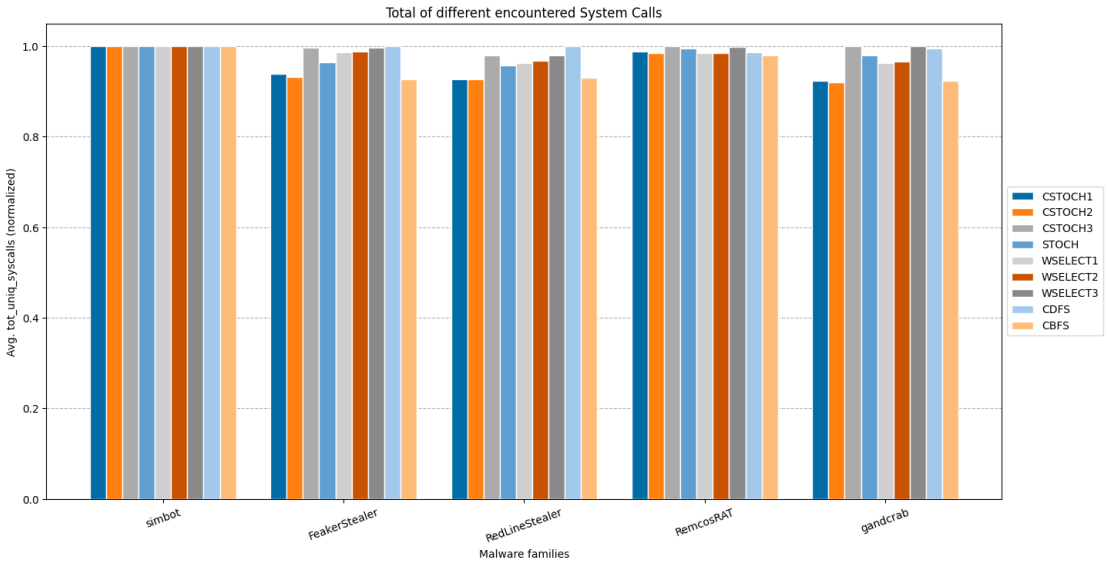

Figure A.3: Number of distinct system calls, timeout of 1min, part 1

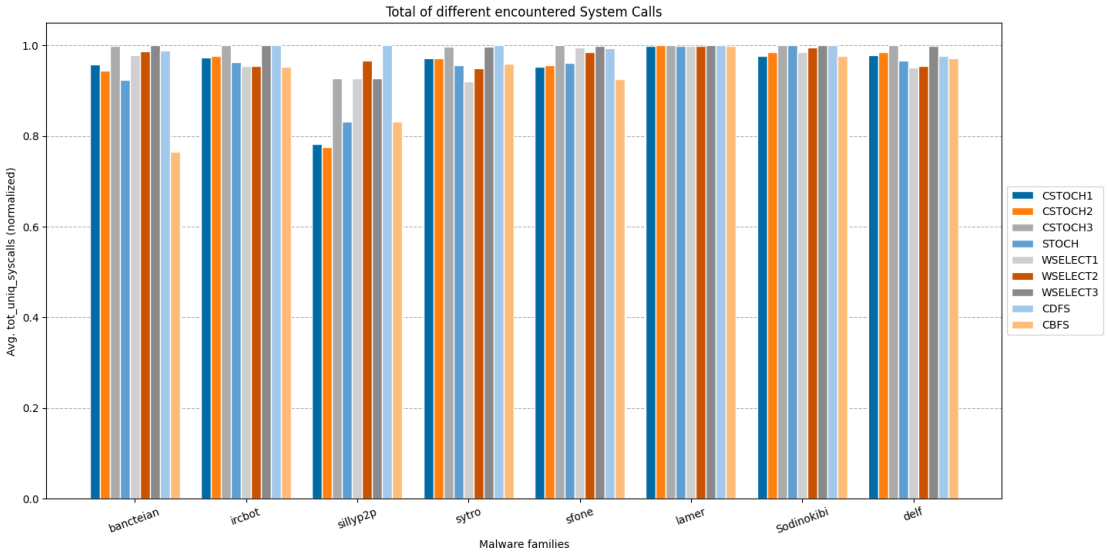

Figure A.4: Number of distinct system calls, timeout of 1min, part 2

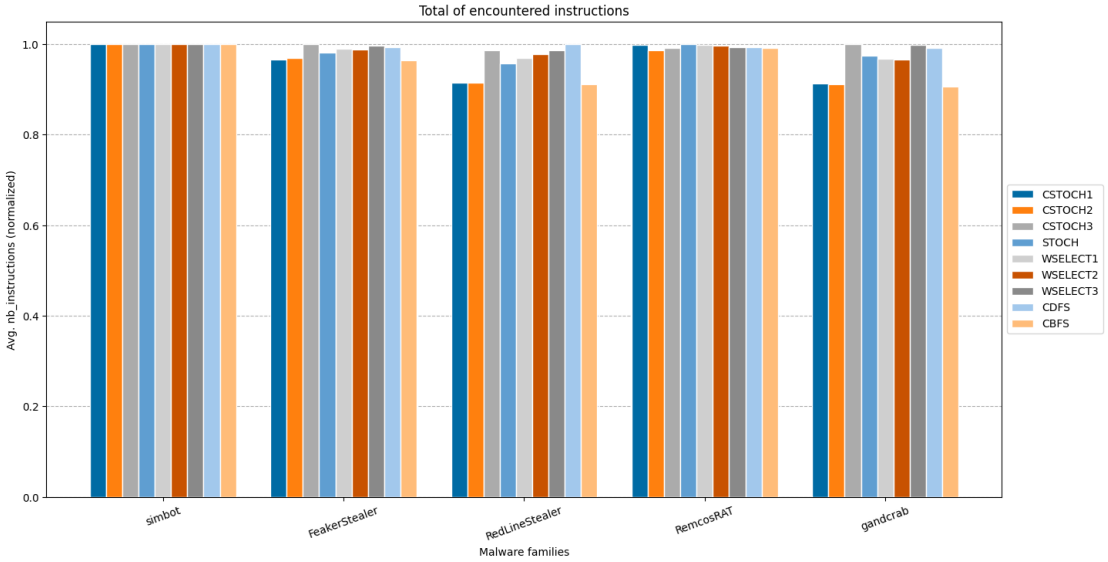

Figure A.5: Number of instructions encountered, timeout of 1min, part 1

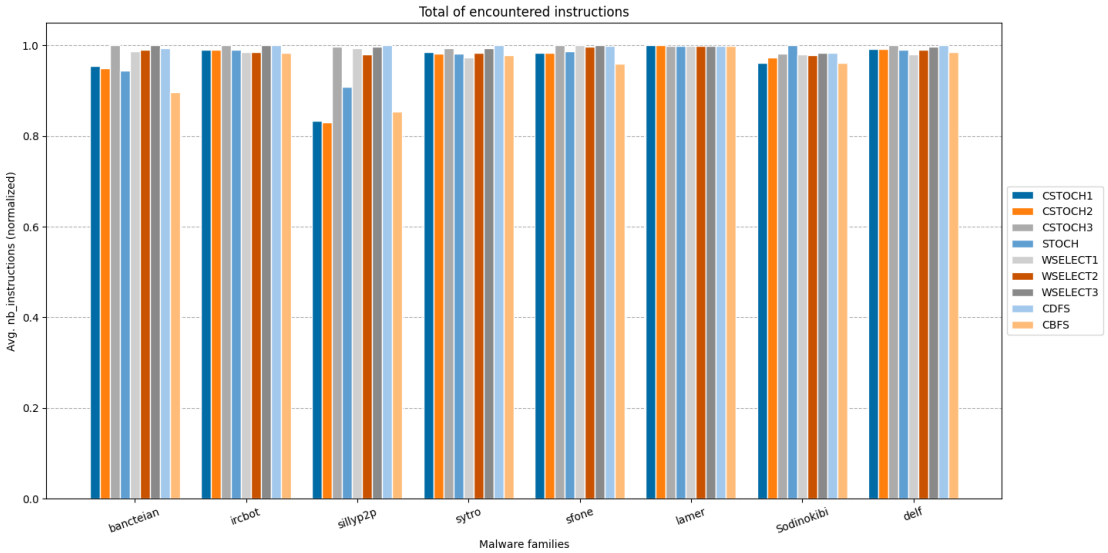

Figure A.6: Number of instructions encountered, timeout of 1min, part 2

A.2 Machine learning evaluation

A.2.1 GIN vs GINJK

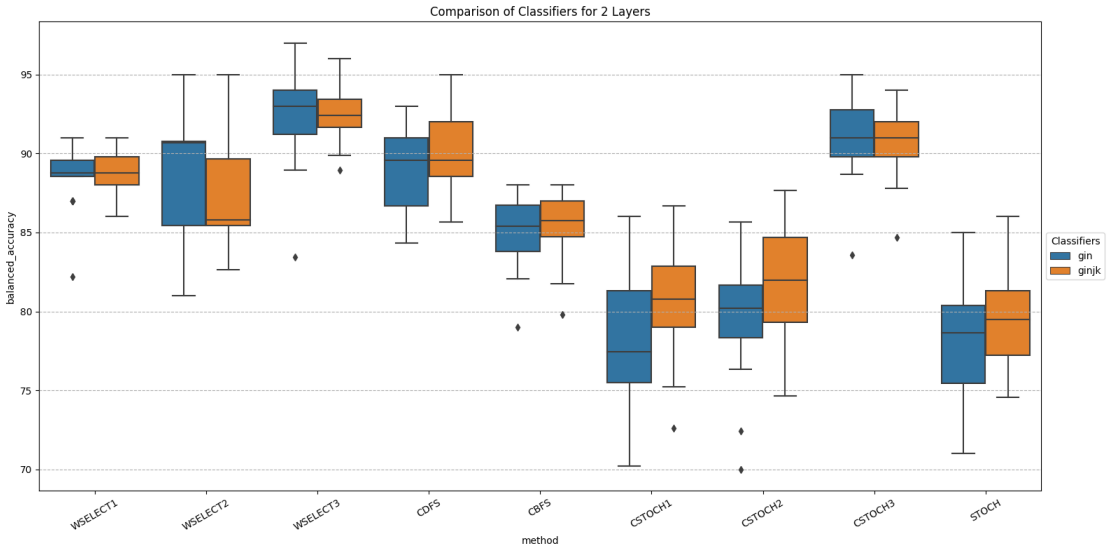

Figure A.7: Performance of GIN and GINJK with 2 layers

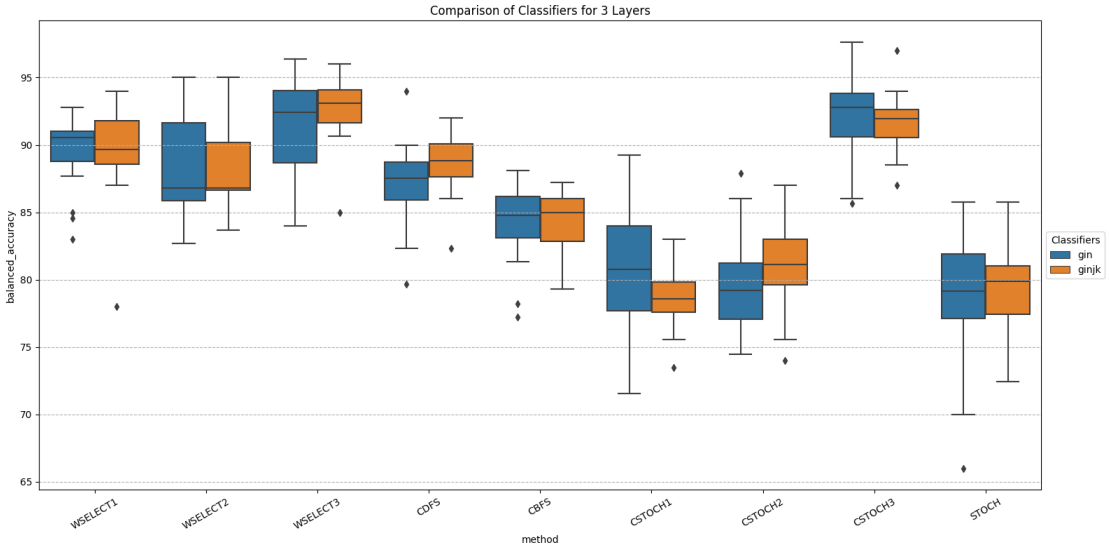

Figure A.8: Performance of GIN and GINJK with 3 layers

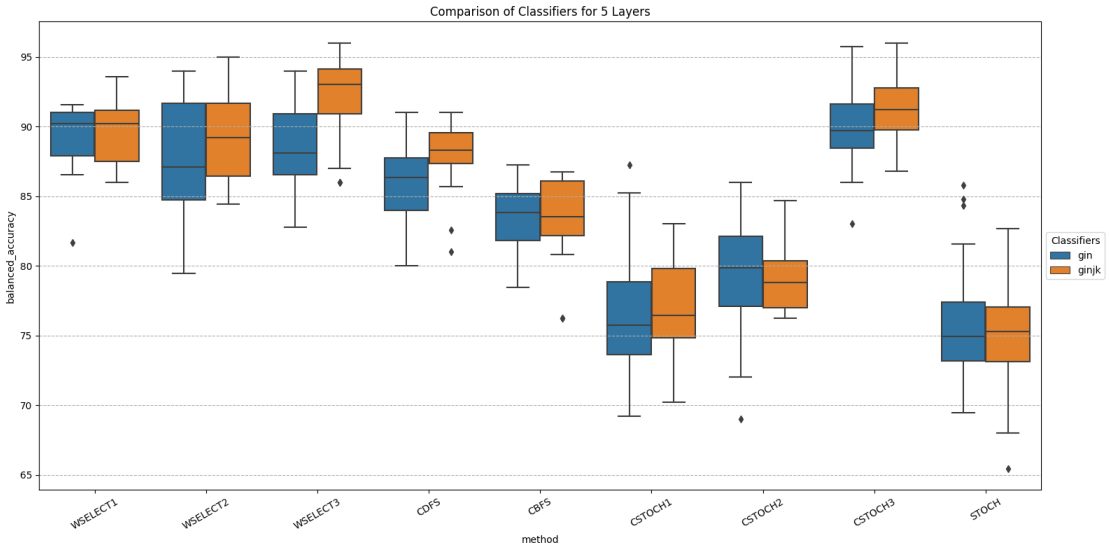

Figure A.9: Performance of GIN and GINJK with 5 layers

A.2.2 Comparison of WL, GIN and GINJK with different layers and different metrics

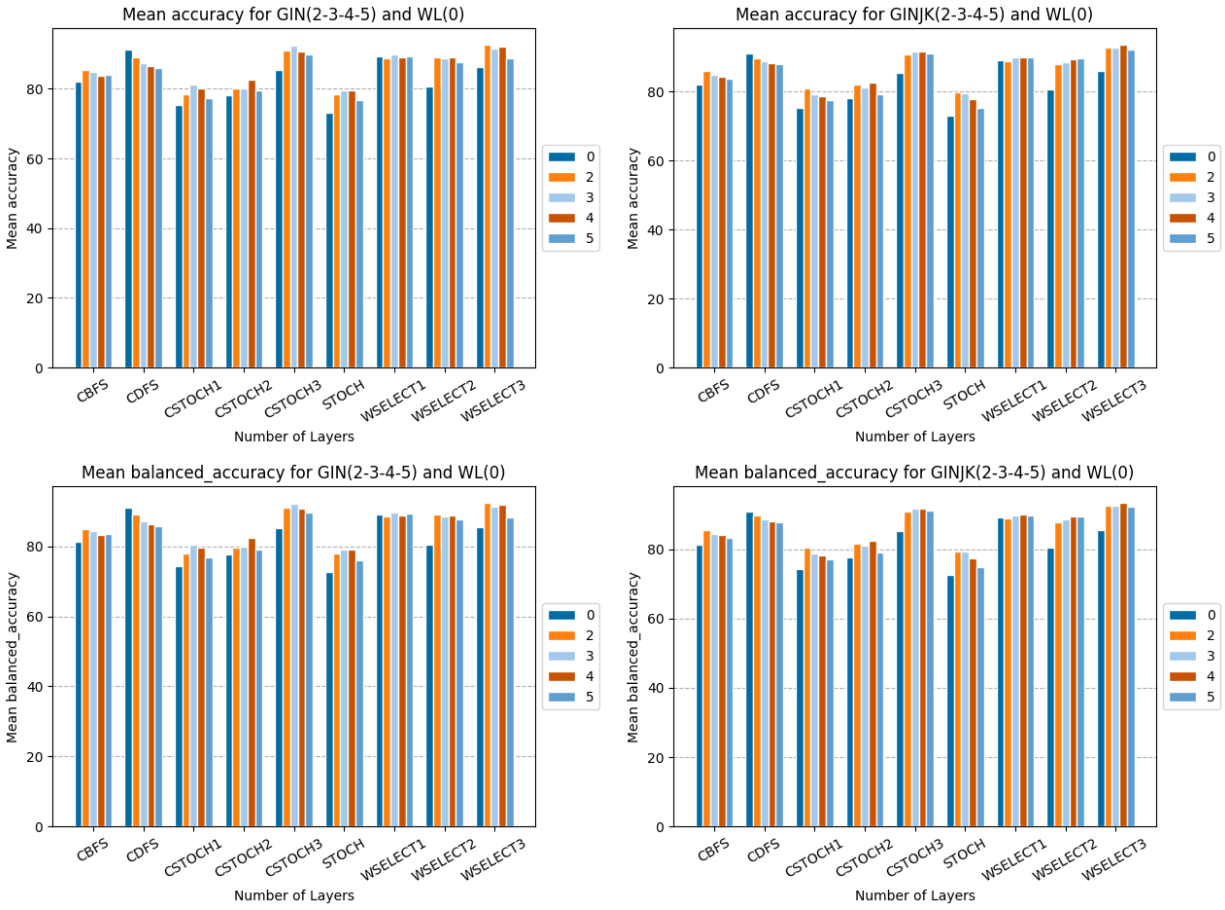

Figure A.10: Mean performance of GIN, GINJK and WL with different layers, part 1

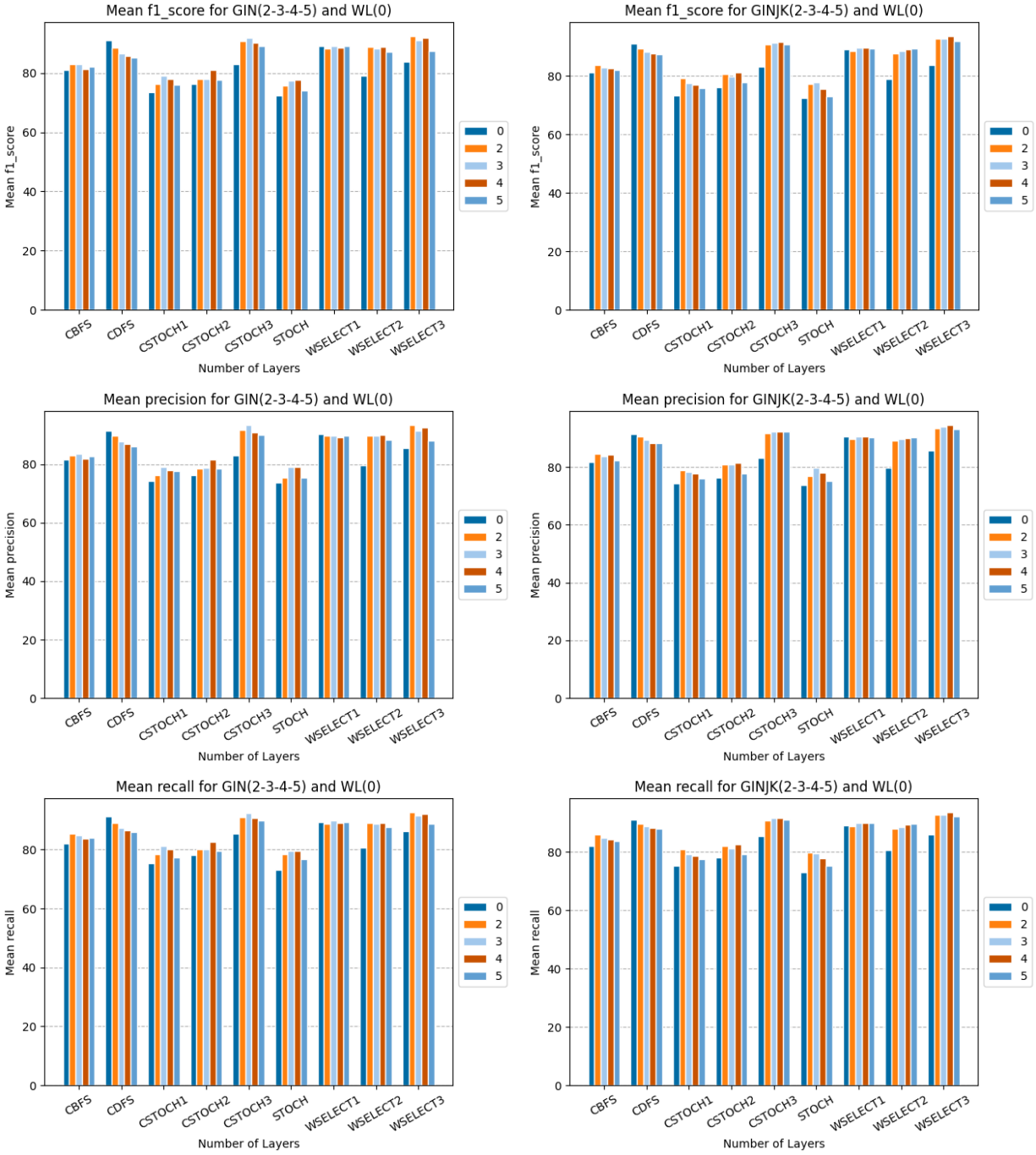

Figure A.11: Mean performance of GIN, GINJK and WL with different layers, part 2

A.3 Integration to SEMA Toolchain

At the time of writing this master thesis, our fork is still not on the production branch of SEMA Toolchain [22]. However it is available on our public fork of the toolchain at `https://github.com/samybtt/SEMA-ToolChain`. You'll need to clone the repository and run the next commands being in `SEMA-ToolChain/src`.

SCDG generation To create a SCDG with particular search heuristic, you can do it in the command line with the following command:
`python3 ToolChainSCDG/ToolChainSCDG.py --method=`*method* `<binary>`
where *method* ∈ `[DFS, BFS, CDFS, CBFS, STOCH, CSTOCH1, CSTOCH2, CSTOCH3, WSELECT1, WSELECT2, WSELECT3]`

Classification To train the model you can use the following command:
`python3 ToolChainClassifier/ToolChainClassifier.py -train --num_layers=`n `--classifier=`clf `<training dataset>`
where n is the number of layers, only useful for graph neural networks. $clf \in$ `[gspan, wl, dl, gin, ginjk]`.
For `gin` and `ginjk`, you can specify the number of layers with the option `--num_layers` (e.g. `--num_layers=4`).

To test the model:
`python3 ToolChainClassifier/ToolChainClassifier.py --classifier=`clf `<testing dataset>`

GNN explainability To use the explainer, you can execute the following command:
`python3 ToolChainClassifier/ToolChainClassifier.py --explain --classifier=`clf `<dataset>`
where $clf \in$ `[gin, ginjk]`.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**