Analyzing binary programs and obfuscation with graph-based representations and machine learning

Roxane Cohen

PSL UNIVERSITÉ PARIS

Quarkslab
Securing every bit of your data

THÈSE DE DOCTORAT
DE L'UNIVERSITÉ PSL

Préparée à Université Paris-Dauphine
Dans le cadre d'une CIFRE avec Quarkslab

Analyzing binary programs and obfuscation with graph-based representations and machine learning

Soutenue par
Roxane COHEN
Le 27/11/2025

École doctorale n°543
Ecole Doctorale SDOSE

Spécialité
Informatique

Composition du jury :

Yann CHEVALEYRE Professeur, Paris-Dauphine-PSL	*Président*
Sébastien ADAM Professeur, Université de Rouen	*Rapporteur*
Valérie VIET TRIEM TONG Professeure, Centrale Supélec	*Rapporteur*
Sébastien BARDIN Chercheur, Commissariat à l'Energie Atomique	*Examinateur*
Robin DAVID Chercheur, Quarkslab	*Enncadrant industriel*
Barbara PILASTRE Chercheuse, Direction Générale de l'Armement	*Examinatrice*
Florian YGER Maître de Conférences, INSA-Rouen Normandy	*Co-encadrant*
Fabrice ROSSI Professeur, Paris-Dauphine-PSL	*Directeur de thèse*

Dauphine | PSL UNIVERSITÉ PARIS

Remerciements

Une thèse est un travail de longue haleine, et celle-ci n'aurait pas été possible sans le soutien sans faille de plusieurs personnes que je tiens à remercier.

Tout d'abord, merci à Robin David, mon encadrant au sein de Quarkslab, pour sa patience, ses conseils et sa disponibilité. Depuis le premier jour où il m'a expliqué ce qu'était une instruction assembleur jusqu'aux derniers moments de cette thèse, il a toujours su m'écouter et me guider, avec bienveillance et bonne humeur, même dans les moments les plus difficiles. Je ne peux imaginer meilleur encadrant.

Merci également à Florian Yger et Fabrice Rossi, mes deux directeurs de thèse, pour m'avoir fait confiance depuis ce stage improvisé en M1 et tout au long de cette thèse, pour leurs conseils et leurs temps.

Je remercie Yann Chevaleyre et Sébastien Bardin pour avoir accepté de faire partie de mon jury de thèse, et tout particulièrement Sébastien Adam et Valérie Viet Triem Tong pour leurs remarques et retours précieux en tant que rapporteurs. Je tiens également à remercier spécifiquement Barbara Pilastre qui, au-delà de son rôle de membre du jury, n'a cessé de me conseiller et de m'accompagner, depuis mon stage au sein de la DGA et en tant que tutrice AID, que ce soit à Rennes ou à Paris.

Au sein de Quarkslab, certains de mes collègues auront été d'une aide et d'un soutien précieux, et je tiens particulièrement à les en remercier. First of all, my thanks go to Riccardo Mori, who never spared his time or his advice in helping me see things more clearly. How many hours did we spend grumbling about BinDiff and trying to figure out why QbinDiff couldn't do any better, or complaining about IDA! Un grand merci également à Sami, mon voisin de bureau, toujours présent pour le soutien moral ou pour répondre à mes questions sur bash, python ou les regex et avec qui les après-midis laborieux passaient avec force ricanements au fond de l'open-space, avec Julio, mon second voisin et complice de bureau, que je remercie tout autant. Merci à Samuel, pour ses conseils avisés sur l'apprentissage par renforcement, sa connaissance du Pcode et sa patience à jouer au canard. Merci également à Jib, pour qui Windows n'a pas de secrets, pour son aide précieuse sur la compilation de malwares et sur l'utilisation de clang dans VisualStudio.

Ces trois ans de thèse auront été parfois longs et denses, mais toujours incroyablement enrichissants et passionnants. Cela, je le dois à mes collègues de Quarkslab, auprès de qui j'ai énormément appris et ri. Je n'aurais pas pu rêver mieux comme lieu où effectuer

ma thèse et mon quotidien à HSM n'aurait pas été le même sans ses membres, passés et présents : Madi, avec qui nos discussions, sur la littérature, le cinéma ou la gastronomie, embêtait tout l'open-space (quand il sortait encore de chez lui), Lo, toujours prompt à me prêter une oreille attentive et à s'assurer que tout allait bien, Ramtine, pour sa gentillesse absolue, Raphaël et son obsession pour les sorties à la boucherie (que je partage entièrement), Benoît, Vincent, Alexandre, Julie, Tom, Loïc, Idann, Mickaël, Jérémy, Célian, Axel, Damiano, Salwa, Kevin et enfin Mathieu Robert, dont l'empreinte laissée sur le Lab en a fait un endroit rêvé pour cette thèse. Un endroit avec ses rituels bien ancrés, entre les expéditions boucherie à 11h45 précises, les sorties cookies, crêpes ou flan portugais (désolée Vincent), ou encore les goûters improvisés dans la cuisine, avec des cannelés ramenés par Renard, les jeux de société ou de Nerf, les apéros à l'inénarrable Frog ou les barbecues chez Robin ou Benoît. Pour tout cela, merci.

Au-delà de Quarkslab, nombre de personnes au sein de l'université ont été d'une aide précieuse. Merci à Marie-Clothilde Quinio et Christian Tresfield pour leur gestion impeccable de tous les aspects administratifs liés à la thèse, à Hubert Radenac pour son aide lors des processus de l'école doctorale et à Olivier Rouyer, pour sa réactivité et sa patience face à mes nombreuses demandes liées aux serveurs de la fac.

Merci également à Aldo et Jason qui ont effectué leurs thèses en même temps que moi et se sont révélés de très bons conseils, notamment lors d'une école d'été à Rouen.

Une thèse ne se limite pas au bureau ou à l'université ; et mes amis et ma famille ont toujours été là pour me soutenir sans relâche. Merci à mes amis du collège et du lycée, pour leur soutien depuis la France et ailleurs, et qui me redemandent à chaque fois le sujet de ma thèse, déterminés à *vraiment* comprendre l'intérêt des graphes. Merci à la Crraim, qui m'a écoutée pester et a pesté avec moi à tous les coups durs et qui n'a cessé de m'encourager.

Un immense merci à mes parents et à mes grands-parents. Les premiers, pour leur présence réconfortante et leur écoute attentive, qui ouvrent de grands yeux lorsque je leur parle de mes travaux mais qui m'ont prodigué le meilleur des conseils, à savoir que "quand tout va mal, il faut être méchant, méchant sans pitié". Les seconds, qui m'ont toujours dit de garder la pêche et m'ont encouragée à aller jusqu'au bout de mes études, pour moi-même. Du plus profond du coeur, merci à ma soeur, pour son soutien indéfectible, qui a fulminé avec moi face aux (nombreux) refus de papiers, a célébré avec moi ceux qui étaient acceptés, et qui a toujours su me redonner le sourire, peu importe les circonstances. J'espère pouvoir être pour toi un soutien aussi constant que tu l'as été pour moi, pendant ta propre thèse.

Enfin, merci à Dahmun, sans qui cette thèse n'aurait sans doute jamais vu le jour, pour avoir fait de ces trois ans une aventure fabuleuse, tant sur le plan professionnel que personnel. Ce n'est que le début. Merci, pour tout.

Contents

List of Figures

List of Tables

Acronyms

AST Abstract Syntax Tree. 39, 40

BB Basic Block. 8, 35, 36, 42, 43, 46, 47, 52, 54, 61, 63, 64, 67, 68, 82–84, 87–91, 113, 114, 116, 119, 125, 127, 150, 152, 154

CDG Control Data Graph. 8, 150–162, 166, 169

CFF Control Flow Graph Flattening. 7, 8, 24, 43, 45–48, 66, 73, 76, 81, 83, 101, 109, 110, 127, 129, 142, 148

CFG Control Flow Graph. 4, 7, 8, 11, 12, 36–40, 42–45, 52, 54, 61, 66, 68, 72, 73, 83–86, 88–107, 113, 116, 119, 123, 127, 134, 148, 150

CG Call Graph. 7, 38, 39, 43, 45, 52, 62–64, 73, 113, 115–119, 123, 127, 132–134, 137, 143, 150

CNN Convolutional Neural Network. 55, 57

CPG Code Property Graph. 40

CPU Central Processing Unit. 23, 29, 31, 35, 90, 122, 185

DFG Data Flow Graph. 7, 38–40

DL Deep Learning. 26, 51–54, 56, 60, 68, 72, 147

DRM Digital Rights Management. 68

FN False Negative. 113

FP False Positive. 113

GAT Graph Attention Network. 56, 83, 88, 94, 98

GCN Graph Convolutional Network. 55, 56, 68, 83, 88, 94, 98, 162, 169

GED Graph Edit Distance. 56, 62

GIN Graph Isomorphism Network. 56, 83, 88, 94, 98, 108

GMN Graph Matching Networks. 61, 114, 125, 132, 133

GNN Graph Neural Networks. 4, 7, 8, 11, 51, 53–58, 60, 61, 82, 88–91, 94, 95, 97, 98, 101–103, 109, 114, 150, 151, 158–160, 166–169, 210

GPU Graphics Processing Unit. 11, 12, 90, 92, 99, 100

GRU Gated Recurrent Unit. 162

IR Intermediary Representation. 3, 4, 8, 11, 12, 30, 33, 49, 68, 84–86, 88, 89, 95–97, 101–103, 105, 107, 150, 152

JNI Java Native Interface. 140

LGBM Light Gradient Boosting Machine. 68, 83

LSTM Long Short-Term Memory. 67, 68, 83, 148, 149

LTO Link-Time Optimization. 76

MBA Mixed Boolean Arithmetic. 4, 7, 9, 24, 42–44, 46–48, 67, 71, 77, 80–83, 109, 127, 145, 147–149, 152, 161, 162, 164–166, 168, 170, 207, 210

MCTS Monte-Carlo Tree Search. 67, 148, 149

MCU Micro Controller Unit. 30, 42, 65, 132

ML Machine Learning. 24, 26, 51, 52, 57, 59, 60, 65, 67–69, 82, 88, 127, 145, 147–149, 167

MLP Multi Layer Perceptron. 68, 83

NAP Network Alignment Problem. 62, 115

NLP Natural Language Processing. 3, 24, 51–53, 58, 60, 61, 90

OOM Out Of Memory. 11, 12, 90, 92, 95, 99, 100, 122, 123

PDG Program Dependence Graph. 40

PIE Position-Independent Executable. 76

PPO Proximal Policy Optimization. 162

RAM Random-Access Memory. 90, 122

RL Reinforcement Learning. 5, 25–27, 145, 147, 149, 150, 152, 157–159, 164, 166, 168–170, 207

SE Symbolic Execution. 52, 63, 65, 67, 68, 148

SMT Satisfiability Modulo Theories. 8, 146, 161, 162, 170

SOG Semantic Oriented Graph. 150–152, 169

SVM Support Vector Machine. 68, 82, 83

t-SNE T-distributed Stochastic Neighbor Embedding. 7, 74–76

TF-IDF Term Frequency Inverse Document Frequency. 52, 68, 83, 88, 91, 94, 98, 108

TP True Positive. 113

TSP Travelling Salesman Problem. 157

VM Virtual Machine. 43, 66

WL Weisfeiler-Lehman. 56, 57

List of Symbols

f A function.

P A program.

G A graph.

V Node set of a graph.

A Directed edge set of a graph.

n The graph order.

d Initial feature size.

k A layer index inside a model.

v A node within a graph.

$h_v^{(k)}$ The embedding of node v at layer k.

X_v The initial node feature matrix.

$\mathcal{N}(v)$ The neighborhood of node v inside a graph.

K Last layer index of a model.

d' Final embedding size.

g A permutation function.

h_G Graph embedding.

H Number of attentions heads in attention networks.

S_p Primary function set.

S_s Secondary function set.

ϕ One-to-one alignment function.

ρ The alignment function.

S Similarity matrix used in QBinDiff.

$\mathcal{X}$ Solution set of the Network Alignment Problem.

s_{ratio} Sparsity ratio used in QBinDiff.

$dist$ Distance used in QBinDiff.

α QBinDiff tradeoff between function similarity and Call-Graph topology.

ϵ QBinDiff relaxation parameter.

γ Reinforcement learning discount factor.

L Maximum size of a graph edit modification sequence.

ψ Node type mapping for heterogeneous graphs.

μ Edge type mapping for heterogeneous graphs.

τ_V Node types for heterogeneous graph.

τ_A Edge types for heterogeneous graph.

F_{cdg} Set of CDG graphs.

$Expr$ Set of expressions.

Var Set of free variables.

$\Diamond_u$ Set of unary operators.

$\Diamond_b$ Set of binary operators.

List of Publications

With proceedings

- Roxane Cohen, Robin David, Florian Yger and Fabrice Rossi. **Identifying Obfuscated Code through Graph-Based Semantic Analysis of Binary Code**. In Proceedings of the 13th International Conference on Complex Networks and their Applications, 2024.
- Roxane Cohen, Robin David, Riccardo Mori, Florian Yger, Fabrice Rossi. **Experimental Study of Binary Diffing Resilience on Obfuscated Programs**. In Proceedings of the 22nd Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), 2025.
- Roxane Cohen, Robin David, Florian Yger and Fabrice Rossi. **Identifying Obfuscated Code through Graph-Based Semantic Analysis of Binary Code**. In Applied Network Science, 2025.

Peer-reviewed

- Roxane Cohen, Robin David, Riccardo Mori, Florian Yger and Fabrice Rossi. **QBinDiff: A modular differ to enhance binary diffing and graph alignment**. At the Symposium sur la sécurité des technologies de l'information et des communications (SSTIC), 2024
- Roxane Cohen, Robin David, Riccardo Mori, Florian Yger and Fabrice Rossi. **Improving binary diffing through similarity and matching intricacies**. At the 6th Conference on Artificial Intelligence for Defense (CAID), 2024

Chapter 1

Introduction

1.1 Context and motivations

Binary executable code constitutes the fundamental basis of modern computing: every computer, smartphone, and connected device relies on it to perform complex computations, renders interactive applications, or enables secure online transactions. Binary code is generated through compilation, a process that translates human-readable source code into a form executable by Central Processing Unit (CPU) and, consequently, by computers. Since its creation, it has been a subject of reverse engineering, a discipline in which analysts seek to understand the behavior and structure of low-level machine instructions.

Such an analysis is motivated by both defensive and offensive perspectives, encompassing legitimate and illegitimate purposes. For instance, a compiled malware disseminated across the Internet must be examined by security engineers from a legitimate defensive point of view, with the aim of understanding its behavior and devising countermeasures to mitigate its propagation and impact. Conversely, analyzing a video game in order to enable cheating at large scale is purely offensive, pursued for unethical ends.

However, binary code is inherently designed for machines rather than humans, making it challenging to interpret without specialized tools and expertise. Manual analysis is thus labor-intensive, time-consuming, and prone to human error. It is further compounded when the code is transformed by aggressive compiler optimizations or, more critically, by obfuscation.

Obfuscation is conceived as a countermeasure against reverse engineering. It intentionally modifies the code to obscure its structure, in order to impede reverse engineering efforts. Like reverse engineering itself, it can serve both legitimate purposes, such as protecting software that falls under intellectual property rights, and illegitimate ones, for instance hindering security analysts attempting to examine the behavior of malware. A wide variety of obfuscation techniques exists, each of them potentially altering different aspects of binary code. In practice, they are frequently combined together to strengthen software protection against reverse engineering. Thus, the analysis of a well-designed obfuscated binary is notoriously difficult, even for highly skilled analysts.

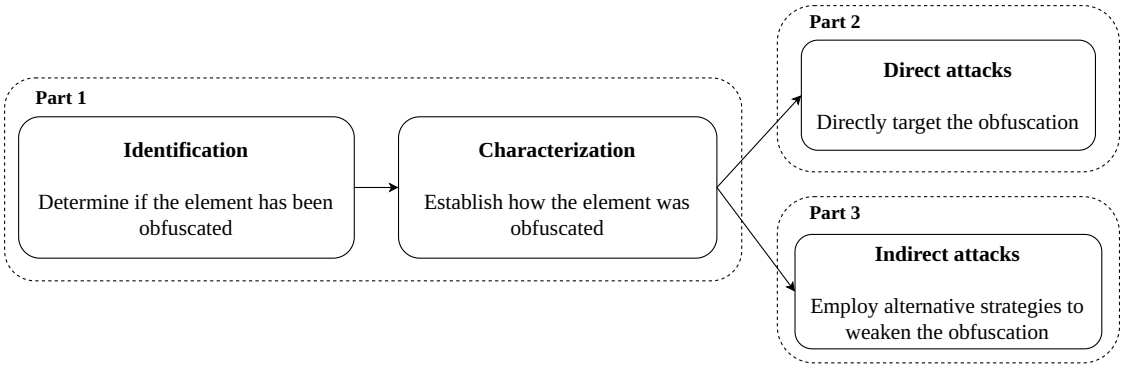


Figure 1.1: Obfuscation analysis pipeline.

In recent years, advances in Machine Learning (ML) have started reshaping binary code analysis. ML techniques have demonstrated impressive success in diverse domains, including computer vision, Natural Language Processing (NLP), and generative modeling. Among these advances, representation learning, the ability of models to automatically extract and encode meaningful patterns from complex data, has proven especially powerful. When combined with traditional classification or retrieval techniques, representation learning enables the automation of tasks that were once the exclusive domain of human experts. This potential has not gone unnoticed in reverse engineering. Novel ML-based approaches now target binary code directly, aiming to extract meaningful representations that support a wide range of analysis tasks.

However, representing binary code is far from straightforward, as it is inherently heterogeneous and complex, varying with the compilation toolchain, target architecture, optimization settings, and any obfuscation layers applied. Designing representations that capture these binary code aspects is therefore essential. In particular, graphs have emerged as a natural and expressive approach for representing binary code, as they can encode both its structural and part of its behavioral properties. Consequently, it motivates the use of advanced graph-based representation learning techniques, now fundamental to tasks such as binary similarity detection, function clustering, and, crucially, obfuscation analysis.

This thesis frames obfuscation analysis as a multi-stage process, illustrated in Figure 1.1. The first step involves obfuscation detection, determining whether the binary or any of its components, has been obfuscated. The second step, obfuscation characterization, seeks to identify which specific techniques were applied. Together, these steps evaluate the stealth of an obfuscation scheme and inform later phases of analysis. Once the obfuscation has been located and identified, it can be challenged via direct and indirect strategies. Direct strategies, or deobfuscation, aim to remove or mitigate specific obfuscation passes, such as Mixed Boolean Arithmetic (MBA), Virtualization or Control Flow Graph Flattening (CFF). These approaches require precise knowledge of the applied ob-

fuscation techniques. However, in practice, the layered aspect of real-world obfuscation, which combines multiple obfuscation techniques, often renders direct attacks against single techniques insufficient. Indirect strategies, by contrast, bypass the obfuscation to recover useful information without explicitly undoing it. They are especially valuable when direct deobfuscation is computationally infeasible or when obfuscation uses novel or hybrid techniques for which no dedicated countermeasures exist.

1.2 Research problems

The study of binary code representation learning, especially in the presence of obfuscation, can be decomposed into distinct but interrelated subproblems, each presenting its own set of technical challenges, as displayed in Figure 1.1.

The first challenge is to determine whether a given binary contains obfuscated regions and, if so, to locate them and identify the obfuscation techniques used. This obfuscation identification and characterization phase is often overlooked in the literature, where it is common to assume that obfuscated regions and their types are already known. In reality, distinguishing between naturally complex code and obfuscated code is non-trivial, given that both may exhibit similar structural irregularities.

> **Research question #1**. How can binary code graph representations be effectively leveraged to evaluate the detection and characterization of obfuscation?

Such representations could also be repurposed for other binary code analysis tasks. For example, in binary diffing, which consists in comparing two variants of the same program, or binary similarity, that aims to obtain similarity scores between two binary code elements, these representations reveal behavioral or structural similarities. This is particularly relevant for indirect attacks, which are used to extract meaningful information from obfuscated binaries without explicit deobfuscation. The critical role of binary diffing and similarity tools, particularly in the presence of obfuscation, gives rise to the following research question.

> **Research question #2**. Once obfuscation has been identified and characterized, can indirect attacks, such as binary diffing or binary similarity, be employed to recover relevant information directly from obfuscated binaries?

While indirect methods are useful in some contexts, they cannot always be applied. In such cases, direct attacks, namely, deobfuscation, may be the only viable approach. Yet, most existing deobfuscation methods are narrowly designed to counter single obfuscation techniques, making them poorly suited for the compound and layered obfuscations seen in practice. Furthermore, none of the existing work fully integrates novel types of enriched graphs for binary code, or advanced obfuscated binary representations, learned with Reinforcement Learning (RL) methods, which have shown promise in sequential decision-making tasks and could be well suited for deobfuscation scenarios. This consideration motivates the following research question:

> **Research question #3**. Once obfuscation has been identified, and indirect attacks are unsuitable, how can graph-based Reinforcement Learning (RL) serve as an effective approach to achieve obfuscation-agnostic deobfuscation?

Addressing these three research questions would yield a comprehensive toolkit for obfuscation analysis in real-world reverse engineering contexts, supporting practitioners at every stage of the process.

1.3 Contributions and structure

This thesis addresses the research questions above through the following outline:

- **Chapter 2, Chapter 3 and Chapter 4**. These Chapters introduce the conceptual and technical background necessary to understand this work. In particular, Chapter 2 presents the fundamental principles of binary code and defines key concepts used throughout the thesis, Chapter 3 surveys representation learning techniques, with a focus on graph-based approaches and Chapter 4 reviews how ML methods have been applied to various binary analysis problems in existing literature.

- **Chapter 5**. It presents the main datasets used in this thesis. It introduces ObfuBench, a large-scale obfuscated binary dataset that surpasses existing benchmarks in terms of size, diversity of obfuscators, and number of obfuscation passes. This dataset is used extensively throughout the thesis to support and validate all subsequent contributions in the field of obfuscation analysis. As such, it is part of our contributions. This dataset is complemented by three other datasets, originated from the binary similarity and obfuscation literatures, as well as two real-world obfuscated programs, serving to demonstrate the validity and generalization of our approaches across this thesis.

- **Chapter 6**. It addresses the first research question and investigates the detection and characterization of obfuscation, by providing an extensive study of graph-based binary representations. The study compares baseline approaches with advanced Deep Learning (DL) models, highlighting the influence of progressively enriched feature sets, ranging from structural to syntactic descriptors, on performance. Special attention is given to the impact of dataset partitioning strategies and compiler optimizations on the results.

- **Chapter 7**. This Chapter, aligned with the second research question, examines the problem of binary diffing as an indirect attack against obfuscation. It begins with an in-depth evaluation of diffing in a non-obfuscated setting, including an ablation study of the QBinDiff tool. The analysis is then extended to obfuscated binaries, assessing how different obfuscation types affect the performance of diffing and similarity tools. The Chapter further explores how insights from obfuscation detection, detailed in Chapter 6, can be leveraged to improve diffing outcomes.

- **Chapter 8**. It focuses on the challenge of deobfuscation, considered here as a direct attack strategy. Following a comprehensive review of the state-of-the art that emphasizes overlooked research gaps, this Chapter presents an exploratory approach combining novel enriched binary graph type, incorporating both data and control-flow information, with RL. The objective is to iteratively simplify the corresponding obfuscated graph through graph edit operations. Although this work remains incomplete, as this thesis ends before being able to conduct this contribution to its ends, the results demonstrate the feasibility of partial deobfuscation and contribute to novel findings in the relatively unexplored area of graph-based RL for binary analysis.

To sum up, this thesis investigates the use and evaluation of novel binary code representations designed to support each stage of the obfuscation analysis process. The proposed contributions are intended as modular tools within a broader reverse engineering framework. These tools can be selectively combined to enhance performance at various stages of the obfuscation analysis pipeline. The overall objective is to bridge the gap between representation learning and practical reverse engineering tasks, with a view toward applicability in real-world scenarios. These complementary contributions are aligned with the obfuscation analysis pipeline, using an experimental validation based on synthetic yet realistic datasets, as well as real-world samples of obfuscated binaries.

Chapter 2

Understanding Binary Code

At the core of every computer program lies binary code, whose inherent complexity makes it challenging to analyze and understand. This difficulty concerns both its structure, which reflects the organization and hierarchy of code elements interacting with specific relationships, and its semantics, which capture the behavior of the code, the operations it is designed to perform and the effects produced during execution.

This Chapter introduces the foundational concepts necessary to understand the work presented in this thesis. Specifically, Section 2.1 presents the nature of binary code, the compilation process that generates it, and the key parameters involved, along with techniques used to reconstruct binary code into higher-level, more interpretable formats. Section 2.2 explores the hierarchical organization of binary code and how it can be modeled using graph-based data structures. Finally, Section 2.3 examines obfuscation, a widely used software protection technique known to significantly distort binary code to hinder and limit its analysis.

2.1 From Source to Binary

A computer program is a structured sequence of instructions, written in a programming language, designed to be executed by a computer. A significant number of these programs are compiled and originate from source code written in a human-readable programming language, which is then translated into low-level machine code through a compilation process. The resulting output is a sequence of machine code instructions encoded in binary (0's and 1's) which can be decoded and executed by a machine CPU. An illustrative example of this process, using the Fibonacci function written in `C`, is presented in the later Figure 2.1.

The compilation process, described in Section 2.1.1, produces a binary whose final machine code is shaped by several factors, including optimization strategies (Section 2.1.2), the linking method (Section 2.1.3), and symbol stripping (Section 2.1.4). Ultimately, the binary is emitted as machine code specifically tailored to the architecture for which it was compiled (Section 2.1.5). During compilation, much of the auxiliary information useful for understanding the code is discarded, as it is unnecessary for execution,

leading to a substantial loss of abstraction (Section 2.1.6). Despite this loss, higher-level structures can be partially reconstructed through disassembly (Section 2.1.7). Such disassembly is, however, architecture-dependent. To overcome this limitation, it can be translated into an architecture-agnostic form using an Intermediary Representation (IR), as described in Section 2.1.8. Finally, in order to facilitate analysis, disassembly results can be exported into user-friendly formats by means of binary exporters (Section 2.1.9).

2.1.1 Compilation

During the compilation process, the source code is first translated into an Intermediary Representation (IR), which serves as an abstraction that facilitates further analysis. This IR is then optimized to meet specific performance objectives, inherent to the characteristics of the original source code, or constraints of the target execution environment, before being translated into binary code. `gcc`[1] and `clang`[2] are widely used compilers, particularly for the `C` programming language, which is the primary focus of this thesis.[3]

2.1.2 Optimization

Compiler optimization tends to influence and alter the structure and content of the resulting binary code. Two programs derived from the same source code, and therefore implementing the same functionality, may nevertheless exhibit different execution times depending on the optimizations applied during compilation. In general, the higher the level of optimization, the faster the resulting binary tends to execute. Consequently, performance-critical applications are typically compiled with optimization levels designed to improve execution speed, such as `-O2` or `-O3`. Inlining is a typical transformation that intervenes frequently with these two optimization levels, that replaces the call site of a function with its body directly in the calling function. Conversely, programs intended to run on Micro Controller Unit (MCU), devices with a single integrated circuit containing a processor, limited memory, and input / output peripherals, are typically compiled with options that minimize code size, such as the `-Os` optimization. In contrast, a code yet at a debugging step will usually be compiled in `-O0`, which disables most optimizations in order to preserve code structure and facilitate debugging.

2.1.3 Static and dynamic linking

Programs frequently rely on external definitions, such as pre-implemented functions or libraries. To enable their execution, the program must resolve their actual memory addresses and link them inside the program. This process, known as linking, is either static or dynamic. In static linking, all required dependencies and resources are directly

[1] https://gcc.gnu.org/

[2] https://clang.llvm.org/

[3] All the methodological contributions presented in this thesis are generic and may be extended to other compiled languages, such as `Rust` or `Go`, subject to appropriate adaptations.

embedded into the resulting binary, yielding a self-contained executable that can be deployed on other systems without external dependencies. Dynamic linking, by contrast, links external libraries or resources at runtime, resulting in smaller binaries that rely on the environment in which they are executed.

2.1.4 Stripping

Additional compiler options can be employed to further reduce the program size, such as stripping. Stripping removes residual metadata, like function names or debugging symbols within the binary, that are not required for program execution. This step reduces the binary size by eliminating in particular function names and their addresses, thereby hindering analysis efforts by obscuring the program's internal structure and the relationships between functions that are no longer explicitly named. While unnecessary for execution, these metadata can reveal valuable information about the program's design and functionality.

2.1.5 Architectures

Compiling source code implies generating binary code that is compatible with the underlying CPU machine code. In practice, this involves generating instructions that can be physically executed by the CPU, whose components are organized in specific ways and operate according to physical constraints. These constraints define the behavior of the processor and the CPU architecture. In this context, a source code compiled on a specific machine will share the same architecture, unless cross-compilation is explicitly specified. For example, the source code compiled on a `x86` machine will produce binary code interpretable by `x86` processors. Various architectures exist, among those `x86`, `ARM` or `MIPS`.

Each processor architecture is also characterized by the width of data it can natively process, typically 32 or 64 bits. Naming conventions for architectures can vary accordingly: for example, `x86` generally refers to the 32-bit `x86` architecture, while `x64` denotes its 64-bit extension, `x86-64`. In the case of `ARM`, the bitness may or may not be explicitly stated (e.g., `ARM-32` or `ARM-64`). Throughout this thesis, without loss of generality, we focus exclusively on the `x64` architecture for the experiments.

2.1.6 Abstraction loss

The final output of the compilation process, referred to as a binary executable, no longer contains any high-level data such as variable names, data types, class identifiers, and other semantic structures, that are irreversibly lost. Only the essential machine-level instructions are preserved, including registers and operands, thereby ensuring fast execution.

Afterwards, commercial software, such as video games, or malware, is usually distributed to users only in the form of machine code, while the original source code remains proprietary and accessible exclusively to the developers. As a consequence, the goal of

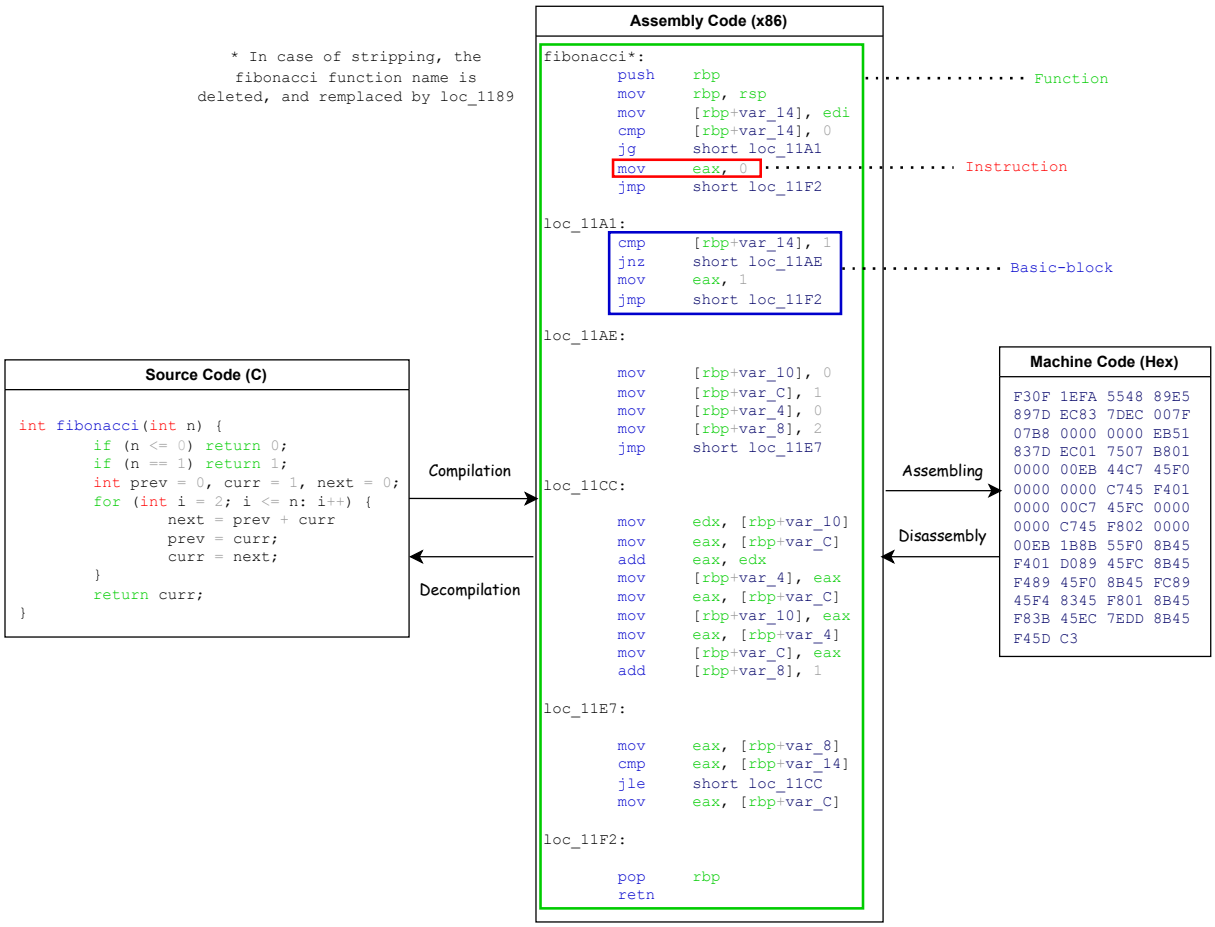


Figure 2.1: Compilation and assembling of the Fibonacci function `C` source code into assembly code and machine code.

reverse engineering is precisely to analyze and understand the functionality and behavior of such a machine code when the executable binary is the only accessible resource to examine due to the source code unavailability.

Direct analysis of machine code is notoriously tedious, due to its low-level nature and the substantial amount of information lost during compilation. While reconstructing the original source code from machine code, a process known as decompilation, is sometimes possible, it is generally challenging and, in many cases, infeasible. A more achievable intermediate step consists in transforming machine code into assembly language which, although still low-level, provides a more human-readable form than raw binary instructions. An overview of these transformations between source code and machine code is displayed in Figure 2.1.

2.1.7 Disassembly

This translation process, known as disassembly, attempts to translate the original binary code into a sequence of assembly instructions, using solely static analysis of the parsed binary code, that is without executing the program. Obtaining such a disassembly from a binary is nevertheless far from trivial and presents numerous technical challenges [102], such as the identification of function boundaries within the binary, resolving indirect control-flow or managing the pointer aliasing problem, known to be undecidable [83]. Tools such as IDA-Pro [4], Ghidra[5], Radare2[6] or BinaryNinja[7] are well-known disassemblers. It is also possible to retrieve a basic disassembly using elementary commands, such as `objdump`. An assembly example is available in Figure 2.1.

2.1.8 Intermediary Representation

Disassembly is derived from the machine code, thus depends on it and on the underlying architecture for which it is designed. In fact, the `x64` and `ARM` assemblies differ significantly, resulting in distinct instruction sets, mnemonics and calling conventions. Although both originate from the same source code, an example of which is shown in Figure 2.2, the resulting assembly varies: specifically, the `x64` architecture uses a distinct set of branching mnemonics, like `jz, jnz, je, jne, ja, jne`, etc. whereas the `ARM` architecture employs instead `b, bl, bx`. Therefore, the same source code, compiled for different architectures, hence machines, leads to different binary codes and assemblies. An IR can, however, be employed to represent a program in a unified, higher-level, architecture-agnostic language. Pcode[8] is the IR behind the Ghidra disassembler.[9] Thus, each architecture-dependent assembly instruction is semantically represented by one or

[4] https://hex-rays.com/ida-pro

[5] https://github.com/NationalSecurityAgency/ghidra

[6] https://github.com/radareorg/radare2

[7] https://binary.ninja/

[8] https://spinsel.dev/assets/2020-06-17-ghidra-brainfuck-processor-1/ghidra_docs/language_spec/html/pcoderef.html

[9] The IR produced by disassemblers are entirely distinct from those generated by compilers; they serve different purposes and are not directly related.

```c
int fibonacci(int n) {
    if (n <= 0) return 0;
    if (n == 1) return 1;

    int prev = 0, curr = 1, next = 0;
    for (int i = 2; i <= n; i++) {
        next = prev + curr;
        prev = curr;
        curr = next;
    }
    return curr;
}
```

(a) Fibonacci source code.

```
fibonacci:
        push    rbp
        mov     rbp, rsp
        mov     [rbp+var_14], edi
        cmp     [rbp+var_14], 0
        jg      short loc_11A1
        mov     eax, 0
        jmp     short loc_11F2

              [...]
```

(b) First instructions of the `x64` Fibonacci binary function.

```
unique[6980:1] = *[ram]RAX
CF = carry(unique[6980:1], AL)
unique[6980:1] = *[ram]RAX
OF = scarry(unique[6980:1], AL)
unique[6980:1] = *[ram]RAX
unique[6980:1] = unique[6980:1] + AL
*[ram]RAX = unique[6980:1]
unique[6980:1] = *[ram]RAX
SF = unique[6980:1] s< 0x0
unique[6980:1] = *[ram]RAX
ZF = unique[6980:1] == 0x0
        [...]
```

(c) First Pcode instructions coming from the `x64` Fibonacci binary code.

```
fibonacci:
        push    {r11}
        add     r11, sp, #0
        sub     sp, sp, #0x1C
        str     r0, [r11,#var_18]
        ldr     r3, [r11,#var_18]
        cmp     r3, #0
        bgt     loc_8268
        mov     r3, #0
        b       loc_82E0
             [...]
```

(d) First instructions of the `ARM` Fibonacci binary function.

```
unique[2680:1] = !ZR
if (unique[2680:1]) goto ram[8248:4]
shift_carry = CY
r0 = r0 & r4
tmpCY = shift_carry
tmpOV = OV
tmpNG = r0 s< 0x0
tmpZR = r0 == 0x0
unique[2680:1] = !ZR
if (unique[2680:1]) goto ram[824c:4]
shift_carry = CY
        [...]
```

(e) First Pcode instructions coming from the `ARM` Fibonacci binary code.

Figure 2.2: The Fibonacci source code, the resulting binary code compiled for `x64` and `ARM`, and their respective underlying Pcode instructions.

more Pcode instructions in an architecture agnostic way, as shown in Figure 2.2. As a consequence, two programs, derived from the same source code but compiled for two different architectures, may not share the same assembly, but will share the same Pcode language, despite variations in the translation outcomes.

2.1.9 Exporting disassembly

Reverse engineering frequently implies extensive manual effort, especially when analyzing a program disassembly. Automating part of this analysis is then necessary, which requires that the disassembled code is easy to manipulate for scripting. It has led to the development of binary exporters, which extract data from the disassembly, given by disassembly tools, and export it to a file designed to be easily manipulated programmatically by analysts. Among the available exporters, BinExport and Quokka are two notable examples.

BinExport[10] is a binary exporter owned by Google. It exports disassembly results to a protobuf file and supports IDA-Pro, Ghidra and BinaryNinja as disassemblers. The resulting protobuf file can be easily manipulated with the `python-binexport` package[11], developed by Quarkslab.

Quokka [21] is another exporter designed to overcome the limitations of BinExport, also conceived by Quarkslab. It exports more data, such as cross-references, that are essential for drawing correlations between code and data in binary analysis, while reducing the storage space and the exporting time[12], although as of this writing, it only supports IDA-Pro. It can be used with the associated python package[13].

2.2 Structure of binary code

2.2.1 Binary code hierarchy

A disassembly captures the structure of a binary program by representing it through a hierarchy of objects with varying levels of granularity, ranging from individual instructions to the entire program. Each of these granularities embeds specific information related to the underlying binary code. These objects can be observed in Figure 2.1 and are defined as follows:

- **Instruction**. It is the smallest unit of execution of a binary program.[14] It combines a mnemonic, namely the action to operate, and optional operands, the parameters of this operation. These operands usually refer to CPU registers, memory addresses or immediate values. For instance, the instruction `mov eax, 0` indicates that the constant immediate value 0 should be moved to the register `eax`. In this context, the register plays the role of a dedicated storage location within the CPU for storing temporary values. `mov` is the mnemonic, while `eax, 0` are respectively the first and second operands.

- **Basic Block**. A sequence of instructions can be grouped together within a Basic Block (BB) if none of the instructions within the sequence is the destination of

[10] https://github.com/google/binexport
[11] https://github.com/quarkslab/python-binexport
[12] https://blog.quarkslab.com/quokka-a-fast-and-accurate-binary-exporter.html
[13] https://github.com/quarkslab/quokka
[14] Microcode is directly governed by the CPU and is inaccessible to the compiled program, which does not interact with it.

an incoming or the source of an outcoming branching instruction. Consequently, a BB is guaranteed to be executed from its beginning to its end in order, without interruption.[15] It usually ends with a control-flow redirection, either a conditional or unconditional branching, a call to another function, or a return to the caller function.

- **Function**. It is a fundamental element within a program, common to almost all programming languages. It is intended to perform a specific task in the program. A function is composed of multiple Basic Blocks (BBs) that are connected together through control-flow constructs, such as conditions or loops. The relationships between the BBs define the logical structure and behavior of the function.
- **Program**. It represents the highest-level code unit and stacks multiple functions together that interact through well-defined calling relationships.

Elements of binary code interact with one another through specific relationships: BBs redirect to other BBs, and functions invoke one another. Graphs are data structures specifically designed to model such relationships.

2.2.2 Program as graphs

Graphs are powerful data structures that represent relationships between objects. They were shown to naturally and effectively model binary code [124, 2, 148], particularly the binary execution flow, commonly referred to as control-flow. Different graph types are designed to capture distinct binary code properties and interactions.

Control Flow Graph

A Control Flow Graph (CFG) is a graph representing a function execution flow. An example of a CFG is available in Figure 2.3.

Definition 2.2.1 (Control Flow Graph)**.** Given a function f within a binary program P, its Control Flow Graph (CFG) is a directed graph $G = (V, A)$, with V a set of nodes representing Basic Blocks (BBs) and A a set of directed edges that link BBs together according to the execution flow of the function.

Since binary code, hence its assembly, is extremely sensitive to compilation parameters, such as optimization level and target architecture, the CFG is not unique for a given source function. Although optimization preserves the program semantics, the same function, compiled then optimized differently, may exhibit multiple CFG variants that all convey the same underlying semantics. The more optimized the code is, the faster it will execute, and the more its assembly will differ from non-optimized assembly code. Thus, entire BBs may disappear, sometimes even entire functions. Figure 2.4 illustrates this CFG variability between `-O0` and `-O2`. Conversely, two distinct functions may share the same CFG structure while differing in the instructions contained within their BBs.

[15] A function call located within a BB, excluding its final instruction, is treated as a non-branching instruction, despite altering the program's control flow.

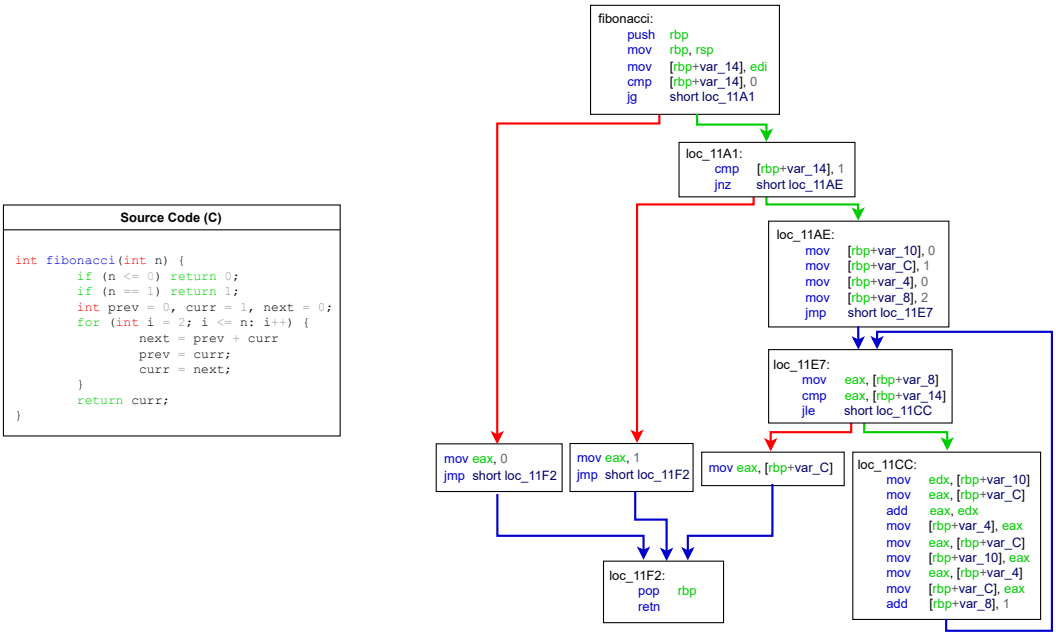


Figure 2.3: The Fibonacci source code and the CFG of the corresponding disassembly, compiled in `-O0`.

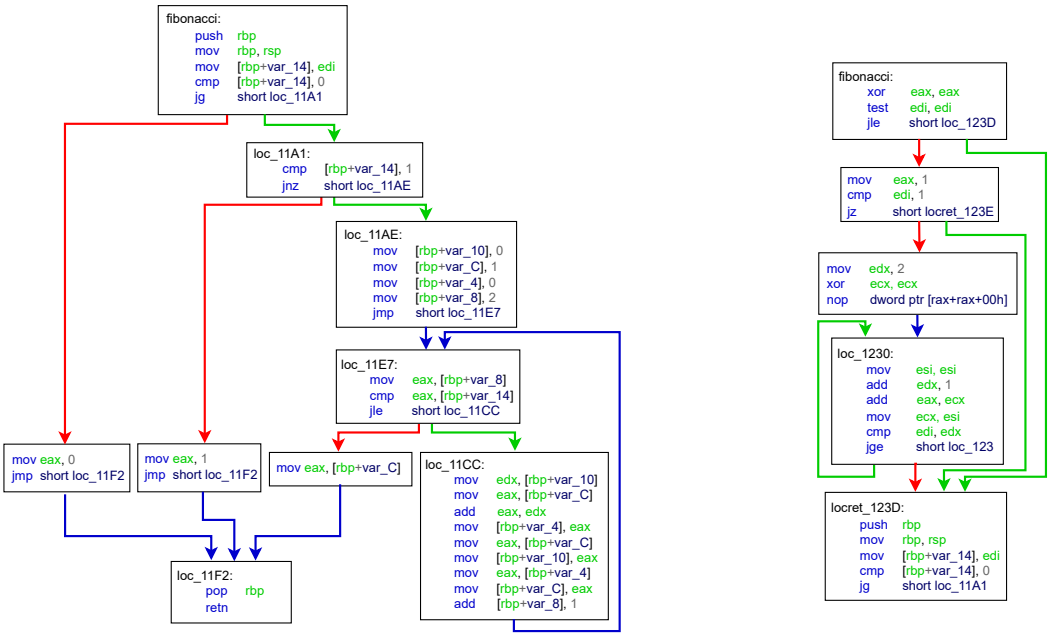


Figure 2.4: Two CFGs of the Fibonacci function, compiled respectively with `-O0` and `-O2`.

Call Graph

A Call Graph (CG) is a graph representing the whole program. An example of a CG is available in Figure 2.5.

Definition 2.2.2 (Call Graph)**.** Given a binary program P, its Call Graph (CG) is a directed graph $G = (V, A)$, with V a set of nodes representing functions and A a set of directed edges that link functions in a caller-callee relationship.

Similar to the CFG, the CG is also subject to variability, mainly due to compiler optimizations. For instance, inlining erases a function from the program and directly embeds its code into the functions that call it, resulting in the elimination of the corresponding function node from the CG and its adjacent edges.

The CG has proven to be a crucial graph format for capturing binary behavior, particularly due to its robustness against various code transformations. It is widely employed across numerous analysis tasks [80, 68].

Data Flow Graph

A Data Flow Graph (DFG) is a graph that describes the data dependencies among a sequence of operations. An example of a DFG is shown in Figure 2.6.

Definition 2.2.3 (Data Flow Graph)**.** Given a block of binary code within a program P, its Data Flow Graph (DFG) is a directed graph $G = (V, A)$, with V a set of nodes representing data-related objects, such as variables or operators, and A a set of directed edges that link each node together according to the data relationships between objects.

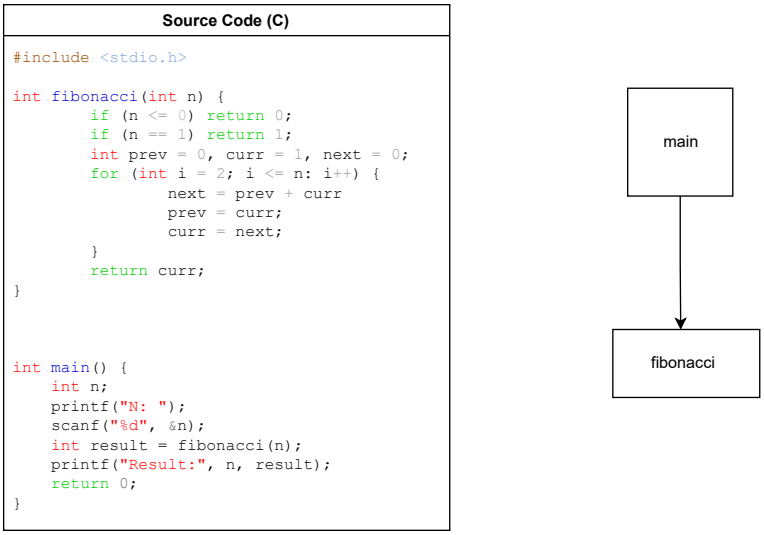


Figure 2.5: A program source code and its corresponding CG.

A DFG is useful to model a sequence of computation, particularly to illustrate the flow from inputs to outputs or vice versa.

While this thesis adopts the conventional definitions of the CFG, CG and DFG as directed graphs, they could theoretically be modeled as multi-directed graphs.[16]

Other graphs

Beyond being solely applied to binary code, graphs also benefit from representing source code. Several graphs are then derived, each of them focusing on different source code aspects.

- **Abstract Syntax Tree (AST)**. These graphs are tree structures that decompose a source code into a hierarchy of nodes, each of them potentially representing a variable, an operation or a condition, and serves as a parameter or child of its parent node.

[16] A multi-directed graph is defined as a directed graph in which multiple arcs are allowed between a given pair of source and destination nodes.

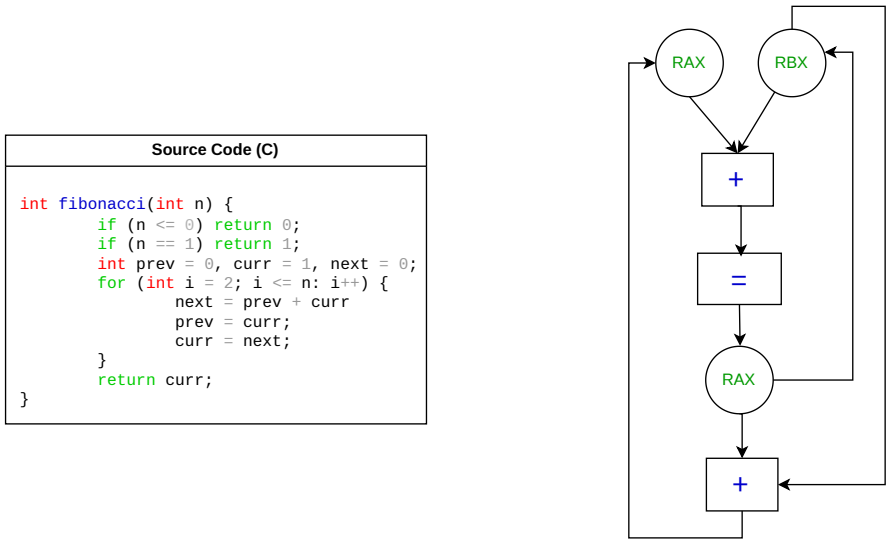


Figure 2.6: The Fibonacci source code and the DFG of the corresponding disassembly.

- **Program Dependence Graph (PDG)**. It captures both control and data dependencies among the source code elements [46]. Each node, potentially representing a variable, an operation or a condition influencing the control-flow, is linked to the entities on which it depends. For example, a loop conditioned on a integer variable will naturally depend on that variable, resulting in an edge between the two corresponding nodes.

- **Code Property Graph (CPG)**. The CPG [154] leverages the previous graph constructions by combining AST, PDG and CFG data into a single heterogeneous graph. Nodes belong to the original AST, thus potentially being variables, operations or types, and are linked together by the AST edges, as well as CFG and PDG edges between the nodes representing the root of each AST.

2.2.3 Semantics, syntax and structure

Binary code can be understood through three fundamental and interrelated concepts: semantics, syntax, and structure, which are extensively used in binary analysis and must be carefully distinguished.

Semantics captures the meaning or behavior of a program: what the code actually accomplishes, the computations it performs, and the side effects it produces on memory, registers, and the environment. To fully comprehend a binary program, it is essential to understand its semantics, as this reflects the true intent and functionality of the code, independent of its specific form of expression.

Syntax, in contrast, refers to the representation of the code, its form and notation. In the context of binary analysis, syntax is often embodied in assembly instructions, which are a direct textual rendering of the underlying machine code. However, syntax may diverge from semantics: the same behavior can be expressed with very different syntactic patterns depending on compilation parameters, optimization strategies, or even the choice of compiler. While semantics remains invariant across these transformations, syntax is far more malleable.

Structure describes the organization and relationships between elements of the binary code. This encompasses control-flow dependencies, data-flow interactions, and higher-level program hierarchies. Structure is crucial because it enables analysts to model and reason about the program as a system of interconnected components, rather than as a flat sequence of instructions. Graph-based representations, as discussed in Section 2.2.2, are the canonical means to capture such structural information, since they naturally encode dependencies and relationships.

These three perspectives are not isolated but complementary, and are used throughout this thesis. In practice, reverse engineering is primarily concerned with recovering the semantics of a program. However, structure and syntax play an indispensable supporting role: they serve as stepping stones to semantic recovery, even though both can be altered or made artificially complex.

2.3 Obfuscation

While assembly code operates at a lower level than programming languages, it can still be analyzed in order to understand its behavior, a process called reverse engineering. It is employed across a wide range of contexts, from legitimate use cases, such as malware analysis, to unethical applications, like reversing a video game to enable cheating. As a consequence, protecting assembly code from reverse engineering has become a critical concern. Obfuscation is intended to offer such software protection and has been employed since the emergence of commercial software and malware. It aims to hide a program underlying logic without altering its functionality, obscuring the program syntax without modifying its semantics, to hinder and complicate reverse engineering.

Achieving perfect obfuscation that will indefinitely protect a program sensitive contents [5] has been proven impossible. Given enough time, expertise and resources, a determined adversary may succeed in breaking an obfuscation and understand the logic of a protected program. Therefore, the primary goal of obfuscation is to increase the code complexity while preserving the semantics, and cost of analysis to the point where it becomes impractical or unattractive for an attacker to pursue. In contrast, compilation-time optimizations perform the inverse operation, seeking to simplify binary code. The opposition between these two processes is examined in Section 2.3.5.

Beyond protecting a program, obfuscation can also serve the purpose of software diversification, aiming to generate multiple variants of a program that appear significantly different. The goal is to produce instances that are difficult to correlate with one another. This technique is commonly employed in software distribution, where each user

receives a uniquely obfuscated version of the same application.

Program obfuscation is typically performed by sequentially applying one or more obfuscation passes, each corresponding to a specific transformation. An obfuscation scheme thus consists of a sequence of such passes. Designing an effective obfuscation strategy often involves balancing security with performance. Applying too many obfuscation passes can severely degrade execution speed, while too few may offer insufficient protection. Determining the appropriate level and type of obfuscation usually requires a deep understanding of both the functional criticality of the code and the constraints of the target system. For example, critical functions that warrant strong protection may also be performance-sensitive, making them particularly vulnerable to the overhead introduced by obfuscation, especially in resource-constrained environments such as MCUs. This tradeoff between security and performance is assessed using dedicated metrics, as outlined in Section 4.3.1.

Applying an obfuscation depends on the underlying programming language; some obfuscations are specifically designed for native code, while others target Java for example. In this thesis, we focus exclusively on native code obfuscation.

Native obfuscation can be categorized into static (Section 2.3.1) and dynamic (Section 2.3.2) approaches. It is typically implemented using obfuscators (Section 2.3.3), each of which employs specific obfuscation techniques with distinct names (Section 2.3.4). For obfuscation to be effective, it must account for the opposing role of compiler optimizations, which are specifically intended to eliminate such transformations, as discussed in Section 2.3.5.

2.3.1 Static obfuscation

Static obfuscation is designed to limit any static reverse engineering, which implies analyzing the binary code without executing the program. This obfuscation type alters the binary code in a way that renders the resulting disassembly challenging to interpret. Within static obfuscation, different obfuscation classes are defined, each of them producing distinct effects on the resulting obfuscated binary:

- **Data obfuscation**. It aims to obscure the program data-flow. Constants, immediates and operations are subject to replacements in such a way that understanding the exact computations becomes significantly more challenging. This obfuscation type is usually applied at the function or Basic Block (BB) level. One of the most widely used techniques in this category is Mixed Boolean Arithmetic (MBA). The MBA pass consists in replacing simple operations or values with a semantically equivalent but syntactically complex sequence of expressions involving arithmetic and logical operations [160]. An example is available in Figure 2.7, compiled with `-O0` using the Tigress obfuscator. Compared to the regular `-O0` CFG displayed in Figure 2.3, the BBs contain more instructions, with a predominant use of new logic and arithmetic mnemonics, such as `mov, xor, add, or, and, shr, sub`. In contrast, the structure of this MBA-obfuscated CFG, namely its adjacency matrix, is left unchanged.

- **Intra-procedural obfuscation**. It alters the execution flow within a function. The structure of the CFG is the first element that will be modified by obfuscations passes falling into this category. One such technique is Control Flow Graph Flattening (CFF), which puts every BBs of a function at the same level and uses a dispatcher to preserve the original execution flow logic [144]. Figure 2.8 presents the Fibonacci function obfuscated with CFF. For clarity and conciseness, instruction-level details within each BB are omitted, as the emphasis is placed on the altered graph structure rather than individual instructions. In fact, the structure of the obfuscated CFG differs significantly from the unobfuscated one, shown in Figure 2.3 and illustrates CFF principle of flattening the control-flow. Another technique consists in adding Opaque Predicates [28], which introduce artificial conditional branches, using in particular MBA-based conditions. These conditions are always evaluated to the same value at runtime, ensuring that the true execution flow is maintained. The unused branch is usually filled with dead code to mislead static analysis. Similarly, Virtualization transforms a regular function into a Virtual Machine (VM) executing its own bytecode.

- **Inter-procedural obfuscation**. It modifies the execution flow at the program level by disturbing the relationships between functions, thereby transforming the CG and CFG structures. For example, an obfuscation may fusion two or more functions inside a single one, leading to the disappearance of the corresponding merged function in the CG. Conversely, it is possible to divide a function into several chunks that are called in the right order to maintain the program execution flow. Copy creates multiple copies of a function, any of which may be invoked in place of the original function. An example of a Split obfuscation is displayed in Figure 2.9, showing the effect of this obfuscation on both the CG and the CFG.

2.3.2 Dynamic obfuscation

Dynamic obfuscation is intended to prevent any analysis at runtime. Contrary to static obfuscation, which does not impede an attacker from executing the program to analyze it, dynamic obfuscation alters the behavior of a program by introducing execution-time transformations, while always maintaining its true semantics. Common dynamic obfuscation techniques include anti-debugging mechanisms, which detect the presence of a debugger and trigger countermeasures to alter the program's behavior [50]; self-modifying code, which alters its own binary instructions during execution [100]; and packing, wherein the original binary is encrypted or compressed and subsequently unpacked at runtime [128].

In this thesis, we do not explore dynamic obfuscation in further detail, as our focus remains on static obfuscation techniques. However, investigating more dynamic obfuscation represents a promising direction for future research.

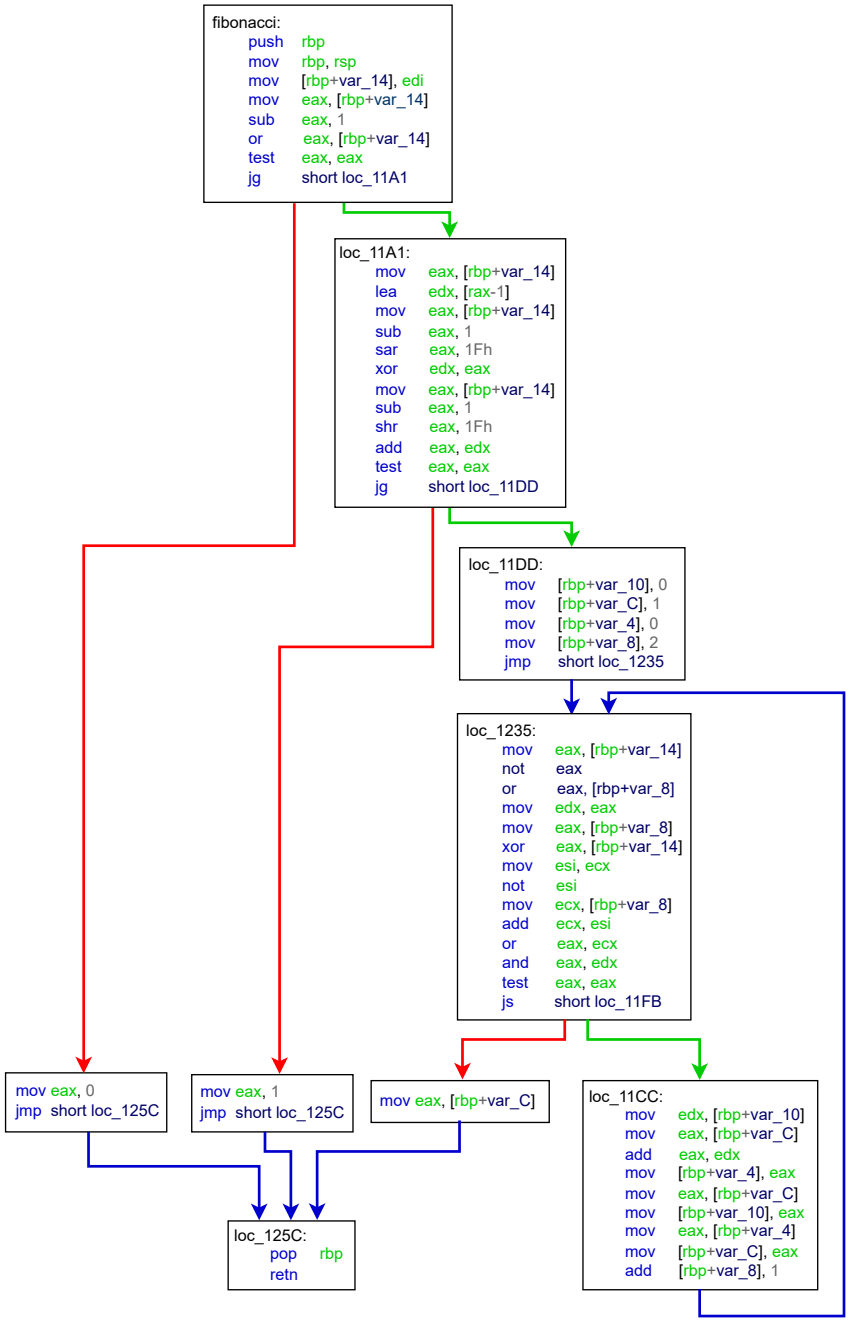


Figure 2.7: The CFG of the `-O0` Fibonacci function obfuscated with a MBA.

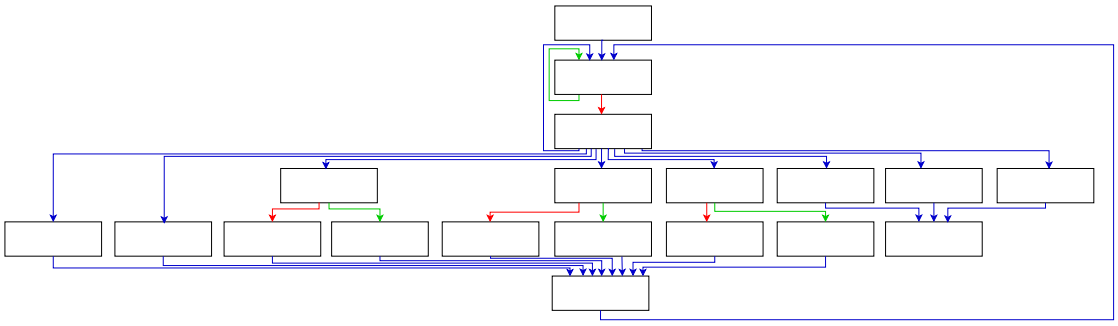


Figure 2.8: The CFG of the `-O0` Fibonacci function obfuscated with a Control Flow Graph Flattening (CFF).

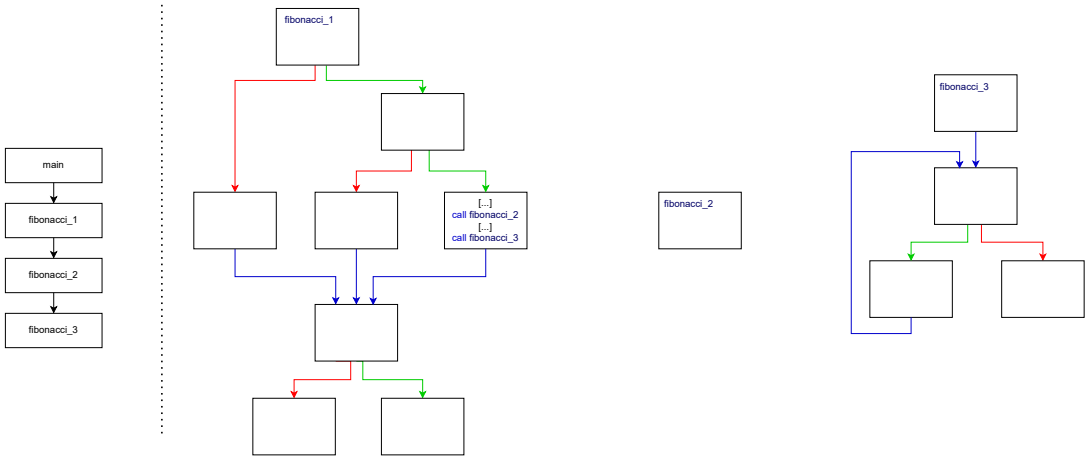


Figure 2.9: The CG of the obfuscated program, with two new split functions and the corresponding CFG for all the `-O0` Fibonacci-related functions.

2.3.3 Obfuscators

Native code obfuscation, applied to programs written in `C` or `C++` for example, can be performed through different approaches: either source-to-source, like the Tigress obfuscator [27] where the source code file is obfuscated and can later be compiled, or integrated into compiler toolchains, such as OLLVM [74].

OLLVM is an obfuscator that directly interfaces with `clang` during compilation. The most recent publicly available version is adapted for `LLVM-4.0`, released in 2017. OLLVM provides only three obfuscation passes: CFF, a Bogus Control-Flow pass and an Instruction Substitution pass. The Bogus Control-Flow inserts a BB guarded by an Opaque Predicate, ultimately redirecting execution to the original next BB. The Instruction Substitution pass replaces binary operations with a complicated sequence of instructions that is semantically equivalent to the original operations. These last two passes are similar to Opaque Predicates and MBA and share the same underlying principles.

Tigress is a source-level obfuscator that offers a broad spectrum of highly configurable obfuscation passes, including both intra and inter-procedural obfuscations, as well as data-related obfuscations. As of the time of writing this thesis, two versions are available: Tigress-3 and Tigress-4. In this work, only Tigress-3 was used, as version 4 was released lately during this thesis. It requires a two-step process: first, an amalgamation phase, that aggregates all the `C` contents within a project into a unique `C` file, and second, the proper obfuscation of the resulting merged `C` file.

While Tigress and OLLVM remain the two most prominent free or open-source obfuscators, several others have been developed, including the eShard obfuscator-LLVM[17], Ssage[18], Pluto[19], Hikari[20], O-MVLL [21], though many of them are either no longer maintained or not viable for practical use. For instance, the eShard obfuscator-LLVM is a fork of OLLVM, providing only a new Split pass, SSage is only available in Chinese, Pluto and Hikari are not maintained anymore and O-MVLL provides a limited number of additional passes compared to those of OLLVM, only available for the `AArch64` architecture.

More recently, a new generation of promising obfuscators, with a significant number of obfuscation passes, is starting to appear, like Polaris[22] or Loki [127].

An overview of the various publicly available obfuscators and their functionalities is provided in Table 2.1.[23]

[17] https://github.com/eshard/obfuscator-llvm
[18] https://github.com/SsageParuders/SsagePass
[19] https://github.com/bluesadi/Pluto
[20] https://github.com/HikariObfuscator/Hikari
[21] https://github.com/open-obfuscator/o-mvll
[22] https://github.com/za233/Polaris-Obfuscator
[23] Commercial products, such as Themida, VMProtect or QShield, are omitted in this work.

Obfuscator name	Maintained	I/O	# obfuscations	Configurable	Data	Intra-procedural	Inter-procedural
Tigress*	Yes	S/S	> 10	+++	✔	✔	✔
eShard obfuscator	Yes	S/B	4	+	✔	✔	✔
Ssage	Yes	S/B	7	+	✔	✔	✔
O-MVLL	Yes	S/B	7	+	✔	✔	✗
Polaris	Yes	S/B	> 10	N/A	✔	✔	✔
Loki	Yes	S/B	1	+	✔	✗	✗
OLLVM	No	S/B	3	+	✔	✔	✗
Pluto	No	S/B	6	N/A	✔	✔	✗
Hikari	No	S/B	5	N/A	✔	✔	✗

∗ Can only be applied to projects contained within a single `C` source file.

Table 2.1: Existing obfuscators and their properties.

2.3.4 Obfuscation pass name

At first glance, OLLVM and Tigress appear to share only one common obfuscation pass, Control Flow Graph Flattening (CFF). However, some obfuscation techniques, despite being referred to by different names, are conceptually equivalent. For instance, the OLLVM Bogus Control-Flow is analogous to Tigress's Opaque Predicates, and the OLLVM Instruction Substitution is similar to the Tigress MBA pass. Consequently, throughout this work, obfuscation pass names are unified based on their underlying transformation logic. Nevertheless, implementation differences may cause these conceptually similar transformations to vary in strength and complexity. A detailed mapping between original pass names and their unified counterparts is provided in Table 2.2.

	Passes	Pass type	Description	Unified name
Tigress	Copy	Inter	Clone a function	Copy
	Split	Inter	Split a function into chunks	Split
	Merge	Inter	Merge multiple function into one	Merge
	CFF	Intra	The function BBs are put at the same level	CFF
	Virtualize	Intra	Transforms a function into an interpreter	Virtualize
	Opaque	Intra	Insert Opaque Predicates	Opaque
	EncodeArithmetic	Data	Replace computations with complex expressions	Enc.A
	EncodeLiterals	Data	Hide literals with less obvious expressions	Enc.L
	Mix	Intra & Data	Combination of CFF, EncodeArithmetic and Opaque	Mix
	Mix + Split	All	Mix + Split	Mix + Split
OLLVM-14	CFF	Intra	The function BBs are put at the same level	CFF
	BogusCF	Intra	Insert BB and OpaquePredicates	Opaque
	InsSub	Data	Substitute operators by more complicated instructions	Enc.A
	Mix	Intra & Data	Combination of CFF, InsSub and BogusCF	Mix

Table 2.2: Obfuscation descriptions and names.

2.3.5 Obfuscation against optimization: opposing methods for the same semantics

Optimization and obfuscation are deeply intertwined. While obfuscation tries to hide the program behavior by deliberately inserting complex code sequences, optimization

applied during compilation conversely attempts to reduce code size and execution time, often removing such artifacts artificially introduced by obfuscation. As a consequence, the compiler can interfere with or even negate the effects of obfuscation, leading to altered obfuscation patterns [93]. Many parameters influence the final efficiency of the obfuscation left: the compiler version, the optimization level that is enabled, the order of the obfuscation passes within the compilation process[24], etc.

In particular, `-O0` optimization reduces optimization to its smallest level ensuring minimal transformation of the source code. Only essential optimizations are applied, such as constant propagation, which consists in replacing values that are known constant in further variables or computation in order to limit the binary code size. This propagation occurs both in the compiler front-end and optimization phase. Without such elementary optimizations, the final binary would execute considerably slower.

The `-O2` optimization level, contrary to `-O0`, activates various optimization passes, including inlining. The higher the optimization level, the greater the risk that the obfuscation is altered or removed. This issue appears with modern compiler versions with aggressive optimizations, such as `clang-14`, when using older obfuscation techniques like OLLVM, based on `LLVM-4`. Even at the optimization level `-O0`, this behavior may still occur, though to a lesser extent, due to constant propagation.

For instance, considering the MBA shown in Figure 2.7, the compiler might decide to remove arithmetic or logic operations. Similarly, for the Split obfuscation, displayed in Figure 2.9, inlining, enabled with the `-O2` optimization level, may eliminate the `Fibonacci_2` and `Fibonacci_3` functions, by inlining them directly in the Fibonacci function, thus countering the effect of the obfuscation.

Source-to-source obfuscators, like Tigress, cannot directly interact with the compiler, thus influence the subsequent optimization transforms. In the case of OLLVM, the CFF and Opaque passes, two out of three, are applied before any optimization, meaning both are particularly vulnerable to simplification.

To mitigate these undesirable compiler side effects, other compiler-toolchain obfuscators insert their transformations within the optimization chain, neither at the beginning, to avoid aggressive subsequent optimizations, nor at the end, to slightly optimize the obfuscated code for performance purposes [61, 18]

Optimization and obfuscation can be seen as two sides of the same coin. In particular, optimization can also be perceived as an obfuscation technique; by heavily optimizing a code, analysis becomes difficult due to hardware-specific efficiencies and low-level instruction usage. For example, setting a register to zero using `xor eax, eax` is preferred over `mov eax, 0` for low-level performance purposes.

In particular, previous work [121] has demonstrated that there exist compiler optimization sequences that maximize, given a fitness function, differences between binary code generated from the same source function. In fact, apart from the conventional `-O0`, `-O1`, `-O2`, `-O3` or `-Os` optimization level, various optimization flags are available that are sparsely used, some of them being particularly efficient to disturb binary similarity measures, often used in reverse engineering. Such an adversarial approach against

[24] if the obfuscator directly interfaces with the compiler, such as OLLVM.

reverse engineering metrics illustrates the effect of optimization on binary code analysis.

2.4 Conclusion

This Chapter provides an overview of compiled programs, tracing the transformation from source code to binary code. It examines the compilation process in detail, highlighting the role of the IR and optimization, target architectures, and the inherent loss of abstraction that occurs during compilation. Furthermore, it explores the techniques employed in reverse engineering to reconstruct higher-level abstractions from binary code, most notably, disassembly into assembly language. To formalize the structure of binary programs, the code is decomposed into binary elements, each capturing specific aspects of the original program. These elements are commonly modeled as graphs, a format widely adopted in reverse engineering. However, such graphs are subject to transformations and syntactic variations, particularly in the presence of obfuscation techniques designed to obscure a program's true behavior. These transformations can affect various facets of the binary and may interact with or oppose compiler optimizations.

Chapter 3

Representation Learning for Binary Code Analysis

The recent advances in ML have enabled the development of automated techniques for binary code analysis, supporting both classification and clustering tasks at various levels of granularity. A common approach involves learning representations of binary code, where a given binary object is mapped to a fixed-size vector. This process, often referred to as geometric learning [7, 16, 17], seeks to embed structured data into a continuous Euclidean space, preserving essential relational and structural properties. More generally, geometric learning leverages the underlying geometry of non-Euclidean data to ensure that similar objects remain close in the embedding space, enabling more effective similarity measurements. Since binary code inherently resides in a non-Euclidean structured space and exhibits various invariances, a central challenge, both for expert-crafted and learned representations, is to produce vector embeddings that are robust to these invariances. Certain models, such as Graph Neural Networks (GNN), naturally exhibit this robustness due to their design.

Approaches for deriving these representations typically fall into two broad categories: hand-crafted features designed by domain experts and processed using traditional ML algorithms, presented in Section 3.1, and features learned directly from data using DL methods. Among the latter, representations based on NLP and graph learning have gained particular prominence and are examined in Sections 3.2 and 3.3, respectively.

3.1 Expert representations

Representing complex objects used to rely on domain experts who possess a deep understanding of the underlying data structure, its key characteristics and properties. In the context of binary code, a substantial body of literature has emerged exploring the most effective ways to represent it using these expert-designed representations [76].

Extracting features from the original binary code enables its analysis in a vector space. However, establishing a relevant set of features remains challenging, primarily

due to the inherent complexity of binary code. These features are generally grouped into several categories, many of which are reused and shared across various studies.

- **Syntactic features** refer to characteristics derived directly from the disassembled code. These include the distribution and frequency of raw assembly [125], as well as groupings of mnemonics based on expert-defined categories, such as arithmetic, logical and control transfer mnemonics [152, 54, 73].
- **Structural features** capture information from graph structures extracted from the binary. For the CFG, it includes the number of nodes, or Basic Blocks (BBs), and edges, cyclomatic complexity, betweenness centrality and loop counts [133, 76, 59, 129]. For the CG, one can extract the program callers and callees [68, 133, 25].
- **Semantic features** aim to capture the proper behavior of the binary code, often through Symbolic Execution (SE) or dynamic analysis. These features include input / output behavior, symbolic BB states or program slices [51, 108, 91, 109, 138].

Once features are extracted, whether at the BB, function or whole-program level, they are usually processed through a conventional ML pipeline. This pipeline commonly involves algorithms such as Decision Tree, Naive Bayes, Random Forests, XGBoost, or Extra-Trees[138, 129]. DL methods are seldom favored, as empirical studies have shown that they often underperform on tabular data compared to traditional ML techniques [134].

3.2 Natural Language Processing-based representations

Rather than manually defining a list of features, a process that is inherently subjective and difficult to standardize, recent approaches aim to learn representations directly through DL algorithms, eliminating the need for expert-designed input.

Disassembled code serves as a particularly suitable input for this purpose. Although it does not constitute natural language in the traditional sense, it can be effectively processed using techniques derived from NLP.

Early methods focused on extracting elementary features from textual representations of code, such as bag-of-words [118]. Subsequent approaches aimed to capture the syntactic structure of binary functions, for instance, through n-gram models computed over instruction mnemonics [13]. Similarly, techniques like Term Frequency Inverse Document Frequency (TF-IDF) are applied to the full assembly code [125], considering finer-grained assembly aspects, such as register usage and memory addresses.

Later developments demonstrated that word embedding models such as word2vec [107] can effectively capture syntactic patterns within binary code [37]. Building on the success of transformer-based architectures, models like BERT [36] have also been adapted to binary code. Typically, such models are trained or fine-tuned on assembly code and incorporate a normalization step to reduce vocabulary size, often by abstracting memory addresses, register names, and other high-variance tokens [86, 116].

3.3 Graph representations

Graph learning is a subset of the more general geometric and structured learning. It aims to represent graph data into a vectorial space. As before, traditional approaches, based on kernels for example, are supplanted by more advanced techniques, such as GNN [22, 149].

3.3.1 Random walks and kernels methods

In addition to expert-designed graph-based representations already discussed in Section 3.1, early graph learning approaches often relied on graph kernels to measure the similarity between graphs [81]. Among these, the random walk graph kernel received significant attention [56, 143, 135], laying the groundwork for subsequent advances in learning techniques that leverage random walk mechanisms.

More recent approaches have shifted toward representing graphs as ordered sequences of nodes, enabling the use of DL models to capture structural information. These approaches are often inspired by the skip-gram model from NLP, aiming to learn latent representations that preserve the graph's underlying topology. To construct these node sequences, random walks are commonly employed, ranging from truncated random walks [117] to biased ones [60].

3.3.2 Graph Neural Networks

Traditional DL models are primarily designed for Euclidean data structures, those that can be represented in regular formats such as matrices, like images, or sequential data, such as text. However, many real-world systems are more naturally modeled as graphs. Extending the DL principles to non-Euclidean domains presents both significant challenges and opportunities. First, due to the absence of a canonical node ordering in graphs, graph-based DL models must be invariant to permutations. Second, unlike images which can be resized to fit a fixed grid, as required by conventional computer vision models, graphs are inherently variable in size and structure. Therefore, graph-based DL models must be capable of handling arbitrary graph topologies, including nodes with varying degrees. Finally, these models must effectively capture both the structural properties of the graph and the individual characteristics of its nodes.

Graph Neural Networks (GNN) have emerged to address these problems. They offer a powerful framework for solving a wide range of graph-related tasks and exhibit properties particularly well suited for graph learning. GNNs have been under active development since their introduction [58], gaining significant popularity only in recent years [77]. Their effectiveness stems from their ability to jointly leverage two fundamental aspects of graphs: the graph structure, represented by an adjacency matrix, and the intrinsic features or attributes of the nodes themselves. In many graph-based domains, nodes are not merely structural entities but correspond to a real-world objects with inherent rich properties. For instance, in a social network, nodes represent users that are associated with attributes such as username, age, or interests. This principle generalizes across

various domains, including binary analysis, where graphs, such as CFG, contain nodes, or BBs (Basic Blocks), that represent sequences of instructions and carry binary data that can be used to derive relevant features. These node features aim to capture the essential characteristics of each node. Spatial GNNs, in particular, use them, together with the graph structure, to iteratively learn better node representations through a message passing mechanism.[1] It enables each node to update the initial node features by aggregating information from its neighbors. In fact, node relationships within a graph are crucial and should be considered in order to learn better node representations, which combine initial node features and the features of node neighbors. Consequently, GNNs produce more informative node representations that integrate both local features and structural dependencies.

To compute these representations, a GNN takes as input the adjacency matrix of a graph, with n nodes, and the associated feature matrix, of size (n, d), with d denoting the feature dimension and where each matrix row encodes the features of a specific node. The GNN model stacks multiple message-passing layers, also known as graph convolutions. At each layer, node representations are iteratively refined by incorporating information from both the node itself, with an arbitrary degree, and its neighbors. Between successive message-passing layers, it is possible to insert pooling operations that reduce the order of the graph by extracting subgraph information into compact representations [53, 84]. Through this iterative learning process, feature information is propagated across the graph, allowing each node to integrate context from increasingly distant nodes. The depth of the GNN model determines the extent to which a node initial feature influences other parts of the graph.

Formally, the k-th message-passing layer of a GNN is described as follows [151]:

$$a_v^{(k)} = AGGREGATE^{(k)}\left(\{h_u^{(k-1)} : u \in \mathcal{N}(v)\}\right)$$
$$h_v^{(k)} = COMBINE^{(k)}\left(h_v^{(k-1)}, a_v^{(k)}\right)$$

where $h_v^{(k)}$ is the feature associated with the node v at the k-th iteration, $h_v^{(0)} = X_v$ is the initial feature of node v and $\mathcal{N}(v)$ denotes the neighborhood of node v. It is possible to consider edge features as well [57], even though in this work, edge features are not included in the GNN design, as most of the graphs considered in this thesis do not provide any.

Finally, after K steps of node feature refinement, each node is associated with an embedding of size d', a learned vector representation of the node that both captures its structural role and its syntactic properties. A GNN schema [147] is available in Figure 3.1.

These GNNs exhibit several of the desirable properties outlined earlier. First, they are able to process graphs of varying sizes and orders, in contrast to traditional DL

[1] GNNs can be broadly categorized into two classes: spatial and spectral. In this thesis, we focus exclusively on spatial GNNs, as they have been substantially more studied and applied than their spectral counterparts. Spectral GNNs, in contrast, rely on spectral graph theory, leveraging the eigenvalues and eigenvectors of the graph Laplacian to perform convolution in the frequency domain, rather than through direct message passing between nodes [15].

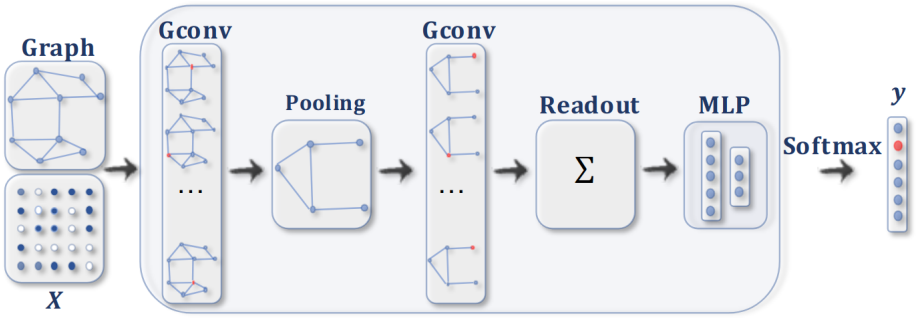


Figure 3.1: A classical GNN architecture, with convolution, and optional pooling and readout layers [147].

models, such as Convolutional Neural Network (CNN), which only operate on fixed-size input images. Second, their node embeddings are equivariant under node permutation, provided that the *AGGREGATE* and *COMBINE* functions are themselves equivariant or invariant to permutation [97].

Definition 3.3.1 (Equivariance)**.** Given a permutation g and a function f, f is equivariant with respect to g if $f(g(x)) = g(f(x))$.

In the context of GNNs, equivariance implies that permuting the nodes of a graph and then applying the GNN yields the same result as first applying the GNN and then permuting the resulting node embeddings accordingly.

These final node embeddings can be directly used for node-level tasks such as node classification. For graph-level tasks, however, a graph embedding of the graph G must be derived using a *READOUT* function that fuses all the node embeddings into a single vector representation of the graph:

$$h_G = READOUT(\{h_v^{(K)} | v \in G\})$$

Assuming that the *READOUT* function is invariant to permutations, the resulting graph embedding is likewise invariant to permutations [97].

Definition 3.3.2 (Invariance)**.** Given a permutation g and a function f, f is invariant with respect to g if $f(g(x)) = f(x)$.

Consequently, any node permutation within a graph does not alter the final graph embedding. This property is particularly valuable as it ensures that isomorphic graphs are mapped to the same representation. Among commonly used invariant *READOUT* functions, such as `sum, max, mean`, the injective `sum` function offers the strongest theoretical guarantees [151].

The *AGGREGATE*, *COMBINE* and optional *READOUT* functions differ depending on the GNN architecture employed. Graph Convolutional Network (GCN) [77] was the first practical GNN using convolutional layers. SAGE [64] improves upon GCN

	AGGREGATE	*COMBINE*	Eventual *READOUT*	Layer complexity
GCN	$a_i = \sum_{j \in \mathcal{N}(i) \cup \{i\}} \frac{e_{j,i}}{\sqrt{\hat{d}_j \hat{d}_i}} x_j$ $\hat{A} = A + I \ \hat{D}_{ii} = \sum_{j=0}^{j=n} \hat{A}_{ij}$	$h_i = \Theta^T a_i$	Any	$\mathcal{O}(\lvert A \rvert d' + ndd')$
SAGE	$a_i = W_2 \times \frac{1}{n} \sum_{j \in \mathcal{N}(i)} x_j$	$h_i = W_1 x_i + a_i$	Any	$\mathcal{O}(\lvert A \rvert d + ndd')$
GIN	$a_i = \sum_{j \in \mathcal{N}(i)} x_j$	$h_i = h_\Theta\left((1+\epsilon) \times x_i + a_i\right)$	$h_G = \sum_{n \in V} h_n$	$\mathcal{O}(\lvert A \rvert d + ndd')$
GAT	$a_i = \sum_{j \in \mathcal{N}(i) \cup \{i\}} \alpha_{i,j} \Theta_t x_j$ $\alpha_{i,j} = \frac{exp(LeakyReLU(a_s^T \Theta_s x_i + a_t^T \Theta_t x_j))}{\sum_{k \in \mathcal{N}(i) \cup \{i\}} exp(LeakyReLU(a_s^T \Theta_s x_i + a_t^T \Theta_t x_k))}$	$h_i = a_i$	Any	$\mathcal{O}(H(\lvert A \rvert d' + ndd'))$

Table 3.1: Description of different commonly used GNNs.

with a more sophisticated *COMBINE* strategy. Graph Isomorphism Network (GIN) [151] provides the most robust theoretical foundations, having been shown to match the discriminative power of the 1-Weisfeiler-Lehman (WL) test. Graph Attention Network (GAT) [142] introduces an attention mechanism into the message passing process, which can be extended to H attention heads, assigning greater importance to influential nodes. UNet [53] draws inspiration from the standard UNet architecture used in computer vision, performing downsampling and subsequent upsampling of the graph structure. These models differ in computational complexity: while GCN, GIN, and SAGE share similar theoretical complexity, even though the latter can benefit from reduced cost through node sampling during the *AGGREGATE* step, GAT generally incurs higher training overhead due to its architecture, which often incorporates multi-head attention mechanisms. A description of commonly used GNNs is presented in Table 3.1.

GNNs have demonstrated promising empirical and experimental results across a wide range of tasks. Substantial efforts have been made to theoretically characterize their expressive power. In particular, GNNs are closely related to several fundamental graph problems. An expressive and powerful GNN should be capable of determining whether two graphs share the same structure, whether their structures are similar or entirely different. Consequently, GNNs are intrinsically linked to the graph isomorphism and Graph Edit Distance (GED) problems. The graph isomorphism problem aims to decide whether two graphs are structurally identical up to a permutation of their nodes and its complexity class is still unknown. The GED, which measures the similarity between two graphs by observing how many operations are necessary to transform one graph into another, is more challenging as it belongs to the NP-hard class and is APX-hard.

For these reasons, and especially given the close relationship between the graph isomorphism problem and the theoretical GNN foundations, the expressive power of a GNN is often evaluated based on its ability to distinguish two non-isomorphic graphs, which differs from traditional expressive power definitions used for other DL models.

In this context, GNN expressivity is commonly compared to the 1-WL test, a recognized heuristic used to test graph isomorphism. Standard GNNs are generally at most

as powerful as the 1-WL test.[2]

Under certain assumptions, it is possible to design a GNN that is as powerful as the 1-WL test [151]. Although the 1-WL test is generally as powerful as, or more powerful than, existing GNNs in distinguishing non-isomorphic graphs, it operates purely in a discrete space, providing a binary indication of graph isomorphism. In contrast, GNNs learn continuous vector representations of graph structures, which can be leveraged for a wide range of downstream tasks.

Despite their interesting properties, GNNs exhibit several notable limitations. One critical consideration is the choice of the GNN depth. Unlike CNNs, which can be very deep with thousands of layers, GNNs usually contain a limited number of layers. Excessive depth leads to the phenomenon known as oversmoothing, where, after a few convolutional layers, the initial node signals are progressively diluted into the graph, causing node features to converge to an indistinguishable constant vector. This effect results from repeated aggregation and combination steps across many layers, significantly impairing the model's ability to learn effective node embeddings [123]. Moreover, GNNs are inherently limited in their computational expressiveness. They are provably incapable of approximating certain graph functions. For instance, GNNs cannot compute the number of cycles within a graph [23]. The GNN performance is also highly sensitive to the quality of initial node features. Ideally, these features should accurately describe the nature of each node. However, in many practical scenarios, users may struggle to design such meaningful features. Empirical evidence indicates that GNN performances vary significantly with the quality of these initial node features, which have a substantial impact on training [44]. Additionally, prior studies have highlighted that GNNs do not always outperform simpler, less computationally expensive baseline methods [44, 47]. Therefore, it is generally advisable to include such baselines for comparison. Finally, GNNs are prone to significant instability during training, which can arise from factors such as weight initialization and data splits [78, 132]. This instability can hinder consistent training and affect overall model performance.

3.4 Conclusion

This Chapter introduces an overview of representation learning for binary code. It emphasizes the non-Euclidean nature of binary code, highlighting the need for suitable geometric learning methods, to derive representations that are robust to inherent invariances. It begins by examining expert-designed representations commonly used to describe program behavior, which are processed using classical ML algorithms. It then explores the growing field of representation learning, focusing on approaches inspired by

[2]The 1-WL test is a widely adopted algorithm for graph isomorphism testing that relies on iterative refinement of node labels. While effective in distinguishing many non-isomorphic graphs, it cannot separate all graph structures. To address this limitation, higher-order WL tests have been proposed, which operate on tuples of nodes rather than individual nodes. These higher-order tests capture more complex structural patterns, enabling the discrimination of graphs that 1-WL cannot distinguish. Building on this principle, higher-order GNNs have been developed to outperform the expressive power of classical GNN, though they are less commonly employed in practice due to increased computational complexity.

both NLP and graph-based techniques. These representations have evolved considerably, from simple n-gram models to transformer architectures, and from random walk-based methods to GNNs. Such models effectively capture the syntax and structure of binary code and support a wide range of downstream tasks.

Chapter 4

How Machine Learning is solving binary analysis problems

The increasing sophistication of ML models, particularly those focused on representation learning, as discussed in Chapter 3, has led to the emergence of specialized approaches tailored to binary code analysis. Among the most promising applications of these techniques are three problems explored in this thesis: the binary similarity problem, presented in Section 4.1, binary diffing, detailed in Section 4.2, and obfuscation analysis, described in Section 4.3. These problems are closely interconnected, often sharing methodological foundations and challenges, with breakthroughs in one area frequently informing and benefiting the others.

4.1 Binary similarity

The binary similarity problem aims to compare two pieces of binary code and assigning them a similarity score that reflects their degree of resemblance. This similarity measure should capture the semantic proximity and two pieces of code exhibiting similar behaviors should receive a high similarity score. Binary similarity constitutes a fundamental problem across various applications, as it provides a meaningful metric for evaluating numerous use cases in reverse engineering:

- **1-day vulnerability detection.** Given a binary function known to contain a specific vulnerability, it is valuable to compare it against a large set of other functions. If a high similarity score is observed between the vulnerable function and another function, this may indicate the presence of the same vulnerability in the latter. This approach is widely employed for 1-day vulnerability discovery, where, following the disclosure of a new (0-day) vulnerability, attackers attempt to identify instances of the same vulnerable code in a broad range of programs that have not yet been patched. Moreover, this vulnerability search can be extended to cross-platform analysis, enabling the detection of similar vulnerabilities across different architectures or operating systems, or to variant analysis that seeks to detect

vulnerable variants of existing 1-day vulnerabilities.

- **Malware analysis.** Given a known malware sample, binary similarity can be employed to detect new variants, including obfuscated or repackaged versions of the same malware. This is achieved by identifying similar payload constructs, as malware samples usually share a subset of core functions that implement common malicious behaviors.[1] Ultimately, binary similarity analysis can support the construction of malware hierarchies or lineages by clustering related samples into families and inferring their chronological development.

- **Patch analysis.** In contrast to vulnerability detection, binary similarity assists in propagating a newly developed patch across a range of binaries that implement the underlying vulnerability. This creates a race between adversaries seeking to exploit the vulnerability and developers working to deploy fixes at scale.

- **Clone analysis.** Detecting code clones across different software projects is valuable for ensuring license compliance and investigating potential violations of intellectual property.

- **Reverse engineering.** Binary similarity technique can be employed to ease automated analysis of binaries. For instance, information from a known binary can be transferred to an unknown one, such as inferring function names based on similarities or identifying statically linked library functions embedded within a binary.

Binary similarity is an active area of research that increasingly leverages ML. Two main approaches are distinguished; traditional ML that processes hand-crafted features and Deep Learning (DL) that learns representations directly from data.

The earliest approaches typically rely on a predefined set of hand-crafted features. These features are either ranked, with the top-ranked ones fed into a Decision Tree to produce a final similarity score [133], or directly compared using specific distance metrics to evaluate their significance [76].

The second approach places increasing emphasis on DL and tends to dominate the research field. Within this domain, two sub-methods have emerged, one inspired by NLP and the other based on GNNs.

NLP-based approaches explore a broad spectrum of techniques aimed at embedding binary functions into continuous vector spaces, enabling the similarity approximation between different functions. For instance, Asm2Vec [37] builds upon an enhanced version of the word2vec model. PalmTree and Trex [86, 116] draw inspiration from the recent successes of transformer architectures in large language models. Specifically, they adapt well-known transformer models, like BERT, and pre-train them on dedicated assembly data through methods like multi-task learning [86] or Masked Language Model [116]. These methods often entirely neglect [90, 99] or only partially consider [37] control-flow

[1] Highly sophisticated and novel malware are relatively rare and often attributed to state-sponsored actors.

information, primarily relying on assembly code.[2] Recent research efforts are beginning to incorporate graph-based information by integrating CFG data, including both inter-procedural or intra-procedural relationships, directly into transformers [145].

GNNs are also gaining popularity, with recent research articles favoring increasingly sophisticated GNNs. Graph Matching Networks (GMN) [87] introduced the concept of jointly learning graph embeddings for pairs of similar graphs, rather than computing independent embeddings. More recent works move away from hand-crafted initial GNN features [87], instead leveraging pre-trained language models on assembly instructions to obtain BB embeddings, which are then used as initial features for GNNs [98]. The idea is further developed with refined GNN architectures or language models [156, 141, 52].

Beyond NLP and GNN methods, alternative representation learning techniques have also been explored, such as `structure2vec` [29] that builds on expert hand-crafted features [152, 54].

Among the broad spectrum of research on binary similarity, several studies have focused their analysis on specific problem aspects. Binary analysis in a cross-compiler, cross-architecture and cross-optimization setting represents some of the most widely explored subjects in the field [37, 116, 156, 95, 76]. Some studies analyze model explainability and feature importance [76] or address the robustness of models against obfuscated binaries on a limited set of obfuscations [37, 76, 133, 116, 69], highlighting the inherent limitations of intra-procedural obfuscation [158].

More recent efforts are shifting toward harder edge cases, such as function inlining similarity, often ignored or neglected by standard binary similarity papers [71, 72], or toward novel graph representations that jointly encode data and control-flow information [67].

Due to the proliferation of papers on the subject, many of which offer comparable methodologies and performance, making direct comparisons challenging, some studies aim to standardize evaluation. These works propose unified experimental setups that include consistent datasets, data splits, and evaluation metrics, which are then reused in subsequent research [95, 76]. These frameworks for testing are further used in other papers [71, 67] to ensure fair and reproducible comparisons.

Finally, it is worth noting that many of these models are vulnerable to adversarial attacks, highlighting the need for further robustness-focused research in binary similarity analysis [20].[3]

[2] Instruction sequences generated with random walks may miss some key aspect about the CFG structures [37].

[3] Adversarial attacks in the context of binary code differ significantly from traditional attacks on simpler data types, such as images. In particular, adversarial modifications must produce a valid, executable binary, which imposes strict constraints on the transformations that can be applied. Unlike images, where adding imperceptible noise preserves the validity of the input, perturbations to a binary program must preserve its syntax validity, semantics correctness, and execution behavior, making the design of effective adversarial attacks considerably more challenging.

4.2 Binary diffing

The binary diffing problem shares common aspects with the binary similarity problem but achieves a distinct goal. Formally, it can be defined as follows:

Definition 4.2.1 (Binary diffing problem)**.** Given two binaries, respectively denoted as primary and secondary, binary diffing aims to find a one-to-one assignment ϕ: (S_p, S_s) $\rightarrow \rho$ where S_p is the primary function set and S_s is the secondary function set. ρ: S_p $\rightarrow S_s$ is an assignment function both partial and injective, so that each function of S_p should be matched to at most one function of S_s.

Binary diffing aims to determine a correspondence between the elements of two binary programs and to highlight their similarities and differences. Semantic equivalence between the two programs is not a strict requirement for effective diffing. Minor variations, such as small patches or differences introduced by compiler optimizations or version updates, can still be captured by robust binary diffing techniques.

Since binary programs are commonly represented as graphs, particularly through their CG, binary diffing fundamentally relates to several distinct graph-theoretical problems.

- The most general problem formulation applicable to binary diffing is the Graph Edit Distance (GED) problem. GED seeks the minimum-cost sequence of edit operations (node/edge insertions, deletions, substitutions) required to transform one graph into another. Within the context of binary diffing, this naturally models the semantic differences between two programs. However, GED is known to be NP-hard and even APX-hard.

- The Network Alignment Problem (NAP) aims to correctly align nodes that represent the same entities across different networks [43, 136]. Solving NAP generally involves taking into account both node similarities and structural consistency at local and global levels. Under certain conditions, NAP has been shown to be equivalent to GED [103], which implies that it is also NP-hard and APX-hard.

Since both GED and NAP are computationally intractable for exact solutions on large instances, practical approaches rely on approximations. A prominent relaxation is the assignment problem, which seeks the minimum-cost correspondence between two sets of elements. In the binary diffing setting, binary similarity tools, presented in Section 4.1, provide similarity scores between program elements; these scores populate a similarity matrix that serves as a cost matrix for the assignment. The resulting assignment can then be interpreted as the diffing output. Unlike GED or NAP, however, the assignment problem does not account for graph topology and relies solely on similarity scores. It admits exact polynomial-time solutions and one classical approach is the Hungarian algorithm, with cubic complexity. Alternatively, auction algorithms, widely applied in other domains, such as optimal transport, can also be employed [8]. Although their theoretical complexity exceeds cubic time, empirical studies have often demonstrated

superior runtime performance compared to the Hungarian method in many practical instances.

Binary diffing usually operates at the function level, even if other definitions extend it to finer program granularities, such as Basic Block (BB) [39], or even to one-to-many assignment [79]. It must be performed without having access to symbols or sources; while existing differs sometimes rely on available function names, this approach mostly fails in real-world scenarios, where binaries are usually stripped. The detailed list of available binary differs and their characteristics is displayed in Table 4.1.

Two of the most widely used static diffing tools are BinDiff [49, 41] and Diaphora [79]. BinDiff operates by first identifying common imported functions, then propagating matches through the CG based on function similarity. Diaphora establishes a set of heuristics used to iteratively match available functions, ranging from high-confidence heuristics, such as function address and name, assembly and function hash, to low-confidence ones, like metrics related to the CG. Both tools rely solely on static analysis, requiring disassembly as a starting point. As such, they are inherently limited by the quality and reliability of static features, which are vulnerable to runtime modifications.

QBinDiff [104, 105] was developed to address several limitations of traditional binary differs. Unlike BinDiff, which offers limited tuning capabilities, except for modifying the function or BB order strategies, and Diaphora, which is challenging to configure due to the multitude of heuristics, some considered reliable and others not, for which there is no clear study of their impact, QBinDiff allows fine-grained control over the diffing process. This modularity enables reverse engineers to tailor the diffing depending on the binaries and their goals, including diffing obfuscated code.

The aforementioned diffing techniques predominantly rely on syntactic features extracted from disassembled code, such as instruction distribution or graph-based features. However, these features are susceptible to transformation, either due to optimization and obfuscation, and may not clearly reflect the true semantics of a binary program. To address this, semantic-based binary diffing has been developed [51, 108, 91, 109]. These methods aim to simulate the true semantics of a binary code, usually at the BB level, by modeling input / output relationships with Symbolic Execution (SE). Similarity is then assessed via symbolic equivalence checking and theorem proving. These semantic features are either static, used to match blocks and functions using custom graph isomorphism algorithm [51] or paths with the longest common subsequences of semantically equivalent blocks [91], or dynamic, that rely on taint analysis and execution traces to narrow the candidate matches [108, 110].

Despite their improved robustness, semantic-based techniques are often computationally expensive. Ongoing research is focused on reducing this overhead through optimization strategies [109].

While the majority of existing binary differs are designed for conventional binary executables, the need to analyze alternative formats has led to the development of domain-specific tools, adapted for Android APKs for example [32]. These tools often adapt traditional diffing algorithms, reusing BinDiff strategies for instance, to the structural characteristics of Android applications.

Differ name	Status	Analysis	Input	Granularity	Matching strategy
BinDiff	Usable	Static	BinExport	Function BB Instruction	Anchoring Similarity match propagation
Diaphora	Maintained	Static	Custom	Function BB Instruction	Heuristic-based priority matching
QBinDiff	Maintained	Static	BinExport Quokka	Functions	Belief-propagation
BinSim	N/A	Dynamic	N/A	Traces	Trace equivalence checking

Table 4.1: Existing binary differs and their characteristics.

Another emerging direction involves cross-modal binary-source matching. This line of work aims to bridge the semantic gap between compiled binaries and their original source code, using advanced cross-modal representation learning techniques [155].

Finally, akin to binary similarity, adversarial approaches aim to develop techniques specifically intended to hinder binary diffing tools. These approaches introduce obfuscation methods that disrupt the core elements upon which binary diffing relies [158], CG in particular.

4.3 Obfuscation analysis

The objective of software protection is to ensure the best protection possible against reverse engineering. Conversely, an attacker tries to weaken the applied obfuscation or to remove it. This cat and mouse game leads to the creation of new obfuscation attacks to withstand increasingly sophisticated attacks. The study of obfuscation has led to a large academic literature and practical works. Several efforts have been made to formally define the properties of obfuscation, as discussed in Section 4.3.1, though these definitions exhibit notable limitations. In parallel, evolving analytical and offensive techniques, detailed in Section 4.3.2, have been devised to target each of these security properties.

4.3.1 Obfuscation properties

Although the primary objective of obfuscation is to protect a program and hinder its exploitation by adversaries, it lacks a formal mathematical foundation or precise definition of what constitutes effective protection. Consequently, several taxonomies have been proposed to define criteria for assessing the quality of an obfuscation. Common metrics include potency, resilience, stealth, and cost [28]:

- **Potency.** It represents the ability of an obfuscation to resist human-assisted attacks. Originally, it was assessed using a variety of program complexity metrics

[28], like the cyclomatic complexity [101] or nesting complexity [65]. The obfuscation precisely aims to modify these features by injecting artificially complex conditional branches, loops or operations, intended to mislead a reverse engineer during manual examination of binary code. However, such metrics often fail to account for the expertise of professional reverse engineers, who may easily circumvent these techniques. As a result, these measures may offer an unreliable or incomplete evaluation of an obfuscation correct potency.

- **Resilience.** It illustrates the capacity of an obfuscation to resist automated attacks, such as Symbolic Execution (SE) or emulation. Quantifying this property through a unique metric remains challenging and several works have proposed various approaches to describe it. Numerous features have been derived as resilience estimates, such as the Kolmogorov complexity [112]. More realistic approaches have emerged, based on empirical evaluation, such as the time or computational effort required to successfully deobfuscate a program, using in particular SE [4]. This study has been extended by predicting deobfuscation time in advance using ML techniques [131], enabling the estimation of resilience without executing a full attack.

- **Stealth.** It denotes the difficulty of identifying or characterizing obfuscation code within a program. In fact, distinguishing a perfectly legitimate function that naturally includes many complex operations from an obfuscated function is often challenging, especially if the obfuscated patterns are well concealed. Figure 4.1 illustrates this ambiguity and explains why non-experts may be misled by an obfuscated function.

- **Cost.** It refers to the computational overhead added by an obfuscation. Since simple code sequences are often replaced with more complex constructions, the resulting binary tends to increase in size. As a consequence, executing the obfuscated program implies a burden both in terms of memory consumption and runtime. Unlike potency, resilience and stealth, cost is not a security property but a performance metric. Ideally, an obfuscation should maximize potency, resilience and stealth while keeping the cost at an acceptable budget. This is explained by either memory or computational constraints, common in embedded systems, where MCUs have limited storage and processing power. In this context, only a small subset of critical functions is obfuscated for security.

Recently, a new evaluation framework has been introduced to address the limitations of earlier definitions [34]. While our work adheres to the original taxonomy [28], we aim to highlight some of these limitations already pointed out [34], as they are illustrated later in the experimental part of this thesis.

Potency and resilience tend to be confused together, as real-world attacks against obfuscation rarely rely solely on manual or automated analysis. In practice, they often result from a combination of both approaches, with reverse engineers increasingly leveraging automated tools such as decompilers or ML techniques [139, 55, 38]. Conversely,

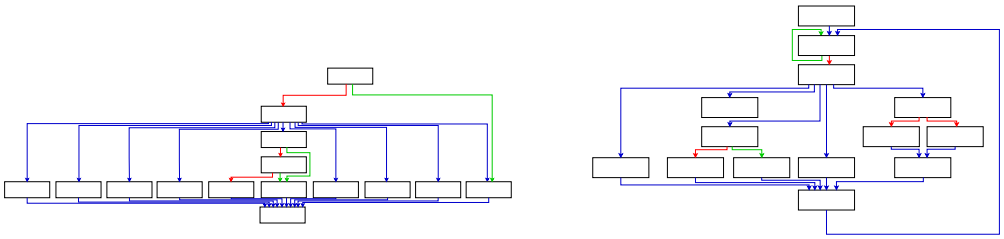


Figure 4.1: The left CFG originates from the function `numarith` embedded into the `minilua` project, the right CFG represents the `minilua genlink` function obfuscated with Control Flow Graph Flattening (CFF). Distinguishing them is tedious without expert knowledge.

most automated attacks usually assume prior knowledge about the obfuscation, which is frequently obtained through a preliminary manual analysis.

Second, these definitions describe too narrowly the problem of obfuscation properties. Other obfuscations security concerns, such as confidentiality or exact program property computation [30], are often omitted by these definitions. This limitation stems from the inherently multi-faceted nature of obfuscation efficacy. Since an obfuscated program must ultimately reveal some behavior during execution, it inevitably exposes an attack surface, making complete protection unachievable. Consequently, a variety of attacks have been developed, each targeting a specific facet of the available attack surface.

4.3.2 Attacking obfuscation

Attacks on obfuscation typically rely either on human-assisted approaches, often based on reverse engineering and primarily aimed at countering obfuscation potency, or on automated techniques designed to compromise its resilience. In contrast, obfuscation detection focuses on the stealth aspect of obfuscation and has attracted growing attention in recent research.

- **Human-assisted attacks against potency.** As mentioned before, purely manual attacks against obfuscation are rarely employed. Semi-manual approaches usually consist in manually detecting the obfuscation and understanding how it is employed, before programming a specific attack algorithm. For instance, in case of CFF, the dispatcher mechanism is first reversed in order to reconstruct the original control-flow [140, 157]. Virtualization is defeated by manually reversing VM handlers and bytecode before writing the corresponding disassembly [12, 122].

- **Automated attacks against resilience.** The resilience property is mainly challenged by automated attacks aiming to weaken or eliminate obfuscation. Automated deobfuscation algorithms, which fall into this category, are designed to target specific obfuscation techniques, exploiting inherent weaknesses in their design.

Most existing deobfuscation approaches operate at the function level and assume prior knowledge of both the presence and type of obfuscation applied. Common targets include MBA [14, 45, 31, 11, 106, 120], Virtualization [126, 106], Opaque Predicates [139] and control-flow obfuscations [153, 96]. Data deobfuscation, especially MBA and, to a lesser extent, some intra-procedural obfuscations, are particularly prevalent in the literature. Some studies, however, can be employed against multiple obfuscations applied simultaneously [153, 106, 96]. These deobfuscation algorithms adopt distinct strategies. Some leverage formal mathematical formulation of the obfuscation, in particular MBA, to develop algebraic attacks [88, 150, 89, 119, 120]. In contrast, many studies consider the deobfuscation problem as a specific instance of the broader program synthesis problem, wherein the objective is to generate semantically equivalent but structurally simpler code. To explore the large search space, these algorithms employ techniques such as Monte-Carlo Tree Search (MCTS) [14], bottom-up iterative search [31, 11] or Iterated Local Search [106] and are often guided by auxiliary data, like input / output, to synthesize the program. Alternatively, expert knowledge is used to design targeted attacks. For instance, in the case of general intra-procedural control-flow obfuscation, trace-based pruning of control-flow skeletons has been shown to effectively simplify any intra control-flow obfuscations [96]. Additional techniques, such as taint analysis, sometimes combined with trace simplification or Symbolic Execution (SE), have also demonstrated efficacy [153, 126]. More recently, ML techniques have been employed to defeat obfuscation mechanisms. For instance, a MBA can be considered as a textual sequence and processed using a Long Short-Term Memory (LSTM) which, augmented with attention, learns to map the MBA to its simplified equivalent [45]. Similarly, Opaque Predicates can be expressed as symbolic states that are analyzed using statistical analysis, such as term frequency-based features, for which Decision Tree is trained to infer the underlying boolean value, either True or False [139].

- **Obfuscation detection against stealth.** Obfuscation stealth is a multifaceted problem, each aspect of which has led to dedicated research efforts. Detection granularity vary, ranging from the program itself [59, 125, 138] to the function level [73, 9, 129]. Since each obfuscation pass targets a specific binary code aspect, such as MBA modifying binary instructions only at the BB level, detection performances are contrasted depending on the obfuscation type being analyzed. Naturally, detecting whether individual functions are obfuscated provides insight into whether the entire program is obfuscated. The problem behind obfuscation stealth is often modeled as a binary classification task [13, 73, 10], where the goal is to determine whether a given binary object, program or function, is obfuscated or not. Extended works aim instead to characterize the obfuscation [59, 73, 138], seeking to determine the specific obfuscation techniques that were applied. While some studies assume that only a single obfuscation is used at a time [125, 59], others extend this principle by combining different obfuscation passes. In such cases, these combined obfuscations are either considered as entirely new obfusca-

tion classes [129] or extend the task to a multi-class and multi-label problem [138]. Such a problem formulation is more flexible and allows a precise identification of potentially arbitrary obfuscation combinations. These latter approaches align more closely with real-word scenarios, such as malware or Digital Rights Management (DRM) protection, [13, 9, 10], where obfuscation schemes are rarely limited to isolated obfuscation passes. Rigorously evaluating the assessed techniques is consequently more challenging in this real-world context, as building a reliable ground truth requires an extensive reverse engineering effort. Effectively addressing the obfuscation identification and characterization problems thus requires a thorough binary code description, as well as its obfuscated components. Substantial research has been devoted to select the most appropriate features to this end. In particular, expert representations, already mentioned in Section 3.1, try to encompass both structural and syntactic attributes and are consequently particularly studied. CFG structural features include code complexity metrics, such as the cyclomatic complexity [59, 129], loop count [59] or the ABC and Halstead metrics [129], as well as expert-derived features involving dominance relations between Basic Blocks (BBs) [9] or the presence of overlapping instructions, often indicative of obfuscation [10]. Statistical features comprise the average BB size per function [10], mnemonic distribution indicators [59], assembly mnemonic n-grams [13], assembly TF-IDF [125], custom mnemonic classes counting features [73] and bag-of-word based on SE expressions [138]. These features serve as a basis for heuristic-based algorithms that usually assign an obfuscation score to a binary code [13, 10, 9], or a n-gram based artificiality metric [75], which reflects how likely the code is obfuscated and how. Most advanced research has increasingly turned to ML techniques, especially classical ones. A wide range of classifiers have been comparatively evaluated, including Decision Tree, Naive Bayes, Support Vector Machine (SVM), Extra-Trees, Light Gradient Boosting Machine (LGBM) or Multi Layer Perceptron (MLP). More advanced DL algorithms are also considered, such as GCN, that has been combined with LSTM, which operates on the resulting GCN graph embedding [73], an approach whose rationale is unusual. Some studies have placed a particular emphasis on ensemble learning implications [138], optimization effects on obfuscation [129, 59], or explainability, provided by classifiers such as XGBoost[59], where the relative importance of the features is analyzed. Some studies restrict themselves to a single architecture [73], leveraging domain-specific knowledge, while others extend their existing work with architecture-agnostic approaches [13], for example by operating at the level of IR. The existing literature relies heavily on publicly available obfuscators, particularly OLLVM and Tigress. OLLVM has been extensively studied [129, 59, 138, 73], due to its simplicity of use. In contrast, deploying Tigress on real-world projects is more challenging. As a result, associated experimental studies are based on small and synthetic code snippets rather than realistic and large projects [125, 138].

4.4 Conclusion

This Chapter presents how ML solves several fundamental problems in binary code analysis. In particular, binary similarity, binary diffing, and obfuscation analysis have emerged as key challenges in the field of reverse engineering. Recent advances in representation learning have led to notable improvements in tackling these tasks. At the core of these approaches lies the extraction of meaningful representations that capture syntactic, structural, and semantic aspects of binary code. Such representations serve as the foundation for computing similarity scores, performing binary diffing, and analyzing or circumventing the security properties introduced by obfuscation. Given their central role across such a wide range of tasks, they exert a significant influence on recent binary program analysis.

Chapter 5

Building and collecting realistic, real-world and obfuscated program datasets

This Chapter places particular emphasis on the datasets employed throughout this thesis. These encompass both standard, unobfuscated binaries and obfuscated ones, including synthetic datasets, constructed specifically for academic purposes, as well as in-the-wild data, drawn from real-world binaries available online. A central contribution of this work is the creation of a novel and extensive obfuscation dataset, ObfuBench, introduced in Section 5.1. In addition, we leverage established datasets, either standard or obfuscated, that have demonstrated their value in prior research, such as BinKit (Section 5.2), the binary similarity Dataset-1 (Section 5.3) or the Loki MBA dataset (Section 5.4), to further support the generalizability of our findings. Finally, we also include in-the-wild binaries, presented in Section 5.5, to strengthen the validity of our approaches. An overview of all the datasets considered in this thesis is provided in Table 5.1.

	Standard	Obfuscated	Purpose	Used in
ObfuBench	✔	✔	Any obfuscation-related task	Obfuscation detection (Chapter 6) Diffing obfuscated binaries (Chapter 7)
BinKit	✔∗	✔	Binary similarity	Diffing obfuscated binaries (Chapter 7)
Dataset-1	✔	✗	Binary similarity	Ablation study (Chapter 7) Diffing standard binaries (Chapter 7)
Loki MBA dataset	✗	✔	Any MBA-related task	Deobfuscation (Chapter 8)
Real-world binaries	✗	✔	Any obfuscation-related task	Obfuscation detection (Chapter 6) Diffing obfuscated binaries (Chapter 7)

∗ Binkit contains standard binaries but they are not used in this thesis.

Table 5.1: A summary of all the data used in this thesis.

5.1 ObfuBench: a new large, realistic and enriched synthetic obfuscated dataset

Research on native binary obfuscation has been significantly hindered by a critical limitation: the absence of any large-scale datasets of labeled, obfuscated program binaries. In fact, learning binary representations often requires ground truth information, including details about which obfuscation passes have been applied and how. However, obfuscated binaries found in the wild, such as malware samples or protected legitimate applications, are difficult to label reliably, as it implies considerable manual reverse engineering and introduces potential bias. In contrast, existing synthetic obfuscated datasets suffer from their own drawbacks. Many are restricted to binaries obfuscated using OLLVM, which is based on the outdated LLVM-4[1], and only propose a narrow set of data and intra-procedural transformations [76, 59, 37, 73]. Furthermore, these datasets frequently consist of simple code snippets, such as sorting algorithms, that fail to imitate the complexity, both structural and semantic, of real-world `C` programs [55, 14]. As a result, models trained on such datasets may struggle to generalize to more realistic and diverse program scenarios, limiting their applicability to practical binary analysis tasks. Others lack the scale necessary for training data-intensive models [4, 14], particularly DL architectures that require substantial volumes of diverse examples.

These limitations primarily stem from the small number of free and open-source obfuscation tools. Among those that do exist, many are either difficult to configure or lack the flexibility needed to generate diverse and well-controlled obfuscation scenarios. To address this gap, and inspired by the state of the art dataset for binary similarity research [95], a new large-scale synthetic dataset of realistic obfuscated binaries, called ObfuBench, was created[2]. It includes original unobfuscated binaries, their obfuscated counterparts, and detailed ground truth metadata specifying the applied obfuscation techniques. Tables 5.2 and 5.3 provide an overview of ObfuBench key characteristics. Further details are provided in Table 6.2 of Chapter 6.

	zlib	lz4	minilua	sqlite	freetype
Source code size (lines)	9,702	7,810	28,989	243,514	166,734
Number of functions	184	240	1,299	2,505	1,592
CFG average size	23.29	92.91	9.09	19.03	21.34
CFG average order	16.95	66.62	7.29	13.99	16.03
CFG average node degree	1.92	1.31	1.67	1.86	1.9

Table 5.2: ObfuBench project overview.

Two obfuscators are evaluated, Tigress-3.1 [27] and OLLVM [74] on five realistic projects written in `C`: `zlib`, `lz4`, `minilua`, `sqlite` and `freetype`. For each obfuscator, a variety of obfuscation passes are selected, including intra-procedural, data, and, when available, inter-procedural. To better reflect real-world scenarios, where multi-

[1] released in 2017.

[2] https://github.com/quarkslab/diffing_obfuscation_dataset

	Passes	Pass type	zlib	lz4	minilua	sqlite	freetype
Tigress	Copy	Inter	✔	✔	✔	✔	✔
	Split	Inter	✔	✔	✔	✔	✔
	Merge	Inter	✔	✔	✗	✗	∼
	CFF	Intra	✔	✔	✔	✔	✔
	Virtualize	Intra	✔	✔	∼	∼	✗
	Opaque	Intra	✔	✔	✔	✗	∼
	EncodeArithmetic (Enc.A)	Data	✔	✔	✔	✔	✔
	EncodeLiterals (Enc.L)	Data	✔	✔	✔	✔	✔
	Mix	Intra & Data	✔	✔	✔	∼	∼
	Mix + Split	All	✔	✔	✔	∼	∼
OLLVM-14	CFF	Intra	✔	✔	✔	✔	✔
	Opaque	Intra	✔	✔	✔	✔	✔
	EncodeArithmetic (Enc.A)	Data	✔	✔	✔	✔	✔
	Mix	Intra & Data	✔	✔	✔	✔	✔

Table 5.3: ObfuBench dataset overview. ✔ compilation success, ✗ no binaries, ∼ some binaries are not available *(depends on obfuscation level or random seed used).*

ple obfuscations are combined to enhance security, certain passes are also associated to create new obfuscation composite classes. In particular, Mix consists of a sequential combination of CFF, EncodeArithmetic, and OpaquePredicates. When feasible, it is extended with the Split pass to form the Mix + Split class. This obfuscation scheme reflects a realistic adversarial scenario in which all program aspects, such as assembly code, CFG, and CG, are significantly modified.

For each project and obfuscation pass, functions are iteratively obfuscated, ranging from 10% of the functions to 100%, in increments of 10%. Due to the inherent randomness of most obfuscation techniques, this process is repeated 10 times with different seeds. The resulting dataset comprises 6,718 binaries and a total of 8,910,962 functions, representing the first publicly available dataset of realistic obfuscated binaries at this scale.

The binaries are compiled for the `x64` architecture using the `-O0` and `-O2` optimization levels, resulting in valid and executable programs. The `-O0` setting disables most compiler effects that could attenuate obfuscation, thus facilitating the analysis of the obfuscations in their unaltered form. Conversely, `-O2`-optimized binaries better reflect real-world conditions, despite potential biases due to the eventual obfuscation alteration or removal that may result in discrepancies between the obfuscated ground truth and the corresponding unobfuscated functions in practice.

To ensure reproducibility and maintain a reasonable computational cost, we limit the exploration of the obfuscation pass parameters to the default obfuscator parameters. Then, default OLLVM and Tigress parameters are used, except for the Tigress `Split` obfuscation pass, where SplitCount is set to 2, and only the top, block, and deep strategies are enabled via the SplitKinds option. Nonetheless, this setting yields a sufficiently large dataset comprising over 6,700 obfuscated binaries.

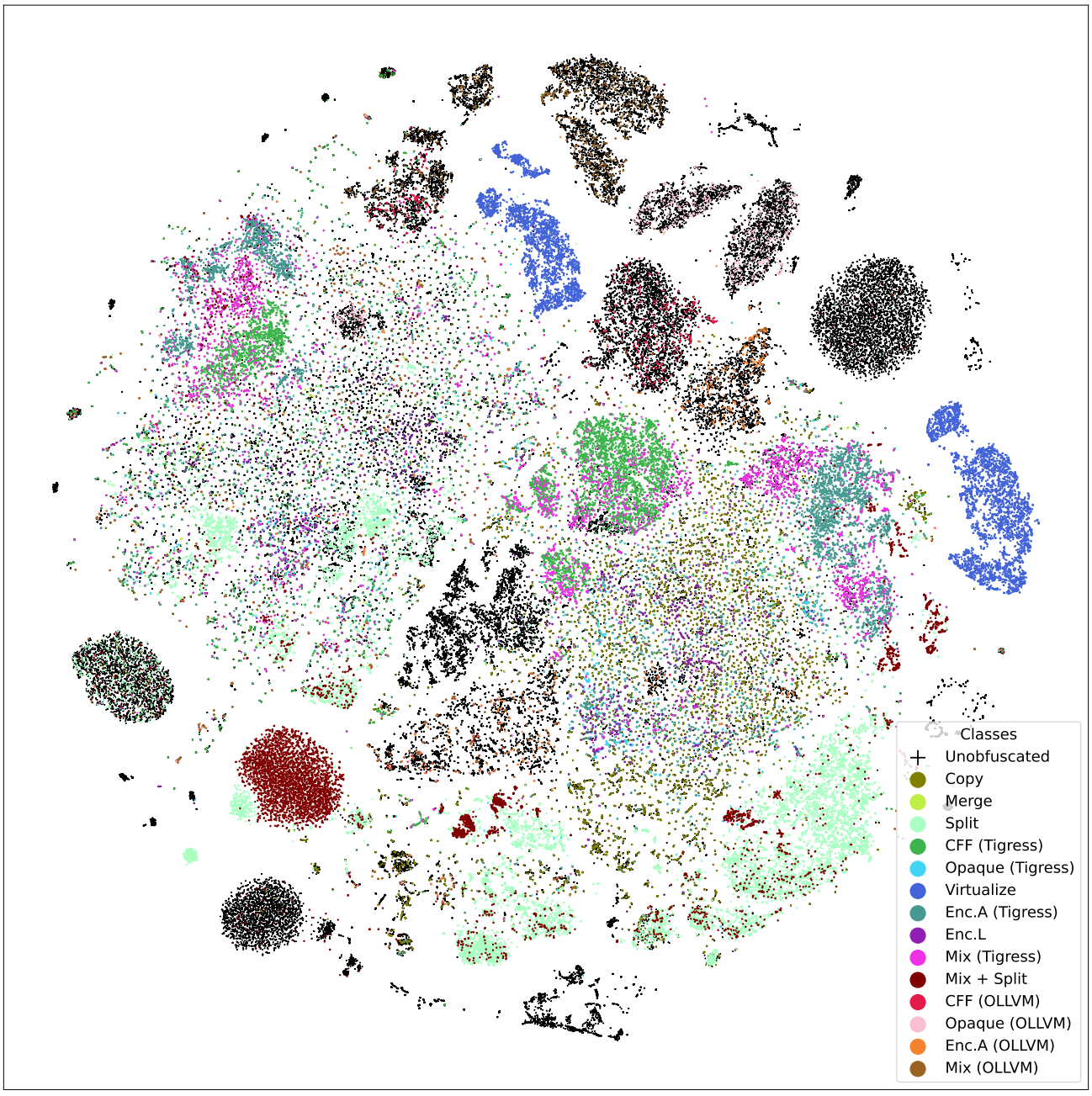


Figure 5.1: A t-SNE visualization, based on the obfuscation classes, of the 128-sized PalmTree embeddings of the ObfuBench functions. The functions are drawn from the standard binaries (unobfuscated) and from the 100%-obfuscated binaries, restricted to one seed.

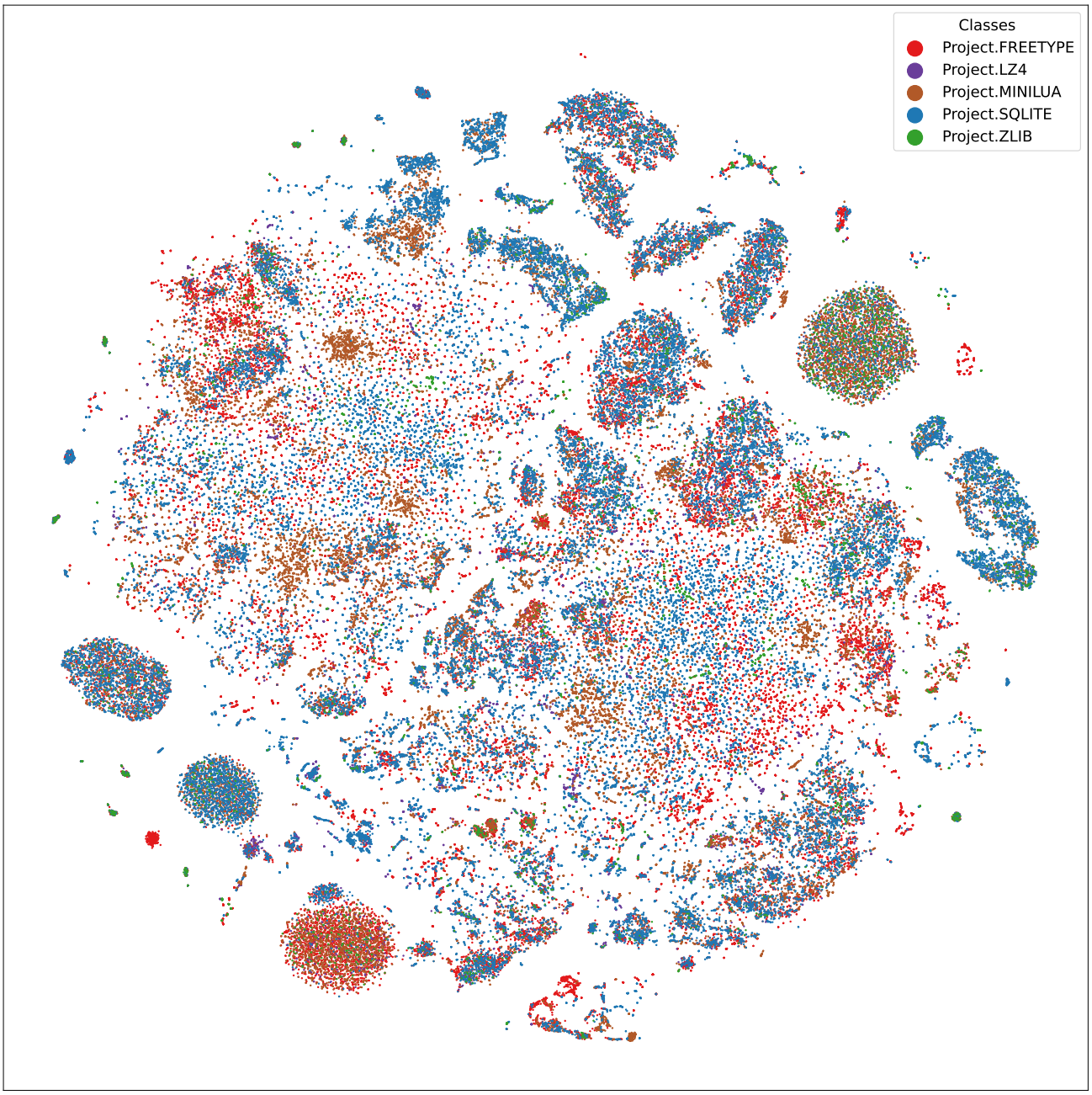


Figure 5.2: A t-SNE visualization, based on the projects, of the 128-sized PalmTree embeddings of the ObfuBench functions. The functions are drawn from the standard binaries (unobfuscated) and from the 100%-obfuscated binaries, restricted to one seed.

Visualizing a dataset can provide valuable insights into the structure and distribution of the data. In this context, both the unobfuscated functions, from the standard binaries,

and their obfuscated variants, from the 100%-obfuscated binaries (restricted to a single obfuscation seed) from the five projects in the dataset are processed using PalmTree [86] to generate 128-dimensional embeddings. These embeddings are then visualized in two dimensions using the T-distributed Stochastic Neighbor Embedding (t-SNE) algorithm [92], using both the obfuscation classes and the project name as labels. The resulting point clouds are respectively shown in Figures 5.1 and 5.2. Several distinct clusters can be observed, particularly among unobfuscated functions, some of which form clearly isolated groups. Likewise, functions obfuscated using Tigress transformations such as Control Flow Graph Flattening (CFF), Virtualization, and Mix + Split appear well separated from the rest of the data. In contrast, other obfuscation techniques, such as Tigress Copy or Merge, or those from OLLVM, result in data points that are more widely dispersed across the space, without forming noticeable patterns. Likewise, the per-project function visualization do not exhibit any well-defined clustering.

While this dataset aims to address existing gaps in the field, it remains constrained by certain limitations inherent to the obfuscation tools employed. OLLVM, originally designed for LLVM-4 but ported to LLVM-14 for this study and referred to as OLLVM-14, supports only three intra-procedural obfuscation passes. While Tigress-3.1 offers more configurable obfuscation passes, it requires source files to be amalgamated into a single `C` file, as its `cilly-merge` functionality is not reliable in practice. This constraint influenced project selection, restricting them to those available in amalgamated form: `zlib, lz4, minilua, sqlite, freetype`. Besides, some Tigress obfuscated source codes could not be generated due to Tigress internal errors or the resulting obfuscated source file cannot be compiled with `gcc-12`, despite significant debugging effort.

This dataset serves as the foundation for our experimental evaluation and is publicly accessible[3]. It can be employed either in its entirety or partially, by constructing smaller datasets through specific partitioning strategies, as detailed in Section 6.2.2.

It is employed extensively throughout this thesis, particularly in Chapters 6 and 7.

5.2 BinKit dataset

The BinKit dataset was originally designed to support research on binary similarity [76]. Its initial versions comprise the source code of 51 GNU packages, compiled under 1,352 distinct settings, yielding more than 243,000 binaries. These settings span multiple dimensions, including architectures, compilers and their versions, optimization levels, and specific compilation flags such as non-inlining, Position-Independent Executable (PIE), and Link-Time Optimization (LTO). In addition to standard binaries, which are not considered in this thesis, the dataset also includes binaries obfuscated with OLLVM, incorporating only intra-procedural and data obfuscations. These obfuscated binaries, necessarily compiled with `clang`, were further diversified by varying optimization levels and target architectures. The subset of obfuscated binaries is used to support experiments reported in Chapter 7.

[3] https://github.com/quarkslab/diffing_obfuscation_dataset

5.3 Binary similarity dataset, Dataset-1

Similar to the BinKit dataset, Dataset-1 [95] comprises seven projects, `clamav`, `curl`, `nmap`, `openssl`, `unrar`, `z3`, and `zlib`, compiled across different compilers, compiler versions, architectures, and optimization levels, with inlining systematically disabled. The dataset consists of more than 5,000 binaries, encompassing a total of 26.8M functions. Unlike BinKit, it does not include any obfuscated data and is therefore restricted to classical binary similarity tasks. It is employed in this thesis to support some experiments presented in Chapter 7.

5.4 Loki MBA dataset

Loki [127] is an open-source academic obfuscator, that has been demonstrated to withstand state of the art deobfuscation techniques. In this thesis, we focus particularly on its ability to produce a large and diverse set of MBA of varying size and complexity. These MBA are expressed as mathematical formulas (e.g., $x + y$). Several such formulas, organized by increasing complexity, are readily available online[4]. They serve as the foundation for the deobfuscation experiments presented in Chapter 8.

5.5 Real-world obfuscated samples

The ObfuBench dataset satisfies the criteria that were previously missing in the existing obfuscation literature. Nonetheless, it remains synthetic: although these obfuscation schemes are realistic, they do not necessarily reflect those encountered in real-world obfuscated programs. Thus, this dataset needs to be complemented by in-the-wild obfuscated binaries to assess the generalization and practical applicability of this thesis.

To enable meaningful and fair obfuscation analysis, real-world obfuscated binaries must meet several criteria. Specifically, multiple variants of the same binary should be available, and the number of functions should remain small enough to permit manual reverse engineering. However, binaries that satisfy these requirements, particularly malware or proprietary software, are scarce.

For the purposes of scalability and generalization, this thesis considers both malware and legitimate proprietary software, as additional data for our experimental evaluation.

5.5.1 XTunnel

In the case of malware, although it is relatively simple to obtain any two samples of the same malware, finding two obfuscated versions is significantly more challenging, though still feasible, as most of them rely on packing techniques rather than obfuscation passes as defined in Section 2.3. Establishing a reliable correspondence between such obfuscated samples, however, becomes substantially more difficult due to the typically large number

[4] https://github.com/RUB-SysSec/loki/blob/main/experiments/experiment_10_mba_formula/data/

of functions and the applied obfuscation. Identifying the precise locations and types of applied obfuscation in these samples is an even more complex task.

XTunnel is one of the few malware that fulfills these criteria, making it particularly well suited for obfuscation analysis. Attributed to the APT28 threat group[5], XTunnel is a malware developed in 2013 for data exfiltration from compromised devices.

It satisfies several essential conditions:

- Multiple obfuscated versions are publicly available;
- The total number of functions, approximately 3,500, is small enough to support detailed manual analysis;
- Previous studies have identified and documented the types and locations of obfuscation applied to the binary [6].

In this thesis, several XTunnel samples are studied, identified by their corresponding MD5 hashes: the unobfuscated sample `42DEE38929A93DFD45C39045708C57DA15D7586C`, denoted in this thesis as `XTunnelPlain` and two obfuscated variants, `C637E01F50F5FBD2160B191F6371C5DE2AC56DE4` and `99B454262DC26B081600E844371982A49D334E5E`, respectively denoted as `XTunnelObfu1` and `XTunnelObfu2`. Selected due to their prior examination in the literature [6], the obfuscated samples comprise 3,775 and 3,558 functions, respectively, of which 1,717 and 1,482 are obfuscated with Opaque Predicates. A portion of the remaining functions are likely third-party libraries statically linked into the binaries.

5.5.2 Rabobank

Banking applications are required to comply with specific security regulations concerning mobile applications. Rabobank is a financial institution that is based in the Netherlands. An Android application is available for customers, that is regularly updated. The versions 35.0 and 37.1 were respectively released in March 2024 and June 2024. Apart from the Android bytecode, it contains several native binaries, that are heavily obfuscated. The `libnative-lib.so` binaries, from both the 35.0 and 37.1 versions, are particularly interesting: they respectively contain only 172 and 161 functions and a preliminary manual analysis quickly reveals that most of them are obfuscated. To the best of our knowledge, no academic studies have previously analyzed these samples, in contrast to XTunnel. An obfuscated function from the `libnative-lib.35.0.so` is displayed in Figure 5.3.

5.6 Conclusion

This Chapter presents the datasets employed throughout this thesis and their respective roles, with particular emphasis on the newly introduced ObfuBench dataset. ObfuBench

[5] https://attack.mitre.org/groups/G0007/

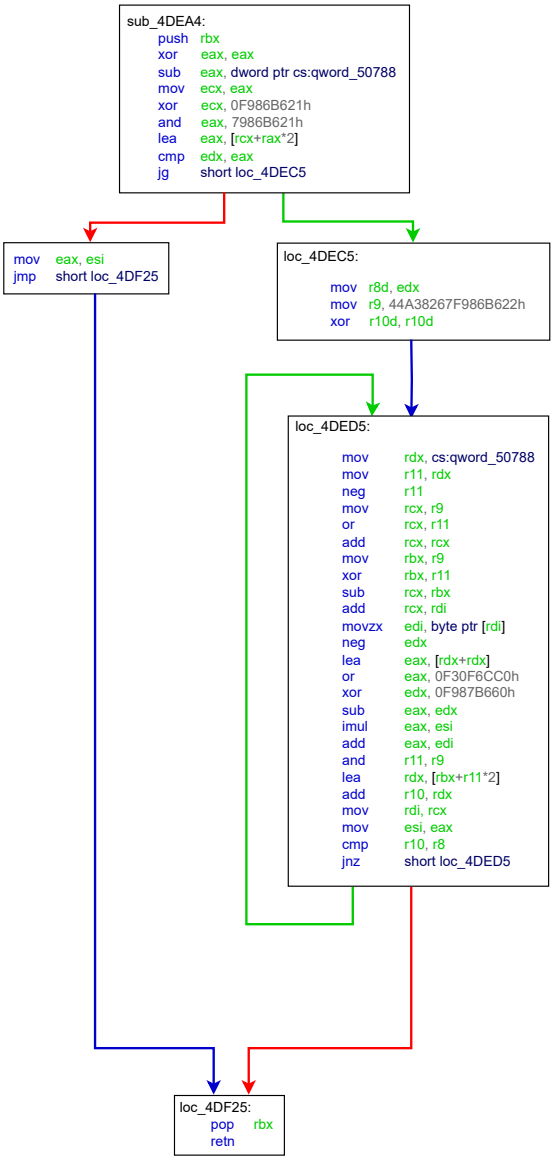


Figure 5.3: An obfuscated function from the `libnative-lib.35.0.so` binary.

surpasses existing resources in terms of size, the diversity of obfuscators, and the range of obfuscation techniques it encompasses. Additional datasets from the literature on binary similarity, such as BinKit (obfuscated) and Dataset-1 (unobfuscated), are also leveraged to support several experiments. The Loki MBA dataset is specifically employed for deobfuscation experiments. Finally, beyond validation on realistic yet synthetic datasets from the literature, most contributions of this thesis are also evaluated on real-world obfuscated programs, including both malware samples and legitimate banking applications.

Chapter 6

Obfuscation detection and characterization

6.1 Introduction

The initial stage of obfuscation analysis involves identifying and characterizing the applied obfuscation techniques. Usually, a reverse engineer might manually inspect selected functions within a binary, looking for patterns indicative of obfuscation, such as extensive use of arithmetic or logical operations in the case of MBA, or control flow structures suggestive of Control Flow Graph Flattening (CFF). However, this manual approach, whether based on intuition or exhaustive inspection, is both time-consuming and potentially unreliable. A more effective and scalable strategy leverages automated algorithms trained to recognize obfuscation patterns. This task can be formulated as location and characterization of obfuscated functions within a binary in order to recognize the specific obfuscation passes applied, their sequence, the obfuscator used, and its associated parameters.

Research on the identification and characterization of obfuscation, two dimensions that largely determine its stealthiness, has gained increasing attention in recent years, giving rise to a number of studies, as already discussed in Section 4.3.2. Each of these works addresses distinct facets of the problem, which are summarized in Table 6.1.

- **Identification.** Only a limited number of studies explicitly address the identification aspect of the problem, while others assume that the locations of obfuscated functions are already known. A part of this work is devoted to this identification problem, as it constitutes a fundamental step in any comprehensive obfuscation analysis pipeline.
- **Characterization.** Conversely, certain studies focus exclusively on the identification phase, without addressing the obfuscation characterization. In this work, we include characterization as a crucial component for thoroughly evaluating the stealthiness of an obfuscation. Unlike prior research that often limits characterization to a narrow set of transformations, we aim to detect a broader range of obfus-

cation passes, encompassing data-oriented, intra-procedural and inter-procedural techniques.

- **Granularity level.** Existing studies typically address obfuscation detection or characterization at a specific level of binary code granularity. While a few works operate at the program level[1], most of the existing papers work directly at the function level. In this study, we also adopt the function-level perspective, as function-wise predictions can be used to infer properties at the program level. To our knowledge, no prior work has specifically targeted obfuscation identification at the Basic Block (BB) level, despite its potential relevance for detecting fine-grained transformations such as MBA.

- **Features.** Various features are considered across studies, such as cyclomatic complexity, assembly mnemonics n-grams or counting vectors. In this study, we build upon expert representations, examining both syntactic and structural features. These features are progressively enriched to assess their relevance based on their capacity to accurately represent binary code for obfuscation detection and characterization tasks.

- **Algorithms.** Techniques used to detect obfuscation vary widely: while some works use heuristics, others leverage traditional ML algorithms, such as Naive Bayes, Random Forests, Decision Tree or SVM. However, only one paper has explored the potential of Graph Neural Networks (GNN) in this context [73]. In this work, we establish and evaluate strong baseline models alongside a comprehensive set of GNN architectures to assess their effectiveness for obfuscation analysis.

- **Obfuscator.** While some studies focus exclusively on either OLLVM or Tigress, this work aligns with more recent efforts that consider both obfuscators in order to provide a broader and more comprehensive evaluation.

Consequently, this Chapter aims to deliver a thorough analysis of obfuscation stealth by analyzing all relevant dimensions of the problem: both the location and characterization of obfuscation, expressed respectively as a binary and multi-class classification problems. It compares a broad set of baseline methods with diverse GNN architectures, investigates the relevance and robustness of different feature types, and evaluates the effects of dataset partitioning strategies and compiler optimization levels.

6.2 Experimental settings

In this Section, experimental settings dedicated to initial data type and distribution, the evaluated features and models, are detailed.

[1] They, however, can be theoretically extended to other levels.

	Date	Identification	Characterization								Level			Representation	Features	Algorithm	Obfuscator	Evaluation	Focus
			CFF	MBA	Opaque Predicates	Virtualization	Split	Merge	Copy	Other	Program	Function	BB						
Kanzaki et al. [75]	2015	✔	✗	✗	✗	✗	✗	✗	✗	✗	✔	✗	✗	Raw assembly	Assembly n-grams	-	Manual Tigress	Artificiality	-
Salem et al. [125]	2016	✗	✔	✔	✔	✗	✗	✗	✗	Jitting	✔	✗	✗	Raw assembly	Assembly TF-IDF	Decision Naive Bayes SVM	Tigress	Accuracy	-
Tofighi Shirazi et al. [138]	2019	✗	✔	✔	✔	✔	✗	✗	✗	Jitting Enc.L Enc.D	✔	✗	✗	Raw assembly	Symbolic bag-of-words	Random Forests	Tigress OLLVM	Accuracy	Multi-label Ensemble methods
Błazytko et al. [10]	2021	✔	✗	✗	✗	✗	✗	✗	✗	✗	✗	✔	✗	CFG	CFG features	Heuristic	-	-	-
Jiang et al. [73]	2021	✔	✔	✔	✔	✗	✗	✗	✗	✗	✗	✔	✗	CFG	Mnemonic classes	GCN + LSTM	OLLVM	Accuracy	-
Blazytko et al. [9]	2021	✗	✔	✗	✗	✗	✗	✗	✗	✗	✗	✔	✗	CFG	CFG features	Heuristic	-	-	-
Błazytko et al. [13]	2023	✔	✗	✗	✗	✗	✗	✗	✗	✗	✗	✔	✗	Raw assembly	n-grams	Heuristic	-	-	-
Schrittwieser et al. [129]	2023	✔	✔	✔	✔	✔	✗	✗	✗	Self-modifying	✗	✔	✗	CFG	CFG features	Random Forests MLP Extra-Trees LGBM	Tigress OLLVM	Accuracy Precision Recall f1-score	-
Greco et al. [59]	2023	✗	✔	✔	✗	✔	✗	✗	✗	✗	✗	✗	✗	CFG	CFG features Assembly features	XGBoost	OLLVM	Accuracy Precision Recall f1-score	Explainability
Our contribution [26]	2024	✔	✔	✔	✔	✔	✔	✔	✔	Enc.L	✗	✔	✗	CFG	CFG features Assembly features	Baselines GCN SAGE GIN GAT UNet	Tigress OLLVM	Balanced accuracy	Comprehensive study

Table 6.1: Literature dedicated to the identification and characterization of obfuscation.

6.2.1 Data types

Binary function code, apart from being represented by the corresponding raw assembly data, is naturally represented by graphs, especially CFG that is well suited for modeling a function execution flow, as discussed in Section 2.2.2.

However, traditional CFG are highly architecture-dependent. In that context, the same function compiled for different architectures yields structurally distinct graphs that do not embed the same assembly code associated to Basic Blocks (BBs). This architecture dependency motivates the creation of an alternative graph type, the CFG-IR.

This type of graph retains the foundation of the standard CFG, but replaces raw assembly instructions with the more abstract Pcode IR. As each assembly instruction is transformed into an equivalent list of more abstract Pcode instructions, it is possible to create a CFG-IR, where each node corresponds to a Pcode block of code, linked by Pcode execution flow.

This CFG-IR differs from the standard CFG as several assembly instructions inside a BB, then without any branching, can be translated as a list of Pcode instructions that contains a branching one. For example, the `x64` assembly instruction `rep movsb` is considered as a non-branching instruction, as it does not modify the control-flow of a function but expanded in Pcode leads to branching conditions, as presented in Figure 6.1.

```
$62592 = int_equal RCX, #0x0
cbranch 0x2, $62592
RCX = int_sub RCX, #0x1
$53248 = RDI
$53376 = int_add RDI, #0x1
$53504 = int_zext DF
$53632 = int_mult #0x2, $53504
RDI = int_sub $53376, $53632
$50176 = RSI
$50304 = int_add RSI, #0x1
$50432 = int_zext DF
$50560 = int_mult #0x2, $50432
RSI = int_sub $50304, $50560
$50816 = load #0x565246ebc240, $50176
$53888 = $50816
store #0x565246ebc240, $53248, $53888
branch 0x0
```

Figure 6.1: Resulting Pcode instructions, including branching conditions, derived from the `rep movsb` instruction in the `x64` architecture.

Then, the assembly BB may be transformed into a Pcode instruction sequence that is

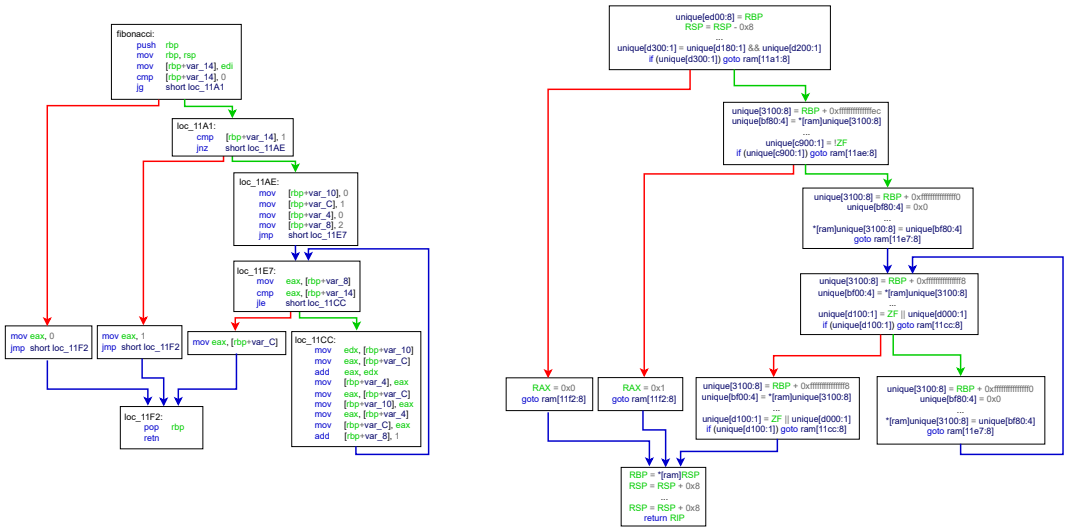


Figure 6.2: The Fibonacci CFG and its corresponding CFG-IR.

no longer basic and that is modeled by two Pcode blocks, linked by an edge. As a result, the topology of the CFG-IR may diverge from that of the standard CFG, usually leading to the creation of new nodes and edges in the CFG-IR graph. Within the dataset, there exist such functions, whose CFG and CFG-IR differ in topology due to the presence of assembly instructions that, once translated, yield a sequence of Pcode instructions containing branching operations. However, these functions are typically large, making the visualization of their corresponding CFG and CFG-IR challenging and of limited clarity.

Nonetheless, if branch-producing instruction sequences are relatively rare, especially in `x64`, the overall structural similarity between CFG and CFG-IR tends to be preserved to a reasonable degree. An example of a CFG and its corresponding CFG-IR, sharing the same topology, is displayed in Figure 6.2.

In this work, both assembly raw data, CFG and CFG-IR are considered as valid data candidates.

6.2.2 Data distribution

As outlined in Chapter 5, the experiments presented here rely on the new ObfuBench dataset. Owing to its design, where binaries progressively incorporate increasing proportions of obfuscated functions, the dataset includes a substantial number of redundant functions, both obfuscated and non-obfuscated. Using the dataset in its entirety without appropriate preprocessing would introduce significant noise and repetition, thus potentially biasing the analysis. To address this, a preprocessing step is applied, resulting in

the construction of two smaller, more curated datasets, each tailored to distinct experimental objectives.

In this context, data leakage poses a significant risk when constructing dataset splits, particularly due to the presence of identical or similar functions across different projects. To mitigate this issue, two distinct data splitting strategies are employed to generate training, validation, and test sets for both datasets.

		-O0 optimization		**-O2 optimization**	
	Train*	3,225 / 48,813		1,846 / 23,151	
	Validation*	803 / 12,135		459 / 5,753	
	Test*	1012 / 15,403		583 / 7,162	
	Ratio binary	(0.11, 0.11, 0.11)		(0.17, 0.17, 0.17)	
		CFG	CFG-IR	CFG	CFG-IR
	Min graph order	1	-	1	2
	Max graph order	5,344	-	1,432	2,761
`Dataset-1`	Mean graph order	15.6	-	17.54	36.54
	Median graph order	7	-	10	21
	Mean graph density	0.116	-	0.132	0.118
	Min graph size	0	-	0	1
	Max graph size	7,122	-	2,396	3,725
	Mean graph size	21.64	-	26.57	45.45
	Median graph size	8	-	13	24
	Min average node degree	0	-	0	1
	Max average node degree	3.93	-	4.44	3.31
	Mean average node degree	1.84	-	2.28	2.16
	Median average node degree	2.4	-	2.67	2.31
	Train*	1,137 / 18,759		610 / 9,019	
	Validation*	279 / 4,652		150 / 2,238	
	Test*	3,948 / 57,627		3012 / 31,760	
	Ratio binary	(0.13, 0.13, 0.11)		(0.14, 0.14, 0.17)	
		CFG	CFG-IR	CFG	CFG-IR
	Min graph order	1	-	1	2
	Max graph order	9,454	-	9,829	23,305
`Dataset-2`	Mean graph order	22.48	-	25.81	54
	Median graph order	7	-	10	22
	Mean graph density	0.112	-	0.127	0.117
	Min graph size	0	-	0	1
	Max graph size	12,764	-	15,620	28,944
	Mean graph size	31.73	-	40,35	68.35
	Median graph size	9	-	14	25
	Min average node degree	0	-	0	1
	Max average node degree	3.95	-	4.85	3.48
	Mean average node degree	1.89	-	2.31	2.17
	Median average node degree	2	-	2.67	2.33

* values expressed in number of functions / samples (obfuscated and unobfuscated variants)

Table 6.2: Characteristics of the two datasets depending on the graph type and the optimization level. The symbol "-" indicates that computing the corresponding graph is unaffordable given our current means.

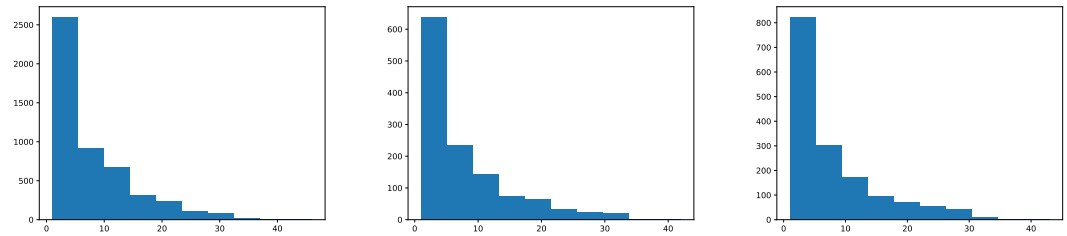


Figure 6.3: Histograms of BBs counts per function for the `Dataset-1` train, validation and test sets.

The first newly constructed dataset, referred to as `Dataset-1`, is built using a **per function split strategy**, wherein a given function and its obfuscated versions are assigned to the same set. To implement this, all the functions from the five projects within the dataset are collected, ensuring the common functions between projects, such as the `libc` functions, are completely removed. The resulting function list is randomly divided into training, validation and test sets following a (64%, 16%, 20%) ratio, using stratified sampling to preserve a similar distribution of Basic Blocks (BBs) counts across sets. Figure 6.3 illustrates such a BBs distribution for each set. For each function in a given set, both its obfuscated and unobfuscated variants are included. This strategy introduces significant class imbalance: the dataset comprises 11 obfuscation classes for only a single unobfuscated class. Notably, this imbalance is atypical in the context of anomaly detection, where normal samples generally dominate and anomalies are rare. In contrast, our controlled experimental environment enables the collection of obfuscated (i.e., abnormal) data without difficulty. However, this abundance does not reflect real-world conditions, where obfuscated functions are often limited in number due to computational constraints. In this context, this choice is motivated by the desire for an exhaustive study of obfuscation types, aiming to broadly investigate all available obfuscation techniques, even at the expense of class imbalance, rather than restricting the analysis to a limited set of obfuscation passes.

The **per binary split strategy** is used to create the second dataset, named `Dataset-2`. All functions from the `zlib, lz4` and `minilua` projects, along with their corresponding obfuscated versions, are used to construct the training and validation sets. These two sets are generated following the same procedure as in `Dataset-1`, using a (80%, 20%) split ratio. The test set, by contrast, consists entirely of functions from `sqlite` and `freetype` projects, including all their obfuscated versions. This configuration simulates a real-world scenario for obfuscation detection and characterization, as the objective is to analyze a previously unseen binary using a model trained exclusively on distinct, unrelated binaries crafted in a controlled environment. As in `Dataset-1`, a pronounced class imbalance is observed, with obfuscated samples being substantially more prevalent than their unobfuscated counterparts.

`Dataset-2` is expected to pose a greater challenge compared to `Dataset-1` as two

projects may exhibit distinct coding styles and differ significantly in the number and nature of functions that are valid candidates for certain types of obfuscation. By ensuring that training, validation and test sets are drawn from entirely separate projects, the model is prevented from relying on project-specific patterns or regularities, thereby promoting a more robust and generalizable evaluation.

For these two datasets, `-O0` and `-O2` binaries are separated, due to the compiler optimization side effects that tend to remove, sometimes completely, the applied obfuscation. The dataset descriptions, especially the number of functions and samples, the class ratio and various graph characteristics, are available in Table 6.2. It illustrates the variability between the `Dataset-1` and `Dataset-2`, the `-O0` and `-O2` optimizations and the CFG and CFG-IR. Some features, in particular the graph order and size, and the average node degree, are displayed as they impact most graph learning algorithms, especially GNNs [151].

6.2.3 Models and features

This study adopts a progressive approach to representation learning, beginning with established baseline ML techniques before advancing to more sophisticated GNN architectures. Specifically, traditional classifiers, such as Random Forests and Gradient Boosting, are employed as baselines, while various GNN architectures, described in detail in Section 3.3.2, including GCN, SAGE, GIN, GAT and UNet, are evaluated.

These models do not operate at the same granularity. Classical ML methods require graph-level features, whereas GNNs need node-level features as input, in addition to the graph structure itself. This necessitates associating each Basic Block (BB), whether in the CFG or CFG-IR, to an appropriate feature vector.

Classical baselines are assessed using two distinct types of graph features:

- A set of graph-level features, applicable to both CFG and CFG-IR, is computed to characterize the structural properties of each function. These include the number of nodes and edges, cyclomatic complexity[2], the number of strongly connected components, the mean degree, graph density, diameter, transitivity, its number of components, maximum and average number of instructions per node, total instruction count per function, maximum and minimum degree. Specifically, these features are complemented with assembly statistics originating from the CFG, such as the counts of `imul`, `shr`, `shl`, `sar`, `div`, `xor`, `add`, `sub` mnemonics in the function, the number of immediate values, and the counts of memory read and write operations within each function. The final feature dimensions are respectively **23** and **13** for the CFG and CFG-IR.

- An assembly TF-IDF feature [125], is considered as a feature for the CFG. It consists of the counts of the **128**-most used assembly tokens of the raw assembly text

[2]Cyclomatic complexity [101] is a software metric that quantifies the structural complexity of a program. It measures the number of linearly independent paths within a code fragment, such as a function. Formally, it is defined as $C = E - N + 2P$, where E denotes the number of edges in the CFG, N the number of nodes, and P the number of connected components of the CFG.

associated with the CFG, inversely weighted by the global frequencies for the assembly. In parallel, a similar feature is constructed for the CFG-IR. Given the more constrained and abstract nature of Pcode, this feature is limited to mnemonics only and has a reduced dimensionality of **72**.

In contrast, GNN features are gradually enriched as follows:

- As a reference, we employ an identity feature vector, a **1**-dimensional vector filled with 1's, which can be applied to both CFG and the CFG-IR. This feature carries no semantic information about the nodes themselves, thereby forcing the GNN to rely solely on the graph topology to iteratively refine node embeddings.
- The first non-trivial feature vector is derived from counting assembly mnemonic classes, inspired by an existing strategy [73] that provides a coarse-grained representation of the assembly mnemonic classes distribution at the BB level. Each assembly mnemonic is assigned to one of **27** general operations categories, such as conditional transfer mnemonics (e.g., `jne`) or logical mnemonics (e.g., `and`). This approach exclusively relies on assembly code and is therefore applicable only to the CFG. Extending this feature to the CFG-IR and its underlying Pcode is not relevant as defining analogous Pcode operation classes is less intuitive and would result in a limited number of categories, insufficient to meaningfully describe a Pcode block in a case of CFG-IR.
- Besides counting user-defined mnemonics classes, a fine-grained feature based on the direct frequency of individual mnemonics can be derived. Specifically for the CFG and its associated assembly, this feature captures the occurrence of all distinct assembly mnemonics, totaling 1,828 unique mnemonics for binaries compiled for the `x64` architecture. This feature is complemented by several BB features[3], resulting in a feature dimension of **1840**.
- Counting assembly mnemonics yields a high-dimensional feature specific to a given architecture. To address this limitation and provide an architecture-agnostic feature, we introduce a Pcode-based alternative, relying on the same mnemonic counting principle. Pcode offers a standardized IR with a fixed set of at most 72 mnemonics, consistent across architectures. Such a feature is available for both the CFG-IR and the CFG, since Pcode can be derived from the assembly code. It is further augmented by specific node features, such as the number of instructions per node[4], resulting in a **77**-dimensional feature.

[3]The complete additional features are a boolean value indicating if the BB is the first one of the function, label encoding depending of the last mnemonic of a node (conditional jump, unconditional jump, call, ret, other), the number of instructions per node, the BB number of successors and predecessors, the number of read-write accesses, of immediates, of syscalls and the different number of calls (external, internal or register).

[4]The complete additional features are a boolean value indicating if the basic block is the first one of the function, label encoding depending of the last mnemonic of a node (conditional jump, unconditional jump, call, ret, other), the number of instructions of a node, the number of its successors and the number of its predecessors.

- Inspired by recent progress in NLP-based approaches for binary similarity analysis [98], each BB code is encoded using PalmTree [86], a transformer trained on `x64` assembly code.[5] The resulting **128**-dimensional textual embedding is then used as the initial node feature for the GNN. Since no transformer model currently exists for Pcode, this feature is limited to the CFG only.

The aforementioned models are evaluated using their respective features on both `Dataset-1` and `Dataset-2`. Hyperparameter tuning is conducted using cross-validation for the baselines and Optuna [1] for the GNNs. The Optuna search is performed with three different random seeds, and the best run, based on validation performance, is used to select the final hyperparameters, such as the number of layers or the hidden dimensions. Each Optuna run is restricted to 20 trials in order to limit the computational burden. The baselines and GNNs are respectively implemented with scikit-learn-1.4.1.post1, with `GridSearchCV` in particular [115], and Pytorch-2.2.1, combined with Pytorch-Geometric-2.5.0 [48].

While the baselines considered in this work rely on graph-level features, GNNs require node-level features. Features were extracted separately for each approach, making it difficult to align them for joint use. Graph-level features, such as cyclomatic complexity, can be incorporated into node-level representations for GNNs by concatenating them to existing node features; however, this provides little additional information for accurately characterizing individual node behavior or properties. Conversely, node-level features cannot be directly aggregated into graph-level features, since metrics such as assembly mnemonics counts over an entire function may be less discriminative than when computed over smaller code units, such as BBs.

6.2.4 Metrics

Due to class imbalance in both binary and multi-class settings, all benchmark evaluations are conducted using the balanced accuracy. This metric imposes penalties when a model favors overrepresented classes at the expense of underrepresented ones, as observed with the unobfuscated class relative to obfuscated classes, thus ensuring that performance is evaluated without bias toward any specific class.

6.2.5 Computational details

All experiments were conducted on a Linux-based server equipped with an Nvidia RTX A6000, 20 cores with 40 threads, 64 GiB of CPU Random-Access Memory (RAM) and 48 GiB of Graphics Processing Unit (GPU) RAM.

[5] Alternative transformer-based models, such as JTrans, are also applicable to this task. However, they typically produce high-dimensional embeddings, which often lead to Out Of Memory (OOM) issues during subsequent GNN computations.

6.2.6 Optimization distinction

Results for the `-O0` and `-O2` optimization levels are presented separately. While both exhibit similar overall trends, the performance under `-O2` is generally lower, likely due to compiler optimizations that interfere with the original obfuscations. More details are available in Section 6.5.

6.3 Locating obfuscation, a binary classification problem

Identifying an obfuscated function within a binary is expressed as a binary classification problem.

6.3.1 CFG

The results obtained using the CFG for both `-O0` and `-O2` optimization levels are reported in Tables 6.3 and 6.4.

Overall, the baseline models perform reasonably well, particularly on the `-O0` CFG, with balanced accuracy scores reaching at least 0.60. The highest performance is achieved by a Gradient Boosting model combined with the assembly TF-IDF feature. This combination consistently outperforms its counterparts, Random Forests and graph-based features. This result can be attributed to the fine-grained nature of assembly distribution features, which more effectively capture the semantic irregularities introduced by obfuscation, in contrast to the coarse and high-level descriptors provided by graph-based features.

These baselines employ algorithms capable of assigning relative importance to the features they utilize. In this context, identifying which features most strongly influence the final decision of these classifiers is of considerable practical interest from a reverse engineering perspective. Notably, the CFG transitivity, its maximum degree, the number of `sub` mnemonics, the total number of instructions, and the minimum CFG degree, rank among the most influential features for binary classification.

Across all evaluated benchmarks, performance is consistently higher on `Dataset-1` compared to `Dataset-2`. This behavior is expected, given the increased difficulty posed by `Dataset-2`. Nonetheless, the performance gap remains within 0.10, indicating that obfuscated and unobfuscated functions can still be effectively distinguished, even in challenging scenarios.

GNN results exhibit contrasted performances. Specifically, using the identity feature yields poor results, often outperformed by simpler baselines. This suggests that relying solely on graph structure is insufficient to distinguish obfuscated from unobfuscated functions, as certain obfuscation techniques operate entirely within BBs, without altering the overall CFG topology. Consequently, the identity feature fails to capture these obfuscated patterns.

The feature based on counting mnemonic classes yields improved performance compared to the identity feature, yet still falls short of baseline methods. This may be attributed to its coarse granularity and reliance on expert-defined mnemonic classes,

Features	**Algorithm**	**Balanced accuracy**	
		Dataset-1	***Dataset-2***
Graph features & assembly (Dim: **#23**)	RandomForest	0.702	0.60
	GradientBoosting	0.725	0.649
TF-IDF on assembly mnemonics (Dim: **#128**)	RandomForest	0.76	0.607
	GradientBoosting	0.80	0.683
Identity (Dim: **#1**)	GCN	0.634	0.608
	Sage	0.615	0.574
	GIN	0.603	0.531
	GAT	0.589	0.539
	UNet	0.616	0.555
Counting mnemonic classes (Dim: **#27**)	GCN	0.659	0.658
	Sage	0.694	0.66
	GIN	0.701	0.673
	GAT	0.655	0.667
	UNet	0.66	0.654
Semantic & counting PCode mnemonics (Dim: **#77**)	GCN	0.761	0.733
	Sage	0.782	0.70
	GIN	0.775	0.69
	GAT	0.77	0.73
	UNet	0.753	0.724
Semantic & counting assembly mnemonics (Dim: **#1840**)	GCN	0.795	0.756
	Sage	0.746	**0.761**
	GIN	**0.806**	0.716
	GAT	0.801	0.733
	UNet	0.788	0.756
PalmTree embeddings (Dim: **#128**)	GCN	0.763	-
	Sage	0.718	-
	GIN	0.715	-
	GAT	0.773	-
	UNet	0.768	-

Table 6.3: Binary classification performance, across different features, algorithms and datasets, for the CFG under `-O0` optimization. "-" indicates a GPU OOM error.

Features	**Algorithm**	**Balanced accuracy**	
		Dataset-1	***Dataset-2***
Graph features & assembly (Dim: **#23**)	RandomForest	0.661	0.583
	GradientBoosting	0.674	0.606
TF-IDF on assembly mnemonics (Dim: **#128**)	RandomForest	0.70	0.565
	GradientBoosting	0.749	0.595
Identity (Dim: **#1**)	GCN	0.581	0.547
	Sage	0.506	0.541
	GIN	0.594	0.56
	GAT	0.508	0.476
	UNet	0.589	0.575
Counting mnemonic classes (Dim: **#27**)	GCN	0.62	0.576
	Sage	0.615	0.57
	GIN	0.63	0.565
	GAT	0.613	0.585
	UNet	0.601	0.549
Semantic & counting PCode mnemonics (Dim: **#77**)	GCN	0.728	0.568
	Sage	0.712	0.559
	GIN	0.672	0.554
	GAT	0.59	0.575
	UNet	0.697	0.572
Semantic & counting assembly mnemonics (Dim: **#1840**)	GCN	**0.787**	0.601
	Sage	0.786	**0.616**
	GIN	0.782	0.603
	GAT	0.778	0.612
	UNet	0.783	0.593
PalmTree on assembly code (Dim: **#128**)	GCN	0.763	0.56
	Sage	0.718	0.566
	GIN	0.792	0.589
	GAT	0.779	0.554
	UNet	0.776	0.564

Table 6.4: Binary classification performance, across different features, algorithms and datasets, for the CFG under `-O2` optimization.

	Features	Algorithm	Training time
Baselines	Graph-based features	Random Forest	0:0:18
		Gradient Boosting	0:0:3
	TF-IDF features	Random Forest	0:0:32
		Gradient Boosting	0:0:8
GNN	Identity features #1	GCN	1:47:00
		SAGE	1:48:00
		GIN	1:49:00
		GAT	2:04:00
		UNet	2:41:00
	Counting mnemonic classes (Dim: #27)	GCN	2:07:00
		SAGE	2:03:00
		GIN	1:55:00
		GAT	2:04:00
		UNet	2:45:00
	Semantic & counting PCode mnemonics (Dim: #77)	GCN	1:47:00
		SAGE	1:59:00
		GIN	2:04:00
		GAT	2:09:00
		UNet	3:00:00
	Semantic & counting assembly mnemonics (Dim: #1,840)	GCN	13:33:00
		SAGE	14:37:00
		GIN	14:36:00
		GAT	18:50:00
		UNet	7:34:00

Table 6.5: Training time in hours:minutes:seconds for each GNN model and features for the CFG `-O0 Dataset-1`. Training time for GNNs using PalmTree initial features is omitted as obtaining PalmTree's embeddings is the limiting timing factor.

which do not necessarily capture obfuscated patterns, particularly since obfuscation can be implemented using common, non-suspicious mnemonics.

Enriching GNN initial features is then fundamental, as more refined features, specifically those based on mnemonic counts, whether derived from assembly or Pcode, substantially improve balanced accuracy, by approximately 10%. These enriched features allow GNNs to match and surpass classical baselines, particularly on `Dataset-2`.

In particular, the Pcode-based counting feature performs slightly below its assembly counterpart. This behavior is primarily due to the Pcode's most abstract and limited mnemonic set, which reduces its capacity to capture specific binary patterns and, consequently, to effectively discriminate obfuscated functions. Nonetheless, its lower dimensionality makes it more efficient to train and broadly applicable across different compiled binaries.

For comparison, Table 6.5 presents the average training time for the baselines and the GNNs, depending in particular on the feature dimensionality and the model architecture. As expected, the baselines remain unmatched in terms of computational efficiency, requiring only a few seconds to execute. In contrast, GNN models demand substantially more time, an effect that becomes increasingly pronounced as the dimensionality of the input features grows. In particular, these results highlight that Pcode-based lower-dimensional features require significantly less training effort compared to the `x64`-based assembly features.

Regarding PalmTree embeddings, unlike findings reported in several state of the art studies on binary similarity [98, 52, 141], initializing GNNs with embeddings generated by a language model trained on assembly code yields lower performance compared to simpler counting-based features. Moreover, obtaining these embeddings presents practical challenges in terms of memory and computational cost. For instance, results could not be computed for functions compiled with the `-O0` optimization level, due to OOM errors, as these functions are significantly larger than those compiled with `-O2`.

6.3.2 CFG-IR

Tables 6.6 and 6.7 report classification scores obtained using CFG-IR.

While previous observations made using CFG are still valid for the CFG-IR, the results obtained with CFG-IR tend to be lower than those achieved with the standard CFG, including traditional baseline models. However, when relying on Pcode counting features, the difference becomes negligible, suggesting that the altered graph topology induced by Pcode-level branching may introduce noise or structural redundancies that are not effectively mitigated by basic feature representations.

Additionally, constructing these CFG-IR is substantially more resource-intensive than generating CFG. This is due to the translation of each assembly instruction into multiple Pcode instructions. For example, the `x64` assembly instruction `packuswb xmm0, xmm2` is expanded into 164 Pcode instructions, which represents the largest translation in our dataset. Among it, some Pcode instructions, though relatively scarce in our `x64`

Features	Algorithm	Balanced accuracy	
		Dataset-1	***Dataset-2***
Graph features & Pcode (Dim: **# 13**)	RandomForest	0.66	-
	GradientBoosting	0.678	-
TF-IDF on Pcode mnemonics (Dim: **#72**)	RandomForest	0.708	0.561
	GradientBoosting	0.742	0.619
Identity (Dim: **#1**)	GCN	0.614	-
	Sage	0.624	-
	GIN	0.618	-
	GAT	0.588	-
	UNet	0.573	-
Semantic & counting Pcode mnemonics (Dim: **#77**)	GCN	0.783	-
	Sage	0.809	-
	GIN	0.77	-
	GAT	0.768	-
	UNet	0.763	-

Table 6.6: Binary classification performance, across different features, algorithms and datasets, for the CFG-IR under `-O0` optimization. "-" indicates that the experiments could not be completed due to the `-O0` CFG-IR processing.

dataset[6], introduce branching behavior, resulting in additional nodes and edges in the CFG-IR. Consequently, the resulting graph becomes significantly larger, particularly for binaries compiled with `-O0`, which naturally produces more verbose binary code than `-O2`. This increase in graph size and the translation from assembly to Pcode lead to pronounced computational overhead in both memory consumption and processing time. As a result, generating CFG-IR for large `-O0` functions in `Dataset-2` becomes infeasible, accounting for the missing results in Tables 6.6 and 6.11. Therefore, the use of CFG-IR is currently limited to smaller-scale experiments unless significant computational resources are available.

Regarding compiler optimization levels, results obtained on `-O0` binaries outperform those on `-O2` by up to 10%. This performance gap is primarily due to the fact that compiler optimizations can partially or entirely eliminate obfuscation patterns within functions, as explained in the latter Section 6.5. As a result, distinguishing obfuscated from unobfuscated functions becomes more challenging under `-O2`, particularly when relying on elementary features, such as the identity one, that may result in a performance

[6]Other architectures, most notably `Arm`, are more prone to this behavior, owing to specific instructions such as `it` and its variants.

Features	Algorithm	Balanced accuracy	
		Dataset-1	***Dataset-2***
Graph features & assembly (Dim: **#13**)	RandomForest	0.631	0.568
	GradientBoosting	0.634	**0.594**
TF-IDF on Pcode mnemonics (Dim: **#72**)	RandomForest	0.643	0.556
	GradientBoosting	0.675	0.584
Identity (Dim: **#1**)	GCN	0.579	0.534
	Sage	0.518	0.481
	GIN	0.575	0.502
	GAT	0.5	0.472
	UNet	0.5	0.5
Semantic & counting Pcode mnemonics (Dim: **#77**)	GCN	0.70	0.574
	Sage	**0.736**	0.578
	GIN	0.703	0.525
	GAT	0.689	0.549
	UNet	0.694	0.586

Table 6.7: Binary classification performance, across different features, algorithms and datasets, for the CFG-IR under `-O2` optimization.

drop up to 10%.

Finally, to evaluate the robustness of these models across different data partitions, we generated two additional training, validation, and testing splits for each dataset and optimization level, using distinct random seeds. For `Dataset-2`, the test set remained fixed due to the dataset construction strategy; therefore, only the training and validation sets are resampled in these experiments. The results, restricted to CFG features for computational reasons, are reported in Figure 6.8, which presents the average balanced accuracy along with the corresponding standard deviations across the three splits. Overall, the models demonstrate strong robustness to variations in data partitioning, although certain GNN models display greater instability compared to the baselines. For computational reasons, PalmTree's features were excluded from this experiment.

6.4 Characterizing obfuscation, a multi-class classification problem

Characterizing an obfuscated function within a binary can be expressed as both a multi-label and multi-class classification problem. A function may be obfuscated using a single

	Features	Algorithm	Dataset-1		Dataset-2	
			-O0	-O2	-O0	-O2
Baselines	Graph-based features	Random Forest	$0.713 \pm 8e{-3}$	$0.661 \pm 4e{-4}$	$0.635 \pm 2e{-2}$	$0.583 \pm 3e{-3}$
		Gradient Boosting	$0.731 \pm 4e{-3}$	$0.679 \pm 4e{-4}$	$0.669 \pm 1e{-2}$	$0.608 \pm 1e{-3}$
	TF-IDF features	Random Forest	$0.773 \pm 9e{-3}$	$0.706 \pm 6e{-3}$	$0.649 \pm 3e{-2}$	$0.563 \pm 1e{-3}$
		Gradient Boosting	$0.807 \pm 6e{-3}$	$0.754 \pm 6e{-3}$	$0.717 \pm 2e{-2}$	$0.597 \pm 4e{-3}$
GNN	Identity features #1	GCN	$0.641 \pm 5e{-3}$	$0.591 \pm 7e{-3}$	$0.593 \pm 1e{-2}$	$0.509 \pm 3e{-2}$
		SAGE	$0.604 \pm 1e{-2}$	$0.542 \pm 2e{-2}$	$0.575 \pm 8e{-4}$	$0.512 \pm 2e{-2}$
		GIN	$0.636 \pm 2e{-2}$	$0.602e{-3}$	$0.588 \pm 4e{-2}$	$0.525 \pm 2e{-2}$
		GAT	$0.581 \pm 5e{-3}$	$0.50 \pm 5e{-3}$	$0.542 \pm 2e{-3}$	$0.488 \pm 9e{-3}$
		UNet	$0.621 \pm 8e{-3}$	$0.588 \pm 9e{-3}$	$0.573 \pm 2e{-2}$	$0.52 \pm 4e{-2}$
	Counting mnemonic classes #27	GCN	$0.664 \pm 3e{-3}$	$0.624 \pm 3e{-3}$	$0.656 \pm 7e{-3}$	$0.565 \pm 9e{-3}$
		SAGE	$0.693 \pm 8e{-3}$	$0.635 \pm 1e{-2}$	$0.675 \pm 1e{-2}$	$0.565 \pm 8e{-3}$
		GIN	$0.707 \pm 4e{-3}$	$0.639 \pm 6e{-3}$	$0.678 \pm 8e{-3}$	$0.552 \pm 1e{-2}$
		GAT	$0.675 \pm 1e{-2}$	$0.616 \pm 7e{-3}$	$0.618 \pm 6e{-2}$	$0.56 \pm 2e{-2}$
		UNet	$0.67 \pm 7e{-3}$	$0.609 \pm 5e{-3}$	$0.643 \pm 2e{-2}$	$0.546 \pm 1e{-2}$
	Semantic & counting Pcode mnemonics #77	GCN	$0.769 \pm 6e{-3}$	$0.713 \pm 2e{-2}$	$0.743 \pm 7e{-3}$	$0.56 \pm 5e{-3}$
		SAGE	$0.718 \pm 9e{-2}$	$0.721 \pm 8e{-3}$	$0.681 \pm 4e{-2}$	$0.554 \pm 4e{-3}$
		GIN	$0.778 \pm 2e{-3}$	$0.692 \pm 1e{-2}$	$0.717 \pm 2e{-2}$	$0.57 \pm 1e{-2}$
		GAT	$0.784 \pm 1e{-2}$	$0.65 \pm 4e{-2}$	$0.737 \pm 9e{-3}$	$0.528 \pm 3e{-2}$
		UNet	$0.767 \pm 1e{-2}$	$0.713 \pm 1e{-2}$	$0.715 \pm 9e{-3}$	$0.565 \pm 1e{-2}$
	Semantic & counting assembly mnemonics #1840	GCN	$0.804 \pm 7e{-3}$	$0.791 \pm 3e{-3}$	$0.749 \pm 3e{-2}$	$0.607 \pm 8e{-3}$
		SAGE	$0.797 \pm 4e{-2}$	$0.796 \pm 7e{-3}$	$0.775 \pm 1e{-2}$	$0.626 \pm 1e{-2}$
		GIN	$0.804 \pm 2e{-3}$	$0.791 \pm 7e{-3}$	$0.758 \pm 3e{-2}$	$0.604 \pm 2e{-2}$
		GAT	$0.572 \pm 3e{-1}$	$0.783 \pm 5e{-3}$	$0.75 \pm 1e{-2}$	$0.612 \pm 6e{-3}$
		UNet	$0.8 \pm 8e{-3}$	$0.793 \pm 8e{-3}$	$0.769 \pm 2e{-2}$	$0.595 \pm 1e{-2}$

Table 6.8: Means and standard deviations of binary classification results for the CFG, given three data-split (including the one already displayed).

obfuscation pass or a combination of multiple passes. Modeling the characterization task within a multi-label framework is particularly advantageous, as it enables the model to independently identify each obfuscation pass, thereby recognizing novel combinations of passes that were not seen during training.

However, multi-label classification is known to be a challenging problem [137], especially when the number of labels is large. Consequently, most existing studies simplify this task by treating it as a multi-class classification problem, where composite obfuscation schema are considered as a novel obfuscation class. While this approach reduces complexity, it sacrifices the modularity offered by the multi-label setting, which allows for a more fine-grained identification of individual obfuscation techniques within composite schemes.

As a result, in this study, the characterization problem is formulated as a multi-class

Features	**Algorithm**	**Balanced accuracy**	
		Dataset-1	***Dataset-2***
Graph features & assembly (Dim: **#23**)	RandomForest	0.65	0.57
	GradientBoosting	0.66	0.594
TF-IDF on assembly mnemonics (Dim: **#128**)	RandomForest	0.697	0.593
	GradientBoosting	0.724	0.579
Identity (Dim: **#1**)	GCN	0.323	0.326
	Sage	0.341	0.347
	GIN	0.414	0.407
	GAT	0.192	0.195
	UNet	0.362	0.299
Counting mnemonic classes (Dim: **#27**)	GCN	0.431	0.462
	Sage	0.498	0.499
	GIN	0.488	0.474
	GAT	0.45	0.342
	UNet	0.439	0.448
Semantic & counting PCode mnemonics (Dim: **#77**)	GCN	0.699	0.693
	Sage	0.611	**0.729**
	GIN	0.706	0.71
	GAT	0.684	0.65
	UNet	0.704	0.627
Semantic & counting assembly mnemonics (Dim: **#1840**)	GCN	0.74	0.659
	Sage	0.738	0.714
	GIN	**0.744**	0.69
	GAT	0.733	0.723
	UNet	0.733	0.68
PalmTree on assembly code (Dim: **#128**)	GCN	0.696	-
	Sage	0.698	-
	GIN	0.693	-
	GAT	0.685	-
	UNet	0.67	-

Table 6.9: Multi-class classification performance, across different features, algorithms and datasets, for the CFG under `-O0` optimization. "-" indicates GPU OOM error.

Features	**Algorithm**	**Balanced accuracy**	
		Dataset-1	***Dataset-2***
Graph features & assembly (Dim: **#23**)	RandomForest	0.418	0.362
	GradientBoosting	0.422	0.368
TF-IDF on assembly mnemonics (Dim: **#128**)	RandomForest	0.469	**0.387**
	GradientBoosting	**0.481**	0.33
Identity (Dim: **#1**)	GCN	0.246	0.177
	Sage	0.135	0.156
	GIN	0.253	0.174
	GAT	0.128	0.1
	UNet	0.245	0.198
Counting mnemonic classes (Dim: **#27**)	GCN	0.268	0.196
	Sage	0.28	0.249
	GIN	0.265	0.144
	GAT	0.261	0.221
	UNet	0.257	0.212
Semantic & counting PCode mnemonics (Dim: **#77**)	GCN	0.36	0.275
	Sage	0.317	0.247
	GIN	0.364	0.271
	GAT	0.353	0.256
	UNet	0.374	0.261
Semantic & counting assembly mnemonics (Dim: **#1840**)	GCN	0.38	0.327
	Sage	0.386	0.269
	GIN	0.381	0.309
	GAT	0.242	0.252
	UNet	0.373	0.307
PalmTree on assembly code (Dim: **#128**)	GCN	0.36	0.288
	Sage	0.355	0.278
	GIN	0.371	0.248
	GAT	0.347	0.282
	UNet	0.28	-

Table 6.10: Multi-class classification performance, across different features, algorithms and datasets, for the CFG under `-O2` optimization "-" indicates GPU OOM error.

classification problem, aiming to assign each obfuscated function to one of 11 predefined obfuscated classes, as presented in Section 5: Copy, Split, Merge, CFF, Virtualize, Opaque, EncodeArithmetic, EncodeLiterals, Mix, Mix + Split, as well as the OLLVM Mix, in practice different from the Tigress Mix.

6.4.1 CFG

The classification results using CFG for `-O0` and `-O2` are reported in Tables 6.9 and 6.10.

These results corroborate several observations previously made in the binary setting. In particular, multi-class classification scores under the `-O0` optimization level are generally 10% lower than in the binary case. Despite the increased complexity of dealing with 11 obfuscation classes, compared to a maximum of 4 in prior studies [73], traditional baseline methods still perform surprisingly well. Conversely, GNNs relying on elementary features, such as the identity feature or mnemonic class counts, yield considerably weaker performance. However, features enriched with semantic information originated from counting mnemonic vector lead to models that outperform baselines, especially on `Dataset-2`. This disparity between features is reinforced in the multi-class experiment.

Features	**Algorithm**	**Balanced accuracy**	
		Dataset-1	***Dataset-2***
Graph features & assembly (Dim: **#13**)	RandomForest	0.535	-
	GradientBoosting	0.538	-
TF-IDF on Pcode mnemonics (Dim: **#72**)	RandomForest	0.645	0.566
	GradientBoosting	0.675	0.589
Identity (Dim: **#1**)	GCN	0.301	-
	Sage	0.306	-
	GIN	0.381	-
	GAT	0.16	-
	UNet	0.287	-
Semantic & counting Pcode mnemonics (Dim: **#77**)	GCN	0.716	-
	Sage	**0.741**	-
	GIN	0.70	-
	GAT	0.679	-
	UNet	0.692	-

Table 6.11: Multi-class classification performance, across different features, algorithms and datasets, for the CFG-IR under `-O0` optimization. "-" indicates that the experiments could not be completed due to the `-O0` CFG-IR processing.

Features	Algorithm	Balanced accuracy	
		Dataset-1	***Dataset-2***
Graph features & assembly (Dim: **#13**)	RandomForest	0.374	0.319
	GradientBoosting	0.393	0.322
TF-IDF on Pcode mnemonics (Dim: **#72**)	RandomForest	0.402	**0.363**
	GradientBoosting	**0.41**	0.307
Identity (Dim: **#1**)	GCN	0.218	0.18
	Sage	0.111	0.197
	GIN	0.244	0.158
	GAT	0.09	0.104
	UNet	0.222	0.1
Semantic & counting Pcode mnemonics (Dim: **#77**)	GCN	0.372	0.244
	Sage	0.364	0.265
	GIN	0.367	0.245
	GAT	0.148	0.236
	UNet	0.345	0.233

Table 6.12: Multi-class classification performance, across different features, algorithms and datasets, for the CFG-IR under `-O2` optimization.

In contrast, the multi-class results under `-O2` optimization reveal a pronounced discrepancy compared to the binary classification setting. While the performance gap between `-O0` and `-O2` was previously modest, it becomes substantially more exacerbated in the multi-class scenario. Even with semantically meaningful features, GNNs struggle to match the performance of simpler baseline methods. This degradation can be attributed to the increased difficulty in distinguishing specific and often subtle obfuscation patterns when they are subject to optimization, which may partially remove or alter these patterns.

6.4.2 CFG-IR

Tables 6.11 and 6.12 contain classification scores for CFG-IR experiment. The trends observed are consistent with those discussed in Sections 6.3.2 and 6.4.1, where the use of CFG-IR leads to a slight reduction in performance compared to CFG. Overall, the multi-class classification scores remain satisfactory, with the notable exception of the `-O2` setting, in which a more pronounced performance drop is observed.

6.5 The role of optimization in the obfuscation stealth analysis

Investigating the intricate relationship between obfuscation and optimization presents significant challenges. Analyzing the discrepancies between the `-O2` and `-O0` results constitutes an important step toward a more comprehensive understanding of these mechanisms.

	Dataset-1			Dataset-2		
	Train	Validation	Test	Train	Validation	Test
Functions labeled as obfuscated	22,646	5,590	6,951	8,696	2,110	34,081
Functions incorrectly labeled as obfuscated	12,057	3,013	3,787	4,283	1,094	19,395

Table 6.13: Refinement effect on the `-O2` ObfuBench dataset.

As discussed in Section 2.3.5, compiler optimization can significantly alter or remove obfuscation, resulting in functions that were intended to be obfuscated but ultimately are not. This issue is particularly pronounced under the `-O2` optimization level. Consequently, the `-O2` dataset is likely affected by label noise, where functions initially labeled as obfuscated are, in practice, indistinguishable from their plain optimized versions[7]. We locate these functions and correct their labels, by considering them as unobfuscated. It should be noted that this labeling correction may still overlook certain functions that were not obfuscated as intended. Table 6.13 details the impact of refinement on the overall dataset.

After correcting the mislabeled functions in the `-O2` version of `Dataset-1` and `Dataset-2`, the previous experiments are run again to quantify the impact of label noise on model performance. Refined binary classification results using the CFG and CFG-IR are presented in Tables 6.14 and 6.15, while the corresponding multi-class results are shown in Tables 6.16 and 6.17.

Fixing the mislabeled functions within the `-O2` dataset has a contrasted effect. In the binary case, baselines show up to a 10% improvement in balanced accuracy, occasionally surpassing GNNs on `Dataset-2`. GNN results are less uniform: while models trained on `Dataset-1` observe minimal change, those trained on `Dataset-2` benefit substantially from labeling correction. In the multi-class setting, performance improvements are even more pronounced across all datasets and models. These findings illustrate the necessity to carefully consider obfuscation when optimization is involved.

[7]Optimized OLLVM-obfuscated binaries, as well as certain Tigress obfuscations, yield identical assembly code for both unobfuscated and obfuscated versions. Moreover, Tigress obfuscation is not always applied as intended: discrepancies were observed between the functions specified for obfuscation and those actually obfuscated in practice.

Features	**Algorithm**	**Balanced accuracy**	
		Dataset-1	***Dataset-2***
Graph features & assembly (Dim: **#23**)	RandomForest	0.723	0.693
	GradientBoosting	0.741	0.679
TF-IDF on assembly mnemonics (Dim: **#128**)	RandomForest	0.788	**0.722**
	GradientBoosting	0.791	0.688
Identity (Dim: **#1**)	GCN	0.604	0.524
	Sage	0.545	0.511
	GIN	0.635	0.552
	GAT	0.524	0.528
	UNet	0.579	0.5
Counting mnemonic classes (Dim: **#27**)	GCN	0.646	0.5
	Sage	0.646	0.582
	GIN	0.657	0.5
	GAT	0.65	0.558
	UNet	0.537	0.602
Semantic & counting PCode mnemonics (Dim: **#77**)	GCN	0.786	0.533
	Sage	0.778	0.59
	GIN	0.762	0.567
	GAT	0.756	0.612
	UNet	0.692	0.546
Semantic & counting assembly mnemonics (Dim: **#1840**)	GCN	0.734	0.608
	Sage	**0.793**	0.689
	GIN	0.781	0.655
	GAT	0.776	0.65
	UNet	0.773	0.623
PalmTree on assembly code (Dim: **#128**)	GCN	0.759	0.588
	Sage	0.752	0.56
	GIN	0.776	0.586
	GAT	0.772	0.528
	UNet	0.764	0.63

Table 6.14: Binary classification performance, across different features, algorithms and datasets, for the CFG under `-O2` optimization on a refined dataset.

Features	**Algorithm**	**Balanced accuracy**	
		Dataset-1	***Dataset-2***
Graph features & assembly (Dim: **#13**)	RandomForest	0.697	0.638
	GradientBoosting	0.676	0.627
TF-IDF on Pcode mnemonics (Dim: **#72**)	RandomForest	0.737	**0.702**
	GradientBoosting	0.748	0.677
Identity (Dim: **#1**)	GCN	0.588	0.524
	Sage	0.496	0.493
	GIN	0.645	0.565
	GAT	0.529	0.523
	UNet	0.583	0.506
Semantic & counting Pcode mnemonics (Dim: **#77**)	GCN	0.771	0.566
	Sage	0.685	0.547
	GIN	**0.772**	0.575
	GAT	0.769	0.51
	UNet	0.747	0.525

Table 6.15: Binary classification performance, across different features, algorithms and datasets, for the CFG-IR under `-O2` optimization on a refined dataset.

Features	**Algorithm**	**Balanced accuracy**	
		Dataset-1	***Dataset-2***
Graph features & assembly (Dim: **#23**)	RandomForest	0.553	0.378
	GradientBoosting	0.597	0.389
TF-IDF on assembly mnemonics (Dim: **#128**)	RandomForest	0.663	0.441
	GradientBoosting	**0.685**	**0.452**
Identity (Dim: **#1**)	GCN	0.263	0.226
	Sage	0.245	0.267
	GIN	0.227	0.185
	GAT	0.2	0.194
	UNet	0.262	0.234
Counting mnemonic classes (Dim: **#27**)	GCN	0.283	0.287
	Sage	0.319	0.244
	GIN	0.285	0.123
	GAT	0.293	0.193
	UNet	0.293	0.25
Semantic & counting PCode mnemonics (Dim: **#77**)	GCN	0.451	0.331
	Sage	0.455	0.319
	GIN	0.443	0.334
	GAT	0.391	0.326
	UNet	0.434	0.354
Semantic & counting assembly mnemonics (Dim: **#1840**)	GCN	0.529	0.435
	Sage	0.46	0.406
	GIN	0.484	0.225
	GAT	0.509	0.404
	UNet	0.521	0.411
PalmTree on assembly code (Dim: **#128**)	GCN	0.507	0.384
	Sage	0.499	0.395
	GIN	0.495	0.239
	GAT	0.487	0.336
	UNet	0.444	0.34

Table 6.16: Multi-class classification performance, across different features, algorithms and datasets, for the CFG under `-O2` optimization on a refined dataset.

Features	**Algorithm**	**Balanced accuracy**	
		Dataset-1	***Dataset-2***
Graph features & assembly (Dim: **#13**)	RandomForest	0.478	0.315
	GradientBoosting	0.484	0.312
TF-IDF on Pcode mnemonics (Dim: **#72**)	RandomForest	0.561	**0.393**
	GradientBoosting	**0.566**	**0.393**
Identity (Dim: **#1**)	GCN	0.25	0.225
	Sage	0.27	0.24
	GIN	0.286	0.235
	GAT	0.206	0.183
	UNet	0.195	0.13
Semantic & counting Pcode mnemonics (Dim: **#77**)	GCN	0.465	0.349
	Sage	0.359	0.331
	GIN	0.40	0.302
	GAT	0.399	0.242
	UNet	0.433	0.262

Table 6.17: Multi-class classification performance, across different features, algorithms and datasets, for the CFG-IR under `-O2` optimization on a refined dataset.

6.6 Real-world examples

6.6.1 XTunnel

In this Section, two obfuscated XTunnel samples are considered, `XTunnelObf1` and `XTunnelObf2`. Among their corresponding 3,775 and 3,558 functions, only 1,717 and 1,482 are considered obfuscated with Opaque Predicates, according to a previously built ground truth [6].

To analyze these samples, we leverage the previously trained models to replicate earlier experiments, aiming to identify obfuscated functions and infer their obfuscation types. The models demonstrating the best performances on the simulated dataset are selected for this task, for both `-O0` and `-O2` optimization levels, since the compilation settings of these samples are unknown. Preference is given to models trained on `Dataset-1`, as they were exposed to a broader diversity of functions compared to those trained on `Dataset-2`. Specifically, the selected models are a `-O0` 1840-sized GIN and a refined `-O2` 1840-SAGE model for binary classification, as well as a `-O0` 1840-sized GIN and a `-O2` refined Gradient Boosting fed with assembly TF-IDF for multi-class classification.

The corresponding results are presented in Table 6.18. Binary scores are computed across all functions, while the multi-class scores are restricted to the subset of obfuscated functions.

When using a `-O2` model, binary balanced accuracy is relatively low, whereas the multi-class classification score is more satisfactory. The `-O0` model achieves decent scores, close to the ones obtained in the previous experiments in Sections 6.3 and 6.4. These findings do not necessarily indicate that the XTunnel samples were compiled with `-O0`, but rather that the `-O0` model is more effective at detecting obfuscation in these samples than its `-O2` counterpart.

Interestingly, a substantial number of functions are classified as obfuscated with Encode Arithmetic instead of Opaque Predicates. This confusion is understandable, as distinguishing between the two is inherently difficult: an Opaque Predicate is essentially an Encode Arithmetic expression that always evaluates to the same boolean value, causing the control flow to consistently follow a single branch. As a result, the models may still capture anomalous patterns indicative of Opaque Predicates, and such predictions can be considered valid.

These findings suggest that the proposed detection and characterization methods are robust, even when applied to real-world malware.

	Binary balanced accuracy	Multi-class balanced accuracy
Sample `XTunnelObf1`	0.737/ 0.357	0.657 / 0.559
Sample `XTunnelObf2`	0.734 / 0.369	0.576 / 0.55

Table 6.18: Obfuscation detection results on two XTunnel samples for both `-O0` / `-O2`.

6.6.2 Rabobank

In this Section, we examine a single obfuscated sample: the `libnative-lib.35.0.so` binary embedded within the Android application Rabobank-35.0. Unlike the XTunnel case, for which a reference ground truth was available, the ground truth here must be constructed entirely manually. Furthermore, while XTunnel was obfuscated exclusively using Opaque Predicates, Rabobank undergoes multiple obfuscation passes. The multi-class framework, described in Section 6.4, is not designed for multi-label classification, and multi-label learning in high-class-count settings remains a relatively underexplored and difficult problem [137]. For each function, we endeavor to establish the most reliable ground truth possible.

In this case, for a function obfuscated with multiple passes, the presence of any individual obfuscation type among the applied passes is treated as a correct label. This approach is inherently limited, as in practice a reverse engineer would ideally seek to identify the complete set of obfuscation passes applied to each function rather than only one.

Functions obfuscated with CFF can be readily identified. However, distinguishing between MBA and Opaque Predicates, and even from dead code insertion (absent from our dataset), proves far more difficult. Consequently, we assign fixed ground truth labels only to functions for which the applied obfuscation can be determined with certainty, discarding the remainder. This filtering results in a set of 118 functions for the final classification task.

The results, presented in Table 6.19, show that binary classification scores for both `-O0` and `-O2` optimization levels are satisfactory and consistent with those reported in Section 6.3. By contrast, multi-class classification performance is markedly lower, particularly for `-O0`. These findings underscore the inherent difficulty of identifying the specific obfuscation type in scenarios where multiple passes are simultaneously applied, leading the model either to select only one of the applied passes or, more often, to incorrectly assign an unrelated class.

	Binary balanced accuracy	Multi-class balanced accuracy
Rabonbank-35.0	0.921 / 0.768	0.017 / 0.319

Table 6.19: Obfuscation detection results on the Rabobank-35.0 `libnative-lib.35.0.so`.

6.7 Conclusion and future works

Our experiments demonstrate that obfuscation detection and characterization can be efficiently performed using classical baselines and features that capture the mnemonics distribution within a function. More importantly, the highest scores are achieved by GNNs leveraging fine-grained semantic features. These results vary substantially in

certain cases, especially when the initial functions are optimized, highlighting the critical need for a better understanding of the link between obfuscation and optimization.

Due to time constraints, we were unable to conduct additional experiments on this topic. In particular, investigating the transferability of the proposed detection approach to previously unseen compilers, architectures, or obfuscators would be highly beneficial for advancing our understanding of obfuscation detection and characterization. Such transferability would, for example, enable a model trained exclusively on `clang`-compiled binaries to be applied to binaries produced by other compilers, or a model trained on `x64` binaries to be adapted to other architectures, particularly with the support of P-code. Similarly, it could allow the recognition of a slightly different Control Flow Graph Flattening (CFF) produced by another obfuscator, even when the model was trained on a specific CFF variant.

Chapter 7

Diffing standard and obfuscated binaries

7.1 Introduction

As discussed in Section 4.2, binary diffing aims to establish a one-to-one correspondence between the functions of two binaries. BinDiff [49, 41] and Diaphora [79] are the most widely used binary differs[1], but present limitations in terms of applicability. Beyond changing the function or basic-block strategy ordering, BinDiff cannot be further parametrized. In contrast, configuring Diaphora is challenging due to the multitude of heuristics, some considered more reliable than others, with little guidance on their ordering or impact on the final diffing outcome. As a result, these tools tend to operate as rigid pipelines, lacking a comprehensive modularity necessary to accommodate diverse analysis objectives.

Yet, binary diffing serves a variety of use cases, including malware analysis, patch detection, identification of statically linked libraries, and clone detection. Each of these scenarios implies specific characteristics and introduces unique constraints. For instance, malware samples are often obfuscated; patch analysis focuses on subtle and local changes [159]; and detecting statically linked libraries is complicated by the absence of imported functions, which are typically used as anchors in the diffing process [49, 41]. Given this diversity, an effective diffing approach must be adaptable, leveraging case-specific characteristics to achieve the best diffing results.

In this thesis, we focus on the impact of obfuscation on existing binary analysis tools, particularly binary diffing and similarity detection techniques. Before conducting an experimental study of binary diffing resilience on obfuscated binaries, described in Section 7.5, we first begin by presenting an analysis of modular diffing, with a specific emphasis on the QBinDiff tool, through an ablation study detailed in Section 7.3, followed by a comprehensive study of binary diffing in a standard, unobfuscated setting, in Section 7.4.

[1] according to popularity metrics (Github downloads, stars, Google Trend).

7.2 Experimental settings

7.2.1 Datasets

As this Chapter seeks to investigate binary diffing in both a standard (i.e., unobfuscated) setting and an obfuscated one, two distinct types of datasets are required.

For the standard case, we rely on a publicly available dataset originally introduced in prior work on binary similarity [95], hereafter referred to as `Dataset-1`, as described in Chapter 5. To support our experiments, `Dataset-1` is partitioned into two disjoint subsets, denoted `Dataset-1A` and `Dataset-1B`. These subsets are used, respectively, for the QBinDiff ablation study and for evaluating diffing performance in the standard setting.

To prevent data leakage between the two experiments, since the results of the ablation study inform the standard diffing evaluation, the subsets must remain entirely disjoint, with no binary appearing in both `Dataset-1A` and `Dataset-1B`. To guarantee this, we proceed as follows: for each project, binaries are divided into two mutually exclusive subsets, `A` and `B`. For example, the binary `x64-clang-7-O1-libz.so.1.2.11` from the `zlib` project is assigned to `Dataset-1A`, while `x64-clang-5.0-O3-libz.so.1.2.11` is assigned to `Dataset-1B`. Similarly, `x64-gcc-7-O0-unrar` belongs to `Dataset-1A`, whereas `x64-clang-3.5-O1-unrar` is assigned to `Dataset-1B`.[2]

Following this procedure, `Dataset-1A` contains 366 binaries from the projects `zlib, unrar, curl, clamav, nmap,` and `openssl`, comprising a total of 425,523 functions. In contrast, `Dataset-1B` includes 474 binaries with 770,544 functions. To construct diffing pairs, each binary is only compared with another version of the same program: for example, `x64-gcc-7-O2-nmap` can be diffed against `x64-clang-3.5-O1-nmap`, but not against `x64-clang-3.5-O1-nping`.[3] The binary pairs are then sampled at random under these constraints and are used in both `Dataset-1A` and `Dataset-1B` to support the QBinDiff ablation study and the standard diffing evaluation.

For the obfuscated scenario, both the ObfuBench dataset and the BinKit dataset are used, the latter being particularly valuable for assessing generalizability, as discussed in Chapter 5. In the case of BinKit, the obfuscation evaluation is restricted to five projects.

Finally, all binaries considered in this Chapter are restricted to the `x64` architecture and are stripped of their function names prior to analysis, to ensure fairness in the diffing process.

7.2.2 Ground truth

A ground truth defines the set of correct function matches between two binaries. For standard unobfuscated binaries, constructing such a ground truth is relatively accessible, as it can be done automatically by aligning functions based on their names, an approach

[2]The number of binaries per project varies: for instance, `unrar` has only one binary, while `openssl` has ten.

[3]This assumption is reasonable for binary diffing, since a reverse engineer rarely attempts to establish an exact mapping between two different binaries (e.g., `nmap` and `nping`), even if they share some functions. However, the assumption does not always hold for other tasks, such as binary similarity [95].

commonly adopted in previous studies [95]. Although this method may be affected by compiler-induced transformations such as inlining, it is generally considered reliable.

In contrast, establishing a ground truth for obfuscated binaries is more complex. In this study, it is constructed as follows:

- Data and intra-procedural obfuscations are applied within the function scope. These techniques will preserve the function name.
- Inter-procedural obfuscation alters the CG structure. Each obfuscation pass has a specific ground truth:
  - A function f in primary split into f_1 and f_2 in the secondary should be matched with both f_1 and f_2.
  - Functions f_1 and f_2 merged into a single function f in the secondary should be matched with both f_1 and f_2 in the primary.
  - A function f cloned into f_1 in the secondary should be matched with both f and f_1 in the secondary.

  This process can be extended to any number of split, merged, or cloned functions.

Both types of ground truth are defined solely at the function-level. Establishing it at a lower level, such as Basic Block (BB), is impractical. In the case of standard binaries, variations in compiler optimizations can produce radically different CFG, rendering accurate BB-level matching infeasible. For obfuscated binaries, while both Tigress and OLLVM can target specific functions, thus preserving the original function mappings, BB-level ground truth cannot be computed for several obfuscations, such as `Virtualization`, which significantly alters the CFG structure.

7.2.3 Metrics

Evaluating a diffing result requires comparing its functions matches against a ground truth that enumerates all correct correspondences. Three standard evaluation metrics are considered: recall ($\mathcal{R}$), precision ($\mathcal{P}$) and f1-score. They are defined as follows:

$$\mathcal{P} = \frac{TP}{TP+FP} \quad \mathcal{R} = \frac{TP}{TP+FN} \quad \text{f1-score} = \frac{2\times\mathcal{P}\times\mathcal{R}}{\mathcal{P}+\mathcal{R}}$$

with True Positive (TP), False Positive (FP) and False Negative (FN).

In the context of binary diffing, TP corresponds to a correctly identified match, FP to an incorrect match identified as correct and FN to a missed correct match.

Intuitively, precision reflects the proportion of retrieved matches that are actually correct, while the recall indicates the proportion of correct matches that have been successfully retrieved. As these two metrics offer complementary perspectives, our primary focus is on maximizing the f1-score, the harmonic mean of precision and recall, which ranges from 0 to 1, as it requires both precision and recall to be high.

7.2.4 Binary tools

As discussed in Section 4.2, there are two main approaches to obtain binary diffing results. The first involves traditional binary diffing tools, which directly produce function-level correspondences between two binaries. The second leverages binary similarity tools that compute similarity scores for pairs of functions, followed by a matching algorithm applied to the resulting similarity matrix, in order to retrieve the function matches.

In this study, we adopt both approaches. The evaluated binary diffing tools include BinDiff, Diaphora-3.0, and QBinDiff. These tools are selected based on their widespread use in practice, their support for function-level matching, and their reliance solely on static analysis, making them computationally efficient. We exclude BB-level differs such as DeepBinDiff [39] and BinSim [110], as their granularity is incompatible with the function-level ground truths employed in this work. To ensure a fair comparison, Diaphora is evaluated without decompiler-based features. Regarding binary export formats, BinDiff utilizes BinExport, Diaphora has its own dedicated export format, and QBinDiff supports both BinDiff and Quokka exports. For the ablation study, QBinDiff is tested with the initial default parameters, previously defined [103], in experiments 7.3.1 and 7.3.3, and with the default features in experiments 7.3.1, 7.3.2 and 7.3.4.

The evaluated binary similarity tools include Asm2vec, GMN, PalmTree and JTrans. From the broad range of existing binary similarity methods, we focus on representative models for each major learning paradigm: GMN [87], notable for its use of GNN that jointly learn graph embeddings; PalmTree and JTrans [86, 145], which employ an effective transformer architecture; and Asm2vec [37], explicitly designed to handle obfuscated binaries. These tools generate embedding-based similarity matrices, from which function correspondences are inferred using the Hungarian algorithm [82]. This choice of matching algorithm is motivated by the exactness of the Hungarian algorithm, meaning that it gives an optimal solution to the alignment problem, which ensures that binary similarity tools are not disadvantaged relative to diffing tools.[4] Our goal was to enable a fair comparison between these two classes of approaches, even at the cost of performance, given that the Hungarian algorithm has a computational complexity of $\mathcal{O}(n^3)$. GMN is trained using the available source code default hyperparameters and graph attributes built with Bag-of-Words over the assembly instruction mnemonics [95]. Asm2vec uses three random walks whereas PalmTree and JTrans are applied with their default configuration. All models, except PalmTree[5] and JTrans[6] for which pre-trained versions are used, are trained on `Dataset-1A`, used exclusively for the QBinDiff ablation study, and evaluated on `Dataset-1B`.

[4] For comparison, the matching algorithm used in QBinDiff, which is based on auction mechanisms, does not offer theoretical advantages over the Hungarian algorithm.

[5] https://github.com/palmtreemodel/PalmTree

[6] https://github.com/vul337/jTrans

7.3 Modular diffing: QBinDiff

QBinDiff [104, 105] was designed as an easy modular diffing tool. Given two binaries, denoted primary and secondary, QBinDiff solves by approximating efficiently a specific instance of the NAP. It identifies a matching or a correspondence between the nodes of the binary CGs that best preserves structural and node similarities. In particular, it approximates its specific NAP problem definition using the following constrained quadratic problem formulation [103], where W_1 and W_2 respectively represent these structural and node similarities and $\mathcal{X}$ indicates the set of solutions of the NAP:

$$\begin{aligned} \mathbf{x}^* = & \quad \arg\max_{\mathbf{x}} \quad \mathbf{x}^T W \mathbf{x} \\ = & \quad \arg\max_{\mathbf{x}} \quad \alpha \mathbf{x}^T W_1 \mathbf{x} + (1-\alpha)\mathbf{x}^T W_2 \mathbf{x} \\ & \text{subject to} \quad \mathbf{x} \in \mathcal{X} \end{aligned}$$

This quadratic optimization problem is solved using a belief propagation algorithm, based on auctions. It uses a graphical model, where nodes indicate variables of the problem and edges denote dependencies between these variables. It iteratively updates values associated with the variables using "messages" sent through graph edges. Once it has converged, marginal probabilities are used to determine the optimal solution to the problem. While almost all the steps of this algorithm can be tuned by a user, no prior work has systematically explored how to parametrize these steps and their impacts on the final result. We address this gap by conducting a comprehensive ablation study of QBinDiff, detailing each aspect of its algorithm and how it contributes to the overall diffing.

Algorithm 1 QBinDiff algorithm.

Require: Primary binary p , Secondary binary s, (*features*, *weights*), *parameters*=(*dist*, s_{ratio}, α, ϵ), Optional list of *pre-passes* and *post-passes*
Ensure: Matching between p and s functions
1: $\mathcal{S} \leftarrow Anchoring(p, s)$
2: **for** $pass_i \in$ *pre-passes* **do**
3: $\quad \mathcal{S} \leftarrow pass_i(p, s, S)$ ▷ similarity matrix refinement passes before feature extraction
4: **end for**
5: $features_p \leftarrow FeaturesExtraction(p, features)$
6: $features_s \leftarrow FeaturesExtraction(s, features)$
7: $\mathcal{S} \leftarrow Similarity(features_p, features_s, \mathcal{S}, weights, dist)$
8: **for** $pass_i \in$ *post-passes* **do**
9: $\quad \mathcal{S} \leftarrow pass_i(p, s, S)$ ▷ refinement passes after feature extraction
10: **end for**
11: $\mathcal{S} \leftarrow Decimation(\mathcal{S}, s_{ratio})$
12: *squares-matrix* $\leftarrow Squares(\mathcal{S}, GetCG(p), GetCG(s))$
13: *match* $\leftarrow$ *Belief Propagation*$(\mathcal{S}, squares\text{-}matrix, GetCG(p), GetCG(s), \alpha, \epsilon)$
14: **return match**

QBinDiff, whose algorithm is shown in Algorithm 1, consists of three main components and several parameters:

- a similarity matrix (S) that captures the pairwise similarities between function nodes across the two CGs, computed using a user-defined set of heuristics, named features, that characterize the function and the whole program.

- a weight matrix (*squares matrix*) that encodes the induced common edges in both graphs for each possible node assignment.

- a number of user-defined parameters, among which the tradeoff α, the distance *dist*, the sparsity s_{ratio} and the relaxation parameter ϵ.

According to Algorithm 1, the first anchoring phase is used to pre-match imported functions, when available.[7] This phase can be supplemented with optional preprocessing passes that either establish more matches or initialize the similarity matrix. This anchoring phase is unnecessary when diffing results are derived solely from combining similarity scores with a matching algorithm, such as the Hungarian method. However, in the context of QBinDiff, it proves highly beneficial, as it establishes certain fixed matches that are subsequently used to define squares between the CGs graphs of the two programs, thereby reinforcing the structural alignment through these anchors.

Subsequently, feature vectors are extracted for both the primary and secondary functions, given a set of features that may include CFG structural features, such as the number of Basic Blocks (BBs) per function, CG properties, like the callees of a function, or low-level assembly-related features, such as instruction mnemonics. QBinDiff provides a total of 33 different features, that represent the default features set.[8] The similarity matrix S is computed as a weighted linear combination of distance measures applied to the primary and secondary feature vectors. Available distance metrics include Canberra, Euclidean, cosine and Haussmann.[9]

The similarity matrix S represents the pairwise similarity between each function node in the primary and the secondary. Conceptually, it is intended to capture domain-specific knowledge relevant to the problem instance that the graphs represent. In the context of binary diffing, an entry $S[i, j]$ close to 1 suggests a high degree of similarity between the node i in the primary and the node j in the secondary. Conversely, values close to 0 indicate a strong dissimilarity between two nodes. In most cases, the similarity matrix tends to be too large and has to be decimated, using the sparsity ratio $s_{ratio} \in [0, 1]$, by removing the lowest similarity scores that are unlikely to produce correct matches. This type of decimation is specific to QBinDiff and is not employed in classical alignment methods, such as those based on the Hungarian algorithm.

The *squares matrix* is derived directly from the graph structure and the similarity matrix S. A "square" is defined as a tuple of nodes (A, B, C, D) satisfying the following conditions:

- Nodes A and D belong to the primary graph.
- Nodes B and C belong to the secondary graph.
- (A, D) is a directed edge in the primary graph.

[7] which is not the case for statically linked binaries.

[8] A complete list is available on the QBinDiff documentation website https://diffing.quarkslab.com/qbindiff/doc/source/features.html.

[9] The Haussmann distance is a unique function defined in QBinDiff that combines both the Jaccard index and the Canberra metric, see https://diffing.quarkslab.com/qbindiff/doc/source/params.html#haussmann.

	zlib	curl	clamav	unrar	nmap	openssl
No anchoring	0.72	0.71	0.71	0.72	0.68	0.69
With anchoring	**0.81**	**0.84**	**0.85**	**0.79**	**0.79**	**0.79**
% gain	+12.5	+18.4	+19.8	+9.8	+16.2	+14.5

Table 7.1: Comparison between QBinDiff without anchoring and with anchoring, with default QBinDiff parameters and features.

- (B, C) is a directed edge in the secondary graph.
- Similarity scores for square nodes are positives: S [A,B]> 0 and S [D,C]> 0.

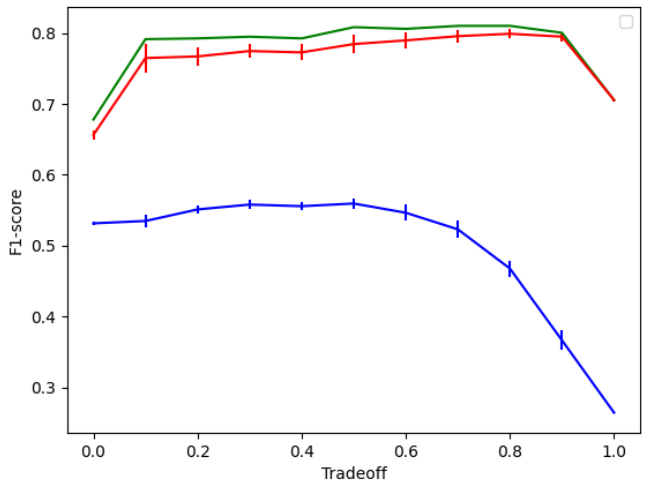
The tradeoff parameter $\alpha \in [0, 1]$ acts as a cursor between function node similarity and graph topology. When $\alpha = 0.0$, only the graph structure is considered to determine the matches, completely ignoring node similarity. Conversely, $\alpha = 1.0$ gives full weight to the similarity values, disregarding the graph structure. $\epsilon \in [0, 1]$ serves as a relaxation parameter that facilitates the convergence of the Belief Propagation [103]. The default parameters, established previously [103] include Canberra distance, $\alpha = 0.75$, s_{ratio}= 0.75 and $\epsilon = 0.5$).

7.3.1 Anchoring

Anchoring (step 1 in Algorithm 1) leverages imported functions as reliable anchors, particularly for dynamically linked binaries, before any further matching step. In this study, we assess the impact of anchoring on QBinDiff convergence behavior and measure the resulting performance improvement. The results reported in Table 7.1 display the average f1-score per project. Notably, the introduction of anchoring yields an important improvement in f1-score: by establishing imported functions as fixed anchor points, the matching process becomes more tractable, as it can now be constrained to function clusters associated with those anchors, significantly reducing computational complexity.

7.3.2 Similarity and CG topology impact

A central QBinDiff parameter is the tradeoff α (step 13 in Algorithm 1) that determines the balance between prioritizing function node similarity and the CG topology during the diffing process. Users may adjust this parameter to emphasize similarity over structural alignment, or vice versa, depending on which yields better results. To evaluate the tradeoff parameter, we systematically vary the tradeoff α under three experimental conditions: first the default QBinDiff configuration, second QBinDiff with a perturbed similarity matrix, and third QBinDiff with altered CG adjacency matrices. The similarity matrix is modified by injecting uniform random noise into its elements, while the CG structure is altered by applying the Metropolis-Hasting algorithm [66], performing 2,000 edge swaps without introducing self-loops. Due to the stochastic nature of these



standard disturbed similarity matrix modified adjacency matrices.

Figure 7.1: f1-score evolution for different QBinDiff instances, depending on the tradeoff, for the `zlib` project.

perturbations, we repeat each diffing experiment with 3 random seeds and report the averaged results. Figure 7.1 presents the findings for the zlib project, from which several key observations emerge:

- Under the standard QBinDiff configuration, the f1-score exhibits two brutal variations as the tradeoff α varies: first, between $\alpha = 0.0$ and $\alpha = 0.1$, and again between $\alpha = 0.9$ and $\alpha = 1.0$. The initial increase from $\alpha = 0.0$ to $\alpha = 0.1$ corresponds to the point at which similarity metrics begin to influence the diffing process, leading to a marked improvement in performance. Conversely, the transition from $\alpha = 0.9$ to $\alpha = 1.0$ removes any consideration of CG topology, depriving the algorithm of crucial structural information and resulting in a sharp decline in f1-score. These abrupt changes underscore the importance of balancing both structural and feature-based similarities when performing binary diffing.

- When the CG adjacency matrices are perturbed, placing significant emphasis on CG topology by setting a low tradeoff α causes the diffing process to rely heavily on noisy structural information, which leads to a degraded f1-score. As α increases, shifting the focus toward the function similarity scores, performance progressively improves and finally converges to the baseline score achieved by the standard QBinDiff configuration. This observation suggests that the algorithm can effectively mitigate the impact of structural noise by prioritizing reliable similarity measures through a higher tradeoff parameter.

- When the similarity matrix is perturbed, increasing the tradeoff from $\alpha = 0.0$ to approximately $\alpha = 0.5$ leads to an improvement in the f1-score. This indicates that, despite the noise, incorporating similarity information remains beneficial when it

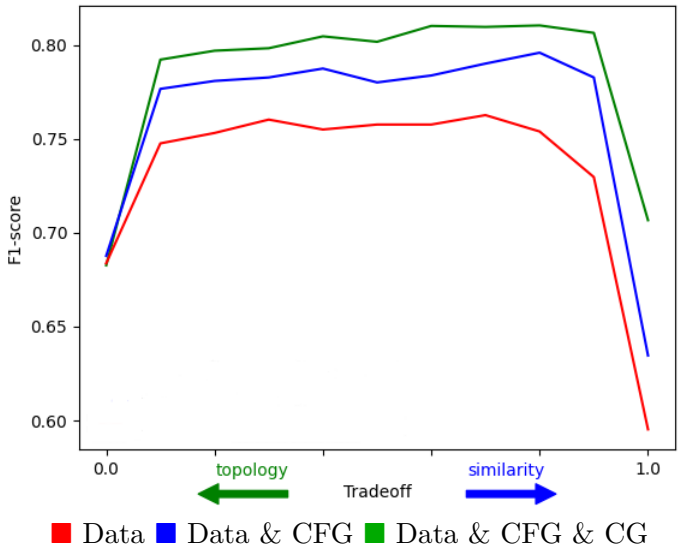


Figure 7.2: QBinDiff features impact on the `zlib` project.

is moderated. However, assigning excessive importance to the noisy similarity, with higher α values, results in a marked decline in performance, highlighting the detrimental impact of over-reliance on altered similarity data.

7.3.3 Features impact

QBinDiff offers a broader range of parameters to adjust beyond the tradeoff α. In particular, the similarity matrix is derived from a weighted linear combination of distances computed over feature vectors (steps 5-6 in Algorithm 1). These features, selected by the user, capture diverse characteristics of the binary, including data-oriented attributes, such as the feature `DatName` that indicates data references, CFG properties, like the feature `BBlockNb` representing the number of Basic Blocks (BBs) inside a function, or CG features, like the `ChildNb` feature, which denotes the number of callees of a given function inside the CG. For this experiment, we define three feature sets, each of them including features from different categories.

Figure 7.2 presents the resulting f1-scores for the `zlib` project, using these feature sets with a varying tradeoff α. When $\alpha = 0.0$, the similarity is entirely ignored in favor of CG topology and then the choice of features has little to no impact, with a f1-score remaining consistent across feature sets. However, as the tradeoff α, so the similarity importance, increases, performances diverge depending on the feature sets. Notably, at $\alpha = 1.0$, where the CG topology is disregarded entirely, features derived from CG structure prove especially beneficial, compensating for the absence of topological information.

The remainder of this work will adopt the default feature set, which encompasses all features currently supported by QBinDiff, including data-related, CFG and CG fea-

tures.[10]

7.3.4 Best parameter search

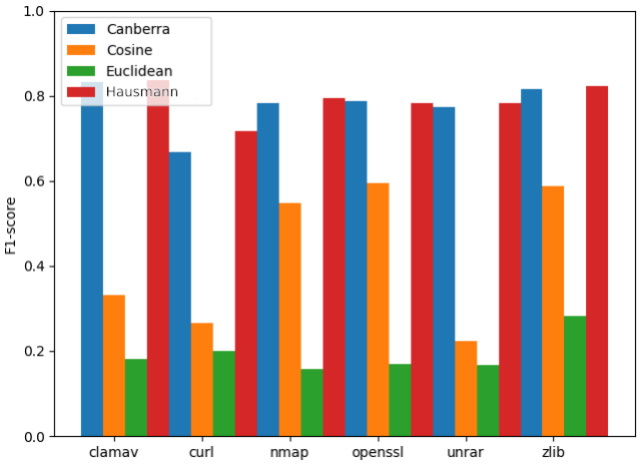


Figure 7.3: Distance (*dist*) impact on the different projects.

As previously mentioned, QBinDiff can be fine-tuned by adjusting its parameters *dist*, ϵ, α, s_{ratio}. Exhaustively searching the full parameter space to identify the optimal configuration is computationally infeasible. Therefore, we begin with the default parameters (Canberra distance, $\alpha = 0.75$, $s_{ratio} = 0.75$ and $\epsilon = 0.5$), established previously [103] and vary only one parameter at a time, replacing the Canberra distance with Haussmann distance for example, to analyze the impact on the differ's behavior. The corresponding plots for *dist* and for ϵ, α and s_{ratio} are respectively presented in Figures 7.3 and 7.4, and their analysis yields several key insights:

- A tradeoff highly focused on the similarity, such as $\alpha = 0.8$ or $\alpha = 0.9$, tends to produce better matching results.
- Higher values of ϵ, such as 0.9 or 1.0, facilitate faster convergence of QBinDiff.
- The Canberra or Haussmann distances are the best distances.
- Increasing the sparsity ratio s_{ratio} does not substantially degrade performance. For small projects, such as `zlib` or `unrar`, only a marginal drop in f1-score is observed as s_{ratio} increases. In contrast, for larger projects, such as `curl`, a moderate sparsity ratio can enhance convergence by decimating the similarity matrix from its unlikely candidate matches, thereby improving performance and reducing computational overhead. However, setting s_{ratio} too high eliminates useful similarity

[10] https://diffing.quarkslab.com/qbindiff/doc/source/api/features.html

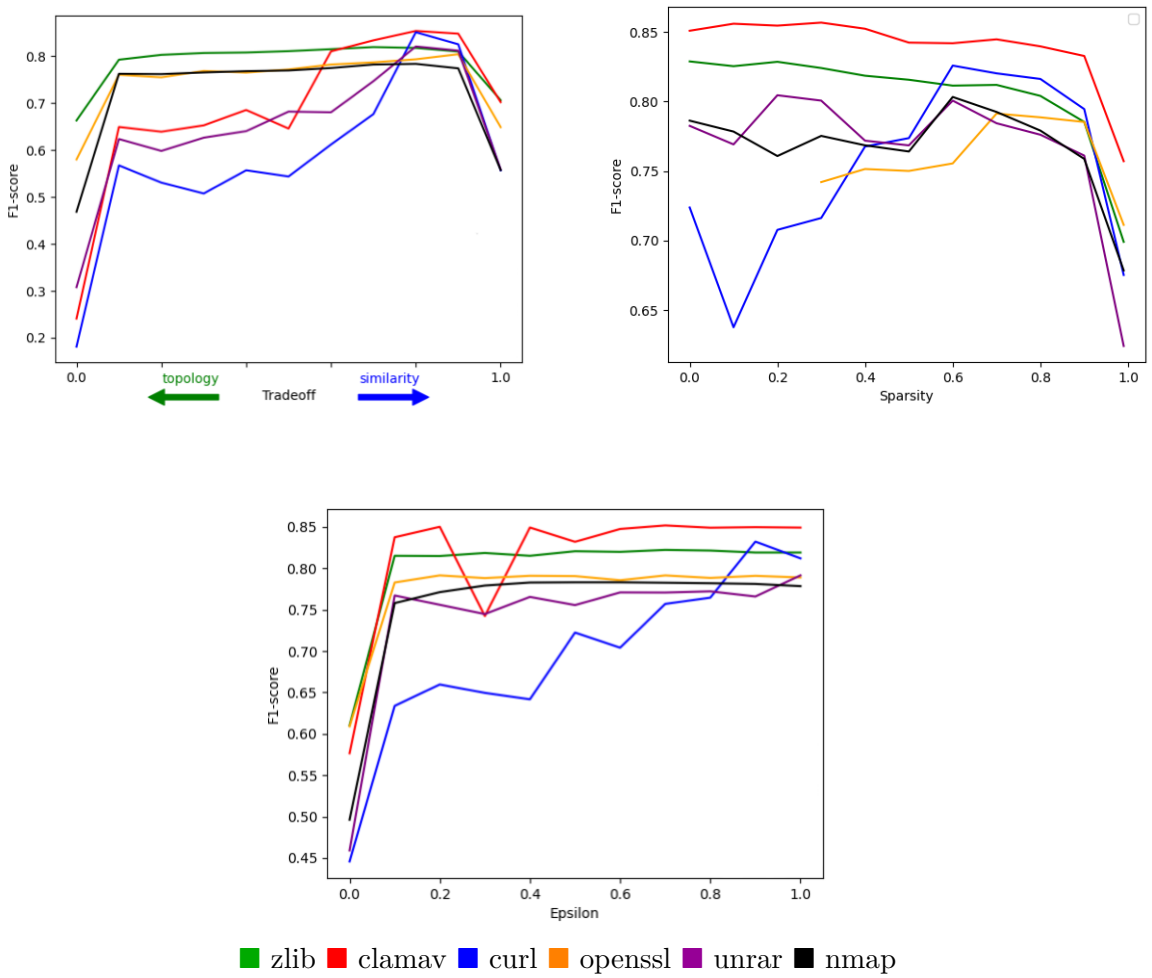


Figure 7.4: Tradeoff (α), epsilon (ϵ) and sparsity (s_{ratio}) impact on the different projects.

data, making matching infeasible. Therefore, for larger binaries, selecting an intermediate value, such as 0.6, appears to offer a good tradeoff between accuracy, memory and runtime efficiency.

Following this analysis, it becomes possible to identify the best parameter configuration for each project, denoted `ppb`. For instance, the `zlib` project achieves its highest performance using the Haussmann distance, $\epsilon = 0.7$ (even though differences among values strictly greater than 0 are negligible), $\alpha = 0.8$ and $s_{ratio} = 0$. In parallel, we also define a project-agnostic best parameter set, called `avb`, which yields the best average performance across all five evaluated projects. It consists of the Haussmann distance, $\epsilon = 0.9$, $\alpha = 0.8$, $s_{ratio} = 0.6$.

7.3.5 Computational resources

Due to its inherent computational complexity, binary diffing can be resource-intensive. Apart from the only QBinDiff ablation study, we evaluate the computational cost of various diffing solutions, including BinDiff, Diaphora and QBinDiff, to assess both time and space efficiency, on a dedicated server equipped with 64 GB of RAM and 16 CPU cores.

	zlib	unrar	curl	clamav	nmap	openssl
BinExport	0.73	2.52	5.04	8.89	9.03	2.41
Diaphora (SQlite)	73	223	450	523	968	496
Quokka	0.71	2.19	4.21	8.37	7.93	2.10

Table 7.2: Averaged exporting time (s) depending on the exporter for each project of `Dataset-1A`.

Table 7.2 compares the time required to export binaries using three different formats: BinExport, used by both QBinDiff and BinDiff, Quokka, exclusively used by QBinDiff, and Diaphora's own exporter, which generates a SQLite database. Among these exports, Quokka demonstrates marginally faster export times compared to BinExport, whereas Diaphora's SQLite export introduces considerable overhead.

Resource consumption during the actual diffing process is also evaluated and Table 7.3 reports the peak memory usage and execution time for each diffing tool. In particular, the s_{ratio} parameter in QBinDiff exhibits a strong effect. Lower sparsity ratios can result

		clamav		nmap		openssl	
		Time	RAM	Time	RAM	Time	RAM
BinDiff		110	298	69	834	56	388
Diaphora3		151	29	651	30	265	30
QBinDiff	$s = 0$	672	823	8675	8339	-	-
	$s = 0.1$	689	801	8702	8329	-	-
	$s = 0.2$	684	712	7424	7167	-	-
	$s = 0.3$	639	617	6712	6171	3844	4884
	$s = 0.4$	656	523	4933	5071	3784	4022
	$s = 0.5$	541	430	4549	3902	2279	3121
	$s = 0.6$	516	347	3146	2900	1509	2301
	$s = 0.7$	489	272	2188	2009	1060	1569
	$s = 0.8$	461	204	1725	1294	1135	999
	$s = 0.9$	447	153	1205	762	528	535
	$s = 0.99$	440	134	880	673	398	408

Table 7.3: Required time (seconds) and memory peaks (MB) needed on the three largest projects for different differs. "-" means the computation was stopped due to an OOM error.

in higher memory consumption and longer runtimes, sometimes causing OOM errors on large binaries, such as those in the `openssl` project. Conversely, moderately higher sparsity ratios not only promote faster convergence but can also yield improved diffing accuracy, as shown by `openssl` scores outperforming `nmap` ones in some configurations.

7.3.6 Conclusion

The QBinDiff ablation study conducted on the standard unobfuscated `Dataset-1A` yields several key insights for effectively adapting and fine-tuning binary diffing. Most notably, the anchoring mechanism, leveraging imported functions when available, demonstrates a marked performance boost by narrowing the scope of function candidates. The study also highlights the critical influence of QBinDiff's parameters on overall diffing score. Specifically, the tradeoff parameter α dictates the balance between function similarity and CG topology; the choice of distance metric *dist* influences the similarity matrix; the sparsity ratio s_{ratio} controls the pruning of unlikely matches; and the relaxation parameter ϵ impacts the convergence behavior of the belief propagation algorithm. Additionally, the selection of features used to compute similarity is crucial, as these features must comprehensively capture the characteristics of binary code, including data-related attributes, CFG and CG structures. Concerning parameters, starting from the default configuration (Canberra distance, $\alpha = 0.75$, $s_{ratio} = 0.75$ and $\epsilon = 0.5$), we identify both the best per-project parameters set `ppb` and a globally best averaged configuration `avb` that provides the best overall performance across the five evaluated projects.

These findings serve as practical guidelines. For the remainder of this work, unless stated otherwise, we default to using the full QBinDiff feature set alongside the averaged best parameter configuration `avb`.

7.4 Diffing standard binaries

Building on the preceding ablation study of QBinDiff, this Section presents a comparative evaluation of various binary diffing solutions using `Dataset-1B`. Table 7.4 reports the results in terms of f1-scores, averaged across the various binaries from each project in the dataset, from which several remarks emerge.

First, the choice of the binary exporter considerably impacts diffing performance. Using QBinDiff with Quokka yields notably better results than with BinExport, with a difference in f1-score of 0.17 for `libcrypto` under the `ppb` configuration. This highlights the practical advantage of Quokka, which exports more information, such as explicit cross-references, not available in BinExport.[11]

Second, diffing tool performance varies substantially across implementations. QBinDiff achieves the highest f1-scores, followed closely by BinDiff, whereas Diaphora consistently performs worse. This is likely due to Diaphora's design to privilege precision over recall. Binary similarity tools, by comparison, generally underperform relative

[11] See https://blog.quarkslab.com/quokka-a-fast-and-accurate-binary-exporter.html for more details.

		BinDiff	Diaphora3	QBinDiff ppb (BinExport)	QBinDiff ppb (Quokka)	QBinDiff avb (BinExport)	QBinDiff avb (Quokka)	GMN	Asm2vec	PalmTree	JTrans
zlib	libz.so.1.2.11	0.85	0.65	0.84	**0.89**	0.82	0.88	0.71	0.19	0.67	0.69
openssl	libssl.so.3	0.81	0.64	0.85	0.85	0.83	**0.86**	0.56	0.17	0.63	0.67
	openssl	0.95	0.68	0.96	**0.98**	0.92	**0.98**	0.59	0.54	0.76	0.72
	libcrypto.so.3	0.76	0.78	0.63	0.80	0.67	**0.82**	0.58	0.01	0.55	0.46
nmap	nping	0.59	0.52	0.74	**0.77**	0.73	**0.77**	0.17	0.17	0.41	0.52
	ncat	0.73	0.58	0.86	**0.92**	0.86	**0.92**	0.24	0.17	0.56	0.67
	nmap	0.8	0.8	0.73	**0.82**	0.73	**0.82**	0.66	0.10	0.61	0.43
clamav	libclamav	0.58	0.46	0.77	**0.81**	0.76	**0.81**	0.43	0.10	0.51	0.53
curl		0.65	0.56	0.83	**0.88**	0.83	**0.88**	0.24	0.22	0.50	0.57
unrar		0.68	0.62	0.82	**0.88**	0.81	0.87	0.22	0.14	0.57	0.69
Averaged		0.74	0.63	0.80	**0.86**	0.80	**0.86**	0.44	0.18	0.58	0.60

Table 7.4: f1-scores for different standard unobfuscated binaries and differs.

to traditional diffing tools. Nevertheless, PalmTree and JTrans approach Diaphora's performance, thanks to their use of enriched representations. In contrast, models like Asm2Vec and GMN yield lower scores due to coarser embeddings: Asm2Vec relies on a basic word2vec architecture that fails to capture fine-grained aspects of binary code, and GMN's input features are limited to distributions of BB mnemonics. JTrans, in contrast, incorporates interprocedural data, leading to more expressive embeddings. Then, when embedding quality is low, similarity scores tend to cluster within a narrow range and exhibit low variance. This undermines the effectiveness of the matching algorithm, as elements across two binaries become indistinguishable in terms of cost, making alignment unreliable.

Third, the difference in f1-score between using `ppb` and `avb` configurations is minimal, suggesting that QBinDiff users do not need to replicate the hyperparameter search described in Section 7.3.4. Instead, they can rely on the best average parameters identified previously. Nonetheless, for specific binaries such as `libssl.so.3` and `libcrypto.so.3`, `ppb` unexpectedly underperforms compared to `avb`. While intuitively this should not occur, the discrepancy arises because the parameter tuning for `ppb` is done incrementally from a default parameter set, modifying one parameter at a time. Thus, `ppb` reflects the best per-dimension configuration relative to the default parameters, while `avb` may occasionally yield better results.

7.5 Diffing obfuscated binaries

In contrast to conventional binaries, where binary diffing techniques have demonstrated consistent and efficient results, diffing obfuscated binaries remains a well-recognized challenge. A prevailing notion within the reverse engineering community is that diffing obfuscated binaries is largely ineffective. As a result, analysts often abandon efforts when confronted with heavily protected programs, such as virtualized binaries.

Moreover, there is limited comprehensive academic research that systematically evaluates the effects of obfuscation on binary analysis tools and quantifies its impact. While some prior works have attempted to define semantic features resilient to specific obfuscation techniques [91, 51, 108], these approaches often struggle to scale to real-world binaries. Likewise, existing diffing and similarity tools are typically not designed for this adversarial context, leading to degraded performance in scenarios where one or both binaries are obfuscated.

As such, the problem of diffing obfuscated binaries remains an open research question. This work aims to fill that gap by conducting an experimental study of binary diffing resilience on obfuscated programs. We compare several binary diffing and similarity-based approaches, assessing their robustness across a diverse range of obfuscation schemes. In parallel, we evaluate the extent to which different obfuscation techniques and obfuscators succeed in undermining the diffing process. Finally, we propose a set of practical guidelines intended to support more effective diffing in the presence of obfuscation.

7.5.1 Motivations

While deobfuscation is rarely applied in practice, primarily because real-world obfuscation schemes rarely align with the assumptions of existing deobfuscation techniques, attackers often circumvent obfuscation through indirect means. These approaches may weaken the obfuscation mechanisms or lead to the leakage of sensitive information. In particular, successfully diffing obfuscated binaries enables adversaries to extract valuable insights from obfuscated programs. For instance, by comparing an unobfuscated binary with a newly obfuscated variant, an attacker can isolate common components and identify newly introduced functionalities.

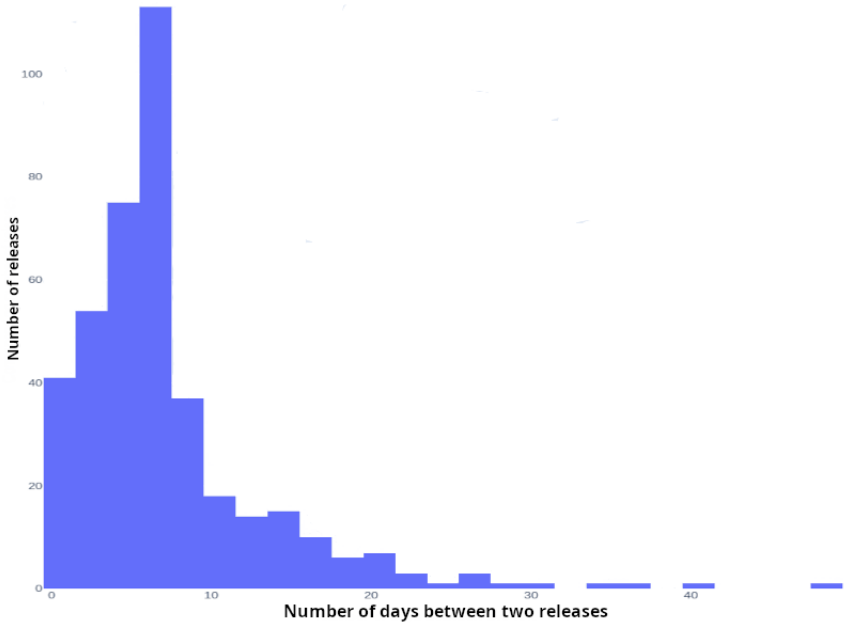


Figure 7.5: Elapsed number of days between every two minor releases of Snapchat Android app, from 2015 to 2024.

If an attacker is able to recover the correct alignment between two binaries, even when one or both have been obfuscated, this indicates that the obfuscation has failed to sufficiently alter the program. In fact, an obfuscation may be considered effective only if it disrupts key program properties to the extent that they can no longer be reliably recovered or computed after obfuscation [30]. This concept defines a threat model in which an attacker can leverage knowledge from one binary to circumvent the obfuscation applied to a new version. Such an approach has been applied to analyze obfuscated Android bytecode [32].

We refer to the setting in which a plain or unobfuscated binary is diffed against its obfuscated counterpart as the `plain-obfuscated` scenario, while comparisons between two obfuscated variants define the `obfuscated-obfuscated` setting. Such scenarios are

frequently encountered, particularly in environments characterized by regular software updates, such as mobile applications. Malware also evolves iteratively across attack campaigns, making variant comparison a practical use case [6].

In fact, obfuscation is largely adopted in the mobile ecosystem, where applications often encapsulate valuable intellectual property and assets. These applications typically follow a dynamic release cycle, resulting in frequent updates. For example, Figure 7.5 illustrates the elapsed time between two minor releases of Snapchat, known to be obfuscated. Updates are published as frequently as weekly, indicating that obfuscated components may be modified and re-released on a similar cadence. This pattern is consistent across the whole mobile ecosystem where similar release cycles apply to most major applications. Likewise, new malware samples, once an initial version is leaked, are often found with varied obfuscation configurations, reflecting the same iterative update behavior.

7.5.2 Resilient diffing

Different types of obfuscation techniques target distinct aspects of binary code, whereas binary diffing tools rely on specific program properties to establish correspondences between two binaries. For instance, inter-procedural obfuscation can significantly alter the CG, which BinDiff uses as a core feature to propagate matches, starting from known imported functions and extending to neighboring ones based on structural and functional similarity. Diaphora, by contrast, employs a hierarchy of heuristics, ranging from high to low confidence, to iteratively align functions. Its most reliable indicators include function addresses, names, assembly and pseudo-code similarity, and function hashes, while less emphasis is placed on CG structure and advanced similarity metrics.

Despite their popularity in practical use, neither BinDiff nor Diaphora was designed to handle heavily obfuscated binaries. In contrast to conventional diffing tools, QBinDiff was designed to be modular in order to leverage binary diffing for specific use cases, particularly adversarial settings such as obfuscated binaries. Its modularity allows precise control over the features used to compute similarity. This adaptability is particularly advantageous when prior knowledge about the applied obfuscation is available. Such knowledge can be obtained manually with reverse engineering as some obfuscations, like CFF or MBA, exhibit very specific patterns, or with ML algorithms that directly infer the presence and type of obfuscation, as discussed in Chapter 6.

Although these classifiers are not without limitations, such as reduced accuracy at higher optimization levels, they nonetheless offer a reliable approximation of the applied obfuscation. This knowledge can then guide the exclusion of features most likely affected by the obfuscation, thereby enhancing diffing robustness. For instance, MBA obfuscation typically increases Basic Block (BB) length without altering the CG or CFG structures, making metrics like cyclomatic complexity or function call counts stable with respect to this obfuscation. Conversely, Virtualization-based obfuscation alters the CFG substantially while preserving the CG. Inter-procedural transformations, such as the Merge pass, disrupt both the CG and the CFG by restructuring both the overall function relationships within the program and the CFG of the merged functions within a unique CFG.

As a consequence, only a limited set of features remain unaffected by this obfuscation type.

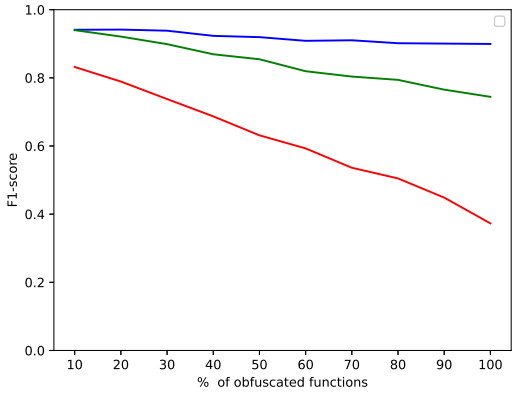
	Merge Split	Copy	Intra	Data
BBlockNb	✗	✔	✗	✔
SCComponents	✗	✔	✗	✔
BytesHash	✗	✔	✗	✗
Cyclomatic Complexity	✗	✔	✗	✔
MDIndex	✗	✔	✗	✔
JumpNb	✗	✔	✗	✔
SmallPrimeNumbers	✗	✔	✗	✔
MaxParentNb	✗	✔	✗	✔
MaxChildNb	✗	✔	✗	✔
MaxInsNb	✗	✔	✗	✗
MeanInsNb	✗	✔	✗	✗
InsNb	✗	✔	✗	✗
GraphMeanDegree	✗	✔	✗	✔
GraphDensity	✗	✔	✗	✔
GraphNbComponents	✗	✔	✗	✔
Graph Diameter	✗	✔	✗	✔
GraphTransitivity	✗	✔	✗	✔
GraphCommunities	✗	✔	✗	✔
Address	✔	✔	✔	✔
DatName	✔	✔	✔	✔
FuncName	✔	✔	✔	✔
ChildNb	✗	✗	✔	✔
ParentNb	✗	✗	✔	✔
RelativeNb	✗	✗	✔	✔
LibName	✗	✗	✔	✔
ImpName	✗	✗	✔	✔
Constant	✗	✗	✔	✔
StrRefs	✔	✔	✔	✗
MnemonicSimple	✔	✔	✔	✗
MnemonicTyped	✔	✔	✔	✗
GroupsCategory	✔	✔	✔	✗
ReadWriteAccess	✔	✔	✔	✗

Table 7.5: QBinDiff stable (✔) and unstable (✗) features list depending on the applied obfuscation.

Based on this understanding, we define two sets of obfuscation-aware features: a stable set comprising features resilient to specific obfuscations, and an unstable set consisting of features known to be altered, displayed in Table 7.5. These sets were determined by comparing unobfuscated binaries with their obfuscated counterparts and analyzing the impact of each obfuscation technique. As a side note, the only features that remain

unaffected by any of the obfuscation techniques considered, and thus are included in all stable feature sets, are the function address, its name[12], and references to the function associated data. In scenarios where the obfuscation type cannot be inferred, all the available QBinDiff features can be naively enabled. Otherwise, the obfuscation-specific stable set should be selected to improve accuracy.

To assess the impact of stable and unstable feature selection on diffing performance, we evaluate QBinDiff across these three configurations, naive, stable and unstable, on the `zlib` project compiled with optimization level `-O0` and obfuscated using Tigress and OLLVM-14. Figure 7.6 illustrates the improvement achieved with the refined QBinDiff$_s$, which only uses stable features, in the case of a Tigress CFF, compared to the standard QBinDiff differ (QBinDiff), where all the features are enabled. Similar results were observed for other obfuscation schemes and are omitted for brevity. This experiment highlights the strong feature dependence of binary diffing: selecting features robust to the applied obfuscation substantially improves match accuracy.



■ stable features (QBinDiff$_s$) ■ all available features (QBinDiff) ■ unstable features.

Figure 7.6: Obfuscated `zlib` (Tigress CFF). QBinDiff f1-scores with respect to the % of obfuscated functions.

The standard set of features should be used as a starting point and can be adapted to account for expert knowledge, by using QBinDiff$_s$ adapted to the applied obfuscation. Moreover, it is fundamental not to only use features that are vulnerable to the same changes, as the unstable features provide degraded performances. For the rest of the paper, QBinDiff denotes the standard QBinDiff algorithm with the default feature set and illustrates the case where a reverse engineer does not have additional knowledge about the obfuscated binary. QBinDiff$_s$ indicates the adjusted QBinDiff with the stable feature set given a specific obfuscation and represents the best-case scenario, where preliminary knowledge about the obfuscation type was extracted with confidence.

[12]Function names can only be removed through stripping.

7.5.3 Diffing a plain binary against its obfuscated variant

This Section examines the `plain-obfuscated` scenario, analyzing the effects of obfuscation on binary diffing and similarity when only the second binary is obfuscated and the first remains unchanged. This scenario is particularly relevant in contexts where updated versions of binaries, such as Android applications or malware, are distributed in obfuscated form.

OLLVM-14	`P-vs-O`	General f1-score				Obfuscated f1-score			
		Mix	*CFF*	*Opaque*	*Enc.A*	*Mix*	*CFF*	*Opaque*	*Enc.A*
10%	Bindiff	0.81	0.78	0.78	0.79	**0.72**	0.71	0.69	0.75
	Diaphora3	0.79	0.78	0.78	0.80	0.45	0.59	0.61	0.78
	GMN	0.66	0.64	0.65	0.69	0.24	0.27	0.36	0.52
	Asm2vec	0.56	0.53	0.56	0.59	0.32	0.40	0.44	0.61
	PalmTree	0.73	0.73	0.72	0.74	0.49	0.64	0.59	0.71
	JTrans	**0.85**	0.85	0.85	**0.86**	0.67	**0.81**	0.79	**0.82**
	QBinDiff	0.84	0.85	0.85	**0.86**	0.58	0.78	0.76	0.81
	QBinDiff$_s$	-	**0.86**	**0.86**	0.81	-	0.80	**0.81**	0.77
50%	Bindiff	0.72	0.76	0.74	0.79	**0.62**	0.67	0.65	0.73
	Diaphora3	0.64	0.70	0.71	0.79	0.40	0.57	0.59	0.78
	GMN	0.44	0.47	0.50	0.62	0.20	0.25	0.31	0.49
	Asm2vec	0.32	0.37	0.44	0.59	0.23	0.32	0.40	0.61
	PalmTree	0.56	0.68	0.62	0.73	0.39	0.61	0.51	0.69
	JTrans	0.70	0.82	0.80	**0.86**	0.57	0.77	0.73	**0.81**
	QBinDiff	**0.74**	0.84	0.82	**0.86**	0.60	0.76	0.73	0.80
	QBinDiff$_s$	-	**0.85**	**0.85**	0.80	-	**0.79**	**0.79**	0.73
100%	Bindiff	0.53	0.65	0.65	0.78	0.47	0.56	0.57	0.72
	Diaphora3	0.40	0.50	0.61	0.79	0.35	0.49	0.56	0.77
	GMN	0.23	0.26	0.33	0.53	0.18	0.22	0.28	0.48
	Asm2vec	0.15	0.17	0.32	0.59	0.17	0.21	0.35	0.60
	PalmTree	0.39	0.59	0.53	0.72	0.34	0.53	0.48	0.67
	JTrans	0.53	0.71	0.73	**0.86**	0.50	0.70	0.70	**0.81**
	QBinDiff	**0.65**	0.74	0.80	0.85	**0.59**	0.69	0.71	0.80
	QBinDiff$_s$	-	**0.77**	**0.84**	0.79	-	**0.73**	**0.78**	0.73

Table 7.6: Averaged f1-score comparison on OLLVM-14 obfuscations, `plain-obfuscated` setting (`P-vs-O`). **First**, second and third best differs are displayed for each obfuscation and obfuscation level.

This experiment considers both the OLLVM-14 and Tigress obfuscators, along with their respective obfuscation passes. All five projects are analyzed under two compiler optimization levels, `-O0` and `-O2`.[13] Different obfuscation levels are evaluated, that corre-

[13]The results for `-O0` and `-O2` are aggregated, as they exhibit identical overall trends. The only notable

Tigress	P-vs-0	General f1-score										Obfuscated f1-score									
		Mix	Mix + Split	Copy	Merge	Split	CFF	Virtualize	Opaque	Enc.A	Enc.L	Mix	Mix + Split	Copy	Merge	Split	CFF	Virtualize	Opaque	Enc.A	Enc.L
10%	Bindiff	0.80	0.72	0.80	0.76	0.74	0.82	0.81	0.79	0.84	0.84	**0.69**	0.02	0.21	0.56	0.09	0.75	0.72	0.69	0.86	0.81
	Diaphora3	0.75	0.67	0.76	0.73	0.70	0.76	0.74	0.75	0.78	0.79	0.34	0.02	0.46	0.34	0.08	0.52	0.04	0.66	0.77	0.78
	GMN	0.51	0.46	0.57	0.52	0.47	0.53	0.48	0.53	0.54	0.59	0.04	0.01	0.34	0.15	0.02	0.08	0.01	0.25	0.16	0.49
	Asm2vec	0.49	0.42	0.51	0.55	0.46	0.49	0.45	0.51	0.56	0.57	0.15	0.01	0.28	0.35	0.03	0.20	0.03	0.45	0.52	0.58
	PalmTree	0.74	0.67	0.78	0.75	0.71	0.78	0.72	0.76	0.78	0.81	0.33	0.02	0.49	0.45	0.08	0.58	0.08	0.65	0.62	0.84
	JTrans	0.80	0.74	0.82	0.78	0.76	0.84	0.80	0.81	0.85	0.85	0.55	0.02	**0.54**	0.35	0.04	0.72	0.36	**0.74**	0.87	0.88
	QBinDiff	**0.84**	**0.77**	**0.86**	0.82	0.81	0.87	0.83	**0.84**	**0.89**	**0.89**	0.55	**0.05**	0.25	0.59	0.19	0.73	0.35	0.69	**0.92**	**0.91**
	QBinDiff_s	-	-	0.85	**0.84**	**0.82**	**0.89**	**0.89**	0.79	0.83	0.84	-	-	0.25	**0.60**	**0.22**	**0.85**	**0.79**	0.65	0.90	0.86
50%	Bindiff	0.63	0.38	0.65	0.63	0.45	0.73	0.66	0.65	0.81	0.83	**0.52**	0.02	0.19	0.43	0.07	0.67	0.60	0.57	0.83	0.79
	Diaphora3	0.57	0.33	0.69	0.59	0.43	0.63	0.46	0.69	0.73	0.78	0.28	0.01	0.48	0.29	0.08	0.46	0.01	0.64	0.74	0.82
	GMN	0.30	0.19	0.49	0.35	0.26	0.32	0.27	0.38	0.38	0.58	0.02	0.01	0.36	0.13	0.02	0.06	0.00	0.20	0.13	0.50
	Asm2vec	0.27	0.16	0.41	0.40	0.26	0.30	0.18	0.40	0.49	0.59	0.10	0.00	0.29	0.28	0.04	0.14	0.01	0.39	0.50	0.64
	PalmTree	0.53	0.34	0.67	0.58	0.42	0.67	0.39	0.66	0.69	0.80	0.28	0.02	0.49	0.35	0.06	0.57	0.04	0.59	0.58	0.82
	JTrans	0.64	0.41	0.71	0.56	0.47	0.76	0.52	**0.74**	0.63	0.85	0.46	0.01	**0.54**	0.16	0.03	0.69	0.23	**0.71**	0.67	**0.90**
	QBinDiff	**0.68**	**0.45**	**0.75**	0.70	0.56	0.79	0.68	**0.74**	**0.87**	**0.89**	0.49	**0.03**	0.26	0.51	0.17	0.72	0.53	0.66	**0.90**	**0.90**
	QBinDiff_s	-	-	0.74	**0.73**	**0.58**	**0.87**	**0.81**	0.68	0.82	0.83	-	-	0.26	**0.58**	**0.20**	**0.84**	**0.77**	0.59	0.88	0.87
100%	Bindiff	0.33	0.10	0.48	0.44	0.22	0.60	0.51	0.47	0.77	0.80	0.23	0.01	0.20	0.28	0.06	0.56	0.48	0.41	0.80	0.68
	Diaphora3	0.27	0.09	0.64	0.38	0.25	0.46	0.10	0.61	0.66	0.76	0.19	0.01	0.50	0.28	0.07	0.43	0.01	0.61	0.71	0.78
	GMN	0.10	0.05	0.42	0.23	0.13	0.12	0.11	0.24	0.25	0.57	0.01	0.01	0.35	0.12	0.02	0.05	0.00	0.19	0.12	0.49
	Asm2vec	0.08	0.04	0.29	0.29	0.13	0.11	0.02	0.32	0.43	0.56	0.08	0.00	0.24	0.32	0.03	0.11	0.00	0.37	0.48	0.65
	PalmTree	0.34	0.14	0.59	0.45	0.24	0.61	0.14	0.58	0.59	0.79	0.27	0.01	0.50	0.32	0.05	0.59	0.02	0.56	0.56	0.78
	JTrans	**0.46**	**0.21**	0.62	0.32	0.28	0.68	0.20	**0.66**	0.60	0.83	**0.43**	0.01	**0.54**	0.14	0.03	0.68	0.16	**0.68**	0.66	**0.89**
	QBinDiff	0.40	0.19	**0.65**	0.57	0.36	0.71	0.49	0.63	**0.85**	**0.87**	0.33	**0.02**	0.26	0.48	0.15	0.70	0.46	0.61	**0.89**	0.87
	QBinDiff_s	-	-	0.64	**0.61**	**0.39**	**0.84**	**0.72**	0.56	0.81	0.82	-	-	0.27	**0.55**	**0.18**	**0.83**	**0.72**	0.53	0.86	0.84

Table 7.7: Averaged f1-score comparison on Tigress obfuscations, `plain-obfuscated` setting (`P-vs-0`). **First**, second and third best differs are displayed for each obfuscation and obfuscation level.

spond to the proportion of functions that are initially obfuscated. The 10% obfuscation level simulates a situation where the defender operates under a constrained obfuscation budget, due to limitations in memory or computational capacity. This is a common characteristic of embedded systems, where MCUs offer limited storage and processing capabilities. In such environments, only a set of critical functions can feasibly be obfuscated for security purposes. The 50% obfuscation level reflects a practical guideline for achieving a reasonable tradeoff between security and performance. Applying this amount of obfuscation often requires a careful selection of the functions to obfuscate based on a deeper understanding of their criticality, as well as an awareness of system constraints. However, some of these critical functions may be especially sensitive to the computational overhead introduced by obfuscation, which can result in unacceptable performance degradation, especially in resource-constrained platforms like MCUs. Finally, the 100% level depicts the worst-case scenario from the attacker's perspective, in which the entire binary is obfuscated.

In contrast to previous experiments conducted on standard, unobfuscated binaries, this analysis reports two distinct f1-scores. The first, referred to as the general f1-score, is computed over all functions, while the second, the obfuscated f1-score, is calculated exclusively on the subset of obfuscated functions. Ideally, these two metrics should be closely aligned; a significant disparity, where the general f1-score is high but the obfuscated f1-score is low, would suggest that the diffing tool performs well only on unobfuscated functions and fails to effectively handle obfuscated ones. This nevertheless indicates that binary diffing tools can still reliably match unobfuscated functions, without being impeded by the presence of obfuscated ones. Together, these scores are critical for assessing the robustness of binary diffing tools in the presence of obfuscation.

Tables 7.6 and 7.7 report the results for OLLVM-14 and Tigress, respectively, and point out noteworthy findings across different dimensions of binary diffing in an obfuscated setting:

- **Overall effectiveness of binary diffing.** Among the evaluated tools, QBinDiff, QBinDiff$_s$, and JTrans emerge as the most effective, achieving f1-scores of 0.80 or higher across numerous obfuscation passes, particularly for data-oriented and intra-procedural obfuscations. In general, traditional binary diffing tools such as BinDiff, Diaphora, and QBinDiff tend to outperform binary similarity techniques combined with a matching algorithm, such as GMN and Asm2Vec. Notably, JTrans does not suffer from the performance limitations observed in GMN or Asm2Vec, thanks to its use of robust function embeddings that incorporate control-flow information, contrary to PalmTree, that presents lower performance scores. QBinDiff consistently surpasses other diffing tools by incorporating both CG structure and function-level similarity. In contrast, JTrans relies solely on embedding-based similarity, while BinDiff emphasizes CG-based match propagation.

distinction is that the performance under `-O2` is slightly lower than that of `-O0`, primarily due to inlining induced by `-O2` optimization level, which eliminates some functions from the binaries, thereby preventing their subsequent matching and leading to a f1-score drop.

- **Resilience to obfuscation, QBinDiff vs QBinDiff$_s$.** QBinDiff and QBinDiff$_s$ exhibit distinct behaviors under different obfuscation scenarios. QBinDiff$_s$ considerably improves performance for most inter and intra-procedural obfuscations relative to QBinDiff, achieving higher scores for both general and obfuscated f1-scores. In some cases, notably for the Virtualization pass, it boosts the obfuscated f1-score by up to 44 points. However, this improvement does not extend to data obfuscations and the Opaque pass, where QBinDiff achieves better results. This difference is due to the data-related obfuscation stable feature set used in QBinDiff$_s$: while features like assembly mnemonics are removed to reduce obfuscation-induced noise, some of these omitted features may still carry valuable information for accurate function matching. Consequently, entirely removing these features from QBinDiff ultimately leads to a decline in performance.

- **Difference between general and obfuscated f1-scores.** The gap between general and obfuscated f1-scores varies depending on the type of obfuscation and the diffing tool employed. Similarity-based tools such as GMN and Asm2Vec often struggle to maintain consistent performance across all function types, typically favoring matches between unobfuscated functions. In contrast, more robust diffing tools like BinDiff, and to a lesser extent, Diaphora, narrow this performance gap to around 10 f1-score points for intra-procedural and data obfuscations. JTrans and both QBinDiff variants reduce this gap even further, typically to about 5 points under lower obfuscation levels. QBinDiff$_s$, in particular, benefits from obfuscation-aware features that enhance its resilience. This result is somewhat unexpected, as one might assume that obfuscation would severely impair diffing performance. However, this trend does not persist for inter-procedural obfuscations: here, all diffing tools exhibit a substantial drop in performance, with f1-score differences sometimes exceeding 60 points. This leads to extremely poor performance for certain obfuscation schemes, with even the most effective binary differ barely reaching a f1-score of 0.02. Such a decline is likely due to two factors: similarity-based tools fail to model inter-procedural relationships, leading to substantial information loss; and CG-based tools rely on structural information that is heavily distorted by such obfuscations, rendering the matching process unreliable, especially under a strict one-to-one mapping constraint.

- **Impact of obfuscation type.** The results indicate that most diffing tools are relatively effective against intra-procedural and data obfuscations, even when a large portion of the binary is obfuscated. This is largely due to the fact that these obfuscation types tend to preserve CG structure, enabling the tools to initiate function matching reliably. In contrast, inter-procedural obfuscations cause a pronounced degradation in performance, particularly in terms of obfuscated f1-scores. This suggests that reverse engineers may encounter greater difficulty when analyzing such binaries and should rely on more adaptive tools, such as QBinDiff, which consistently achieve superior performance. From a software protection perspective, this finding implies that inter-procedural obfuscations should be prioritized when

the objective is to resist reverse engineering through diffing.

- **Obfuscator robustness.** On average, binaries obfuscated with Tigress yield lower f1-scores compared to those obfuscated with OLLVM-14. While OLLVM-14 does not include inter-procedural obfuscations, which are particularly challenging for diffing, its data and intra-procedural obfuscations are also less aggressive. As a result, Tigress-obfuscated binaries tend to score 10 to 20 points lower. This performance gap is largely due to Tigress's use of more sophisticated transformations such as Virtualization, as well as its more aggressive implementation of shared passes like Opaque.

- **Effect of obfuscation level.** Interestingly, increasing the degree of obfuscation, i.e., the proportion of functions obfuscated, results in only a slight decline in f1-scores, particularly for intra-procedural and data-related transformations. This trend is particularly pronounced for OLLVM-14, where performance remains nearly constant across different obfuscation levels. This behavior is not observed with Tigress, suggesting two possible interpretations: first, increasing the number of obfuscated functions does not necessarily enhance the binary's resistance to analysis, especially when using OLLVM; second, although OLLVM-14 passes have been updated for compatibility with LLVM-14, they may not be fully adapted to the latest optimization strategies introduced in that version. These optimizations tend to eliminate coarse-grained obfuscations, like those applied by OLLVM, thus making it easier for diffing tools to operate effectively.

7.5.4 Diffing two obfuscated variants

This Section is dedicated to the `obfuscated-obfuscated` experiment, in which two obfuscated versions of the same binary are compared. This scenario is particularly relevant for tracking the evolution of obfuscation techniques over time or for identifying binary variants that may be more vulnerable than others. Tables 7.8 and 7.9 present results analogous to those reported earlier in Section 7.5.3, but within this specific experimental setting.

Several aspects of this setup differ from the previous `plain-obfuscated` experiment. First, we focus on a selected subset of `obfuscated-obfuscated` pairs of obfuscated binaries, prioritizing intra-intra, inter-inter, and inter-intra combinations when available. This choice is motivated by the results of the first `plain-obfuscated` experiment, in which data-related obfuscations produced less informative outcomes compared to other obfuscation types. The Mix-Mix pairing denotes two binaries obfuscated using the same Mix scheme but with different random seeds. Second, results for QBinDiff_s are not reported in this setting. When comparing two obfuscated variants, particularly when the obfuscation types differ, it becomes challenging to identify a set of features that remains robust across different obfuscation types. Neither the CG nor the CFG retains structural consistency when, for example, one binary is obfuscated with Split and the other with Virtualization.

OLLVM-14	`O-vs-O`	General f1-score				Obfuscated f1-score			
		Mix - Mix	*CFF - Opaque*	*Opaque - Enc.A*	*CFF - Enc.A*	*Mix - Mix*	*CFF - Opaque*	*Opaque - Enc.A*	*CFF - Enc.A*
10%	BinDiff	**0.89**	**0.94**	**0.94**	**0.94**	**0.79**	**0.75**	0.77	0.78
	Diaphora3	0.84	0.89	0.90	0.89	0.61	0.70	0.74	0.73
	GMN	0.76	0.83	0.83	0.82	0.44	0.43	0.50	0.46
	Asm2vec	0.64	0.63	0.45	0.63	0.41	0.35	0.36	0.44
	PalmTree	0.76	0.83	0.82	0.83	0.47	0.50	0.52	0.57
	JTrans	0.86	0.93	**0.94**	**0.94**	0.63	0.74	**0.78**	**0.79**
	QBinDiff	0.87	0.93	0.92	0.93	0.69	0.73	0.74	0.76
50%	BinDiff	0.73	0.88	**0.89**	**0.91**	0.65	**0.70**	**0.73**	0.75
	Diaphora3	0.61	0.81	0.83	0.82	0.55	0.67	0.72	0.70
	GMN	0.50	0.64	0.65	0.64	0.36	0.37	0.42	0.41
	Asm2vec	0.44	0.44	0.18	0.44	0.37	0.30	0.15	0.37
	PalmTree	0.55	0.73	0.71	0.75	0.41	0.43	0.45	0.52
	JTrans	0.64	0.87	0.88	**0.91**	0.56	0.68	0.71	**0.76**
	QBinDiff	**0.75**	**0.90**	0.88	0.90	**0.67**	0.69	0.71	0.73
100%	BinDiff	0.64	0.66	0.79	**0.80**	0.55	0.56	0.65	0.64
	Diaphora3	0.51	0.62	0.73	0.64	0.51	**0.63**	**0.68**	0.63
	GMN	0.39	0.43	0.48	0.46	0.31	0.36	0.37	0.39
	Asm2vec	0.40	0.22	0.15	0.24	0.36	0.25	0.13	0.26
	PalmTree	0.48	0.55	0.60	0.64	0.34	0.37	0.40	0.43
	JTrans	0.59	0.72	0.81	0.79	0.49	0.59	0.67	**0.65**
	QBinDiff	**0.83**	**0.79**	**0.83**	**0.80**	**0.66**	0.59	0.66	0.62

Table 7.8: Averaged f1-score comparison on OLLVM-14 obfuscations, `obfuscated-obfuscated` setting (`O-vs-O`). **First**, second and third best differs are displayed for each obfuscation and obfuscation level.

Although this evaluation scenario appears more challenging than the `plain-obfuscated` case, the `obfuscated-obfuscated` results reflect a similar trend to previous findings. QBinDiff, JTrans, and BinDiff generally perform the best. However, JTrans exhibits diminished performance in this context, as it was trained on unobfuscated code and is consequently more dependent on features present in unprotected binaries.

Both general and obfuscated f1-scores remain reasonably high for binaries that do not employ inter-procedural obfuscation. In particular, scores of at least 0.70 are observed for many binary pairs at low levels of obfuscation. While performance declines as the proportion of obfuscated functions approaches 100%, the results nonetheless demonstrate that diffing between obfuscated variants remains feasible, especially when dealing with

Tigress	O-vs-O		General f1-score										Obfuscated f1-score									
			Mix - Mix	Split - Merge	Merge - Copy	Split - Copy	CFF - Split	CFF - Merge	CFF - Copy	CFF - Opaque	Opaque - Enc.A	CFF - Enc.A	Mix - Mix	Split - Merge	Merge - Copy	Split - Copy	CFF - Split	CFF - Merge	CFF - Copy	CFF - Opaque	Opaque - Enc.A	CFF - Enc.A
10%	BinDiff	0.81	0.68	0.79	0.72	0.77	0.80	0.83	0.82	0.83	0.87	**0.70**	0.08	0.24	0.16	**0.35**	**0.68**	0.50	**0.75**	**0.83**	**0.77**
	Diaphora3	0.74	0.65	0.74	0.68	0.71	0.73	0.77	0.76	0.77	0.79	0.40	0.09	**0.42**	0.17	0.18	0.51	**0.53**	0.71	0.79	0.64
	GMN	0.62	0.50	0.67	0.56	0.59	0.62	0.68	0.64	0.67	0.70	0.09	0.03	0.30	0.06	0.05	0.16	0.10	0.22	0.33	0.11
	Asm2vec	0.55	0.47	0.55	0.46	0.50	0.37	0.56	0.56	0.58	0.61	0.19	0.05	0.31	0.08	0.11	0.27	0.21	0.40	0.57	0.25
	PalmTree	0.72	0.66	0.74	0.67	0.72	0.73	0.77	0.77	0.78	0.80	0.22	0.09	0.38	0.17	0.21	0.39	0.41	0.62	0.69	0.57
	JTrans	0.79	0.70	0.79	0.71	0.77	0.78	0.82	0.82	0.83	0.86	0.43	0.05	0.37	0.15	0.26	0.49	0.50	**0.75**	**0.83**	0.73
	QBinDiff	**0.83**	**0.73**	**0.82**	**0.75**	**0.81**	**0.83**	**0.85**	**0.85**	**0.86**	**0.88**	0.53	**0.15**	0.25	**0.20**	0.33	0.66	0.48	0.69	0.80	0.69
50%	BinDiff	0.55	0.40	0.57	0.38	0.46	0.59	0.60	0.63	0.71	0.80	0.40	0.06	0.18	0.11	0.24	0.43	0.38	0.51	0.66	**0.69**
	Diaphora3	0.53	0.40	0.60	0.37	0.39	0.52	0.58	0.65	0.69	0.67	0.41	0.07	**0.35**	0.15	0.13	0.36	**0.43**	0.64	**0.72**	0.55
	GMN	0.32	0.25	0.43	0.24	0.27	0.32	0.36	0.34	0.40	0.42	0.09	0.02	0.18	0.04	0.03	0.07	0.06	0.11	0.18	0.05
	Asm2vec	0.32	0.21	0.29	0.19	0.20	0.27	0.29	0.32	0.50	0.39	0.20	0.04	0.16	0.06	0.07	0.20	0.13	0.28	0.53	0.16
	PalmTree	0.48	0.39	0.55	0.34	0.41	0.47	0.55	0.62	0.64	0.66	0.26	0.05	0.25	0.11	0.15	0.25	0.33	0.51	0.54	0.45
	JTrans	0.57	0.43	0.59	0.36	0.45	0.53	0.62	**0.72**	0.55	0.61	0.38	0.02	0.24	0.11	0.19	0.24	**0.43**	**0.66**	0.50	0.47
	QBinDiff	**0.61**	**0.48**	**0.63**	**0.43**	**0.52**	**0.67**	**0.66**	**0.72**	**0.78**	**0.81**	**0.43**	**0.11**	0.19	**0.17**	**0.28**	**0.55**	**0.43**	0.61	0.70	0.64
100%	BinDiff	0.49	0.25	0.36	0.14	0.13	0.38	0.24	0.40	0.39	0.53	0.32	0.04	0.08	0.05	0.06	0.15	0.13	0.21	0.40	0.48
	Diaphora3	0.67	0.23	0.32	0.22	0.14	0.32	0.37	0.55	0.39	0.33	**0.61**	0.03	**0.15**	**0.14**	0.09	0.24	0.36	**0.53**	**0.54**	0.42
	GMN	0.30	0.18	0.25	0.10	0.08	0.23	0.13	0.16	0.16	0.13	0.18	0.01	0.04	0.03	0.02	0.04	0.04	0.02	0.06	0.04
	Asm2vec	0.41	0.06	0.08	0.08	0.05	0.16	0.08	0.18	0.33	0.14	0.35	0.03	0.06	0.05	0.05	0.17	0.09	0.18	0.44	0.13
	PalmTree	0.61	0.28	0.36	0.18	0.20	0.42	0.36	0.54	0.43	0.44	0.44	0.03	0.08	0.09	0.12	0.23	0.29	0.40	0.33	0.35
	JTrans	**0.71**	0.27	0.35	0.19	0.21	0.34	0.43	0.64	0.36	0.41	0.52	0.01	0.06	0.09	0.15	0.12	**0.38**	0.51	0.29	0.34
	QBinDiff	0.70	**0.32**	**0.41**	**0.25**	**0.24**	**0.61**	**0.46**	**0.66**	**0.48**	**0.56**	0.51	**0.07**	0.12	0.13	**0.18**	**0.49**	**0.38**	0.49	0.39	**0.60**

Table 7.9: Averaged f1-score comparison on Tigress obfuscations, `obfuscated-obfuscated` setting (`O-vs-O`). **First**, second and third best differs are displayed for each obfuscation and obfuscation level.

intra-procedural or data obfuscations that preserve CG structure.

These findings suggest that data and intra-procedural obfuscations, although effective at impeding manual reverse engineering, do not entirely prevent binary similarity or diffing tools from working. Notably, these types of obfuscation are the most commonly implemented in open source or freely available obfuscators. In contrast, comparisons involving binaries obfuscated with inter-procedural passes are substantially more difficult. In such cases, f1-scores drop substantially, often falling below 0.50 once obfuscation exceeds 50%, highlighting the limitations of current tools in handling transformations that disrupt the program's function relationships.

7.5.5 Extension to the BinKit dataset

As discussed in Section 7.2.1, extending this experimental study on diffing resilience against obfuscation to an additional dataset is essential for improving the generalizability of our findings.

Only the general f1-score is shown, as the obfuscation is applied at the program level directly, contrary to the function-level of the ObfuBench dataset. The results, summarized in Table 7.10, align with the trends observed in our earlier experiments, though the differences are even more pronounced. Binary similarity tools exhibit much lower performance than binary diffing tools. Among the latter, QBinDiff, and especially its variant QBinDiff$_s$, consistently outperforms both BinDiff and Diaphora.

These results, obtained on a distinct dataset, provide further validation of our earlier findings and strengthen the generalizability of our contributions.

Binkit	Plain-obfuscated					Obfuscated-obfuscated				
	bcf	*cpio*	*cflow*	*ccd2cue*	*a2ps*	*bcf*	*cpio*	*cflow*	*ccd2cue*	*a2ps*
BinDiff	0.9	0.63	0.78	0.94	0.7	0.8	0.42	0.61	0.84	0.44
Diaphora3	0.66	0.6	0.71	0.71	0.63	0.57	0.45	0.4	0.57	0.39
GMN	0.41	0.39	0.30	0.53	0.22	0.40	0.39	0.31	0.53	0.23
Asm2vec	0.37	0.29	0.22	0.55	0.15	0.34	0.25	0.19	0.38	0.13
PalmTree	0.73	0.65	0.72	0.82	0.54	0.64	0.51	0.48	0.69	0.36
JTrans	0.86	0.80	0.84	0.90	0.69	0.70	0.55	0.55	0.66	0.42
QBinDiff	0.96	0.92	0.91	0.98	0.82	0.9	0.82	0.82	**0.91**	0.7
QBinDiff$_s$	**0.97**	**0.94**	**0.93**	**0.99**	**0.87**	**0.92**	**0.86**	**0.86**	**0.91**	**0.80**

Table 7.10: Averaged f1-score comparison for the BinKit obfuscated dataset. **First**, second and third best differs are displayed for each obfuscation and obfuscation level.

7.5.6 Real-world examples

Finally, this Section extends the previous experiments, conducted on realistic yet simulated datasets, to real-world obfuscated samples. As in Chapter 6, both XTunnel and

Rabobank are considered, and only the diffing tools that achieved satisfactory performance on the simulated dataset are evaluated, as reported in Sections 7.5.3 and 7.5.4: BinDiff, JTrans, QBinDiff, and QBinDiff$_s$.

XTunnel

We replicate both the previous `plain-obfuscated` and `obfuscated-obfuscated` experiments by diffing three XTunnel samples: `XTunnelPlain` and the two obfuscated samples `XTunnelObf1` and `XTunnelObf2`.

In the `plain-obfuscated` experiment, ground truth is constructed manually by first aligning functions with identical hashes, followed by manual reverse engineering to identify additional matches. This process is labor-intensive and may introduce bias, particularly due to discrepancies in the number of primary and secondary functions, often the result of the compiler inlining, which complicates accurate function matching. As a result of this uncertainty, the diffing pair `XTunnelPlain` and `XTunnelObf1` leaves 431 primary and 1211 secondary functions unmatched. Similarly, the pair `XTunnelPlain` and `XTunnelObf2` results in 417 unmatched primary functions and 931 unmatched secondary functions.

In the `obfuscated-obfuscated` experiment, which involves diffing the pair `XTunnelObf1` and `XTunnelObf2`, constructing the ground truth manually is even more challenging than in the previous case. To approximate the matches, the unobfuscated binary `XTunnelPlain` is used as a reference: its corresponding matches with each obfuscated version are leveraged to infer alignments between the two obfuscated samples. However, this approach provides only a partial view of the actual correspondence, as it may amplify biases inherited from the initial ground truths used in the mapping process.

The results of these two experiments are presented in Tables 7.11 and 7.12, respectively. While all diffing tools, except JTrans which performs notably below the others, achieve high overall f1-scores on the full set of functions, only QBinDiff and QBinDiff$_s$ yield promising results on the subset of obfuscated functions. In contrast, JTrans and BinDiff perform poorly in this regard. Notably, the performance of QBinDiff$_s$ on this real-world malware is particularly encouraging, demonstrating that a resilient diffing approach can scale effectively in practical scenarios.

Based on the QBinDiff$_s$ results, precision-recall curves are generated for the two obfuscated XTunnel samples and are presented in Figure 7.7. These curves are constructed by varying the threshold that determines how many diffing output matches are considered, and by analyzing the resulting balance between precision and recall. Concretely, the diffing output matches are first sorted in decreasing order of similarity. Starting with the top-ranked match, precision and recall are computed relative to the complete ground truth. At this initial stage, if the single selected match is correct, precision equals one, and recall typically remains low since only one correct match has been retrieved out of the entire ground truth set. The evaluation then proceeds iteratively by incrementally incorporating additional matches according to their ranking, recomputing precision and recall at each step, until all matches have been considered. Plotting the sequence of these precision-recall pairs produces the precision-recall curve, which explicitly captures

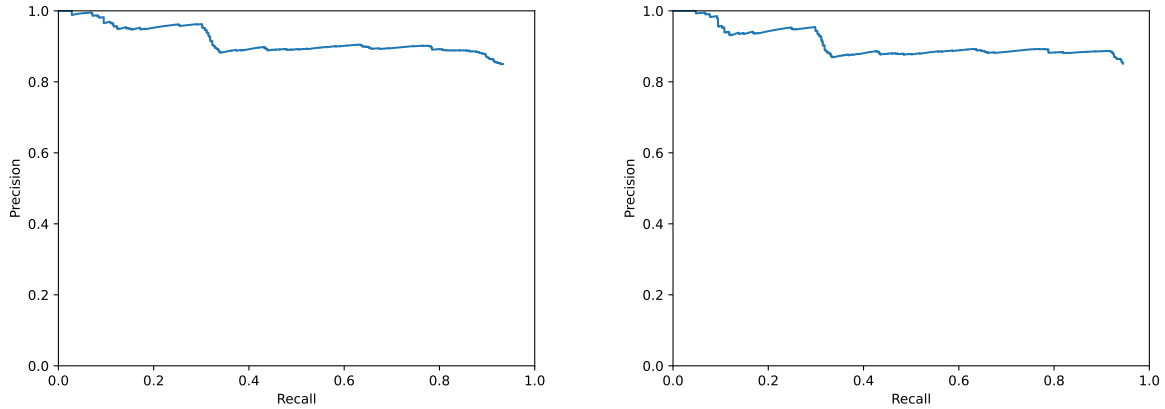


Figure 7.7: Precision-recall curves of `XTunnelObf1` (left) and `XTunnelObf2` (right) for the QBinDiff$_s$ diffing results.

the inherent trade-off between the two metrics. The resulting XTunnel precision-recall curves exhibit a favorable profile, closely resembling the precision-recall curve of an ideal classifier, thereby demonstrating the robustness of the diffing procedure in this case.

	Differ	General f1-score	Obfuscated f1-score
`XTunnelPlain` vs `XTunnelObf1`	BinDiff	0.957	0.156
	JTrans	0.812	0.026
	QBinDiff	0.939	0.757
	QBinDiff$_s$	0.952	0.802
`XTunnelPlain` vs `XTunnelObf2`	BinDiff	0.967	0.303
	JTrans	0.882	0.033
	QBinDiff	0.96	0.787
	QBinDiff$_s$	0.963	0.915

Table 7.11: f1-scores for the `plain-obfuscated` experiment variant between the unobfuscated sample `XTunnelPlain` and two obfuscated samples `XTunnelObf1` and `XTunnelObf2`.

For the `obfuscated-obfuscated` experiment, the results warrant more cautious interpretation, especially the f1-scores on obfuscated functions which are notably high, as these are derived through inferred alignments based on the previously established ground truths, potentially introducing bias.

Rabobank

To the best of our knowledge, no unobfuscated version of Rabobank is publicly available. Consequently, we limit our analysis to the `obfuscated-obfuscated` setting, comparing

`XTunnelObf1` vs `XTunnelObf2`	Differ	General f1-score	Obfuscated f1-score
	BinDiff	0.816	0.824
	JTrans	0.712	0.38
	QBinDiff	0.804	1
	QBinDiff$_s$	0.81	1

Table 7.12: f1-scores for the `obfuscated-obfuscated` experiment variant between the obfuscated samples `XTunnelObf1` and `XTunnelObf2`.

the `libnative-lib.so` binaries from versions 35.0 and 37.1. As in the XTunnel case, ground truth must be manually constructed; however, in this instance, no prior reference is available apart from the matches obtained by aligning functions with identical hashes.

Establishing this manual ground truth proved substantially more challenging than in the XTunnel case. Despite considerable effort, several function matches could not be determined, suggesting that the two binaries are, in fact, highly dissimilar and likely reflect significant differences in their corresponding source code. As a result of these limitations, the diffing pair `libnative-lib.35.0.so` and `libnative-lib.37.1.so` leaves 79 primary and 73 secondary functions unmatched.

The results of this `obfuscated-obfuscated` experiment are presented in Table 7.13. Since all functions appear to be obfuscated[14], the general and obfuscated f1-scores coincide and are hereafter referred to simply as the f1-score. In contrast to the XTunnel experiment, the scores are substantially lower, indicating that these diffing and similarity tools face greater difficulty aligning the two binaries. Notably, JTrans achieves the highest performance, closely followed by the QBinDiff variant, whose stable release slightly underperforms the default configuration, and then by BinDiff.

`libnative-lib.35.0` vs `libnative-lib.37.1.so`	Differ	f1-score
	BinDiff	0.385
	JTrans	0.525
	QBinDiff	0.464
	QBinDiff$_s$	0.439

Table 7.13: f1-scores for the `obfuscated-obfuscated` experiment variant between the obfuscated samples `libnative-lib.35.0.so` and `libnative-lib.37.1.so`.

While the number of matched functions across versions is relatively close, suggesting at first glance that semantic differences might be limited, further examination reveals that the two versions differ more substantially. Specifically, despite both binaries being stripped, some functions still retain names, albeit obfuscated. These are dynamically loaded at runtime through the `JNI_OnLoad` function provided by the Java Native Interface (JNI).[15] However, the number of such functions is inconsistent across versions,

[14] With the exception of very small routines, such as trampolines.

[15] For instance, examples of such names include `BKld0` or `YzqDWEt`.

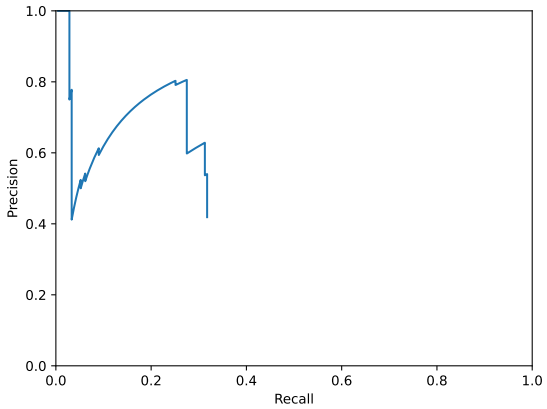


Figure 7.8: Precision-recall curve of the Rabobank libnative binary for the QBinDiff$_s$ diffing results.

pointing to potentially significant dissimilarities between the samples and providing a plausible explanation for the observed degradation in diffing scores.

The associated precision-recall curve for the QBinDiff$_s$ diffing results is shown in Figure 7.8. In contrast to the precision-recall curves obtained for XTunnel, which exhibit a consistent and satisfactory trend, this curve shows a pronounced drop in performance after incorporating only the first few matches. Such an atypical precision-recall profile is more likely to reflect inaccuracies in the constructed ground truth, as previously noted. We suspect that, although the two binary versions are closely related, they nonetheless contain substantial semantic differences which render the ground truth somewhat artificial and thereby account for the degraded results observed in comparison with the XTunnel case.

7.6 Limitations and future works

This experimental study, conducted on both unobfuscated and obfuscated binaries, represents an initial step toward a deeper understanding of binary code analysis in the presence of obfuscation. However, several limitations must be acknowledged:

- The binary diffing and similarity tools evaluated in this study, like most such tools, rely on disassembly as the foundational step for analysis. We initially assumed that disassemblies produced by tools such as IDA-Pro or Ghidra are sufficiently accurate. In practice, however, these disassemblers may struggle with complex scenarios involving anti-disassembly techniques or unconventional compiler optimizations. As a result, all tools inherit the limitations of relying on potentially flawed disassembly outputs.

- Our experiments are constrained to one-to-one function matching, which may be

inadequate in cases where obfuscation or compiler optimizations result in one-to-many or many-to-one function transformations. This limitation is non-trivial, and although some practical solutions have been proposed [79], no academic work has yet demonstrated their effectiveness. While many-to-many alignment does not in itself constitute a particular challenge, its applicability to graphs, in particular, binary diffing, remains an open research question.

- As discussed in Section 6.5, compiler behavior may interfere with or even neutralize applied obfuscations. Higher optimization levels increase the likelihood that obfuscation effects are altered or undone. Therefore, determining whether an obfuscation has been successfully applied, and remains intact in the final binary, is non-trivial and lies beyond the scope of this study. Consequently, in the absence of deeper insight into this aspect, our evaluation methodology may be subject to bias, and our results overly optimistic.

- This work primarily focuses on individual obfuscation techniques, with the exception of the Mix and Mix + Split configurations. In the Mix setting, a Control Flow Graph Flattening (CFF) pass is followed by another intra-procedural obfuscation and a data obfuscation pass. However, the order of these transformations was fixed, and it may not represent the most effective sequence for resisting reverse engineering efforts.

Nonetheless, this work lays the groundwork for numerous promising directions in future research:

- While this study strives to be as comprehensive as possible, practical constraints, particularly time, have limited the exploration of certain tools, models, and configurations. Future work could broaden the benchmarks by incorporating cross-architecture binaries, additional binary diffing and similarity tools, such as SAFE, and binaries compiled with a wider range of optimization levels, from `-O1` to `-Os`. Furthermore, investigating the impact of compiler inlining on binary diffing in obfuscated contexts represents a particularly promising subject.

- Obfuscation remains a complex research challenge, due to the interactions between compiler optimizations, obfuscation passes, and obfuscators. The ObfuBench dataset introduced in this work constitutes an initial attempt to address a gap in obfuscation research. Expanding it to include more obfuscators and software projects would greatly enhance its generalizability and utility. We intend to progressively extend the ObfuBench dataset over time.

- The relationship between obfuscation and diffing can be viewed as a form of adversarial interaction, akin to the ongoing arms race between defenders and attackers. This perspective invites further research into adversarial refinement on both sides. For instance, developers of obfuscation tools could leverage binary diffing techniques to detect vulnerabilities in their passes and iteratively strengthen them. Such a feedback loop could lead to the development of more robust and resilient obfuscation schemes.

7.7 Conclusion

This comprehensive study on binary diffing, conducted on both standard and obfuscated binaries, yields several valuable insights for the reverse engineering community. First, it presents a detailed ablation study of QBinDiff, a modular diffing tool with several noteworthy properties. This is followed by a systematic benchmarking of existing binary diffing and similarity tools on standard binaries, which enables the establishment of a performance ranking and highlights the benefits of modularity in diffing approaches. This modular design proves particularly effective when addressing the challenges posed by obfuscated binaries. The extensive experimental analysis of obfuscated programs offers a deep understanding of how different obfuscation techniques affect diffing performance, and how well each diffing tool is able to recover original function mappings. The results show that most tools remain effective against intra-procedural and data obfuscations, where structural features such as the CG remain largely intact. In contrast, inter-procedural obfuscations considerably degrade diffing accuracy by disrupting these structural relationships among the program. From a reverse engineering standpoint, this underscores the need for developing more sophisticated diffing algorithms capable of coping with structural disruptions introduced by inter-procedural obfuscations. Conversely, from a software protection perspective, these results advocate for a stronger emphasis on inter-procedural transformations, as they significantly hinder automated diffing.

Chapter 8

Deobfuscation

8.1 Introduction

Deobfuscation remains a central objective for reverse engineers, as it enables recovering the original or a semantically equivalent but simplified version, from its obfuscated form.

However, this task is inherently challenging, both theoretically and practically. Under specific assumptions, deobfuscation has been shown to be NP-easy [3], while certain obfuscation techniques have been formally proven to induce PSPACE-complete deobfuscation problems [24]. These theoretical insights are complemented by a range of practical studies, which propose diverse strategies to deobfuscate binaries. Nevertheless, to the best of our knowledge, none of these efforts have yet fully leverage advanced ML, especially Reinforcement Learning (RL), to tackle this problem.

In this Chapter, we present the first exploratory investigation into the use of ML-based techniques for deobfuscation, with a particular focus on RL. Section 8.2 introduces the problem, outlines our motivations, and surveys the state of the art in related areas such as program synthesis [62]. Section 8.3 then details the approach proposed in this thesis for applying ML to deobfuscation.

8.2 Problem formalization and state of the art

Deobfuscation refers to the process of partially or fully removing obfuscation from binary code. It can be applied at different levels of abstraction, for instance, directly on raw textual representations, such as in the simplification of MBA expressions [88, 119, 120], on assembly code [14] or obfuscated source code [96]. Regardless of the abstraction level, deobfuscation relies on several principles:

- **Semantic preservation.** The deobfuscated code must exhibit the same behavior as the original obfuscated code. No transformation can alter the program's functional output or side effects.
- **Structure validity.** Ideally, the resulting deobfuscated output should conform to the syntactic and structural rules of the output format. For instance, valid

deobfuscated assembly code should be structured as a sequence of instructions, each comprising a mnemonic followed by the appropriate operands. However, in certain contexts, such as source-code deobfuscation for example, the resulting deobfuscated code may fail to compile and thus violate strict syntactic constraints; nonetheless, it can still provide significant insight and utility for reverse engineering purposes.

- **Simplification.** The deobfuscated code should ideally be simpler than the original. However, there is no universal consensus on what constitutes 'simplified' code, as it can be interpreted in terms of readability, conciseness, or other criteria. In this thesis, we adopt the perspective that simplicity primarily targets human reverse engineers comprehension. It is important to note, however, that what is simpler for a human may not be considered simpler from a computational standpoint. For example, a logical formula containing universal quantifiers may be easier for a human to understand, whereas a Satisfiability Modulo Theories (SMT) solver might prefer an equivalent formula expressed as a longer propositional statement.

Given these constraints, deobfuscation can be framed as an instance of program synthesis, in which the input program serves as the specification to synthesize.

In traditional program synthesis, the goal is to construct a program that satisfies a given specification which may, in certain cases, be expressed as a subset of input / output pairs. In this context, this task bears similarities to supervised learning, where the objective is to learn a function mapping inputs to outputs. However, in program synthesis, the goal is not merely to approximate an abstract mathematical function, but to generate an explicit and executable program.

Usually, a synthesis engine generates candidate programs and evaluates them against the given specification. If a candidate satisfies the specification, it is accepted; otherwise, the process is repeated. This paradigm, where synthesis relies solely on the specification without access to internal program structure, is typically characterized as black-box program synthesis. The term black-box is adopted here in accordance with previous deobfuscation studies [14, 106]. It indicates that outputs are obtained solely by querying the program as a black-box model. The subsequent program synthesis, performed with the initial inputs and newly obtained outputs, is carried out without any access to or interaction with the original program. In this setting, the program to be synthesized needs only be executable to generate the input / output pairs.

By contrast, grey-box approaches combine program execution to obtain input / output behavior with program reading, directly allowing access to the program's internal contents.

In this work, deobfuscation corresponds more closely to a white-box synthesis task. The obfuscated program is fully known and accessible, and the objective is not to generate a program from scratch, but to simplify a known one while preserving its semantics. This positions deobfuscation as a problem of white-box program synthesis via symbolic compression: the aim is to transform the given obfuscated program into a functionally equivalent but more compact representation.

Program synthesis has given rise to a rich body of literature, encompassing several distinct theoretical paradigms:

- **Iterative search-based synthesis.** This strategy incrementally explores the space of candidate programs, usually guided by a context-free grammar. Programs, more and more complex, are enumerated and tested against the specification until a correct match is found. Although formal correctness is not guaranteed, verification is conducted over a finite set of input / output examples, often comprising randomly sampled or manually selected test cases. This method is exhaustive in principle but may become computationally expensive or infeasible as the search space grows [14, 31, 113, 85, 106].

- **Neural-guided program synthesis.** These approaches leverage DL models as a core mechanism for guiding program synthesis. They may either directly map input / output examples to candidate programs expressed in a domain-specific language [35, 45], or they may serve to bias a symbolic search over program candidates, effectively prioritizing promising regions of the program space and facilitating the discovery of reusable abstractions [42]. Existing work typically confine these methods to well-defined, constrained domains, such as visual or textual problem-solving tasks.

- **Reinforcement Learning (RL) approaches.** RL is a subdomain of ML and frames the synthesis problem as an agent interacting with an environment: the agent chooses actions, namely candidate program modifications, observes the resulting behavior on the environment and receives a reward based on how closely the output matches the specification. The objective is to learn a policy that maximizes cumulative rewards over time. In the context of program synthesis, this reward can be defined according to various criteria, such as functional correctness, simplicity, or syntactic validity [114, 19]. RL-based synthesis is commonly modeled over token sequences, like source code or symbolic representations.

While many contributions in the program synthesis literature adopt a general perspective and address a wide range of problem types [42, 114], a subset, summarized in Table 8.1, focuses specifically on deobfuscation. These deobfuscation-oriented approaches typically target a single obfuscation technique, such as MBA or Virtualization. While this narrow focus simplifies the formulation of the deobfuscation problem, it substantially restricts applicability in real-world contexts, where obfuscation typically involves multiple combined passes to strengthen protection.

Broadly, existing methods fall into four main categories: expert-designed algorithms tailored to specific obfuscation techniques and grounded in domain knowledge, generic program synthesis approaches, algebraic methods specifically designed for MBA deobfuscation, and ML-based solutions.

Historically, expert-crafted deobfuscation algorithms were the first to emerge. These methods rely on detailed reverse engineering insights and deep understanding of obfuscation mechanisms to develop deterministic algorithms capable of removing specific

	Obfuscation					Algorithm class	Method	Obfuscators
	MBA	Opaque Predicates	Virtualization	Intra-procedural	Other			
[153] (2015)	✗	✗	✔	✗	✔	Expert-derived	Taint analysis	Code Virtualizer EXECryptor Themida VMProtect
[14] (2017)	✔	✗	✔	✗	✗	Program synthesis	MCTS	Tigress Themida VMProtect
[126] (2018)	✗	✗	✔	✗	✗	Expert-derived	Taint analysis and SE	Tigress
[139] (2019)	✗	✔	✗	✗	✗	ML Decision Tree	Symbolic term-frequency	OLLVM Tigress
[45] (2020)	✔	✗	✗	✗	✗	ML LSTM	Assembly tokens	MBA algorithm [160]
[31] (2020)	✔	✗	✔	✗	✗	Program synthesis	Dynamic SE	Tigress
[106] (2021)	✔	✗	✔	✗	✗	Program synthesis	Iterative local search	Reuse [31, 14]
[11] (2021)	✔	✗	✗	✗	✗	Program synthesis	Dynamic SE	Malware
[120] (2023)	✔	✗	✗	✗	✗	Algebric computation	System solving	Reuse [45, 14, 31] MBA-obfuscator MBA-solver
[96] (2024)	✗	✗	✗	✔	✗	Expert-derived	CFG skeleton analysis	Tigress

Table 8.1: Existing deobfuscation approaches.

transformations. Initially confined to relatively simple intra-procedural Control Flow Graph Flattening (CFF) [153], such approaches have since been extended to support more complex intra-procedural obfuscation schemes [96].

Algebraic methods [88, 119, 120] have been proposed to specifically address the deobfuscation of MBA expressions, leveraging formal mathematical definitions to simplify such constructs. However, these approaches often fall short when confronted with more recent variants of MBA expressions, particularly polynomial MBA, which lie outside the scope of the simplifications afforded by traditional algebraic techniques.

Among the various approaches explored in the literature, program synthesis appears to yield the most promising results for deobfuscation, leading to the development of tools such as Syntia [14], QSynth [31], and Xyntia [106]. These tools all share the foundational idea of incrementally constructing a candidate program that satisfies a predefined set of input / output specifications. However, they differ significantly in the strategies employed to navigate the search space. Syntia leverages Monte-Carlo Tree Search (MCTS) to synthesize small segments of binary code by exploring the space of possible programs. QSynth, in contrast, adopts a table-driven strategy: it precomputes large equivalence tables that map input / output vectors to semantically equivalent expressions. During syn-

thesis, it attempts to retrieve suitable expressions directly from these tables; if a match is not found, it recursively decomposes the problem into smaller subexpressions. Xyntia replaces MCTS with an Iterated Local Search approach, aiming to improve efficiency in navigating the program space. Despite their innovations, each of these methods exhibits notable limitations. MCTS, as used in Syntia, has been shown to perform comparably to brute-force search, failing to effectively prioritize promising branches, an ability that underpins the theoretical advantage of MCTS. QSynth's reliance on exhaustive precomputed tables introduces scalability issues, as these tables become prohibitively large in both depth and variable count. Xyntia, while more scalable in principle, still inherits challenges inherent to local search methods in high-dimensional, discrete spaces.

In parallel, ML-based techniques are beginning to emerge in this domain. Early approaches use traditional algorithms, such as Decision Tree, to infer values of Opaque Predicates [139]. More advanced models, based on Long Short-Term Memory (LSTM) networks, have been applied for deobfuscating MBA expressions, aiming to learn a direct mapping from obfuscated expressions to their original semantics [45].

This state of the art review reveals several important gaps in current deobfuscation research:

- **Lack of modern ML-based solutions.** To date, no existing approach effectively addresses the deobfuscation problem using ML techniques that leverage recent advances in the field. In particular, RL remains entirely unexplored in the deobfuscation context.

- **Limited use of enriched code structures.** Most existing methods operate directly on raw text formats, such as mathematical expressions of MBAs, assembly code, or obfuscated source code, without exploiting richer, semantically informed representations. Notably, graph-based forms of binary code, which better capture structural and behavioral properties, have yet to be explored for deobfuscation.

- **Narrow focus on single obfuscation techniques.** The majority of current contributions target a specific obfuscation pass, often disregarding scenarios where multiple obfuscation layers are applied concurrently. While this constraint stems from the intrinsic difficulty of the task, it significantly limits the applicability of these techniques in real-world settings, where compound obfuscation is the norm.

To address these shortcomings, we pursue an exploratory research direction that investigates the use of RL methods for deobfuscation purposes, applied to graph-based representations of binary code.

8.3 Our approach

The deobfuscation problem, presented in this thesis as a form of white-box program synthesis via symbolic compression, can be summarized as follows:

- Obfuscated functions are represented using an enriched graph that jointly captures data-flow and control-flow. This graph conveys more comprehensive information about the binary code and contains all the elements required to resolve both data-oriented and intra-procedural obfuscations.
- A RL framework is used, in which a GNN agent iteratively simplifies the obfuscated graph by using graph edit operations until a valid deobfuscated graph is obtained.

An example of the desired deobfuscation process is illustrated in Figure 8.1. Although closely related to the classical graph edit problem, our objective is fundamentally different. The graph edit problem seeks to transform a source graph into a known target graph using the minimal number of edit operations. In contrast, our setting involves an obfuscated input graph, but the corresponding unobfuscated version is unknown, making the transformation goal implicit rather than explicitly defined.

We begin by introducing a new graph format, called Control Data Graph (CDG), as discussed in Section 8.3.1, before presenting our RL-based simplification algorithm in Section 8.3.2.

8.3.1 A new graph combining control and data-flow

Existing binary code graphs mostly consist of CFGs and CGs, which respectively capture relationships between Basic Blocks (BBs) and functions. However, these structures lack explicit representation of data-related information, which is often implicitly embedded within the BBs of the CFG.

Recent efforts have focused on designing richer graph types, capable of capturing more nuanced aspects of binary code. The motivation behind these developments is that more expressive graphs may yield higher-quality graph representation learning embeddings, which in turn can enhance performance across a variety of downstream tasks.

To the best of our knowledge, the first significant attempt to define such a graph led to the introduction of the Semantic Oriented Graph (SOG) [67]. The SOG is a heterogeneous graph in which nodes correspond to various assembly-level entities, including mnemonics, registers, and immediate values, interconnected through three distinct types of edges: control-flow edges, reflecting the branching logic of the CFG; data-flow edges, encoding data movements and def-use relationships; and effect edges, which model execution order and side effects introduced by certain instructions, such as store operations. This graph is constructed from Ghidra's IR, P-code, and possesses several notable properties that make it particularly well suited for binary code analysis:

- **Instruction reordering invariance.** In binary code, successive instructions may be semantically independent, that is, their execution order does not affect the program's behavior. For instance, the two instructions `mov eax, 0` and `mov ebx, 1` can be safely reordered without altering the code semantics. The SOG captures this independence by omitting any ordering or dependency edges between such instructions. This invariance is beneficial when training graph-based representation learning models, as it encourages the generation of identical embeddings for

binaries that differ only in the ordering of semantically independent instructions, an expected and desirable property for semantic-preserving transformations.

- **Instruction shifting invariance.** In binary code, sequences of instructions may be displaced within a code segment due to the insertion of additional instructions by the compiler. In many cases, such shifts do not alter the overall semantics of the binary code. The SOG, by disregarding the positional information of assembly instructions, is inherently agnostic to instruction shifting. Consequently, two semantically equivalent assembly sequences that differ only by such shifts will be mapped to the same embedding.

- **Register substitution robustness.** In many contexts, certain registers can be used interchangeably when they are not otherwise constrained or already used. To account for this, the SOG abstracts away register-specific identifiers and is a register-agnostic graph that better reflects the underlying semantics, rather than syntactic artifacts tied to register naming.

In addition, the SOG incorporates specific nodes, such as Phi nodes. These nodes are introduced when the value of a register is not statically determinable, typically in cases where the register is conditionally modified. For instance, if a register is updated only when a specific branch condition is satisfied, its value at a merge point may depend on the control-flow path taken. This ambiguity is modeled by a Phi node, which reflects the possible multiple states the register may assume, thereby preserving the semantic complexity inherent in the program's execution.

Unfortunately, this graph format poses several practical challenges. When processed for handling by a GNN, it lacks essential metadata such as node and edge types, making it difficult to interpret or relate the graph back to the original assembly code. Additionally, the construction of the SOG is time-consuming, the resulting graphs are often too large for efficient processing, especially for complex functions, and the generation script is both difficult to understand and cumbersome to adapt.

As a result, rather than directly reusing the SOG, we draw inspiration from its underlying principles to develop our own graph format, the Control Data Graph (CDG). The CDG retains the core ideas of the SOG while offering a more lightweight and adaptable structure for downstream analysis.

Definition 8.3.1 (Control Data Graph (CDG))**.** Given a function f within a binary program P, its Control Data Graph (CDG) is a directed graph $G = (V, A, \mu, \psi)$, where V is the set of nodes, representing code elements, A the set of edges, representing the relationships between the code elements, μ: $V \to \tau_V$ is a node mapping function that assigns to each node a type, with τ_V = (Input, Output, Constant, Operation, Phi, Memory), ψ: $A \to \tau_A$ the edge mapping function that assigns to each edge a type, with τ_V = (data, control).

The CDG node types can be described as follows:

- **Input nodes.** They represent the input registers at function entry. A graph may contain one or multiple inputs, and these nodes have no predecessors, like the initial `RAX` and `RBX` in Figure 8.1.
- **Constant nodes.** They correspond to fixed values, such as immediates, that remain unchanged. These likewise have no predecessors.
- **Output nodes.** They denote the registers holding the final computed values of the functions. There may be one or more per graph, and they have no successors, like the last `RAX` and `RCX` in Figure 8.1.
- **Operation nodes.** They capture the operations performed within the function. Every function contains at least one such node. Unary operation nodes, such as the `not` operation, have a single predecessor, while binary operation nodes, like the `xor` operation, have two predecessors.
- **Phi nodes.** They serve a similar purpose to the SOG Phi nodes, capturing the multiple possible states of a register that may arise from different execution paths converging at a given point.
- **Memory nodes.** They are employed to partially model accesses to memory.

The graph is further structured with two types of edges: data-flow and control-flow edges. The integration of both data-flow and control-flow edges enables the simultaneous representation of data and control dependencies, thereby theoretically supporting the deobfuscation of both data-oriented and intra-procedural obfuscation within a unified RL framework. In the illustrative example, computed on a simple MBA, of Figure 8.1, all instructions lie within a single Basic Block (BB), so no control-flow redirection occurs and only data-flow edges exist.

Given a disassembled function, the CDG is constructed by first translating the assembly instructions into the Pcode IR. The resulting sequence of Pcode instructions is then analysed to recover def-use relationships, ultimately yielding the CDG.

The CDG complies with several structural constraints that ensure the syntactic validity of the graph:

- A `COPY` operation node has exactly one predecessor.
- A unary operation node has one predecessor, whereas a binary operation node has two predecessors.
- An output node has exactly one predecessor, which must be either an operation node or a Phi node.
- Whereas control-flow edges may only connect operation nodes, data edges are only permitted between specific node types, namely:
  - Between two operation nodes;

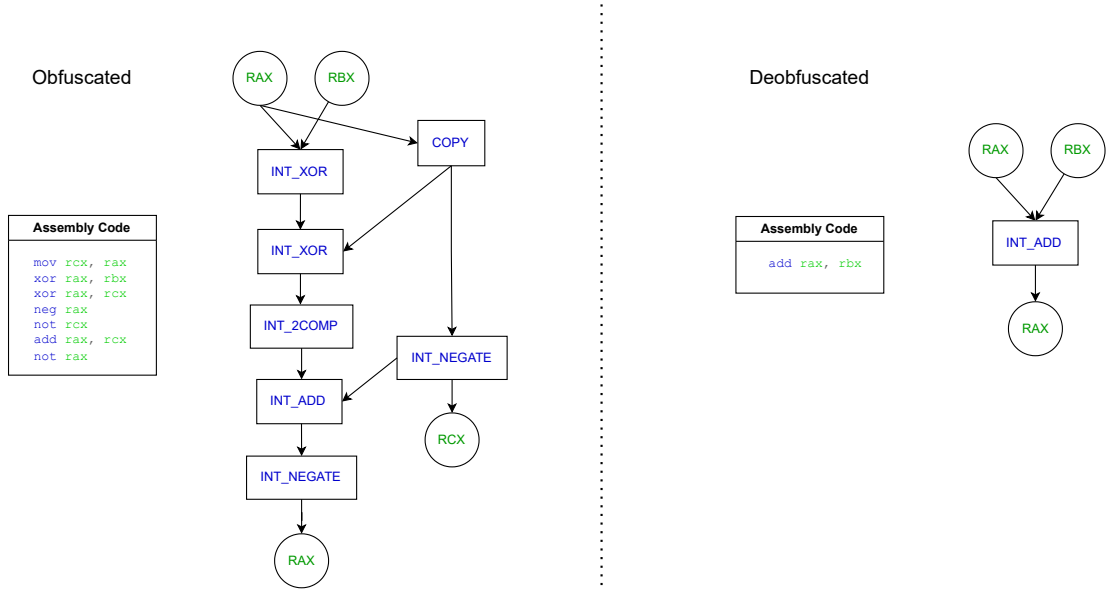


Figure 8.1: Illustration of the input / output desired deobfuscation process, where the CDG of an obfuscated assembly function is simplified to recover the original unobfuscated function.

- From an operation node to an output node;
- From an operation node to a Phi node;
- From an input node to an operation node;
- From a constant node to an operation node;
- From a Phi node to an operation node;
- From a Phi node to an output node;
- From a memory node to an operation node;
- From a memory node to a Phi node;
- From an operation node to a memory node;
- From a Phi node to a memory node.

Generating such CDG can vary in difficulty depending on the specific function under consideration. Accordingly, we define different function categories: functions exhibiting only data-flow ($\mathbb{F}$), functions with branching conditions ($\mathbb{F}_b$), functions involving memory address computations ($\mathbb{F}_m$), functions that include Phi nodes ($\mathbb{F}_{phi}$), functions with function calls ($\mathbb{F}_c$), and finally, functions containing loops ($\mathbb{F}_l$). A given function may belong to multiple categories simultaneously. Although these classes do not form a strict hierarchy of difficulty with respect to deobfuscation, deobfuscating functions that involve

memory address computations or loops is substantially more challenging than handling functions with simple data flow.

Figures 8.2, 8.3, 8.4, 8.5, 8.6 and 8.7 provide illustrative examples of assembly code, for each function complexity class, and their corresponding CDG. In these examples, red arrows represent control flow, while black arrows denote data flow. Two operation nodes may be connected by a control-flow edge if one is executed after the other.[1] Certain edges are indexed to indicate operand order.[2]

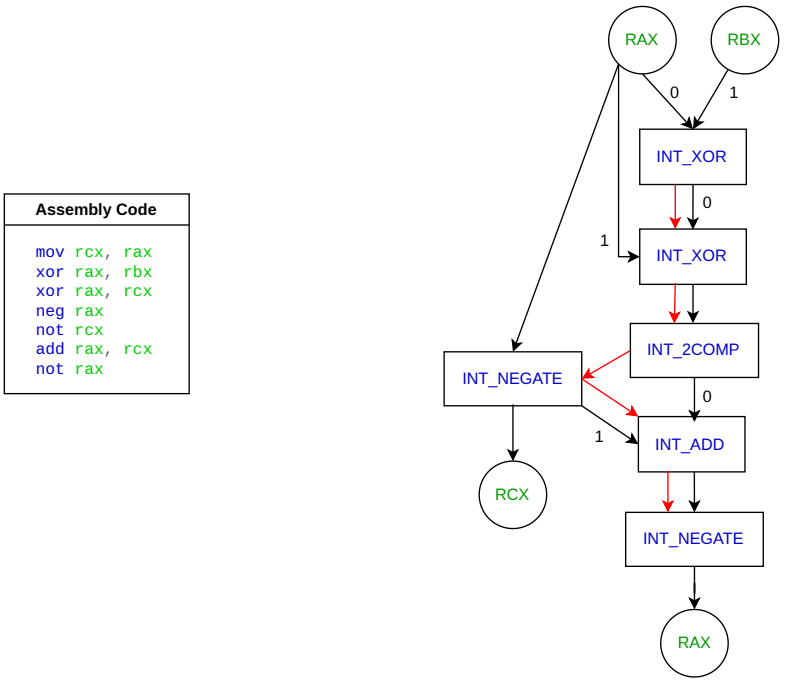


Figure 8.2: Assembly code from the class $\mathbb{F}$ and its corresponding CDG.

However, this CDG formulation suffers from several limitations. First, memory management is not fully developed, as memory nodes do not carry additional information about the corresponding memory locations. For instance, considering the assembly instruction `mov eax, [rbx+0x39]`, the graph can successfully represent the addition `rbx + 0x39`, but it cannot provide further details about the corresponding memory location, instead representing the instruction in a simplified manner as a memory node linked

[1] Within a Basic Block (BB), control flow is implicit, since instructions are executed in sequential order, though it is nonetheless present.

[2] For instance, the CDG of the instruction `xor rax, rbx` contains two edges: the edge linking the register node `rax` to the operation node `xor` is indexed as 0 (first operand), while the edge linking the register node `rbx` is indexed as 1 (second operand). Interchanging these indices is generally not permissible, as it would alter program semantics; `xor rax, rbx` is not semantically equivalent to `xor rbx, rax`.

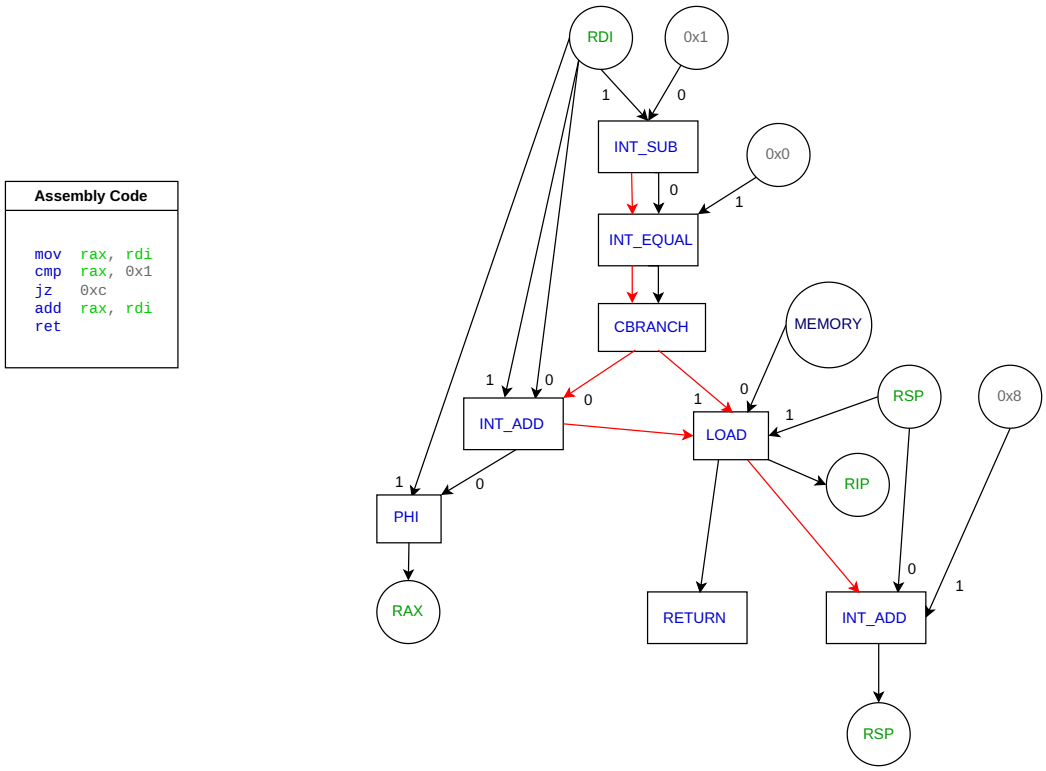


Figure 8.3: Assembly code from the class $\mathbb{F}_b$ and its corresponding CDG.

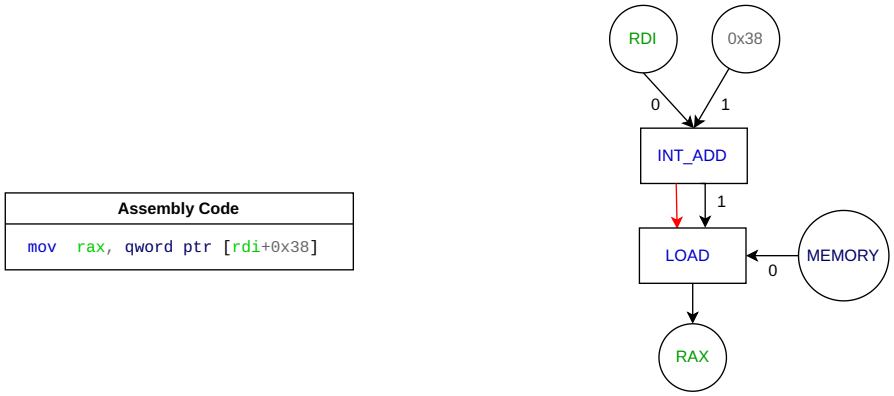


Figure 8.4: Assembly code from the class $\mathbb{F}_m$ and its corresponding CDG.

to a load operation, as shown in Figure 8.4. As a consequence, pointer aliasing is not addressed in this graph construction, as implementing it would require substantial ad-

Assembly Code

```
mov  qword ptr [rsp+0x40], rax
mov  rbp, qword ptr [rsp]
mov  rbx, qword ptr [rsp+0x8]
add  rsp, 0x10
```

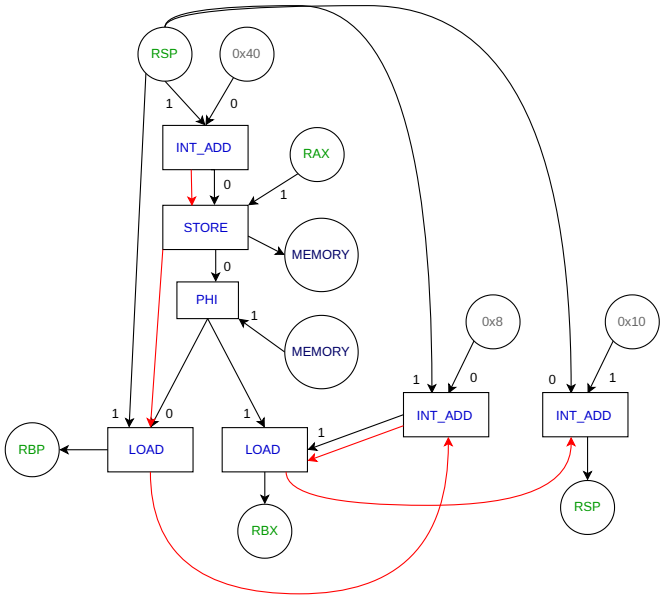


Figure 8.5: Assembly code from the class $\mathbb{F}_{phi}$ and its corresponding CDG.

Assembly Code

```
mov  rdi, 0x4
call 0xcafe
imul rcx, rax
```

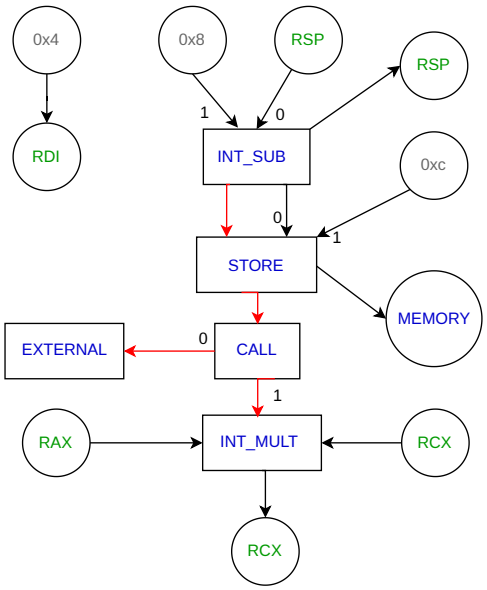


Figure 8.6: Assembly code from the class $\mathbb{F}_{c}$ and its corresponding CDG.

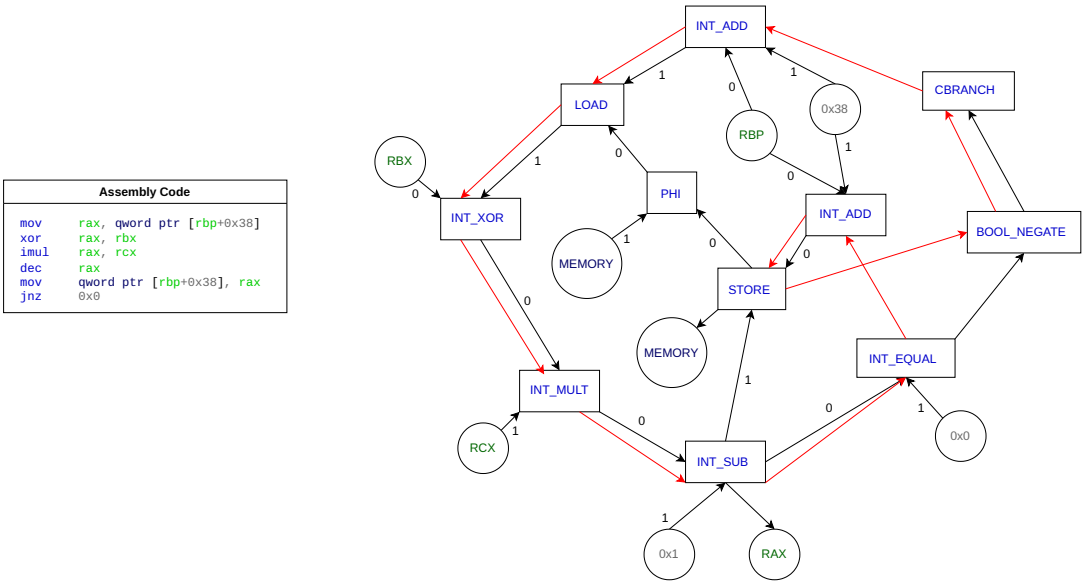


Figure 8.7: Assembly code from the class $\mathbb{F}_l$ and its corresponding CDG.

ditional work beyond the current scope. Thus, the CDG may be unsound for certain classes of graphs, particularly those involving loops or memory computations. In the case of loops, this limitation can be mitigated through loop unrolling.

8.3.2 RL on graphs

Despite the extensive research on RL, its application to graph-structured environments remains relatively underexplored. While classical RL has yielded a wide range of algorithms and techniques [146, 111, 130, 63], extending these methods to graphs presents unique challenges, primarily due to the nature of the underlying data structure.

- **Large and potentially unbounded discrete action space dependent on the input environment.** In many conventional RL settings, such as maze navigation, agents typically operate within a fixed and predefined action space (e.g., moving left, right, up, or down), independent of the specific problem instance. By contrast, graph-based environments often induce an action space whose size is directly determined by the input. A concrete example is the Travelling Salesman Problem (TSP), where the number of possible actions, namely the cities to visit, scales directly with the order of the input graph. Consequently, the action space grows combinatorially with input size. Moreover, while in most conventional RL settings, the action space is either discrete and finite, such as selecting a direction in a navigation task, or continuous but bounded, such as choosing an acceleration

value in robotic control, the setting considered here is more complex. Specifically, since the agent can apply arbitrary graph edit operations to simplify an obfuscated graph, particularly introducing new nodes, the action space, though discrete, can become unbounded and effectively infinite. This characteristic poses significant challenges for existing RL algorithms, which are generally not designed to cope with environments involving such large and discrete action spaces [40, 70, 161].

- **Order sensitivity of the action space.** In conventional RL environments, such as maze navigation, the action space is fixed and semantically stable: selecting an action such as move left always carries the same interpretation, regardless of the specific instance of the task. By contrast, in graph-based environments, the action space is inherently tied to the input graph. Selecting a node, for example the fourth node, as an action does not have a consistent semantic meaning across graphs, or even across different permutations of the same graph, since node identifiers are arbitrary and lack intrinsic meaning. This permutation-dependence, directly related to the graph isomorphism problem, implies that two isomorphic graphs with different node labeling could result in distinct action spaces, potentially leading to divergent agent behaviors, even though the underlying task remains identical. This label sensitivity undermines the generalization of learned policies, as the RL agent may overfit to specific permutations of node indices. Training a separate agent for every possible labeling is clearly intractable, particularly as graph size grows. This challenge highlights the need for RL methods that are invariant or equivariant to node permutations.

Despite the aforementioned limitations, RL on graphs holds significant promise. Notably, the connection between structured prediction problems, where the goal is to incrementally transform a structured input into a structured output, and RL has been formally established [94]. In particular, RL-based approaches have been shown to outperform classical structured prediction methods in several settings. This suggests that graph-structured prediction tasks, which inherently involve complex and sequential decision-making, could benefit from RL algorithms specifically designed to operate over graph representations.

The RL framework, designed to simplify obfuscated CDG, is characterized by:

- **An agent and environment.** In this work, the agent is a GNN with multiple output layers, denoted as multi-headed, that computes embeddings for various elements of the graph. The environment is the current obfuscated graph to deobfuscate. Employing a GNN is particularly advantageous, as it mitigates the dependency of the action space on arbitrary node labeling by producing equivariant and invariant embeddings. A schema of the GNN architecture is shown in Figure 8.8.

- **An action space.** At each step, the agent must select an action and, when applicable, its parameters. For instance, if the chosen action is to remove a node, the agent must select which node to delete; similarly, removing an edge requires

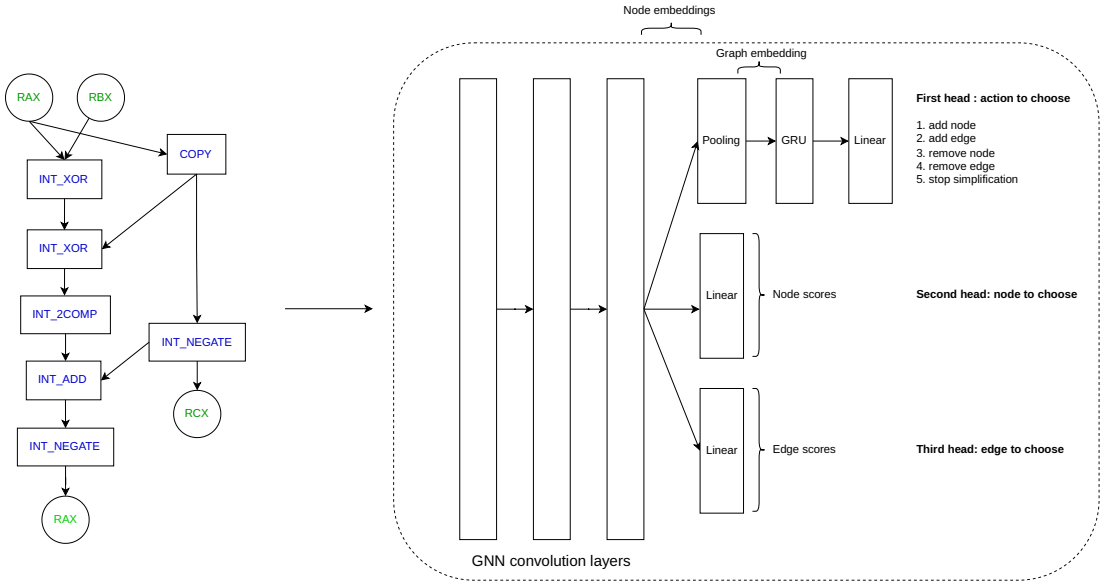


Figure 8.8: Schema of the multi-headed GNN agent architecture.

selecting the edge, adding an edge requires specifying its source and destination, and adding a node requires selecting the appropriate parameters. The agent may also decide to terminate the simplification process.

- **A reward function.** This function evaluates the simplified CDG and returns a reward score based on several criteria. First, the resulting graph must remain structurally valid, demonstrating that the agent has learned to preserve the syntactic constraints of the CDG. Second, it must be semantically equivalent to the original obfuscated graph. Third, it should be smaller than the initial graph, otherwise the simplification has failed. Careful design of the reward function is crucial to avoid undesirable RL behaviors. For example, if structural and semantic correctness are rewarded too heavily relative to simplification, the agent may learn that doing nothing yields a higher return than actively simplifying the graph.

The overall RL-based simplification proceeds as follows. Starting from an obfuscated CDG, the GNN agent attempts to simplify the graph by iteratively applying graph edit operations, while preserving both the structural syntax validity and the semantics of the original graph. The agent is allowed to perform at most L such modifications. At each step, it may either decide to terminate the simplification process or carry out a graph edit operation:

- choosing a node to remove;

- choosing an edge to remove;
- choosing to add a node and selecting all the parameters to create it;
- choosing a non-existing edge to add.

To choose its actions, the GNN stacks multiple graph convolution layers. Then, a first head predicts the action type among the available options (remove node, remove edge, add node, add edge, stop). A second head assigns scores to all nodes in the graph, while a third head produces scores for candidate positive and negative edges. If the selected action is to remove a node, the node is sampled according to the node-score-based distribution[3]; if an edge needs to be deleted, an existing edge is sampled from the positive-edge-score-based distribution; and if an edge has to be added, a source and destination are sampled from the negative-edge-based distribution. To add a node, the agent must also choose the node type and associated parameters: for example, adding an operation node implies determining its mnemonic, an input or output node requires to choose its name and a constant its value.[4]

Some actions are inherently invalid, such as adding a self-loop, and are therefore masked with zero probability. Other actions may temporarily violate structural constraints but are still permitted with the expectation that subsequent edits may restore a valid state. For example, deleting an operation's predecessor might later be compensated by reattaching valid predecessors elsewhere. The agent continues this editing process until it has applied L modifications or decides to stop, thereby concluding the episode.

In certain situations, the agent may select a valid action for which no valid parameters are available. For example, if all nodes in the graph have already been removed, resulting in an empty graph, and the agent attempts to delete another node, the action cannot be executed as no nodes remain. Similarly, the agent might choose to add an edge, but all possible candidate edges may be structurally invalid, such as forming self-loops, resulting in a zero probability of selection for all non-existing candidate edges. In such cases, the action itself is valid in principle, but no admissible parameters exist to realize it. When this occurs, the episode is terminated.

After each sequence of edits, a reward is computed based on three criteria:

- Structural validity, which is checked using a syntactic algorithm, ensuring that all CDG well-defined constraints are respected, such as no operation node has more than two predecessors;

[3]The effect of a node deletion operation depends on the type of node selected. If the node is a constant, input, or output node, it is simply removed along with all its adjacent edges. In contrast, if the node is a Phi or operation node, it undergoes contraction rather than direct deletion, and its incoming edges are redirected to its successor.

[4]Selecting only these elements is not sufficient to ensure that the resulting simplified graph is structurally valid. In particular, adding an operation node not only requires specifying its opcode, but also its predecessors, i.e., the operation's arguments. Since these correspond to edges, they must be introduced in subsequent steps by the agent, if it decides to connect the newly added operation node to other nodes in the graph.

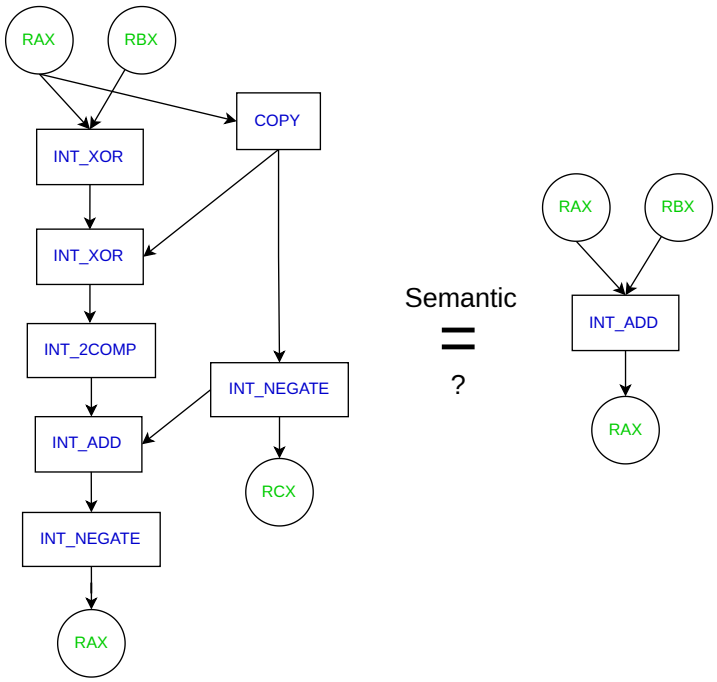


Figure 8.9: Semantic equivalence checking between two CDG using a SMT solver.

- Degree of simplification, typically measured as the difference in node count between obfuscated and simplified graphs;
- Semantic equivalence between obfuscated and simplified graphs. Ideally, it should be assessed using Satisfiability Modulo Theories (SMT)-based verification, which is the only way to guarantee correctness. However, such guarantees may be unattainable in practice due to the presence of cycles or prohibitive verification cost, particularly when more than two variables or complex nonlinear operations are involved. A weaker, albeit approximate, alternative is to evaluate the proportion of input values for which the simplified graph produces correct outputs. In this work, semantic checking between two CDGs, the first representing a MBA and the second its deobfuscated counterpart, as illustrated in Figure 8.9, is asserted using a SMT-solver, Z3 [33], restricted to the quantifier-free bit-vector logic, denoted `QF_BV`. In this context, let us define F_{cdg} as the set of CDG graphs, and $Expr$ as the set of expressions, where an expression denotes bit-level operations performed on bit-vectors. We further define Var as the set of free variables encoded as bit-vectors of width n; $\Diamond_u$ as the set of unary operators such as $\Diamond_u : Expr \mapsto Expr$; and $\Diamond_b$ as the set of binary operators such as $\Diamond_b : (Expr, Expr) \mapsto Expr$. Additionally, we define the function $Transform : F_{cdg} \mapsto Expr$, which converts a CDG into a symbolic expression[5], and the function $Solve : Expr \mapsto \{True, False\}$. Under these definitions,

[5]The *Transform* function cannot produce a correct expression for certain types of CDG, particularly

checking whether two CDGs, cdg_1 and cdg_2, are semantically equivalent reduces to verifying that $Solve(Transform(cdg_1) \neq Transform(cdg_2)) = False$. Although such SMT solving is theoretically sound for MBA obfuscation[6], it becomes intractable in the presence of loops or branching, which commonly arise in intra-procedural obfuscation.

8.3.3 Experimental settings and first results

Initially, we restrict our experiments to the simplest $\mathbb{F}$ function class, which comprises functions exhibiting only data-flow and no control-flow branching, that corresponds to functions obfuscated using MBA transformations. Specifically, our first evaluation focuses on the MBA example illustrated in Figure 8.1. Our long-term goal is to progressively extend this approach to learn deobfuscation strategies for increasingly complex obfuscation types, associated to more challenging function classes.

We begin our evaluation with the REINFORCE algorithm [146], which is known to be conceptually simpler than the more advanced Proximal Policy Optimization (PPO) algorithm. The agent architecture integrates a GCN with a Gated Recurrent Unit (GRU), enabling it to retain the temporal evolution of the graph simplification process. This design allows the agent to preserve information about the original obfuscated graph, even after deleting nodes or edges, thereby maintaining awareness of the initial simplification objective. We employ a batched implementation with 64 parallel environments, trained over 500 epochs with a discount factor $\gamma = 0.99$ and a maximum of $L = 11$ steps per episode, sufficient to transform the selected MBA variant, presented in Figure 8.1, into its unobfuscated form. The progression of the average reward over time is illustrated in Figure 8.10.

Initially, the average reward increases steadily, suggesting early learning progress. However, it soon plateaus and declines brutally to zero. This pattern indicates that the algorithm converges to a suboptimal local minimum from which it cannot escape, repeatedly generating graph transformations that yield a zero reward. Figure 8.11 presents the five most frequently produced simplified graphs at the end of an episode.

Unfortunately, the leftmost and rightmost graphs both fail to preserve semantic correctness and violate the structural constraints of the CDG, despite being syntactically simpler. The third and fourth graphs maintain structural validity but do not guarantee semantic equivalence. Among the five graphs, only the second one satisfies both structural and semantic correctness, while also being more concise than the original obfuscated graph. As such, it can be considered a valid deobfuscated variant, even though it is not the minimal syntactically and semantically correct form. In this instance, the algorithm reaches a satisfactory intermediate state, achieving partial deobfuscation. Although not complete, this result constitutes a promising result and can still provide valuable support to an analyst. Such a non-minimal deobfuscation suggests that the agent ceases exploration prematurely, becoming trapped in a narrow region of the policy

those containing loops.

[6] As it does not introduce control-flow redirection.

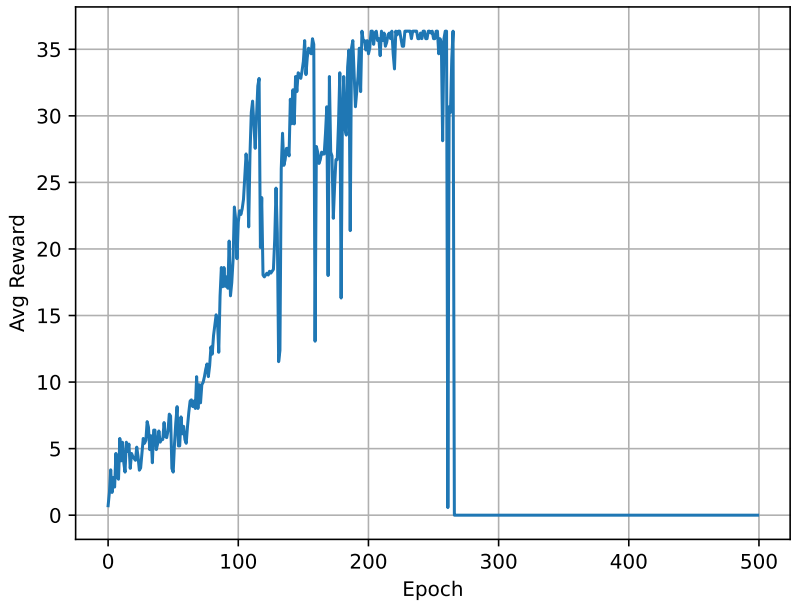


Figure 8.10: Average rewards depending on the number of epochs for the REINFORCE algorithm.

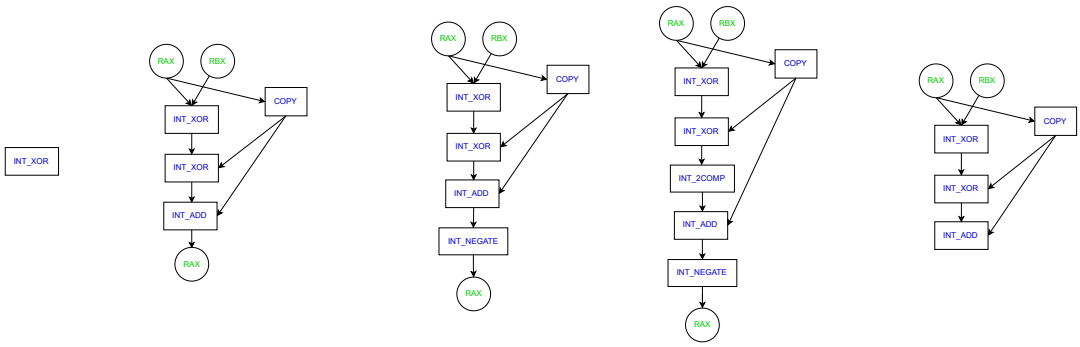


Figure 8.11: The five most frequently generated simplified graphs produced by the REINFORCE algorithm at the end of an episode are shown from left to right. Their respective numbers of occurrences are 13,904; 6,803; 2,115; 884; and 427.

space where it repeatedly generates the most familiar suboptimal graphs.

Such a result highlights the intrinsic difficulty of the problem. In the partially deobfuscated graph, shown in the second position from the left in Figure 8.11, achieving the minimally deobfuscated form would require removing only the two XOR nodes and the COPY node. However, reaching this state necessitates executing a very specific sequence

of actions, namely the successive deletion of exactly those nodes. This illustrates a key challenge: successful deobfuscation often depends on discovering and performing precise action sequences, which a RL agent may fail to encounter through exploration alone, and which can be even more difficult to generalize and learn from. This explains why the agent may produce partially deobfuscated results.

As a side observation, another graph that satisfies all three criteria (syntactic validity, semantic correctness, and simplification) is the original obfuscated graph with the COPY node removed. However, this variant can not be considered deobfuscated.

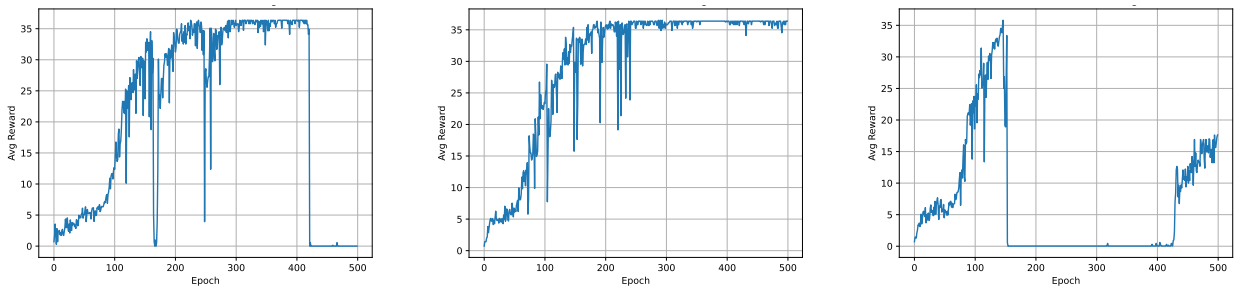


Figure 8.12: Average rewards, depending on the number of epochs, of the entropy-regularized, the intrinsic-enhanced and the combination of both REINFORCE algorithms.

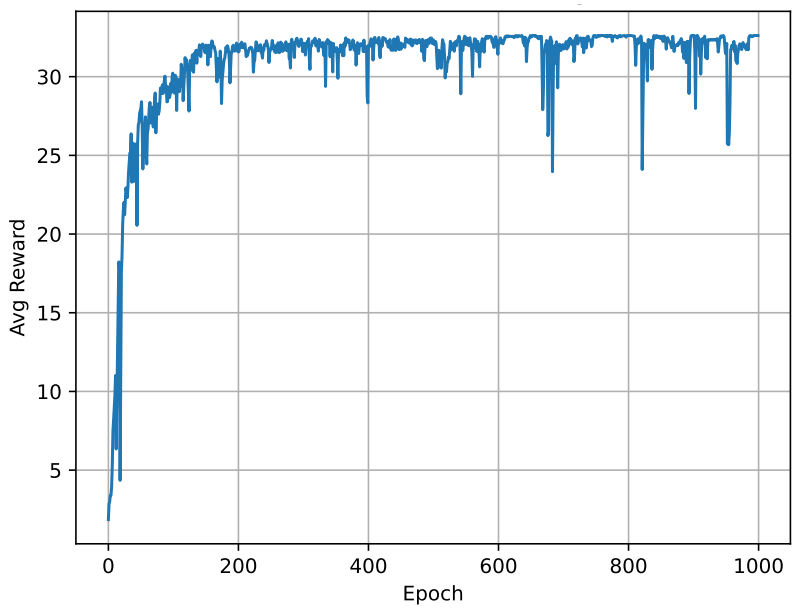


Figure 8.13: Average rewards depending on the number of epochs of the entropy-regularized agent with intrinsic rewards trained on multiple MBA variants of $x + y$.

To mitigate this limitation and encourage more effective exploration, we integrate en-

tropy regularization and intrinsic reward mechanisms into the learning process. Entropy regularization consists in adding a term to the loss function, derived from the entropy of the action and parameter distributions. This additional term penalizes low-entropy policies, which correspond to high certainty and premature convergence toward deterministic behavior, thereby mitigating the risk of reduced exploration. Intrinsic rewards provide a mechanism to incentivize exploration by supplementing the external reward signal with an additional term at each step. This auxiliary reward is designed to encourage the agent to visit underexplored states. In our setting, the intrinsic reward is defined as the inverse square root of the visitation count of graphs characterized by their number of nodes and edges. Consequently, graphs with specific structural properties that have been less frequently encountered yield a higher intrinsic reward, thereby promoting their exploration, while those already observed multiple times contribute proportionally less.

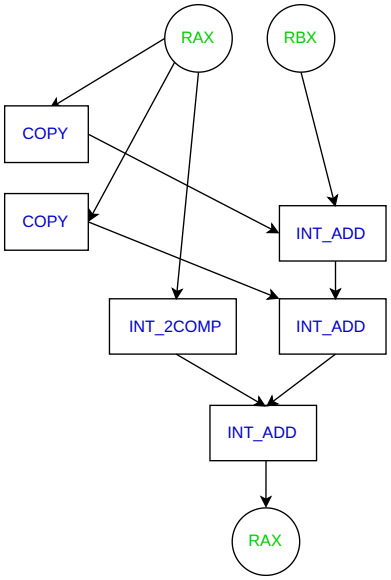


Figure 8.14: The first most frequently generated simplified graph produced by the REINFORCE algorithm, with entropy regularization and intrinsic rewards, on the Loki MBA dataset, at the end of an episode, with a number of occurrence of 27,868.

Figure 8.12 illustrates the impact of entropy regularization, intrinsic rewards, and their combination, respectively, on the REINFORCE algorithm. While entropy regularization alone does not lead to substantial improvements, aside from prolonging the reward plateau before decline, incorporating intrinsic rewards has a more pronounced effect. It successfully prevents the collapse of the average reward, sustaining a satisfactory performance level over time. As a result, the majority of the simplified graphs produced correspond to the second graph shown in Figure 8.11, indicating that the agent consistently achieves partial deobfuscation under this setting.

Such partial deobfuscation, however, is initially demonstrated on a single MBA variant, illustrated in Figure 8.1 for the expression $x + y$. To avoid the risk of the agent overfitting to a single graph structure, the experiment must be extended to encompass multiple graphs, corresponding to different MBA variants of the same expression $x + y$. For this purpose, we rely on the Loki-generated MBA, as introduced in Section 5.4. Specifically, all mathematical MBA variants of depth 1[7] of $x + y$[8] are translated into assembly then CDG, and incorporated to augment the agent's training dataset. The resulting average reward, measured on 1000 epochs for an entropy-regularized agent with intrinsic rewards, and presented in Figure 8.13, demonstrates that the agent is still able to generalize beyond a single instance and achieve valid and correct partial deobfuscations across diverse MBA variants. Since these variants introduce greater structural diversity, the corresponding deobfuscated graphs differ from those in Figure 8.11. For example, the graph displayed in Figure 8.14, both structurally valid and semantically correct, corresponds to the first most frequent deobfuscated graph of the Loki dataset, limited to MBA of depth 1 for the expression $x + y$.

8.4 Conclusion

In this Chapter, we formulate deobfuscation as a white-box symbolic compression problem. Each obfuscated function is represented using a CDG, a novel binary graph capturing both control and data-flow. This graph is then iteratively simplified through graph edit operations until a valid, simplified deobfuscated graph is produced by a GNN-based RL agent. Although time constraints precluded further exploration of the subject, the preliminary results are promising, demonstrating that partial deobfuscation of MBA obfuscation can be achieved using this approach.

[7] In this context, the depth of a Loki-MBA denotes the number of rewriting operations required, based on precomputed MBA equivalences, to transform an expression into a MBA [127].

[8] https://github.com/RUB-SysSec/loki/blob/main/experiments/experiment_10_mba_formula/data/add_depth1.txt

Chapter 9

Conclusion and perspectives

9.1 Contribution summary

This thesis aims to employ and evaluate representation learning methodologies in the context of obfuscation analysis. The proposed contributions can be viewed as a set of complementary tools designed to assist reverse engineers in their tasks. They are summarized as follows:

A new, diverse, and large-scale obfuscated dataset. To facilitate broader and more realistic experimentation in obfuscation analysis, this thesis introduced ObfuBench, a novel dataset specifically constructed for this purpose. By virtue of its scale and diversity, ObfuBench currently represents the most comprehensive resource available for tackling obfuscation-related research problems.

Obfuscation detection and characterization. A systematic and in-depth investigation was carried out on binary and multi-class classification in the context of obfuscation detection and characterization. This study examined and compared multiple dimensions of the problem, including the initial graph format, feature engineering strategies, ML models, and data partitioning schemes. Experimental results demonstrate that simple baselines, such as those leveraging graph-derived features or assembly distribution, can yield satisfactory performance. However, these are consistently outperformed by more advanced GNNs, provided that the GNNs are initialized with informative node features that capture fine-grained binary code properties, such as assembly or Pcode mnemonic frequencies. In contrast, coarse-grained features produce disappointing results. GNNs also exhibit superior generalization capabilities, particularly in scenarios where the data split is unfavorable to the model. Furthermore, this work highlights the significant impact of compiler optimization levels on detection performance, as well as the performance gains achievable by correcting labeling errors in the dataset. All findings are validated both on the ObfuBench and BinKit datasets, as well as on real-world obfuscated samples.

Experimental study of binary diffing and similarity in a standard and obfuscated contexts. Binary diffing in obfuscated environments has been largely overlooked in prior research. To address this gap, an extensive experimental study was

conducted, covering both unobfuscated and obfuscated scenarios. We first performed an ablation study of QBinDiff, deriving practical guidelines for its parameter tuning according to specific analysis objectives. Subsequently, a comparative evaluation of state of the art binary diffing and similarity approaches on unobfuscated binaries revealed the superiority of certain tools, most notably JTrans, BinDiff, and QBinDiff. These trends persist in obfuscated settings, regardless of whether one or both binaries are obfuscated. In particular, we demonstrate that a refined version of QBinDiff, leveraging features resilient to specific obfuscation types, achieves significantly better results than the standard configuration. The study further reveals that most data and intra-procedural obfuscations offer limited resistance to diffing-based attacks, regardless of obfuscation intensity. By contrast, inter-procedural obfuscations substantially hinder the effectiveness of binary diffing and similarity tools, thereby supporting their use in software protection contexts, particularly when employing aggressive obfuscators such as Tigress.

Graph-based deobfuscation with RL. This exploratory research investigated the potential of advanced graph representations for deobfuscation tasks. Specifically, we adapted RL algorithms to operate on graph-structured data with the goal of simplifying obfuscated graphs through graph edit operations, thereby recovering their deobfuscated counterparts. Although time constraints precluded the completion of all planned experiments, several promising results were obtained. Notably, partial deobfuscation of MBA obfuscations was achieved using the REINFORCE algorithm and a GNN-based agent, particularly when intrinsic rewards were employed.

9.2 Community contributions

The guiding principle of this thesis is that its contributions are intended to function as modular tools within a broader toolkit, enabling reverse engineers to select and combine them as needed to achieve optimal results. This approach aims to bridge the gap between academic research and practical reverse engineering. In line with this philosophy, the open-sourcing of the materials developed or enhanced during this work is of central importance. During the course of this thesis, QBinDiff was open-sourced in collaboration with Robin David and Riccardo Mori, while ObfuBench was released to facilitate future research on obfuscation detection. Whenever feasible, scripts and artifacts associated with our contributions have also been made publicly available to ensure reproducibility.

9.3 Future works

This work opens the way to numerous further improvements.

Disassembly dependency. Throughout this work, several simplifying assumptions were made that may not hold in practical scenarios. Notably, it was assumed that disassemblies produced by tools such as IDA-Pro or Ghidra are accurate. In reality, these tools can struggle to produce correct disassemblies, particularly when faced with advanced obfuscation techniques or unconventional compiler optimizations. Consequently, all tools relying on disassembly, whether to construct graph structures or to perform

binary diffing, inherit these limitations. Future research could therefore benefit from approaches that either bypass the disassembly step entirely, by working directly with raw binary code, or improve existing disassembly techniques. Such developments would represent significant progress for reverse engineering in obfuscated contexts.

Dataset limitations and extensions. Similarly, the scope of this thesis is restricted to `x64` binaries compiled with `-O0` and `-O2`. Extending this work to additional architectures and compilation settings would enhance its generalization potential. The dataset introduced here constitutes an initial effort to address the shortage of resources for obfuscation research, yet building a realistic obfuscated dataset entails substantial implementation challenges. Considerable effort was made to represent a wide variety of obfuscators and obfuscation schemes within the available resources. However, due to the inherent interplay between obfuscation and compiler optimizations, it is difficult to guarantee that obfuscations are preserved, especially at higher optimization levels such as `-O2`, where transformations may weaken or eliminate them. Expanding ObfuBench to include more obfuscators and obfuscations, in particular dynamic ones, a broader range of projects, and varied optimization levels would enable a deeper investigation into the interaction between obfuscation and optimization. Such an expansion could also support systematic studies of compilation behaviors and their influence on binary analysis tools, with particular interest in underexplored phenomena such as function inlining and the combined effects of optimization and obfuscation. We hope to incrementally enhance ObfuBench to address these challenges.

Towards a unified framework for practical obfuscation analysis. This thesis introduces several contributions, each addressing a specific stage of the obfuscation analysis pipeline. While these components have been studied independently, a promising direction for future work lies in their integration into a unified framework. Such a framework would enable a systematic workflow capable of detecting and characterizing obfuscation before applying either direct or indirect attacks. Moreover, extending the generalization of these methods to numerous real-world obfuscated binaries, evaluated under realistic deployment conditions, remains essential to uncovering practical limitations and identifying opportunities for further improvement.

Extending the deobfuscation work. Chapter 8 presents promising preliminary results that warrant further investigation. These findings open multiple research directions:

- **Agent and learning algorithm.** The current results were obtained using the REINFORCE algorithm with a GCN-based agent. Both choices have more advanced counterparts in the literature. Extending the proposed approach to alternative RL algorithms and GNN architectures could yield deeper insights into the problem and potentially improve performance.

- **Environment design.** In our formulation, the deobfuscation environment is modeled as a graph. Inspired by the recently introduced SOG, the CDG proposed in this work captures both data and control-flow information. This underlines the importance of designing richer graph formats and features that better encode

function semantics, a principle that extends beyond deobfuscation to other binary analysis tasks, such as obfuscation detection and characterization.

- **Reward design.** The reward function used in this study balances structural validity, semantic correctness, and a simplification factor. However, the simplification factor explored here is only one of many possible formulations. Likewise, current measures for structural and semantic correctness are binary, which can hinder learning in RL settings by providing sparse feedback. Developing soft metrics, such as rewarding semantic correctness proportionally to the number of test inputs producing correct outputs, could offer a denser learning signal. While this would relax the strict semantic correctness requirement, it could improve scalability to more complex obfuscations. Notably, SMT-based rewards may be practical for short MBA patterns but become computationally infeasible for more complex ones, making input / output-based evaluation essential for larger function classes, particularly those with branching logic.
- **Obfuscation Diversity.** Due to time limitations, this work focuses exclusively on MBA-based obfuscation. Extending the approach to other obfuscation schemes, particularly intra-procedural transformations, would provide a more comprehensive evaluation of its applicability.

In summary, this thesis offers several contributions to the field of binary code representation learning for obfuscation analysis. By addressing these identified limitations and pursuing the outlined research directions, we hope to inspire and support future advances in the discipline.

Bibliography

[1] Takuya Akiba et al. "Optuna: A next-generation hyperparameter optimization framework". In: *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*. 2019, pp. 2623–2631.

[2] Frances E. Allen. "Control flow analysis". In: *Proceedings of a Symposium on Compiler Optimization*. Urbana-Champaign, Illinois: Association for Computing Machinery, 1970, pp. 1–19. ISBN: 9781450373869. DOI: 10.1145/800028.808479. URL: https://doi.org/10.1145/800028.808479.

[3] Andrew Appel. "Deobfuscation is in NP". In: *Princeton University, Aug* 21.2 (2002).

[4] Sebastian Banescu, Martín Ochoa, and Alexander Pretschner. "A Framework for Measuring Software Obfuscation Resilience against Automated Attacks". In: *2015 IEEE/ACM 1st International Workshop on Software Protection*. 2015, pp. 45–51. DOI: 10.1109/SPRO.2015.16.

[5] Boaz Barak et al. "On the (im) possibility of obfuscating programs". In: *Annual international cryptology conference*. Springer. 2001, pp. 1–18.

[6] Sébastien Bardin, Robin David, and Jean-Yves Marion. "Backward-Bounded DSE: Targeting Infeasibility Questions on Obfuscated Codes". In: *2017 IEEE Symposium on Security and Privacy (SP)*. 2017, pp. 633–651. DOI: 10.1109/SP.2017.36.

[7] Aurélien Bellet, Amaury Habrard, and Marc Sebban. "A survey on metric learning for feature vectors and structured data". In: *arXiv preprint arXiv:1306.6709* (2013).

[8] Dimitri Bertsekas. "New auction algorithms for the assignment problem and extensions". In: *Results in Control and Optimization* 14 (2024), p. 100383. ISSN: 2666-7207. DOI: https://doi.org/10.1016/j.rico.2024.100383. URL: https://www.sciencedirect.com/science/article/pii/S2666720724000134.

[9] Tim Blazytko. *Automated Detection of Control-flow Flattening*. https://synthesis.to/2021/03/03/flattening_detection.html. 2021.

[10] Tim Blazytko. *Automated Detection of Obfuscated Code*. https://synthesis.to/2021/08/10/obfuscation_detection.html. 2021.

[11] Tim Blazytko. *Practical MBA Deobfuscation with msynth*. https://synthesis.to/2021/11/11/practical_mba_deobfuscation.html. 2021.

[12] Tim Blazytko. *Writing Disassemblers for VM-based Obfuscators*. https://synthesis.to/2021/10/21/vm_based_obfuscation.html. 2021.

[13] Tim Blazytko. *Statistical Analysis to Detect Uncommon Code*. https://synthesis.to/2023/01/26/uncommon_instruction_sequences.html. 2023.

[14] Tim Blazytko et al. "Syntia: Synthesizing the semantics of obfuscated code". In: *26th USENIX Security Symposium (USENIX Security 17)*. 2017, pp. 643–659.

[15] Deyu Bo et al. "A survey on spectral graph neural networks". In: *arXiv preprint arXiv:2302.05631* (2023).

[16] Michael M Bronstein et al. "Geometric deep learning: going beyond euclidean data". In: *IEEE Signal Processing Magazine* 34.4 (2017), pp. 18–42.

[17] Michael M Bronstein et al. "Geometric deep learning: Grids, groups, graphs, geodesics, and gauges". In: *arXiv preprint arXiv:2104.13478* (2021).

[18] Pierrick Brunet et al. "Epona and the Obfuscation Paradox: Transparent for Users and Developers, a Pain for Reversers". In: *Proceedings of the 3rd ACM Workshop on Software Protection*. SPRO'19. London, United Kingdom: Association for Computing Machinery, 2019, pp. 41–52. ISBN: 9781450368353.

[19] Rudy Bunel et al. "Leveraging grammar and reinforcement learning for neural program synthesis". In: *arXiv preprint arXiv:1805.04276* (2018).

[20] Gianluca Capozzi et al. "Adversarial Attacks against Binary Similarity Systems". In: *arXiv preprint arXiv:2303.11143* (2023).

[21] Alexis Challande, Robin David, and Guënaël Renault. "Quokka: A Fast and Accurate Binary Exporter". In: *GreHack 2022-10th International Symposium on Research in Grey-Hat Hacking*. 2022.

[22] Fenxiao Chen et al. "Graph representation learning: a survey". In: *APSIPA Transactions on Signal and Information Processing* 9 (2020), e15.

[23] Zhengdao Chen et al. "Can graph neural networks count substructures?" In: *Advances in neural information processing systems* 33 (2020), pp. 10383–10395.

[24] Stanley Chow et al. "An approach to the obfuscation of control-flow of sequential computer programs". In: *International Conference on Information Security*. Springer. 2001, pp. 144–155.

[25] Roxane Cohen, Florian Yger, and Fabrice Rossi. "Adding semantic to level-up graph-based Android malware detection". In: *The 10th International Conference on Complex Networks and their Applications (Complex Networks 2021)*. 2021, pp. 235–237.

[26] Roxane Cohen et al. "Identifying Obfuscated Code through Graph-Based Semantic Analysis of Binary Code". In: *The 13th International Conference on Complex Networks and their Applications*. 2024.

[27] Christan Collberg. *The Tigress C obfuscator*. https://tigress.wtf/index.html. 2016.

[28] Christian Collberg, Clark Thomborson, and Douglas Low. *A taxonomy of obfuscating transformations*. Tech. rep. Department of Computer Science, The University of Auckland, New Zealand, 1997.

[29] Hanjun Dai, Bo Dai, and Le Song. "Discriminative embeddings of latent variable models for structured data". In: *International conference on machine learning*. PMLR. 2016, pp. 2702–2711.

[30] Mila Dalla Preda et al. "Code obfuscation and malware detection by abstract interpretation". In: (2007).

[31] Robin David, Luigi Coniglio, and Mariano Ceccato. "Qsynth-a program synthesis based approach for binary code deobfuscation". In: *BAR 2020 Workshop*. 2020.

[32] Robbe De Ghein et al. "ApkDiff: Matching Android App Versions Based on Class Structure". In: *Proceedings of the 2022 ACM Workshop on Research on Offensive and Defensive Techniques in the Context of Man At The End (MATE) Attacks*. Checkmate '22. Los Angeles, CA, USA: Association for Computing Machinery, 2022, pp. 1–12. ISBN: 9781450398817.

[33] Leonardo De Moura and Nikolaj Bjørner. "Z3: an efficient SMT solver". In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'08/ETAPS'08. Budapest, Hungary: Springer-Verlag, 2008, pp. 337–340. ISBN: 3540787992.

[34] Bjorn De Sutter. "A New Framework of Software Obfuscation Evaluation Criteria". In: *arXiv preprint arXiv:2502.14093* (2025).

[35] Jacob Devlin et al. "Robustfill: Neural program learning under noisy i/o". In: *International conference on machine learning*. PMLR. 2017, pp. 990–998.

[36] Jacob Devlin et al. "Bert: Pre-training of deep bidirectional transformers for language understanding". In: *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*. 2019, pp. 4171–4186.

[37] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization". In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2019.

[38] Luke Dramko et al. "Quantifying and Mitigating the Impact of Obfuscations on Machine-Learning-Based Decompilation Improvement". In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Ed. by Manuel Egele et al. Cham: Springer Nature Switzerland, 2025, pp. 244–266. ISBN: 978-3-031-97620-9.

[39] Yue Duan et al. "Deepbindiff: Learning program-wide code representations for binary diffing". In: *Network and distributed system security symposium*. 2020.

[40] Gabriel Dulac-Arnold et al. “Deep reinforcement learning in large discrete action spaces”. In: *arXiv preprint arXiv:1512.07679* (2015).

[41] Thomas Dullien and Rolf Rolles. “Graph-based comparison of executable objects (english version)”. In: *Sstic* 5.1 (2005), p. 3.

[42] Kevin Ellis et al. “Dreamcoder: Bootstrapping inductive program synthesis with wake-sleep library learning”. In: *Proceedings of the 42nd acm sigplan international conference on programming language design and implementation*. 2021, pp. 835–850.

[43] Frank Emmert-Streib, Matthias Dehmer, and Yongtang Shi. “Fifty years of graph matching, network alignment and network comparison”. In: *Information sciences* 346 (2016), pp. 180–197.

[44] Federico Errica et al. “A Fair Comparison of Graph Neural Networks for Graph Classification”. In: *International Conference on Learning Representations*. 2020. URL: https://openreview.net/forum?id=HygDF6NFPB.

[45] Weijie Feng et al. “Neureduce: Reducing mixed boolean-arithmetic expressions by recurrent neural network”. In: *Findings of the Association for Computational Linguistics: EMNLP 2020*. 2020, pp. 635–644.

[46] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. “The program dependence graph and its use in optimization”. In: *ACM Trans. Program. Lang. Syst.* 9.3 (July 1987), pp. 319–349. ISSN: 0164-0925. DOI: 10.1145/24039.24041. URL: https://doi.org/10.1145/24039.24041.

[47] Maurizio Ferrari Dacrema, Paolo Cremonesi, and Dietmar Jannach. “Are we really making much progress? A worrying analysis of recent neural recommendation approaches”. In: *Proceedings of the 13th ACM Conference on Recommender Systems*. RecSys ’19. ACM, 2019. DOI: 10.1145/3298689.3347058.

[48] Matthias Fey and Jan E. Lenssen. “Fast Graph Representation Learning with PyTorch Geometric”. In: *ICLR Workshop on Representation Learning on Graphs and Manifolds*. 2019.

[49] Halvar Flake. “Structural comparison of executable objects”. In: *DIMVA 2004, July 6-7, Dortmund, Germany* (2004).

[50] Michael N. Gagnon, Stephen Taylor, and Anup K. Ghosh. “Software Protection through Anti-Debugging”. In: *IEEE Security and Privacy* 5.3 (2007), pp. 82–84. DOI: 10.1109/MSP.2007.71.

[51] Debin Gao, Michael K Reiter, and Dawn Song. “Binhunt: Automatically finding semantic differences in binary programs”. In: *International Conference on Information and Communications Security*. Springer. 2008, pp. 238–255.

[52] Hao Gao et al. “FUSION: Measuring Binary Function Similarity with Code-Specific Embedding and Order-Sensitive GNN”. In: *Symmetry* (2022).

[53] Hongyang Gao and Shuiwang Ji. "Graph U-Nets". In: *Proceedings of the 36th International Conference on Machine Learning*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, Sept. 2019, pp. 2083–2092. URL: https://proceedings.mlr.press/v97/gao19a.html.

[54] Jian Gao et al. "Vulseeker: A semantic learning based vulnerability seeker for cross-platform binary". In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 2018, pp. 896–899.

[55] Peter Garba and Matteo Favaro. "SATURN - Software Deobfuscation Framework Based On LLVM". In: *Proceedings of the 3rd ACM Workshop on Software Protection*. SPRO'19. London, United Kingdom: Association for Computing Machinery, 2019, pp. 27–38. ISBN: 9781450368353. DOI: 10.1145/3338503.3357721. URL: https://doi.org/10.1145/3338503.3357721.

[56] Thomas Gärtner, Peter Flach, and Stefan Wrobel. "On graph kernels: Hardness results and efficient alternatives". In: *Learning Theory and Kernel Machines: 16th Annual Conference on Learning Theory and 7th Kernel Workshop, COLT/Kernel 2003, Washington, DC, USA, August 24-27, 2003. Proceedings*. Springer. 2003, pp. 129–143.

[57] Liyu Gong and Qiang Cheng. "Exploiting edge features for graph neural networks". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2019, pp. 9211–9219.

[58] M. Gori, G. Monfardini, and F. Scarselli. "A new model for learning in graph domains". In: *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.* Vol. 2. 2005, 729–734 vol. 2. DOI: 10.1109/IJCNN.2005.1555942.

[59] Claudia Greco et al. "Explaining Binary Obfuscation". In: *2023 IEEE International Conference on Cyber Security and Resilience (CSR)*. July 2023, pp. 22–27. DOI: 10.1109/CSR57506.2023.10224825.

[60] Aditya Grover and Jure Leskovec. "node2vec: Scalable feature learning for networks". In: *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*. 2016, pp. 855–864.

[61] Serge Guelton et al. "Combining Obfuscation and Optimizations in the Real World". In: *18th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*. Madrid, Spain: IEEE, Sept. 2018, pp. 24–33. DOI: 10.1109/SCAM.2018.00010. URL: https://hal.science/hal-02062166.

[62] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. "Program synthesis". In: *Foundations and Trends® in Programming Languages* 4.1-2 (2017), pp. 1–119.

[63] Tuomas Haarnoja et al. "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor". In: *International conference on machine learning*. Pmlr. 2018, pp. 1861–1870.

[64] William L. Hamilton, Rex Ying, and Jure Leskovec. "Inductive representation learning on large graphs". In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS'17. Long Beach, California, USA: Curran Associates Inc., 2017, pp. 1025–1035. ISBN: 9781510860964.

[65] Warren A. Harrison and Kenneth I. Magel. "A complexity measure based on nesting level". In: *SIGPLAN Not.* 16.3 (Mar. 1981), pp. 63–74. ISSN: 0362-1340. DOI: 10.1145/947825.947829. URL: https://doi.org/10.1145/947825.947829.

[66] W. K. Hastings. "Monte Carlo sampling methods using Markov chains and their applications". In: *Biometrika* 57.1 (Apr. 1970), pp. 97–109. ISSN: 0006-3444. DOI: 10.1093/biomet/57.1.97. eprint: https://academic.oup.com/biomet/article-pdf/57/1/97/23940249/57-1-97.pdf. URL: https://doi.org/10.1093/biomet/57.1.97.

[67] Haojie He et al. "Code is not natural language: Unlock the power of semantics-oriented graph representation for binary code similarity detection". In: *33rd USENIX Security Symposium (USENIX Security 24), PHILADELPHIA, PA*. 2024.

[68] Xin Hu, Tzi-cker Chiueh, and Kang G Shin. "Large-scale malware indexing using function-call graphs". In: *Proceedings of the 16th ACM conference on Computer and communications security*. 2009, pp. 611–620.

[69] Yikun Hu et al. "Binmatch: A semantics-based hybrid approach on binary code clone analysis". In: *2018 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE. 2018.

[70] Ayush Jain, Andrew Szot, and Joseph J Lim. "Generalization to new actions in reinforcement learning". In: *arXiv preprint arXiv:2011.01928* (2020).

[71] Ang Jia et al. "1-to-1 or 1-to-n? Investigating the Effect of Function Inlining on Binary Similarity Analysis". In: *ACM Transactions on Software Engineering and Methodology* 32.4 (2023), pp. 1–26.

[72] Ang Jia et al. "Cross-inlining binary function similarity detection". In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 2024, pp. 1–13.

[73] Shuai Jiang et al. "Function-level obfuscation detection method based on Graph Convolutional Networks". In: *J. Inf. Secur. Appl.* 61.C (Sept. 2021). ISSN: 2214-2126. DOI: 10.1016/j.jisa.2021.102953. URL: https://doi.org/10.1016/j.jisa.2021.102953.

[74] Pascal Junod et al. "Obfuscator-LLVM–Software Protection for the Masses". In: *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*. Ed. by Brecht Wyseur. 2015.

[75] Yuichiro Kanzaki, Akito Monden, and Christian Collberg. "Code artificiality: A metric for the code stealth based on an n-gram model". In: *2015 IEEE/ACM 1st International Workshop on Software Protection*. IEEE. 2015, pp. 31–37.

[76] Dongkwan Kim et al. "Revisiting Binary Code Similarity Analysis using Interpretable Feature Engineering and Lessons Learned". In: *IEEE Transactions on Software Engineering* (2022), pp. 1–23.

[77] Thomas N. Kipf and Max Welling. "Semi-Supervised Classification with Graph Convolutional Networks". In: *International Conference on Learning Representations*. 2017. URL: https://openreview.net/forum?id=SJU4ayYgl.

[78] Max Klabunde and Florian Lemmerich. "On the prediction instability of graph neural networks". In: *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer. 2022, pp. 187–202.

[79] Joxean Koret. *Diaphora*. https://github.com/joxeankoret/diaphora. 2015.

[80] Orestis Kostakis et al. "Improved call graph comparison using simulated annealing". In: *Proceedings of the 2011 ACM Symposium on Applied Computing*. 2011, pp. 1516–1523.

[81] Nils M Kriege, Fredrik D Johansson, and Christopher Morris. "A survey on graph kernels". In: *Applied Network Science* 5.1 (2020), p. 6.

[82] Harold W Kuhn. "The Hungarian method for the assignment problem". In: *Naval research logistics quarterly* 2.1-2 (1955), pp. 83–97.

[83] William Landi. "Undecidability of static analysis". In: *ACM Lett. Program. Lang. Syst.* 1.4 (Dec. 1992), pp. 323–337. ISSN: 1057-4514. DOI: 10.1145/161494.161501. URL: https://doi.org/10.1145/161494.161501.

[84] Junhyun Lee, Inyeop Lee, and Jaewoo Kang. "Self-attention graph pooling". In: *International conference on machine learning*. pmlr. 2019, pp. 3734–3743.

[85] Woosuk Lee. "Combining the top-down propagation and bottom-up enumeration for inductive program synthesis". In: *Proc. ACM Program. Lang.* 5.POPL (Jan. 2021). DOI: 10.1145/3434335. URL: https://doi.org/10.1145/3434335.

[86] Xuezixiang Li, Yu Qu, and Heng Yin. "Palmtree: Learning an assembly language model for instruction embedding". In: *Proceedings of the 2021 ACM SIGSAC conference on computer and communications security*. 2021, pp. 3236–3251.

[87] Yujia Li et al. "Graph matching networks for learning the similarity of graph structured objects". In: *International conference on machine learning*. PMLR. 2019, pp. 3835–3845.

[88] Binbin Liu et al. "MBA-Blast: Unveiling and Simplifying Mixed Boolean-Arithmetic Obfuscation". In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021, pp. 1701–1718.

[89] Binbin Liu et al. "An In-Place Simplification on Mixed Boolean-Arithmetic Expressions". In: *Security and Communication Networks* 2022.1 (2022), p. 7307139.

[90] Bingchang Liu et al. “Alpha-Diff: cross-version binary code similarity detection with DNN”. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ASE ’18. Montpellier, France: Association for Computing Machinery, 2018, pp. 667–678. ISBN: 9781450359375. DOI: 10.1145/3238147.3238199. URL: https://doi.org/10.1145/3238147.3238199.

[91] Lannan Luo et al. “Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection”. In: *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*. 2014, pp. 389–400.

[92] Laurens van der Maaten and Geoffrey Hinton. “Visualizing data using t-SNE”. In: *Journal of machine learning research* 9.Nov (2008), pp. 2579–2605.

[93] Matias Madou et al. “On the effectiveness of source code transformations for binary obfuscation”. In: *Proceedings of the International Conference on Software Engineering Research and Practice (SERP06)*. CSREA Press. 2006, pp. 527–533.

[94] Francis Maes, Ludovic Denoyer, and Patrick Gallinari. “Structured prediction with reinforcement learning”. In: *Machine learning* 77.2 (2009), pp. 271–301.

[95] Andrea Marcelli et al. “How machine learning is solving the binary function similarity problem”. In: *31st USENIX Security Symposium (USENIX Security 22)*. 2022.

[96] Benjamin Mariano et al. “Control-Flow Deobfuscation using Trace-Informed Compositional Program Synthesis”. In: *Proceedings of the ACM on Programming Languages* 8.OOPSLA2 (2024), pp. 2211–2241.

[97] Haggai Maron et al. “Invariant and equivariant graph networks”. In: *arXiv preprint arXiv:1812.09902* (2018).

[98] Luca Massarelli et al. “Investigating graph embedding neural networks with unsupervised features extraction for binary analysis”. In: *2nd Workshop on Binary Analysis Research (BAR)*. 2019.

[99] Luca Massarelli et al. “SAFE: Self-Attentive Function Embeddings for Binary Similarity”. In: *Proceedings of 16th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*. 2019.

[100] Nikos Mavrogiannopoulos, Nessim Kisserli, and Bart Preneel. “A taxonomy of self-modifying code for obfuscation”. In: *Computers & Security* 30.8 (2011), pp. 679–691.

[101] T.J. McCabe. “A Complexity Measure”. In: *IEEE Transactions on Software Engineering* SE-2.4 (1976), pp. 308–320. DOI: 10.1109/TSE.1976.233837.

[102] Xiaozhu Meng and Barton P Miller. “Binary code is not easy”. In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 2016, pp. 24–35.

[103] Elie Mengin. “Binary Diffing as a Network Alignment Problem”. PhD thesis. Universite Paris 1-Pantheon Sorbonne, 2021.

[104] Elie Mengin and Fabrice Rossi. "Binary diffing as a network alignment problem via belief propagation". In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2021, pp. 967–978.

[105] Elie Mengin and Fabrice Rossi. "Improved Algorithm for the Network Alignment Problem with Application to Binary Diffing". In: *Procedia Computer Science* 192 (2021), pp. 961–970.

[106] Grégoire Menguy et al. "Search-based local black-box deobfuscation: understand, improve and mitigate". In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2021, pp. 2513–2525.

[107] Tomas Mikolov et al. "Efficient estimation of word representations in vector space". In: *arXiv preprint arXiv:1301.3781* (2013).

[108] Jiang Ming, Meng Pan, and Debin Gao. "iBinHunt: Binary hunting with inter-procedural control flow". In: *International Conference on Information Security and Cryptology*. Springer. 2012, pp. 92–109.

[109] Jiang Ming, Dongpeng Xu, and Dinghao Wu. "Memoized Semantics-Based Binary Diffing with Application to Malware Lineage Inference". In: *ICT Systems Security and Privacy Protection*. Ed. by Hannes Federrath and Dieter Gollmann. Cham, 2015. ISBN: 978-3-319-18467-8.

[110] Jiang Ming et al. "BinSim: Trace-based semantic binary diffing via system call sliced segment equivalence checking". In: *26th USENIX Security Symposium (USENIX Security 17)*. 2017, pp. 253–270.

[111] Volodymyr Mnih et al. "Asynchronous methods for deep reinforcement learning". In: *International conference on machine learning*. PmLR. 2016, pp. 1928–1937.

[112] Rabih Mohsen and Alexandre Miranda Pinto. "Evaluating Obfuscation Security: A Quantitative Approach". In: *Foundations and Practice of Security*. Ed. by Joaquin Garcia-Alfaro, Evangelos Kranakis, and Guillaume Bonfante. Cham: Springer International Publishing, 2016, pp. 174–192. ISBN: 978-3-319-30303-1.

[113] Augustus Odena et al. "BUSTLE: Bottom-up program synthesis through learning-guided exploration". In: *arXiv preprint arXiv:2007.14381* (2020).

[114] Julian Parsert and Elizabeth Polgreen. "Reinforcement learning and data-generation for syntax-guided synthesis". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 38. 9. 2024, pp. 10670–10678.

[115] Fabian Pedregosa et al. "Scikit-learn: Machine learning in Python". In: *the Journal of machine Learning research* 12 (2011), pp. 2825–2830.

[116] Kexin Pei et al. "Trex: Learning execution semantics from micro-traces for binary similarity". In: *arXiv preprint arXiv:2012.08680* (2020).

[117] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. "Deepwalk: Online learning of social representations". In: *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2014, pp. 701–710.

[118] Babak Bashari Rad and Maslin Masrom. "Metamorphic virus variants classification using opcode frequency histogram". In: *arXiv preprint arXiv:1104.3228* (2011).

[119] Benjamin Reichenwallner and Peter Meerwald-Stadler. "Efficient deobfuscation of linear mixed boolean-arithmetic expressions". In: *Proceedings of the 2022 ACM Workshop on Research on offensive and defensive techniques in the context of Man At The End (MATE) attacks*. 2022, pp. 19–28.

[120] Benjamin Reichenwallner and Peter Meerwald-Stadler. "Simplification of general mixed boolean-arithmetic expressions: Gamba". In: *2023 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE. 2023, pp. 427–438.

[121] Xiaolei Ren et al. "Unleashing the hidden power of compiler optimization on binary code difference: an empirical study". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. Virtual, Canada: Association for Computing Machinery, 2021, pp. 142–157. ISBN: 9781450383912. DOI: 10.1145/3453483.3454035. URL: https://doi.org/10.1145/3453483.3454035.

[122] Jack Royer. *LLVM-powered deobfuscation of virtualized binaries*. https://blog.thalium.re/posts/llvm-powered-devirtualization/. 2024.

[123] T Konstantin Rusch, Michael M Bronstein, and Siddhartha Mishra. "A survey on oversmoothing in graph neural networks". In: *arXiv preprint arXiv:2303.10993* (2023).

[124] B.G. Ryder. "Constructing the Call Graph of a Program". In: *IEEE Transactions on Software Engineering* SE-5.3 (1979), pp. 216–226. DOI: 10.1109/TSE.1979.234183.

[125] Aleieldin Salem and Sebastian Banescu. "Metadata recovery from obfuscated programs using machine learning". In: *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering*. 2016, pp. 1–11.

[126] Jonathan Salwan, Sébastien Bardin, and Marie-Laure Potet. "Symbolic deobfuscation: From virtualized code back to the original". In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2018, pp. 372–392.

[127] Moritz Schloegel et al. "Loki: Hardening code obfuscation against automated attacks". In: *31st USENIX Security Symposium (USENIX Security 22)*. 2022, pp. 3055–3073.

[128] Sebastian Schrittwieser et al. "Protecting software through obfuscation: Can it keep pace with progress in code analysis?" In: *Acm computing surveys (csur)* 49.1 (2016), pp. 1–37.

[129] Sebastian Schrittwieser et al. “Modeling Obfuscation Stealth through Code Complexity”. In: *European Symposium on Research in Computer Security*. Springer. 2023, pp. 392–408.

[130] John Schulman et al. “Proximal policy optimization algorithms”. In: *arXiv preprint arXiv:1707.06347* (2017).

[131] Banescu Sebastian, Collberg Christian, and Pretschner Alexander. “Predicting the resilience of obfuscated code against symbolic execution attacks via machine learning”. In: *Proceedings of the 26th USENIX Security Symposium (USENIX Security 17), Vancouver, BC, Canada*. 2017, pp. 16–18.

[132] Oleksandr Shchur et al. “Pitfalls of graph neural network evaluation”. In: *arXiv preprint arXiv:1811.05868* (2018).

[133] Paria Shirani, Lingyu Wang, and Mourad Debbabi. “Binshape: Scalable and robust binary library function identification using function shape”. In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2017, pp. 301–324.

[134] Ravid Shwartz-Ziv and Amitai Armon. “Tabular data: Deep learning is not all you need”. In: *Information Fusion* 81 (2022), pp. 84–90.

[135] Mahito Sugiyama and Karsten Borgwardt. “Halting in random walk kernels”. In: *Advances in neural information processing systems* 28 (2015).

[136] Rui Tang et al. “Network alignment”. In: *Physics Reports* 1107 (2025). Network Alignment, pp. 1–45. ISSN: 0370-1573. DOI: https://doi.org/10.1016/j.physrep.2024.11.006. URL: https://www.sciencedirect.com/science/article/pii/S0370157324004034.

[137] Adane Nega Tarekegn, Mohib Ullah, and Faouzi Alaya Cheikh. “Deep learning for multi-label learning: A comprehensive survey”. In: *arXiv preprint arXiv:2401.16549* (2024).

[138] Ramtine Tofighi-Shirazi, Irina Măriuca Asăvoae, and Philippe Elbaz-Vincent. “Fine-grained static detection of obfuscation transforms using ensemble-learning and semantic reasoning”. In: *Proceedings of the 9th Workshop on Software Security, Protection, and Reverse Engineering*. SSPREW9 ’19. San Juan, Puerto Rico, USA: Association for Computing Machinery, 2019. ISBN: 9781450377461. DOI: 10.1145/3371307.3371313. URL: https://doi.org/10.1145/3371307.3371313.

[139] Ramtine Tofighi-Shirazi et al. “Defeating opaque predicates statically through machine learning and binary analysis”. In: *Proceedings of the 3rd ACM Workshop on Software Protection*. 2019, pp. 3–14.

[140] S.K. Udupa, S.K. Debray, and M. Madou. “Deobfuscation: reverse engineering obfuscated code”. In: *12th Working Conference on Reverse Engineering (WCRE’05)*. 2005, 10 pp.–54. DOI: 10.1109/WCRE.2005.13.

[141] Sami Ullah and Heekuck Oh. "BinDiff NN: Learning Distributed Representation of Assembly for Robust Binary Diffing against Semantic Differences". In: *IEEE Transactions on Software Engineering* 48.9 (2021), pp. 3442–3466.

[142] Petar Veličković et al. "Graph Attention Networks". In: *International Conference on Learning Representations*. 2018. URL: https://openreview.net/forum?id=rJXMpikCZ.

[143] S Vichy N Vishwanathan et al. "Graph kernels". In: *The Journal of Machine Learning Research* 11 (2010), pp. 1201–1242.

[144] Chenxi Wang. *A security architecture for survivability mechanisms*. University of Virginia, 2001.

[145] Hao Wang et al. "JTrans: Jump-Aware Transformer for Binary Code Similarity Detection". In: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2022. Virtual, South Korea, 2022, pp. 1–13. ISBN: 9781450393799.

[146] Ronald J Williams. "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Machine learning* 8.3 (1992), pp. 229–256.

[147] Zonghan Wu et al. "A comprehensive survey on graph neural networks". In: *IEEE transactions on neural networks and learning systems* 32.1 (2020), pp. 4–24.

[148] Tobias Wüchner, Martín Ochoa, and Alexander Pretschner. "Robust and effective malware detection through quantitative data flow graph metrics". In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2015, pp. 98–118.

[149] Feng Xia et al. "Graph learning: A survey". In: *IEEE Transactions on Artificial Intelligence* 2.2 (2021), pp. 109–127.

[150] Dongpeng Xu et al. "Boosting SMT solver performance on mixed-bitwise-arithmetic expressions". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2021, pp. 651–664.

[151] Keyulu Xu et al. "How Powerful are Graph Neural Networks?" In: *International Conference on Learning Representations*. 2019. URL: https://openreview.net/forum?id=ryGs6iA5Km.

[152] Xiaojun Xu et al. "Neural network-based graph embedding for cross-platform binary code similarity detection". In: *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 2017, pp. 363–376.

[153] Babak Yadegari et al. "A Generic Approach to Automatic Deobfuscation of Executable Code". In: *2015 IEEE Symposium on Security and Privacy*. 2015, pp. 674–691. DOI: 10.1109/SP.2015.47.

[154] Fabian Yamaguchi et al. "Modeling and Discovering Vulnerabilities with Code Property Graphs". In: *2014 IEEE Symposium on Security and Privacy*. 2014, pp. 590–604. DOI: 10.1109/SP.2014.44.

[155] Zeping Yu et al. "Codecmr: Cross-modal retrieval for function-level binary source code matching". In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 3872–3883.

[156] Zeping Yu et al. "Order matters: Semantic-aware neural networks for binary code similarity detection". In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 34. 2020, pp. 1145–1152.

[157] Zerotistic. *Breaking Control Flow Flattening: A Deep Technical Analysis*. https://zerotistic.blog/posts/cff-remover/. 2024.

[158] Peihua Zhang et al. "Khaos: The Impact of Inter-procedural Code Obfuscation on Binary Diffing Techniques". In: *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*. CGO '23. Montréal, QC, Canada: Association for Computing Machinery, 2023, pp. 55–67. ISBN: 9798400701016.

[159] Lei Zhao et al. "PatchScope: Memory object centric patch diffing". In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 2020, pp. 149–165.

[160] Yongxin Zhou et al. "Information hiding in software with mixed boolean-arithmetic transforms". In: *International Workshop on Information Security Applications*. Springer. 2007, pp. 61–75.

[161] Jie Zhu, Fengge Wu, and Junsuo Zhao. "An overview of the action space for deep reinforcement learning". In: *Proceedings of the 2021 4th international conference on algorithms, computing and artificial intelligence*. 2021, pp. 1–10.

Appendix A

Résumé détaillé

A.1 Introduction

Le code binaire exécutable constitue la base fondamentale de l'informatique moderne : chaque ordinateur, smartphone et objet connecté en dépend pour effectuer des calculs complexes, exécuter des applications ou des transactions en ligne sécurisées. Ce code binaire est le résultat de la compilation, un processus qui traduit le code source, lisible par l'humain, en une forme exécutable par le Central Processing Unit (CPU) et donc par les ordinateurs. Depuis son apparition, il fait l'objet de rétro-ingénierie, une discipline dans laquelle les analystes cherchent à comprendre son comportement et sa structure, au moyen de ses instructions bas-niveau.

Une telle analyse se justifie à la fois par des considérations défensives et offensives, englobant des finalités légitimes comme illégitimes. Par exemple, un logiciel malveillant compilé et diffusé sur Internet est étudié dans une optique défensive et parfaitement légitime, afin d'en comprendre le fonctionnement et de concevoir des contre-mesures destinées à limiter sa propagation. À l'inverse, analyser un jeu vidéo dans le but de faciliter la triche à grande échelle relève d'une démarche purement offensive, poursuivie à des fins non éthiques.

Cependant, le code binaire est intrinsèquement conçu pour être lu par des machines plutôt que par des êtres humains, ce qui le rend difficile à comprendre pour les analystes [102], sans outils et compétences spécialisés. L'analyse manuelle s'avère ainsi particulièrement difficile, chronophage et sujette à l'erreur humaine. La difficulté s'accroît d'autant plus que le code est transformé par des optimisations, parfois poussées, du compilateur ou, plus encore, par l'obfuscation.

L'obfuscation, conçue comme une contre-mesure à la rétro-ingénierie, modifie intentionnellement le code binaire afin d'en dissimuler la structure, dans le but de compliquer les tentatives d'analyse. À l'instar de la rétro-ingénierie elle-même, elle peut répondre à des objectifs légitimes, comme la protection de logiciels relevant du droit à la propriété intellectuelle, comme à des objectifs illégitimes, tels qu'entraver la rétro-ingénierie d'un logiciel malveillant. Une grande variété de techniques d'obfuscation existe [28], chacune pouvant modifier différents aspects du code binaire. En pratique, ces techniques

sont fréquemment combinées afin de renforcer la protection d'un programme. Ainsi, l'analyse d'un binaire soigneusement obfusqué s'avère notoirement difficile, même pour des analystes expérimentés.

Récemment, les avancées en apprentissage automatique ont profondément transformé l'étude du code binaire. Ces techniques ont démontré un succès remarquable dans des domaines variés, tels que la vision par ordinateur, le traitement automatique du langage naturel et les modèles génératifs. Parmi ces avancées, l'apprentissage de représentations s'est révélé particulièrement prometteur : la capacité de modèles à extraire et à encoder automatiquement des schémas pertinents à partir de données complexes peut être combinée à des techniques classiques de classification ou de régression, afin d'automatiser des tâches autrefois réservées à l'expertise humaine.

Ce potentiel n'a pas échappé à la rétro-ingénierie, où de nouvelles approches fondées sur l'apprentissage automatique ciblent désormais directement le code binaire [38, 95, 59, 73, 139, 14], dans le but d'en extraire des représentations adaptées à un large éventail de tâches d'analyse.

Cependant, représenter le code binaire demeure un défi majeur, en raison de son hétérogénéité et de sa complexité, liées notamment à la chaîne de compilation, à l'architecture cible, aux options d'optimisation ou encore aux techniques d'obfuscation appliquées. L'étude de représentations capables de capturer ces aspects est donc cruciale. En particulier, les graphes se sont imposés comme une approche naturelle et efficace pour décrire le code binaire, puisqu'ils peuvent refléter à la fois sa structure et une partie de ses propriétés sémantiques. Par conséquent, l'adoption de techniques avancées d'apprentissage de représentations, fondées sur les graphes, est particulièrement indiquée. Ces méthodes sont désormais essentielles pour des tâches telles que la détection de similarité binaire, le clustering, et, de manière générale, l'analyse d'obfuscation.

Cette thèse modélise l'analyse d'obfuscation comme un processus en plusieurs étapes. La première consiste à détecter l'obfuscation, c'est-à-dire à déterminer si un programme binaire, ou l'une de ses composantes, a été obfusqué. La seconde étape, la caractérisation de l'obfuscation, vise à identifier quelles techniques spécifiques ont été appliquées. Ces deux étapes permettent d'évaluer la furtivité d'un schéma d'obfuscation et de guider les phases d'analyse ultérieures.

Une fois l'obfuscation localisée et caractérisée, elle peut être analysée selon deux grandes stratégies : directe ou indirecte. Les stratégies directes, également appelées désobfuscation, visent à supprimer ou atténuer des techniques d'obfuscation précises, telles que les expressions mixtes arithmético-booléennes, la virtualisation ou l'aplanissement de graphes de flot de contrôle, et requièrent souvent une connaissance approfondie des mécanismes liés à ces techniques. Toutefois, l'obfuscation, telle qu'observée en pratique dans des programmes existants, combine généralement plusieurs techniques, ce qui rend souvent insuffisantes les approches de désobfuscation ciblant une seule d'entre elles.

À l'inverse, les stratégies indirectes contournent l'obfuscation afin de récupérer des informations exploitables par l'analyste, sans chercher à supprimer l'obfuscation explicitement. Elles s'avèrent particulièrement précieuses lorsque la désobfuscation directe est irréalisable en pratique, ou lorsque l'obfuscation emploie des techniques inédites ou

hybrides pour lesquelles aucun algorithme de désobfuscation dédié n'existe.

A.2 Problématiques

L'étude de l'apprentissage de représentations du code binaire, en particulier en présence d'obfuscation, peut être décomposée en sous-problèmes distincts mais interdépendants, chacun présentant ses propres défis techniques.

Le premier défi consiste à déterminer si un programme binaire donné contient des régions obfusquées et, le cas échéant, à les localiser et à identifier les techniques d'obfuscation utilisées. Cette phase d'identification et de caractérisation de l'obfuscation est souvent négligée dans la littérature, où il est courant de supposer que les régions obfusquées et leurs types sont déjà connus. En pratique, toutefois, distinguer un code naturellement complexe d'un code obfusqué n'est pas trivial, car tous deux peuvent présenter des irrégularités structurelles similaires.

> **Question de recherche #1.** Comment les représentations de graphes du code binaire peuvent-elles être efficacement exploitées pour améliorer la détection et la caractérisation de l'obfuscation ?

Ces représentations peuvent également être réutilisées pour d'autres tâches d'analyse du code binaire. Par exemple, appliquées au diffing binaire, qui consiste à comparer deux variantes d'un même programme, ou à la similarité binaire, qui vise à calculer un score de similarité entre deux sections de code, de telles représentations permettent de révéler des correspondances structurelles ou comportementales. Ces approches s'avèrent particulièrement pertinentes dans le cadre d'attaques indirectes, qui cherchent à obtenir des informations à partir de binaires obfusqués sans procéder à une désobfuscation explicite. Le rôle crucial des outils de diffing et de similarité binaires, notamment dans un contexte d'obfuscation, conduit à la question de recherche suivante :

> **Question de recherche #2.** Une fois l'obfuscation identifiée et caractérisée, peut-on recourir à des attaques indirectes, telles que le diffing ou la similarité binaires, afin de récupérer des informations pertinentes à partir de binaires obfusqués ?

Néanmoins, les méthodes indirectes ne sont pas toujours applicables. Dans certains cas, la désobfuscation peut représenter la seule approche viable. Or, la plupart des méthodes existantes sont conçues de manière à contrer une technique d'obfuscation unique, ce qui les rend peu adaptées aux obfuscations composites rencontrées en pratique. De plus, aucune approche n'exploite pleinement la richesse intrinsèque de données telles que les graphes ou des représentations apprises via de l'apprentissage par renforcement. Ce type d'apprentissage a pourtant démontré son efficacité dans le cadre de tâches de prise de décision séquentielle et pourrait se révéler particulièrement adapté aux scénarios de désobfuscation. Cette considération motive la question de recherche suivante :

Question de recherche #3. Une fois l'obfuscation identifiée, et lorsque les attaques indirectes sont inadaptées, comment l'apprentissage par renforcement, appliqué à des graphes, peut-il constituer une approche efficace pour déobfusquer du code binaire, sans considération de l'obfuscation appliquée ?

Répondre à ces trois questions de recherche permettrait d'offrir un cadre méthodologique complet dédié à l'analyse d'obfuscation, dans un contexte réaliste de rétro-ingénierie, offrant ainsi aux praticiens des outils adaptés à chaque étape du processus.

A.3 Contributions et structure

Cette thèse répond aux questions de recherche précédemment formulées selon le plan suivant :

- **Chapitres 2, 3 et 4**. Ces Chapitres introduisent les bases conceptuelles et techniques nécessaires à la compréhension de cette thèse. Plus précisément, le Chapitre 2 présente les principes fondamentaux du code binaire et définit les notions clés mobilisées tout au long de cette thèse. Le Chapitre 3 propose un état de l'art des techniques d'apprentissage de représentations, orientées spécifiquement vers les graphes. Enfin, le Chapitre 4 examine l'application des méthodes d'apprentissage automatique aux différents problèmes d'analyse binaire recensés dans la littérature.

- **Chapitre 5**. Il présente les principaux jeux de données utilisés dans cette thèse. Ce chapitre introduit en particulier ObfuBench, un jeu de données de programmes binaires obfusqués qui surpasse les jeux de données existants en termes de taille, de diversité des obfuscateurs et de nombre de techniques d'obfuscation. Utilisé tout au long de ce manuscrit pour étayer et valider l'ensemble de nos travaux, ce jeu de données constitue en lui-même une contribution en soi. Il est complété par trois autres jeux de données, issus des littératures sur la similarité binaire et l'obfuscation, ainsi que par deux programmes obfusqués, publiquement disponibles, permettant de démontrer la validité et la généralisation des approches proposées.

- **Chapitre 6**. Ce Chapitre répond à la première question de recherche et s'intéresse à la détection et à la caractérisation de l'obfuscation à travers une étude approfondie de représentations, basées sur les graphes, du code binaire. Cette étude compare des approches de référence à des modèles avancés d'apprentissage profond, en mettant en évidence l'impact de descripteurs progressivement enrichis, allant de structurels à syntaxiques, sur les performances. Une attention particulière est portée à l'influence des stratégies de partitionnement des jeux de données et aux optimisations du compilateur, sur les résultats.

- **Chapitre 7**. En lien avec la deuxième question de recherche, ce Chapitre étudie le problème du diffing binaire, en tant qu'attaque indirecte ciblant l'obfuscation.

Il débute par une évaluation approfondie du diffing dans un contexte non obfusqué, comprenant une étude d'ablation du differ QBinDiff. L'analyse est ensuite étendue aux binaires obfusqués, afin d'évaluer l'impact de différents types d'obfuscation sur les performances des outils de diffing et de similarité. Ce Chapitre explore également comment les résultats liés à la détection et à la caractérisation de l'obfuscation, présentés au Chapitre 6, peuvent être exploités pour améliorer les performances du diffing binaire.

- **Chapitre 8**. Ce Chapitre se concentre sur la désobfuscation, considérée comme une stratégie d'attaque directe. Après une revue approfondie de l'état de l'art, mettant en lumière les lacunes de la littérature sur le sujet, il propose une approche exploratoire, combinant de nouveaux types de graphes, plus complets, qui intégrent à la fois des informations de flot de données et de contrôle, avec de l'apprentissage par renforcement. Cette approche vise à simplifier de manière itérative le code obfusqué par des opérations d'édition de graphes. Bien que ce travail reste inachevé, cette thèse se terminant avant de pouvoir en mener le développement complet, les résultats obtenus démontrent la faisabilité d'une désobfuscation partielle et apportent des résultats inédits dans le domaine encore peu exploré de l'apprentissage par renforcement, appliqué aux graphes, dans le cadre de l'analyse binaire.

En résumé, cette thèse explore l'usage et l'évaluation de nouvelles représentations du code binaire, conçues pour être utilisées à chaque étape du processus d'analyse d'obfuscation. Les contributions proposées sont pensées comme des outils modulaires au sein d'un cadre plus large de rétro-ingénierie, qui peuvent être combinées de manière sélective au cours du processus d'analyse. L'objectif global est de combiner l'apprentissage de représentations aux besoins pratiques de la rétro-ingénierie, en vue d'applications concrètes dans des scénarios réels. Ces contributions, complémentaires, s'inscrivent dans une analyse générale de l'obfuscation et s'appuient sur une validation expérimentale reposant à la fois sur des jeux de données synthétiques mais réalistes, et sur des échantillons de programmes binaires obfusqués issus du monde réel.

A.4 Contexte et notions

A.4.1 Code binaire

Un programme informatique compilé provient d'un code source écrit dans un langage de programmation lisible par un être humain. Celui-ci est d'abord traduit en une représentation intermédiaire, optimisé, puis transformé en code machine de bas niveau au terme d'un processus de compilation. Le résultat final de la compilation, appelé binaire exécutable, ne contient plus aucune information de haut niveau, telles que les noms de variables, les types de données, les identifiants de classes ou encore d'autres structures sémantiques, qui sont alors perdues de manière irréversible.

Par la suite, les programmes commerciaux, tels que les jeux vidéo, ou les malwares, sont généralement distribués aux utilisateurs uniquement sous forme de code machine,

tandis que le code source d'origine demeure propriétaire et accessible exclusivement aux développeurs. En conséquence, l'objectif de la rétro-ingénierie est précisément d'analyser et de comprendre la fonctionnalité et le comportement de ce code machine lorsque le binaire exécutable constitue la seule ressource disponible à examiner.

L'analyse directe du code machine est notoirement fastidieuse, en raison de son faible niveau d'abstraction et de la quantité substantielle d'informations perdues lors de la compilation. Il est toutefois possible de transformer le code machine en langage assembleur, lequel, bien que toujours de bas niveau, offre plus de lisibilité, pour un être humain, que les instructions binaires brutes : ce processus est appelé désassemblage. Le résultat de ce désassemblage, qui peut être exporté grâce à des exporteurs binaires, afin de faciliter une analyse ultérieure, est hiérarchisé de la façon suivante :

- Uune **instruction** représente la plus petite unité d'exécution d'un programme binaire ;
- Un **bloc de base** regroupe plusieurs instructions exécutées séquentiellement sans interruption ;
- Une **fonction** est composée de plusieurs blocs de base reliés entre eux par le flot de contrôle ;
- Le **programme** complet se compose de plusieurs fonctions interagissant entre elles.

En particulier, les interactions entre blocs de base au sein d'une fonction, et les interactions entre fonctions au sein du programme, sont représentées par deux types de graphes : le graphe de flot de contrôle, appelé CFG, et le graphe d'appels de fonctions, appelé CG. Ces graphes ne sont pas uniques et peuvent prendre de nombreuses formes, en fonction des optimisations appliquées lors de la compilation ou d'une éventuelle obfuscation.

L'obfuscation vise à dissimuler la logique sous-jacente d'un programme sans en altérer la fonctionnalité, en obscurcissant la syntaxe du programme tout en préservant sa sémantique, dans le but de gêner et de compliquer la rétro-ingénierie. L'obfuscation statique, qui constitue l'objet de cette thèse, peut être appliquée via différentes techniques : l'obfuscation des données, comme les expressions mixtes arithmético-booléennes, l'obfuscation intra-procédurale, qui modifie le flot de contrôle d'une fonction, comme l'aplanissement du CFG ; l'obfuscation inter-procédurale, qui modifie les interactions entre fonctions au niveau du programme, comme le fait de couper une fonction en plusieurs morceaux. Pour ce faire, différents obfuscateurs existent, parmi lesquels Tigress et OLLVM, qui figurent parmi les plus utilisés dans le milieu académique.

A.4.2 Apprentissage de représentations

Les récents progrès de l'apprentissage automatique ont permis le développement de nouvelles approches dédiées à l'analyse du code binaire. L'une d'elles consiste à apprendre des représentations du code binaire, où un objet binaire donné est projeté dans un espace euclidien [16], et par conséquent, représenté par un vecteur de taille fixe.

Historiquement, obtenir un tel vecteur reposait sur l'utilisation de caractéristiques conçues manuellement par des experts en rétro-ingénierie, qui étaient, par la suite, traitées par des algorithmes traditionnels d'apprentissage automatique [59, 79, 9, 133, 139]. Ces caractéristiques sont généralement regroupées en plusieurs catégories : les caractéristiques syntaxiques, obtenues directement des instructions désassemblées ; les caractéristiques structurelles, qui capturent les informations issues des graphes extraits du binaire et les caractéristiques sémantiques, qui visent à refléter le comportement effectif du code binaire, souvent au moyen d'exécution symbolique ou d'analyse dynamique.

Plutôt que de définir manuellement une liste de caractéristiques, un processus intrinsèquement subjectif et difficile à standardiser, les approches plus récentes cherchent à apprendre directement des représentations du code binaire à l'aide d'algorithmes d'apprentissage profond, supprimant ainsi le besoin de concevoir des descripteurs experts. Le code désassemblé se prête particulièrement bien à ce type d'approche. Bien qu'il ne constitue pas un langage naturel au sens strict, il peut être efficacement traité au moyen de techniques issues du traitement automatique du langage naturel. Les premières méthodes de ce type se concentraient sur l'extraction de caractéristiques élémentaires, issues de représentations textuelles du code, telles que les sacs de mots, les n-grammes ou le TF-IDF [118, 125, 13]. Par la suite, il a été démontré que des modèles de plongement, tel que word2vec, peuvent efficacement capturer des motifs syntaxiques dans le code binaire [37]. S'appuyant sur le succès des architectures basées sur les transformeurs, des modèles tels que BERT ont également été adaptés au code binaire [86, 145]. Typiquement, ces modèles sont entraînés ou ajustés sur du code assembleur et intègrent une étape de normalisation afin de réduire la taille du vocabulaire, souvent en abstrayant les adresses mémoire, les noms de registres et d'autres tokens à forte variabilité.

Cependant, comme le code binaire se modélise naturellement sous forme de graphes, l'apprentissage de représentations de graphes [22] s'avère particulièrement pertinent dans ce contexte. Outre les noyaux de graphes [81], l'essor récent des réseaux de neurones de graphes [58, 147] a permis de relever les défis propres à ce type de données. L'efficacité de ces modèles repose sur leur capacité à exploiter conjointement deux aspects fondamentaux des graphes : leur structure, représentée par une matrice d'adjacence, et les caractéristiques intrinsèques des nœuds eux-mêmes.

À chaque couche d'un réseau de neurones de graphes, les représentations des nœuds sont raffinées de manière itérative en intégrant les informations provenant à la fois du nœud courant, quel que soit son degré, et de ses voisins. Ce processus d'apprentissage itératif permet de propager l'information au sein du graphe, de sorte que chaque nœud intègre progressivement le contexte de nœuds de plus en plus éloignés.

Les réseaux de neurones de graphes présentent plusieurs des propriétés recherchées dans le contexte d'apprentissage sur graphes. Premièrement, ils sont capables de traiter des graphes de tailles et d'ordres variables, contrairement aux modèles d'apprentissage profonds traditionnels qui opèrent uniquement sur des entrées de taille fixe. Deuxièmement, leurs plongements de nœuds sont équivariants face aux permutations des nœuds, à condition que certains fonctions, définissant en partie ces modèles, soient elles-mêmes équivariantes ou invariantes aux permutations. Troisièmement, le plongement du

graphe obtenu est invariant aux permutations, sous réserve que la fonction qui aggrège les plongements de noeuds, soit elle-même invariante.

Parmi les réseaux de neurones les plus utilisés figurent les réseaux convolutionnels de graphes (GCN) [77], SAGE [64], le réseau isomorphe de graphe (GIN) [151] et le réseau d'attention sur graphe (GAT) [142].

A.5 Créer et collecter des données réalistes et obfusquées

La recherche sur l'obfuscation de binaires natifs a été considérablement freinée par une limitation majeure : l'absence de jeux de données à grande échelle constitués de programmes binaires obfusqués et annotés. En effet, l'apprentissage de représentations binaires requiert souvent une vérité de terrain, incluant des informations précises sur les techniques d'obfuscation appliquées et leurs paramètres. Or, les binaires obfusqués observés en pratique, tels que des échantillons de logiciels malveillants ou des applications légitimes et protégées, sont difficiles à annoter de manière fiable, malgré un travail manuel important de rétro-ingénierie, et avec un risque élevé de biais. À l'inverse, les jeux de données synthétiques existants présentent leurs propres limites. Beaucoup se restreignent à des binaires obfusqués via OLLVM, basé sur une version obsolète de LLVM, et n'intègrent qu'un ensemble restreint de transformations, principalement intra-procédurales et liées aux données. De plus, ces jeux de données consistent fréquemment en des fragments de code basiques, tels que des algorithmes de tri, qui ne reflètent pas la complexité, tant structurelle que sémantique, des programmes `C` rencontrés en pratique. D'autres ne contiennent pas des données en nombre suffisant, ne permettant donc pas l'entraînement de modèles d'apprentissage profond, qui exigent de larges volumes d'exemples diversifiés.

Pour combler ce déficit, un nouveau jeu de données synthétique de grande ampleur, appelé ObfuBench, a été élaboré. Celui-ci inclut des binaires originaux non obfusqués, leurs équivalents obfusqués, ainsi que des métadonnées détaillées décrivant les techniques d'obfuscation appliquées. Deux obfuscateurs sont évalués, Tigress-3.1 et OLLVM, sur cinq projets réalistes écrits en `C` : `zlib`, `lz4`, `minilua`, `sqlite` et `freetype`. Pour chaque obfuscateur, un panel de techniques d'obfuscation a été sélectionné, incluant des transformations intra-procédurales, de données, et, lorsque cela est possible, inter-procédurales. Les binaires ont été compilés pour l'architecture `x64`, avec les niveaux d'optimisation `-O0` et `-O2`.

Si ObfuBench constitue le principal jeu de données de cette thèse, d'autres jeux de données sont néammoins utilisés, afin de favoriser la généralisation des contributions proposées. Ainsi, la partie obfusquée du jeu de données BinKit [76], le jeu de données d'expressions mixte arithetico-booléennes Loki [127], ainsi que des échantillons obfusqués issus de programmes réels, sont également étudiés. Certains binaires standards, employés pour quelques expériences au cours de cette thèse, proviennent du Dataset-1 [95].

A.6 Détection et caractérisation d'obfuscation

La première étape au cours de l'analyse d'obfuscation consiste à identifier et à caractériser les techniques appliquées. D'ordinaire, un ingénieur en rétro-ingénierie inspecte manuellement certaines fonctions d'un binaire afin d'y détecter des motifs caractéristiques d'obfuscation. Toutefois, cette approche manuelle, qui repose soit sur l'intuition, soit sur une inspection exhaustive, est à la fois chronophage et peu fiable. Une stratégie plus efficace consiste à recourir à des algorithmes automatisés, entraînés pour reconnaître ces motifs d'obfuscation. La recherche sur l'identification et la caractérisation d'obfuscation, deux aspects qui déterminent en grande partie son degré de furtivité, a suscité un intérêt croissant récemment, donnant lieu à plusieurs travaux, détaillés au Chapitre 6.

Dans cette thèse, nous étudions la furtivité de l'obfuscation en abordant conjointement les problèmes d'identification et de caractérisation, modélisés respectivement comme des tâches de classification binaire et multi-classe. Ces tâches sont effectuées au niveau des fonctions, puisque des prédictions concernant les fonctions peuvent être utilisées pour inférer des propriétés au niveau du programme général. Les travaux existants utilisent différentes approches, allant des algorithmes d'apprentissage automatique élémentaires aux modèles avancés de réseaux de neurones de graphes, ainsi qu'une grande variété de descripteurs, tels que la complexité cyclomatique, les n-grammes d'instructions assembleur ou encore les vecteurs de comptage.

Ainsi, afin de détecter l'obfuscation au sein d'une fonction et d'identifier son type, deux formats de graphes sont utilisés : le CFG classique et le CFG-IR, basé sur une représentation intermédiaire. Ce dernier conserve les fondements du CFG standard, mais remplace les instructions assembleur brutes par du Pcode, plus abstrait, et indépendant de l'architecture. Nous exploitons le jeu de données ObfuBench, que nous divisons en deux sous-ensembles selon deux stratégies distinctes. Le premier, appelé `Dataset-1`, regroupe une fonction donnée et ses versions obfusquées dans le même ensemble, où les ensembles d'entraînement, de validation et de test sont obtenus en séparant toutes les fonctions du jeux de données, selon un ratio (64%, 16%, 20%), avec un échantillonnage stratifié, afin de préserver une distribution similaire du nombre de blocs de base entre les ensembles. Le second, appelé `Dataset-2`, est construit à partir de toutes les fonctions issues des projets `zlib, lz4` et `minilua`, avec leurs versions obfusquées, utilisées pour constituer les ensembles d'entraînement et de validation selon un ratio (80%, 20%). L'ensemble de test, quant à lui, est composé exclusivement des fonctions issues des projets `sqlite` et `freetype`, avec leurs versions obfusquées. Cette configuration simule un scénario réaliste de détection et de caractérisation de l'obfuscation, où l'objectif est d'analyser un binaire jamais rencontré jusqu'à présent, à l'aide d'un modèle entraîné uniquement sur des binaires distincts, générés dans un environnement contrôlé. Ces deux stratégies induisent un déséquilibre de classes marqué, avec 11 classes obfusquées contre une seule non obfusquée. Par ailleurs, le `Dataset-2` présente une difficulté accrue par rapport au `Dataset-1`, puisque les projets de l'ensemble d'entraînement et de test diffèrent par leurs styles de programmation et par le nombre et la nature des fonctions

susceptibles d'être obfusquées. En forçant les ensembles d'entraînement, de validation et de test à provenir de projets distincts, le modèle doit nécessairement s'appuyer sur des régularités propres à un projet donné, favorisant ainsi une évaluation plus robuste et généralisable.

Cette thèse se propose d'étudier en détail les différents modèles et caractéristiques employés dans le cadre de ces classications binaire et multi-classes. Les classifieurs traditionnels, tels que les forêts aléatoires et le Gradient Boosting, servent de références, tandis que diverses architectures de réseaux de neurones de graphes sont évaluées. Ces modèles n'opèrent pas tous au même niveau de granularité. Les méthodes classiques requièrent des caractéristiques au niveau du graphe, tandis que les réseaux de neurones de graphes nécessitent des caractéristiques au niveau des nœuds, en complément de la structure du graphe elle-même. Par conséquence, chaque bloc de base, qu'ils proviennent du CFG ou CFG-IR, doivent être associés à un vecteur de caractéristiques approprié.

Ainsi, les méthodes de référence sont évaluées à partir, d'une part, de caractéristiques globales de graphe, telles que la complexité cyclomatique, et, d'autre part, d'une représentation de la distribution assembleur obtenue via une pondération TF-IDF appliquée aux mnémoniques et opérandes. À l'inverse, les caractéristiques utilisées pour les réseaux de neurones de graphes sont enrichies progressivement : d'abord un descripteur identité, dépourvu d'information sémantique sur les nœuds, puis des caractéristiques plus élaborées décrivant précisément chaque nœud du CFG ou du CFG-IR. Ces dernières incluent un comptage de classes prédéfinies de mnémoniques, un vecteur de comptage d'instructions assembleur, spécifiques à l'architecture `x64`, son équivalent indépendant de l'architecture, basé sur le Pcode, et enfin un transformer pré-entraîné sur l'assembleur, utilisé pour obtenir des représentations vectorielles des blocs de base.

L'ensemble des modèles est évalué avec leurs caractéristiques respectives sur les `Dataset-1` et `Dataset-2`. Le réglage des hyperparamètres est réalisé avec une recherche par grille pour les baselines et avec Optuna pour les réseaux de neurones de graphes. En raison du déséquilibre de classes, aussi bien en classification binaire qu'en multi-classes, toutes les évaluations reposent sur l'exactitude équilibrée. Les résultats correspondant aux niveaux d'optimisation `-O0` et `-O2` sont présentés séparément. Si les tendances générales restent similaires, les performances sous `-O2` sont généralement plus faibles, vraisemblablement en raison des optimisations du compilateur, qui perturbent les schémas d'obfuscation initiaux.

Les résultats obtenus en classification binaire montrent que les modèles de référence présentent des performances globalement satisfaisantes, en particulier concernant le CFG compilé en `-O0`, avec des scores d'exactitude atteignant au moins 0,60. La meilleure performance est obtenue par un modèle de Gradient Boosting, combiné au descripteur TF-IDF du code assembleur. Cette combinaison surpasse systématiquement ses homologues, à savoir les forêts aléatoires et les descripteurs fondées sur les graphes. Ce résultat peut être attribué à la granularité plus fine de la distribution du code assembleur, qui permet de mieux capter les irrégularités syntaxiques introduites par l'obfuscation, contrairement aux descripteurs de haut niveau, plus grossiers, dérivés des graphes. Sur l'ensemble des résultats obtenus, les performances liées au `Dataset-1` sont systématique-

ment supérieures à celles du `Dataset-2`. Ce comportement est attendu, compte tenu de la difficulté accrue posée par `Dataset-2`. Néanmoins, l'écart de performance demeure inférieur à 0.10, ce qui montre que les fonctions obfusquées et non obfusquées peuvent encore être efficacement distinguées, même dans des scénarios exigeants.

Les résultats obtenus avec les réseaux de neurones de graphes sont plus contrastés. En particulier, l'utilisation du descripteur identité conduit à des performances médiocres, souvent inférieures à celles des modèles de référence plus simples. Cela suggère que s'appuyer uniquement sur la structure du graphe est insuffisant pour distinguer les fonctions obfusquées des fonctions non obfusquées, certaines techniques d'obfuscation opérant exclusivement à l'intérieur des blocs de base, sans modifier la topologie globale du CFG. L'utilisation d'une caractéristique fondée sur le comptage de classes de mnémoniques améliore les performances par rapport au descripteur identité, mais reste inférieure aux méthodes de référence. Cela peut s'expliquer par sa granularité trop grossière et sa dépendance à des classes de mnémoniques manuellement définies.

L'enrichissement des caractéristiques initiales des réseaux de neurones de graphes s'avère donc essentiel : les descripteurs plus raffinées, notamment ceux basés sur le comptage des mnémoniques, qu'elles soient dérivées de l'assembleur ou du Pcode, améliorent substantiellement l'exactitude (d'environ 10%). Grâce à ces enrichissements, les réseaux de neurones de graphes parviennent à égaler, voire à dépasser, les méthodes de référence classiques, en particulier sur le `Dataset-2`. La version basée sur le Pcode obtient toutefois des résultats légèrement inférieurs à son équivalent assembleur, principalement en raison de la nature plus abstraite et limitée de ses mnémoniques, qui réduit sa capacité à capturer certains schémas spécifiques et, par conséquent, à discriminer efficacement les fonctions obfusquées.

Les observations précédemment faites avec le CFG classique demeurent valables pour le CFG-IR, bien que les résultats obtenus avec ce dernier tendent à être inférieurs à ceux du CFG standard, y compris pour les modèles de référence traditionnels.

Concernant les niveaux d'optimisation du compilateur, les résultats obtenus sur les binaires compilés en `-O0` dépassent ceux des binaires compilés `-O2` de près de 10%. Cet écart de performance s'explique principalement par le fait que les optimisations du compilateur peuvent partiellement ou totalement supprimer certains schémas d'obfuscation. Ainsi, distinguer les fonctions obfusquées des non obfusquées devient plus difficile avec le niveau d'optimisation `-O2`, surtout lorsque seuls des descripteurs élémentaires, tels que le descripteur identité, sont utilisés.

Afin d'évaluer la robustesse des modèles face aux différentes partitions de données, deux nouveaux partitionnements des `Dataset-1` et `Dataset-2` (entraînement, validation et test) sont obtenus, pour chaque jeu de données et chaque niveau d'optimisation, à l'aide de graines aléatoires distinctes. Les résultats, restreints aux seuls CFG, illustrent l'exactitude moyenne ainsi que les écarts-types correspondants sur les trois partitions de données. Globalement, les modèles démontrent une bonne robustesse face aux variations de partitionnement, bien que certains réseaux de neurones de graphes affichent une instabilité plus marquée que les modèles de référence.

En ce qui concerne le problème de caractérisation, celui-ci peut être formulé à la

fois comme un problème de classification multi-label et de classification multi-classe. En effet, une fonction peut être obfusquée par une seule technique d'obfuscation ou par une combinaison de plusieurs. La formulation multi-label présente l'avantage de permettre au modèle d'identifier indépendamment chaque technique d'obfuscation, et ainsi de reconnaître des combinaisons inédites, non observées durant l'entraînement. Toutefois, la classification multi-label est réputée difficile, en particulier lorsque le nombre de labels est important. Par conséquent, la majorité des travaux existants simplifient cette tâche en l'exprimant comme une classification multi-classes, où chaque schéma composite d'obfuscation est considéré comme une nouvelle classe. Bien que cette approche réduise la complexité du problème, elle sacrifie la modularité offerte par le cadre multi-label, qui permet une identification plus fine des techniques individuelles au sein de schémas composites.

Dans cette étude, nous retenons la formulation d'une classification multi-classe, visant à associer chaque fonction obfusquée à l'une des 11 classes prédéfinies. Les résultats obtenus dans ce cadre corroborent plusieurs observations précédemment faites au sujet de la classification binaire. En particulier, les scores de classification multi-classes correspondant au niveau d'optimisation `-O0` sont en moyenne 10% inférieurs à ceux observés en classification binaire. Malgré la complexité accrue liée à la gestion de 11 classes d'obfuscation (contre un maximum de 4 dans les travaux antérieurs), les méthodes classiques de référence obtiennent encore des performances étonnamment robustes. À l'inverse, les réseaux de neurones de graphes reposant sur des descripteurs élémentaires, telles que l'identité ou les comptages de classes de mnémoniques, affichent des performances nettement plus faibles. En revanche, les descripteurs enrichis, issus de vecteurs de comptage de mnémoniques, permettent d'obtenir des modèles surpassant les méthodes de référence, notamment sur `Dataset-2`.

Cependant, les résultats multi-classes correspondant au niveau d'optimisation `-O2` révèlent un écart bien plus marqué par rapport à la classification binaire. Alors que l'écart entre les niveaux `-O0` et `-O2` reste relativement modeste en classification binaire, il devient nettement plus prononcé dans le cas multi-classes. Même avec des caractéristiques plus détaillés, les réseaux de neurones de graphes peinent à égaler les performances des méthodes de référence plus simples. Cette dégradation s'explique par la difficulté accrue à distinguer des schémas d'obfuscation spécifiques et souvent subtils, lorsqu'ils sont soumis à des optimisations, lesquelles peuvent les altérer ou les supprimer partiellement.

A.7 Comparer des programmes standards et obfusqués

Lors de l'analyse d'un code binaire obfusqué, certaines attaques directes peuvent être trop coûteuses à mettre en œuvre. Dans ce cas, il est possible de recourir à des attaques indirectes, qui visent à permettre à l'attaquant d'acquérir des connaissances sur les binaires protégés sans avoir à les désobfusquer intégralement. Ces attaques indirectes peuvent s'appuyer sur des tâches d'analyse binaire déjà existantes, telles que la similarité et le diffing binaire.

Le diffing binaire a pour objectif d'établir une correspondance un-à-un entre les fonc-

tions de deux binaires. Il répond à une variété de cas d'usage, parmi lesquels l'analyse de logiciels malveillants, la détection de correctifs, l'identification de bibliothèques statiquement liées ou encore la détection de clones. Chacun de ces cas d'usage présente des caractéristiques spécifiques et introduit des contraintes particulières. Par exemple, les échantillons de malwares sont fréquemment obfusqués ; l'analyse de correctifs recherche des changements subtils et localisés ; et la détection de bibliothèques statiquement liées est rendue complexe par l'absence de fonctions importées, qui servent généralement de points d'ancrage dans le processus de diffing. Face à ces contraintes, le diffing binaire se doit d'être adaptable, en exploitant les caractéristiques propres à chaque situation, afin d'obtenir de meilleurs résultats. BinDiff [49, 41] et Diaphora [79] sont les outils de diffing binaire les plus utilisés en pratique, bien qu'ils présentent des limites en termes d'applicabilité, car dépourvues de la modularité nécessaire pour s'adapter à la variété des objectifs d'analyse.

Dans cette thèse, nous nous intéressons à l'impact de l'obfuscation sur les outils d'analyse binaire existants, en particulier les techniques de diffing et de similarité. Nous étudions le potentiel de ces outils en tant que vecteurs d'attaques indirectes contre l'obfuscation.

Deux grandes approches permettent d'obtenir des résultats de diffing : la première repose sur les outils traditionnels de diffing, qui produisent directement des correspondances au niveau des fonctions entre deux binaires, tandis que la seconde exploite les outils de similarité binaire, lesquels calculent des scores de similarité pour des paires de fonctions ; un algorithme de correspondance est ensuite appliqué à la matrice de similarité résultante afin de déterminer les correspondances de fonctions.

Dans cette étude, nous étudions les deux approches, en considérant le f1-score comme mesure d'efficacité d'un résultat de diffing par rapport à la vérité terrain. Les outils de diffing évalués comprennent BinDiff, Diaphora-3.0 et QBinDiff [104, 105], sélectionnés en raison de leur large adoption en pratique, de leur granularité fonctionnelle et de leur recours exclusif à l'analyse statique. Les outils de similarité binaire évalués incluent Asm2vec [37], GMN [87], PalmTree [86] et JTrans [145]. Ces outils produisent des matrices de similarité fondées sur des plongements, à partir desquelles les correspondances fonctionnelles sont déduites à l'aide de l'algorithme hongrois. Le choix de cet algorithme est motivé par son exactitude, puisqu'il fournit une solution optimale au problème de correspondence.

Tout d'abord, nous conduisons une étude d'ablation sur un differ binaire, appelé QBinDiff, conçu pour pallier les limites existantes et le manque de modularité de BinDiff et Diaphora. Étant donnés deux binaires, appelés primaire et secondaire, QBinDiff résout par approximation une instance spécifique du problème d'alignement de réseaux. Il identifie une correspondance entre les nœuds des CGs des deux binaires, qui préserve au mieux les similarités structurelles et sémantiques. Plus précisément, il représente ce problème à l'aide d'une formulation quadratique contrainte, qui est résolue au moyen d'un algorithme de propagation de croyances, basé sur des enchères [8]. Celui-ci repose sur un modèle graphique, où les nœuds représentent les variables du problème et les arêtes indiquent les dépendances entre les variables. Les valeurs associées aux variables

sont mises à jour de manière itérative au moyen de « messages » transmis via les arêtes du graphe. Une fois la convergence atteinte, les probabilités marginales sont utilisées pour déterminer la solution optimale du problème.

L'algorithme de QBinDiff se compose de trois éléments principaux et de plusieurs paramètres :

- Une matrice de similarité (S) qui capture les similarités deux-à-deux entre les nœuds des deux CGs, calculées à l'aide d'un ensemble de descripteurs définis par l'utilisateur, caractérisant la fonction correspondante et l'ensemble du programme ;
- Une matrice de carrés qui encode les similarités topologiques entre les deux CGs ;
- Un ensemble de paramètres définis par l'utilisateur, parmi lesquels un paramètre de compromis α, la distance $dist$, le taux de parcimonie s_{ratio} et le paramètre de relaxation ϵ.

La première phase de QBinDiff, dite d'ancrage, permet de pré-apparier les fonctions importées, en les exploitant comme des points d'ancrage fiables, en particulier dans le cas des binaires dynamiquement liés, avant tout autre processus de correspondance. Cette phase peut être complétée par d'autres procédures de prétraitement qui établissent davantage de correspondances ou initialisent la matrice de similarité.

Ensuite, des vecteurs de caractéristiques sont extraits pour les fonctions primaires et secondaires, à partir d'un ensemble de features pouvant inclure des caractéristiques structurelles liées CFG, des propriétés du CG, ou des caractéristiques bas niveau liées à l'assembleur. QBinDiff propose par défaut un total de 33 caractéristiques différentes. La matrice de similarité S est alors calculée comme une combinaison linéaire pondérée de la distance choisie, appliquée aux vecteurs de caractéristiques primaires et secondaires. Parmi les distances disponibles figurent les distances de Canberra, Euclidienne, cosinus et Haussmann. La matrice de similarité S tend à être de grande taille et doit généralement être réduite et décimée à l'aide du ratio de parcimonie s_{ratio}, en supprimant les plus faibles scores de similarité, peu susceptibles de produire des correspondances correctes.

Le paramètre α agit, quant à lui, comme un curseur entre la similarité des fonctions, représentées par des noeuds au sein du CG, et la topologie de ce même graphe. Lorsque $\alpha = 0.0$, seule la structure du graphe est prise en compte, et la similarité des nœuds est ignorée. À l'inverse, $\alpha = 1.0$ indique que seule la similarité est considérée, en négligeant totalement la structure. Le paramètre ϵ joue un rôle de relaxation facilitant la convergence de l'algorithme.

Nous analysons successivement l'impact de l'ancrage, l'arbitrage entre la similarité et la topologie via α, l'impact des features, et effectuons une étude des différents paramètres et de leur combinaison optimale.

Cette étude d'ablation révèle que l'introduction de l'ancrage entraîne une amélioration notable du f1-score, d'au moins 10% : en fixant les fonctions importées comme points d'ancrage, le processus de correspondance devient plus aisé, puisqu'il peut être

contraint à des clusters de fonctions associés à ces ancres, réduisant ainsi considérablement la complexité computationnelle.

Concernant le paramètre de compromis α, qui équilibre le poids donné à la similarité des fonctions et à la topologie du CG, nous le faisons varier dans trois conditions expérimentales : la configuration par défaut de QBinDiff, une version de QBinDiff avec une matrice de similarité perturbée, et enfin une variante de QBinDiff avec des matrices d'adjacence perturbées. Cette expérience montre que le f1-score connaît deux variations abruptes : entre $\alpha = 0.0$ et $\alpha = 0.1$, puis entre $\alpha = 0.9$ et $\alpha = 1.0$. La première hausse traduit l'effet bénéfique de l'introduction des mesures de similarité dans le processus de diffing. En revanche, la seconde transition élimine toute considération de la topologie, privant l'algorithme d'informations structurelles cruciales et entraînant une chute brutale du f1-score. Ces variations soulignent l'importance de combiner similarités structurelles et fonctionnelles.

Lorsque les matrices d'adjacence sont perturbées, donner un poids excessif à la topologie (avec un faible α) amène l'algorithme à s'appuyer sur une information bruitée, ce qui dégrade le f1-score. À mesure que α augmente, la performance s'améliore et converge vers celle de la configuration par défaut. Cela suggère que QBinDiff peut atténuer l'effet du bruit structurel en priorisant les mesures de similarité fiables.

A l'inverse, lorsque la matrice de similarité est perturbée, augmenter α de 0.0 à environ 0.5 améliore le f1-score, montrant qu'une information bruitée reste utile si elle demeure limitée. En revanche, lui accorder une importance excessive entraîne une dégradation marquée de la performance, illustrant les limites d'une dépendance trop forte à des similarités fonctionnelles bruitées.

Par ailleurs, QBinDiff offre également une large gamme de paramètres ajustables, notamment le choix des descripteurs utilisés pour calculer la matrice de similarité. Nous définissons trois ensembles de descripteurs, qui sont utilisés alors même que α varie. Lorsque $\alpha = 0.0$, la similarité est ignorée, et le choix des features n'a pas d'impact. En revanche, à mesure que α augmente, les performances divergent selon les ensembles de descripteurs choisis. Lorsque $\alpha = 1.0$, les features dérivées de la structure du CG deviennent particulièrement utiles, compensant l'absence d'information topologique.

D'autres paramètres influencent également le comportement de QBinDiff. Ainsi, des valeurs élevées de α (0.8–0.9) donnent de meilleurs résultats de correspondance ; des valeurs élevées de ϵ (0.9–1.0) favorisent une convergence plus rapide ; les distances de Canberra et d'Haussmann sont les plus performantes ; l'augmentation du ratio de parcimonie s_{ratio} n'entraîne pas de baisse marquée des performances et peut même améliorer la convergence sur de larges programmes binaires, tandis qu'un ratio de parcimonie s_{ratio} trop élevé supprime des correspondances utiles, rendant le résultat de diffing non pertinent.

À l'issue de cette analyse, la meilleure configuration de paramètres pour chaque projet, appelé `ppb`, du jeu de données utilisé, appelé Dataset-1 et précédemment utilisé dans le cadre de la similarité binaire, peut être obtenue, ainsi qu'un jeu de paramètres générique (`avb`), offrant la meilleure performance moyenne sur l'ensemble des cinq projets évalués.

Suite à cette étude d'ablation, l'objectif consiste désormais à analyser l'efficacité des outils de diffing binaire et de similarité sur des binaires standards, non obfusqués. Nous menons une évaluation comparative de plusieurs solutions de diffing binaire à l'aide d'un jeu de données contenant divers binaires non obfusqués. Cette comparaison met en évidence plusieurs constats. Tout d'abord, le choix de l'exporteur binaire influence considérablement les performances de diffing. L'utilisation de QBinDiff avec Quokka [21] produit des résultats nettement supérieurs à ceux obtenus avec BinExport. Ces résultats soulignent l'avantage notable de Quokka, qui exporte des informations, telles que des références croisées explicites, absentes de BinExport.

Par ailleurs, les performances des outils de diffing varient fortement selon les approches utilisées. QBinDiff obtient les meilleurs f1-scores, suivi de près par BinDiff, tandis que Diaphora obtient de moins bons résultats. Les outils de similarité binaire, en comparaison, se révèlent généralement moins performants que les outils de diffing traditionnels. Néanmoins, PalmTree et JTrans s'approchent des performances de Diaphora grâce à l'utilisation de représentations plus adaptées et riches. À l'inverse, des modèles tels qu'Asm2Vec et GMN produisent des scores inférieurs en raison de plongements plus grossiers.

Enfin, l'écart de f1-scores entre les configurations ppb et avb est minimal, ce qui suggère que les utilisateurs de QBinDiff n'ont pas nécessairement besoin de reproduire la recherche d'hyperparamètres, menée au cours de cette étude d'ablation. Ils peuvent s'appuyer directement sur les meilleurs paramètres moyens (avb), identifiés précédemment, pour analyser leurs propres programmes binaires.

Après cette analyse des binaires standards non obfusqués, nous abordons le cœur de ce Chapitre, à savoir, le diffing de binaires obfusqués comme vecteur d'attaques indirectes. Contrairement aux binaires classiques, pour lesquels les techniques de diffing binaire se sont révélées efficaces, le diffing de binaires obfusqués demeure un défi. Ce travail vise à combler les lacunes existantes de la littérature en menant une étude expérimentale sur la résilience des outils de diffing face aux programmes obfusqués. Une telle étude présente un intérêt pratique puisque les attaquants contournent fréquemment l'obfuscation par des moyens indirects. Ces approches peuvent fragiliser les mécanismes de protection ou conduire à des fuites d'informations sensibles. Par exemple, en comparant un binaire non obfusqué avec une variante nouvellement obfusquée, un attaquant peut isoler les composants communs entre les programmes et identifier les fonctionnalités nouvellement introduites.

Si un attaquant parvient à retrouver une correspondance correcte entre deux binaires, lorsque l'un ou les deux programmes ont été obfusqués, cela indique que l'obfuscation n'a pas suffisamment altéré le programme. En réalité, une obfuscation ne peut être considérée comme efficace que si elle perturbe les propriétés essentielles du programme au point de les rendre inaccessibles après transformation. Ce concept définit un modèle de menace dans lequel un attaquant exploite la connaissance d'un binaire pour contourner l'obfuscation appliquée à une nouvelle version d'un programme. Une telle approche a déjà été employée pour analyser du bytecode Android obfusqué. Nous utilisons le terme `standard-obfusqué`, pour désigner le scénario où seul un binaire est obfusqué, tandis que

le scénario `obfusqué-obfusqué` considère le cas où les deux binaires sont obfusquées. De tels cas se rencontrent fréquemment, notamment dans des environnements caractérisés par des mises à jour régulières, comme les applications mobiles. Les logiciels malveillants évoluent également de manière itérative au fil du temps, rendant ce type de comparaison particulièrement pertinent.

Bien qu'ils soient populaires en pratique, ni BinDiff ni Diaphora n'ont été conçus pour traiter des binaires fortement obfusqués. QBinDiff, à l'inverse, a été spécifiquement pensé pour être modulaire, afin d'exploiter le diffing binaire pour des cas d'usage variés, notamment adversariaux, comme ceux impliquant l'obfuscation. Sa modularité permet un contrôle précis des descripteurs utilisés pour calculer la similarité. Cette adaptabilité constitue un avantage notable lorsqu'une connaissance au-préalable des obfuscations appliquées est disponible. Une telle connaissance peut être obtenue manuellement, par rétro-ingénierie ou bien automatiquement, par des algorithmes d'apprentissage automatique capables d'inférer directement la présence et le type d'obfuscation (cf. Chapitre 6). Bien que ces classifieurs présentent des limites, telles qu'une précision réduite à haut niveau d'optimisation, ils offrent néanmoins une approximation fiable des transformations appliquées. Ces informations guident ensuite l'exclusion des descripteurs les plus susceptibles d'être affectées par l'obfuscation utilisée, renforçant ainsi la robustesse du processus de diffing.

En conséquence, nous définissons deux ensembles de descripteurs pour chaque technique d'obfuscation : un ensemble stable, comprenant les descripteurs résilients face à l'obfuscation considérée et un ensemble instable, regroupant les caractéristiques connues pour être altérées. Ces ensembles sont établis en comparant des binaires non obfusqués à leurs variantes obfusquées, puis en analysant l'impact de chaque technique d'obfuscation. Dans les scénarios où le type d'obfuscation ne peut être inféré, toutes les caractéristiques de QBinDiff peuvent être activées par défaut. En revanche, lorsqu'il est identifié, l'ensemble stable spécifique à l'obfuscation utilisée, doit être privilégié afin d'améliorer la précision. L'évaluation de QBinDiff selon ces trois configurations (défaut, stable et instable) met en évidence un net gain de performance, obtenu grâce à la variante stable de QBinDiff, appelée QBinDiff_s. Cet expérience montre la forte dépendance du diffing binaire vis-à-vis des descripteurs choisis, et sélectionner des caractéristiques robustes face à l'obfuscation appliquée améliore donc substantiellement l'exactitude des résultats.

Le premier axe d'expérimentation concernant le diffing binaire dans un contexte obfusqué, porte sur le scénario `standard-obfusqué`. Cet expérience inclut les obfuscateurs OLLVM-14 et Tigress, ainsi que leurs techniques respectives. Les cinq projets du dataset ObfuBench sont analysés vis-à-vis de deux niveaux d'optimisation, `-O0` et `-O2`. Différents niveaux d'obfuscation sont évalués, correspondant à la proportion de fonctions initialement obfusquées : 10%, qui correspond à un scénario simulant un usage limité de l'obfuscation, réaliste dans le cadre de systèmes contraints, comme les MCUs, 50% qui représente un compromis pragmatique entre sécurité et performance et 100%, cas le plus défavorable à l'attaquant, où le binaire complet est obfusqué.

Contrairement aux expériences menées sur des binaires non obfusqués, cette analyse

rapporte deux f1-scores distincts : un f1-score général, calculé sur l'ensemble des fonctions, et un f1-score obfusqué, calculé uniquement sur les fonctions obfusquées. Idéalement, ces deux scores doivent rester proches ; un écart significatif, avec un f1-score général élevé mais un f1-score obfusqué faible, signalerait que l'outil de diffing réussit à faire correspondre uniquement les fonctions non obfusquées, échouant sur celles obfusquées.

Les résultats obtenus avec OLLVM et Tigress mettent en évidence plusieurs observations essentielles.

- Efficacité globale du diffing binaire. Parmi les outils évalués, QBinDiff, QBinDiff$_s$ et JTrans se distinguent comme les plus efficaces, atteignant des f1-scores supérieurs ou égaux à 0.80, pour de nombreuses techniques d'obfuscation, en particulier pour les obfuscations de données et intra-procédurales. De manière générale, les outils traditionnels de diffing binaire, tels que BinDiff, Diaphora et QBinDiff, surpassent les techniques de similarité binaire, combinées à un algorithme d'appariement, comme GMN et Asm2Vec. Notons que JTrans ne souffre pas des limitations de performance observées avec GMN ou Asm2Vec, grâce à son recours à des plongements de fonctions robustes, intégrant des informations de flot de contrôle, contrairement à PalmTree, qui présente des performances plus faibles. QBinDiff dépasse systématiquement les autres outils, en combinant la structure des CGs et la similarité des fonctions. À l'inverse, JTrans repose uniquement sur la similarité fondée sur les plongements, tandis que BinDiff met l'accent sur la propagation des correspondances au sein du graphe de flot de contrôle.

- Résilience face à l'obfuscation: QBinDiff vs QBinDiff$_s$. QBinDiff et QBinDiff$_s$ montrent des comportements distincts selon les scénarios d'obfuscation. QBinDiff$_s$ améliore considérablement les performances pour la majorité des obfuscations inter et intra-procédurales par rapport à QBinDiff, obtenant de meilleurs f1-scores, tant généraux que spécifiques aux fonctions obfusquées. Dans certains cas, notamment en ce qui concerne la Virtualization, il améliore le f1-score obfusqué de 44 points. Toutefois, cette amélioration ne s'étend pas aux obfuscations liées aux données et à la technique de Prédicats Opaques, où QBinDiff obtient de meilleurs résultats. Cette différence provient de l'ensemble de descripteurs stables utilisés par QBinDiff$_s$ pour les obfuscations de données : bien que des caractéristiques comme les mnémoniques assembleur soient supprimées afin de réduire le bruit induit par l'obfuscation, certaines de ces caractéristiques peuvent néanmoins contenir des informations précieuses à un appariement correct. Leur suppression complète entraîne donc, in fine, une baisse de performance.

- Écart entre scores f1 généraux et obfusqués. L'écart entre les f1-scores généraux et obfusqués varie en fonction du type d'obfuscation et de l'outil utilisé. Les outils fondés sur la similarité, tels que GMN et Asm2Vec, peinent souvent à maintenir des performances constantes sur l'ensemble des fonctions, privilégiant généralement les correspondances entre fonctions non obfusquées. En revanche, des outils plus robustes comme BinDiff, et dans une moindre mesure Diaphora, réduisent cet

écart à environ 10 points pour les obfuscations intra-procédurales et de données. JTrans et les deux variantes de QBinDiff réduisent cet écart encore davantage, à environ 5 points pour de faibles niveaux d'obfuscation. QBinDiff$_s$, en particulier, bénéficie de descripteurs stables face à l'obfuscation qui renforcent sa résilience. Ce résultat est quelque peu inattendu, dans la mesure où l'on pourrait supposer que l'obfuscation altère fortement la performance des outils de diffing. Toutefois, cette tendance ne se maintient pas pour les obfuscations inter-procédurales : dans ce cas, tous les outils présentent une chute de performance marquée, avec des écarts de f1-scores dépassant parfois 60 points. Cela conduit à des performances extrêmement faibles pour certains schémas d'obfuscation, où même les meilleurs outils n'atteignent qu'un f1-score de 0.02. Ce déclin s'explique probablement par deux facteurs : d'une part, les outils basés sur la similarité échouent à modéliser les relations inter-procédurales, ce qui entraîne une perte substantielle d'information ; d'autre part, les outils basés sur le CG exploitent des structures profondément altérées par ce type d'obfuscation, rendant le processus d'appariement peu fiable, en particulier sous la contrainte d'une correspondance une-à-une stricte.

- Impact du type d'obfuscation. Les résultats montrent que la plupart des outils de diffing restent relativement efficaces face aux obfuscations intra-procédurales et de données, même lorsque la majorité du binaire est obfusquée. Cela s'explique principalement par le fait que ces types d'obfuscation préservent la structure du CG, permettant ainsi un appariement initial fiable des fonctions. À l'inverse, les obfuscations inter-procédurales entraînent une dégradation marquée des performances, en particulier sur les f1-scores obfusqués. Cela suggère que les ingénieurs en rétro-ingénierie rencontrent davantage de difficultés face à de tels binaires et doivent recourir à des outils plus adaptatifs, tels que QBinDiff, afin d'obtenir de meilleures performances. Du point de vue de la protection logicielle, ce constat implique que les obfuscations inter-procédurales doivent être privilégiées lorsque l'objectif est de résister à la rétro-ingénierie via le diffing binaire.

- Robustesse des obfuscateurs. En moyenne, les binaires obfusqués avec Tigress produisent des f1-scores inférieurs à ceux obtenus avec OLLVM-14. Ce dernier n'inclut pas d'obfuscations inter-procédurales, particulièrement difficiles à traiter du point de vue du diffing, et ses obfuscations, liées aux données et intra-procédurales, se révèlent également moins agressives. Ainsi, les binaires obfusqués avec Tigress obtiennent en moyenne des scores inférieurs de 10 à 20 points. Cet écart de performance s'explique principalement par l'utilisation par Tigress, de transformations plus sophistiquées, comme la Virtualisation, ainsi que par une mise en œuvre plus agressive de techniques partagées avec OLLVM-14, telles que les Prédicats Opaques.

- Effet du niveau d'obfuscation. Enfin, l'augmentation du taux d'obfuscation, c'est-à-dire de la proportion de fonctions obfusquées, n'entraîne qu'un léger déclin du f1-score, notamment pour les transformations intra-procédurales et de données.

Cette tendance est particulièrement marquée pour OLLVM-14, dont les performances demeurent quasi constantes, quel que soit le niveau d'obfuscation. Ce comportement n'est cependant pas observé avec Tigress.

Ces observations se confirment dans le scénario `obfusqué-obfusqué`, bien que ce cas d'usage puisse paraître plus exigeant. QBinDiff, JTrans et BinDiff restent les meilleurs differs, bien que JTrans voit ses performances diminuer, car entraîné sur du code non obfusqué.

Enfin, nous étendons ces expériences au jeu de données BinKit, où les résultats se révèlent encore plus marqués, confirmant les constats précédents, et montrant la supériorité constante de QBinDiff, et plus encore de sa variante stable.

Pour démontrer la généralisation de nos approches, nous concluons par deux études de cas sur des binaires réels obfusqués : XTunnel et Rabobank. Les résultats montrent que seul le differ QBinDiff obtient des performances prometteuses dans le cas de XTunnel, tandis que pour Rabobank, les performances sont globalement plus faibles et contrastées. Cette disparité suggère des différences sémantiques substantielles entre les deux versions étudiées de Rabobank, rendant la vérité terrain partiellement artificielle et expliquant les résultats dégradés, par rapport à XTunnel.

A.8 Désobfuscation

Une fois l'obfuscation détectée et caractérisée, l'attaquer directement via la désobfuscation peut être la seule stratégie viable pour comprendre réellement le comportement d'un programme binaire. La désobfuscation demeure un objectif central en rétro-ingénierie, puisqu'elle permet de récupérer la version originale, ou une version simplifiée mais sémantiquement équivalente, d'un code obfusqué. Cependant, cette tâche est intrinsèquement complexe, tant sur le plan théorique que pratique. Les travaux existants n'ont pas encore pleinement exploité les méthodes d'apprentissage automatique, et en particulier l'apprentissage par renforcement, pour aborder ce problème.

La désobfuscation repose sur plusieurs principes : la préservation sémantique, qui garantit que le code déobfusqué présente le même comportement que le code obfusqué original ; la validité structurelle, qui devrait idéalement assurer que le code déobfusqué respecte, dans la mesure du possible, les règles syntaxiques et structurelles du format de sortie ; et la simplification, où le code déobfusqué devrait idéalement être plus simple que l'original, même s'il n'existe pas de consensus universel sur ce qui constitue un code simplifié. Compte tenu de ces contraintes, la désobfuscation est considérée, dans cette thèse, comme une instance de synthèse de programmes, en particulier la synthèse de programmes en boîte blanche au moyen d'une compression symbolique. L'objectif est alors de transformer le programme obfusqué en une représentation fonctionnellement équivalente mais plus compacte.

Les limitations de la littérature existante sur la synthèse de programmes et la désobfuscation, présentées au Chapitre 8, motivent l'utilisation d'un nouveau type de graphe. Ce graphe représente le code binaire à l'aide d'une représentation intermédiaire combinant flot de données et de contrôle dans un cadre unifié, permettant ainsi théoriquement

la désobfuscation, à la fois des obfuscations de données et intra-procédurales. L'objectif est de simplifier ce graphe en appliquant des opérations d'édition de graphe jusqu'à obtenir un graphe déobfusqué. À cette fin, nous introduisons le Control-Data Graph (CDG), un graphe dirigé hétérogène dont les nœuds peuvent représenter des entrées, des sorties, des constantes, des opérations, des nœuds Phi ou des nœuds mémoire, reliés par des arêtes représentant le flot de données ou de contrôle. Ce graphe, soumis à de nombreuses contraintes structurelles, présentent certaines limitations, en particulier concernant certaines classes de fonctions, notamment celles impliquant des calculs d'adresses mémoire ou des boucles.

Ce graphe est utilisé dans le cadre d'un apprentissage par renforcement.

- L'**agent** est un réseaux de neurones de graphes, doté de plusieurs couches de sortie, qui calcule des plongements pour divers éléments du graphe, tandis que l'**environnement** correspond au graphe obfusqué à simplifier. L'utilisation d'un réseau de neurones de graphes est particulièrement avantageuse, car elle réduit la dépendance de l'espace d'action à l'étiquetage arbitraire des nœuds, en produisant des plongements équivariants et invariants.

- L'**espace d'action** est défini comme suit : à chaque étape, l'agent doit sélectionner une opération d'édition de graphe et, le cas échéant, ses paramètres. Par exemple, si l'action choisie est de supprimer un nœud, l'agent doit déterminer lequel. L'agent peut également décider de terminer le processus de simplification.

- La **fonction de récompense** évalue le CDG simplifié et renvoie un score basé sur plusieurs critères : le graphe résultant doit rester structurellement valide, démontrant que l'agent a appris à préserver les contraintes syntaxiques du CDG ; il doit être sémantiquement équivalent au graphe obfusqué original ; enfin, il doit être plus petit que le graphe initial, faute de quoi la simplification n'est pas atteinte.

Le processus global de simplification se déroule donc ainsi : à partir d'un CDG obfusqué, l'agent tente de simplifier le graphe en appliquant itérativement des opérations d'édition tout en préservant la validité structurelle et sémantique du graphe original. La validité structurelle est vérifiée via un algorithme syntaxique garantissant le respect de toutes les contraintes du CDG, tandis que l'équivalence sémantique, entre le graphe obfusqué et simplifié, doit idéalement être évaluée via une vérification par Satisfiabilité Modulo des Théories. Dans ce travail, limité pour l'heure aux expressions mixtes arithmético-booléennes, cette vérification est effectuée avec Z3.

Cette expérience est réalisée avec l'algorithme REINFORCE. Les résultats montrent que la récompense moyenne, que l'on cherche à maximiser, augmente régulièrement au fil du temps, suggérant un apprentissage initial prometteur. Cependant, elle plafonne rapidement puis chute brutalement à zéro, indiquant que l'algorithme converge vers un minimum local sous-optimal dont il ne peut s'échapper, générant de manière répétée des transformations de graphe produisant une récompense nulle. Pour pallier cette limitation et favoriser une exploration plus efficace, nous intégrons une régularisation par entropie et des mécanismes de récompense intrinsèque. Ces ajustements ont un effet notable

sur l'apprentissage : la régularisation par entropie seule n'apporte pas d'amélioration substantielle, hormis le prolongement du plateau de récompense avant sa chute, tandis que l'intégration des récompenses intrinsèques a un effet plus marqué, empêchant l'effondrement de la récompense moyenne et maintenant un niveau de performance satisfaisant au cours du temps. En conséquence, la majorité des graphes simplifiés produits respectent à la fois la validité structurelle et la correction sémantique, tout en étant plus concis que le graphe obfusqué original, sans être pour autant la version déobfusquée minimale. Ils peuvent donc être considérés comme des variantes déobfusquées valides, même s'ils ne constituent pas la forme minimale syntaxiquement et sémantiquement correcte. L'algorithme atteint ainsi un état intermédiaire satisfaisant, réalisant une désobfuscation partielle. Ces résultats sont étendus à une multitude d'expressions mixtes arithmético-booléennes, pour la même expression, et démontrent la généralisation de cette approche, qui permet d'obtenir une désobfuscation partielle d'expressions mixtes arithmético-booléennes à l'aide d'apprentissage par renforcement, basé sur les graphes.

A.9 Conclusion

Pour conclure, cette thèse a pour objectif d'exploiter et d'évaluer des méthodologies d'apprentissage de représentations dans le contexte de l'analyse de l'obfuscation. Les contributions proposées peuvent être envisagées comme un ensemble d'outils complémentaires conçus pour assister les ingénieurs en rétro-ingénierie dans leurs tâches. Elles se résument comme suit :

Un nouveau jeu de données obfusqué, diversifié et de grande ampleur. Afin de faciliter des expérimentations plus larges et plus réalistes en analyse de l'obfuscation, cette thèse introduit ObfuBench, un nouveau jeu de données spécifiquement construit à cette fin. Par son ampleur et sa diversité, ObfuBench constitue actuellement la ressource la plus complète disponible pour aborder les problématiques de recherche liées à l'obfuscation.

Détection et caractérisation de l'obfuscation. Une étude systématique et approfondie a été menée sur la classification binaire et multi-classes dans le cadre de la détection et de la caractérisation de l'obfuscation. Ce travail a examiné et comparé plusieurs dimensions du problème, incluant le format initial des graphes, le choix de descripteurs, les modèles d'apprentissage automatique, ainsi que les schémas de partitionnement des données. Les résultats expérimentaux montrent que des approches de base, telles que celles exploitant des caractéristiques dérivées des graphes ou la distribution d'assembleur, peuvent fournir des performances satisfaisantes. Cependant, elles sont systématiquement surpassées par des réseaux de neurones de graphes plus avancés, à condition que ceux-ci soient initialisés avec des caractéristiques de nœuds informatives, capturant des propriétés détaillées du code binaire, telles que le comptage des mnémoniques assembleur ou Pcode. À l'inverse, des caractéristiques plus grossières conduisent à des résultats décevants. Les réseaux de neurones de graphes présentent également de meilleures capacités de généralisation, en particulier dans des scénarios où le partitionnement des données est défavorable au modèle. En outre, ce travail met en

évidence l'impact significatif des niveaux d'optimisation du compilateur sur les performances de détection. L'ensemble des résultats est validé à la fois sur ObfuBench, sur d'autres jeux de données de la littérature, et sur des échantillons obfusqués issus du monde réel.

Étude expérimentale du binary diffing et de la similarité dans des contextes standards et obfusqués. Le diffing binaire, dans un contexte obfusqué, a été négligé dans les recherches antérieures. Pour combler cette lacune, une étude expérimentale approfondie a été réalisée, couvrant à la fois des scénarios non obfusqués et obfusqués. Nous avons d'abord conduit une étude d'ablation de QBinDiff, permettant d'obtenir des recommandations pratiques pour le réglage de ses paramètres en fonction des objectifs d'analyse. Nous avons ensuite mené une évaluation comparative des approches existantes en diffing et similarité binaires, sur des binaires non obfusqués, révélant la supériorité de certains outils, notamment JTrans, BinDiff et QBinDiff. Ces tendances se maintiennent dans un contexte obfusqué, que l'un ou les deux binaires à comparer soient obfusqués. En particulier, nous montrons qu'une version raffinée de QBinDiff, exploitant des caractéristiques robustes face à certains types d'obfuscation, permet d'obtenir des résultats nettement supérieurs à la configuration standard. Ces travaux révèlent également que la plupart des obfuscations de type data et intra-procédurales offrent une résistance limitée aux attaques par diffing, indépendamment de l'intensité de l'obfuscation. En revanche, les obfuscations inter-procedural entravent fortement l'efficacité des outils de diffing et de similarité, soutenant ainsi leur usage dans les contextes de protection logicielle, en particulier lorsque des obfuscateurs efficaces, tels que Tigress, sont utilisés.

Désobfuscation fondée sur les graphes avec RL. Cette recherche exploratoire cherche à étudier le potentiel des représentations de graphes pour les tâches de désobfuscation. Plus précisément, nous avons adapté des algorithmes de RL à des données structurées, telles que les graphes, dans le but de simplifier des graphes obfusqués via des opérations d'édition, afin d'en retrouver des versions désobfusquées. Bien que des contraintes de temps aient empêché l'achèvement de l'ensemble des expériences prévues, plusieurs résultats prometteurs ont été obtenus. Notamment, une désobfuscation partielle d'obfuscations de type MBA a été réalisée, à l'aide de l'algorithme REINFORCE et d'un agent basé sur des réseaux de neurones de graphes, en particulier lorsqu'une fonction de récompense, incluant un aspect intrinsèque, est employé.

RÉSUMÉ

La protection du contenu sensible d'un programme constitue un enjeu majeur, tant dans des contextes légitimes que malveillants, l'obfuscation étant l'une des techniques les plus couramment employées à cette fin. En transformant le code binaire de programmes compilés, l'obfuscation complique et retarde les opérations de rétro-ingénierie, contraignant ainsi les attaquants à suivre une série d'étapes analytiques spécifiques pour contourner ces protections. Cette thèse présente plusieurs contributions, toutes fondées sur des méthodes avancées d'apprentissage de représentations de graphes, et couvrant différentes étapes du processus d'analyse d'obfuscation. Dans un premier temps, nous abordons la détection et la caractérisation de l'obfuscation sous forme de problèmes de classification binaire et multi-classes, en comparant divers formats de graphes, des modèles d'apprentissage de référence, ainsi que des ensembles de caractéristiques, tout en évaluant l'impact de la distribution des données et des optimisations du compilateur. Nos résultats montrent que les modèles avancés de réseaux de neurones de graphes surpassent les approches de référence lorsqu'ils sont alimentés par des caractéristiques capturant finement la sémantique du code binaire, bien que leurs performances demeurent sensibles au niveau d'optimisation et aux stratégies de partitionnement des données. Dans un second temps, nous menons une étude approfondie du diffing binaire et des techniques de similarité, en partant d'un contexte standard et non obfusqué, au travers d'une analyse d'ablation de QBinDiff et d'une comparaison approfondie des techniques existantes, jusqu'à des scénarios obfusqués, dans lesquels un attaquant cherche à extraire des informations du programme protégé, sans procéder à une désobfuscation complète. Les résultats mettent en évidence les avantages de QBinDiff qui permet un diffing plus fin et montrent que la plupart des obfuscations intra-procédurales et de données offrent une protection limitée, contrairement aux obfuscations inter-procédurales, plus robustes mais rarement mises en œuvre dans les outils open source. Enfin, nous explorons la désobfuscation directe, en combinant l'apprentissage par renforcement et l'apprentissage de représentations de graphes. Nous modélisons cette tâche comme un problème de compression symbolique, où une fonction obfusquée est représentée dans un format de graphe innovant, combinant les flots de contrôle et de données, puis simplifiée de manière itérative via des opérations d'édition de graphe, guidées par un agent basé sur des réseaux de neurones de graphes. Les premiers résultats obtenus sur les expressions mixtes arithmético-booléennes montrent un succès partiel mais prometteur, ouvrant des perspectives intéressantes pour de futurs travaux.

MOTS CLÉS

Graphes, Apprentissage Automatique, Reverse engineering, Obfuscation, Analyse de programmes, Réseaux de neurones de graphes.

ABSTRACT

Protecting sensitive program content is a critical concern in both legitimate and malicious contexts, with obfuscation being one of the most widely used techniques. By transforming the binary code of compiled programs, obfuscation hinders and delays reverse engineering, forcing attackers to follow specific analytical steps to bypass such protections. This thesis presents several contributions, all grounded in advanced graph representation learning, covering different stages of the obfuscation analysis process. First, we address obfuscation detection and characterization as binary and multi-class classification problems, comparing graph formats, baseline and advanced deep learning models, and enriched feature sets, while assessing the impact of data distribution and compiler optimizations. Our results show that advanced Graph Neural Networks models outperform baselines when provided with features capturing fine-grained code semantics, though performance remains sensitive to optimization levels and dataset partitioning. Second, we conduct an extensive study of binary diffing and similarity, from the standard unobfuscated case, through an ablation study of QBinDiff and a comparison of existing tools, to obfuscated scenarios where attackers aim to extract knowledge from an obfuscated program, without full deobfuscation. Results highlight QBinDiff's advantages for fine-grained diffing and reveal that most intra-procedural and data-flow obfuscations offer limited protection compared to the stronger, yet rarely used, inter-procedural obfuscations. Finally, we explore direct deobfuscation by integrating reinforcement learning and graph representation learning. We model the task as a white-box symbolic compression problem, representing obfuscated functions in a novel graph format, combining data and control flow, and simplifying them iteratively through graph edit operations guided by a Graph Neural Networks-based agent. Preliminary results on Mixed Boolean Arithmetic obfuscation demonstrate partial but promising success, opening avenues for further research.

KEYWORDS

Graphs, Machine Learning, Reverse engineering, Obfuscation, Program analysis, Graph Neural Networks.