Data-driven malware classification assisted by machine learning methods

Cassius Puodzius

▶ **To cite this version:**

Cassius Puodzius. Data-driven malware classification assisted by machine learning methods. Other [cs.OH]. Université de Rennes, 2022. English. NNT : 2022REN1S090 . tel-04072990v4

HAL Id: tel-04072990

https://theses.hal.science/tel-04072990v4

Submitted on 18 Apr 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES 1

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *INFO*

Par
Cassius PUODZIUS

Data-Driven Malware Classification Assisted by Machine Learning Methods

Thèse présentée et soutenue à Rennes, le 19 décembre 2022
Unité de recherche : INRIA

Rapporteurs avant soutenance :

Jacques KLEIN PR Université du Luxembourg
Amedeo NAPOLI DR Emérite (HDR) CNRS, LORIA

Composition du Jury :

Président :	Nathalie BERTRAND	PR (HDR) Université Rennes 1
Examinateurs :	Eric ALATA	MCF INSA Toulouse
	Olivier BARAIS	PR Université Rennes 1
	Jacques KLEIN	PR Université du Luxembourg
	Amedeo NAPOLI	DR Emérite (HDR) CNRS, LORIA
	Olivier ZENDRA	CR INRIA Rennes
Dir. de thèse :	Ludovic ME	DR INRIA Rennes

ACKNOWLEDGEMENT

I express my heartfelt gratitude to my advisors, *Olivier Zendra* and *Annelie Heuser*, for their unrelenting support, guidance, and encouragement throughout my thesis. Their patience, knowledge, and wisdom have been indispensable in guiding me through the research process. Their insights, feedback, and suggestions have been essential in shaping my research and this thesis. I am thankful for the privilege of working with such dedicated advisors.

I extend my gratitude to my thesis supervisor, *Ludovic Mé*, for their support throughout my thesis. I am truly grateful for the chance to work with such an accomplished researcher who was always available to offer valuable assistance whenever needed.

I also extend my thanks to the members of my thesis committee, *Nicolas Rougier* and *Veronica Valeros*, for their insightful feedback and suggestions. I am grateful for their time, expertise, and invaluable contributions to the process.

I acknowledge the support and resources provided by INRIA, through the TAMIS and DiverSE teams, which made this research possible. I am thankful for the opportunities and facilities provided by the institution that helped me conduct my research and complete this thesis.

Over the almost five years that I have been in INRIA I had the privilege of meeting and working with many talented colleagues. I extend my gratitude to my numerous colleagues for their support and encouragement throughout my thesis journey. Their camaraderie and professional insights have been a source of motivation and inspiration. I am grateful for the positive work environment and the opportunity to learn from such brilliant individuals.

In terms of my personal life, this thesis engendered major changes. Venturing to France and starting a new life in Rennes was a formidable challenge. Therefore, I want to express my greatest gratitude to my family and friends (old ones and new ones), who have supported me throughout this long and adventurous journey. Their encouragement and support helped me to stay motivated and focused during the challenging times. I am truly grateful for their love and support.

Special thanks to my dear sweetheart, *Angèle Chevillard*, for her unwavering support and tireless efforts in overcoming the challenges we faced together, making the fulfillment of

thesis possible. I extend my gratitude to my in-laws, who provided me with immense support and comfort throughout my thesis journey. Their love and encouragement have been a source of strength, and I am grateful for the bond we have formed.

Deepest gratitude to my beloved friends who have been a part of this journey. Their encouragement, companionship, and support have been integral to this experience and have made it truly memorable. To mention but a few: *Jean Carlos Padilha*, *Gustavo Santos Guimarães* (our *Guga*), *Lucas Souza*, my dear friends of the "Petit Comité", Collectif Caxangá, Toucouleurs. There are no words to express my gratitude for everything we have been through together and I am deeply thankful for it.

Last but not least, my family. My deepest gratitude to my family, who have been my source of support and encouragement throughout this journey. Despite the challenges posed by the COVID-19 pandemic, despite time and distance, I always felt their presence. I am truly thankful for their encouragement and for always being there for me, through thick and thin. Their constant love and belief in me have provided me with the strength to face the challenges and pursue my dreams. I am really grateful to have such a supportive family.

My utmost gratitude goes to my parents, *Vera Lopes de Oliveira* and *Douglas Napoleão Puodzius*, for their steadfast love and caring throughout this journey. Despite the challenges we faced, their belief in me has pushed me to reach this far and I couldn't be more grateful. I am proud to call you my parents and I will be forever grateful for the role you have played in shaping who I am today. Thank you for always being there for me and for always believing in me.

Finally, I would like to dedicate this thesis to the memory of my dear uncle, *Jairo Luis Soares Tenório*, who was a constant source of inspiration. His untimely passing during the course of my studies was a great loss, but his spirit remains with me, pushing me forward. He was devoted to helping me from the very beginning of this journey and his shining example served as a guiding light throughout. I am forever grateful for his guidance and inspiration, and this achievement would not have been possible without him. I hope that in some way, this dedication will serve as a testament to the impact he had on my life and to the legacy he left behind.

RÉSUMÉ ÉTENDU

Les logiciels malveillants, ou malwares, sont un problème majeur dans le domaine de la sécurité informatique. Leur capacité à se dissimuler et à éviter la détection les rend extrêmement dangereux pour les entreprises, les gouvernements et les particuliers. Historiquement, l'analyse des malwares était une tâche manuelle, nécessitant une expertise humaine pour identifier ses caractéristiques et pour les classer en fonction de leurs comportements. Cependant, cette approche est coûteuse en ressources et chronophage, ce qui ne permet pas de faire face aux scénarios modernes de cybermenaces.

L'automatisation de l'analyse des malwares est donc devenue un enjeu majeur dans le domaine de la sécurité informatique. L'objectif de cette thèse est d'améliorer l'automatisation de l'analyse des malwares en adoptant une approche dirigée par les données et en évitant autant que possible l'intervention humaine. Pour cela, nous nous concentrons sur la classification des malwares, qui permet d'optimiser le traitement de grands corpus de malwares en identifiant les ressemblances entre des instances similaires.

Afin de travailler sur les thématiques principales, cœur de notre recherche, nous avons établi l'axiologie de notre étude. Cette étape préliminaire nous a permis de poser les bases de notre recherche en définissant les fondements qui ont guidé notre démarche scientifique tout au long de cette thèse.

En résumé, dans ce chapitre, nous avons discuté des aspects ontologiques et épistémologiques de l'analyse des malwares. Nous avons adopté une approche en deux volets pour établir une ontologie de malware : une approche historique et une approche analytique. L'approche historique consiste à examiner l'histoire de l'émergence des concepts dans le domaine de l'analyse des malwares, tandis que l'approche analytique se concentre sur un examen conceptuel de ce dernier. Nous avons également discuté de l'épistémologie de la recherche en malware. Pour cela, nous avons souligné que la recherche en malware est principalement *fondationnaliste* et nous avons identifié certaines limites à cette approche. Nous avons également reconnu que la sécurité informatique est une activité socialement enracinée, qui ne peut être complètement exemptée d'intervention humaine. Or, par ailleurs, nous pouvons nous appuyer sur l'apprentissage machine pour largement alléger et optimiser l'effort d'analyse.

Dans l'ensemble, notre discussion ontologique et épistémologique nous a permis de formuler un paradigme de recherche pour la suite de notre étude de classification des malwares. Ensuite, nous avons structuré notre travail en trois étapes principales, qui ont permis d'approfondir et de compléter les objectifs que nous nous étions fixés.

Dans un premier temps, nous avons étudié de nouvelles façons d'exploiter l'exécution symbolique dans l'analyse des malwares, en accordant une attention particulière à l'utilisation optimale des tactiques des solveurs SMT et aux réglages des hyperparamètres. La méthode d'exécution symbolique est une alternative aux méthodes d'analyse plus courantes, telles que l'analyse statique et l'analyse dynamique. En effet, l'analyse dynamique présente une importante limitation vis-à-vis de la couverture du code, tandis que l'analyse statique a une vue complète du code binaire, mais ne peut pas l'exécuter. Quant à l'analyse symbolique, elle permet d'émuler l'exécution du code binaire tout en gardant un contrôle précis sur tous les éléments de l'analyse, ce qui peut résoudre ces limitations. Dans ce contexte, nous avons évalué l'exécution symbolique, ainsi qu'amélioré son utilisation dans le cadre de l'analyse des malwares. Nous nous sommes intéressés en particulier à la détermination précise des appels effectués lors de l'exécution, y compris les appels qui peuvent être initialement obscurcis dans le code source. Nous avons évalué différentes techniques pour guider et optimiser l'analyse symbolique, en mesurant leur impact sur la classification des malwares. Les résultats obtenus montrent que l'exécution symbolique peut être utile en pratique pour l'analyse des malwares; de plus, l'optimisation des paramètres de l'analyseur peut largement améliorer sa performance.

Dans un deuxième temps, nous nous sommes concentrés sur la représentation du comportement des logiciels, en accordant une attention particulière à sa précision et à sa robustesse. Nous proposons ainsi une méthode d'analyse basée sur les graphes de dépendance des appels externes (ECDG) pour la représentation du comportement, ainsi qu'une fonction de similarité (σ^{ECDG}) pour leur comparaison. Cette représentation est compacte, pour autant elle n'engendre pas de perte d'information; ainsi elle permet d'accélérer son traitement par des algorithmes d'analyse des graphes. Cette étude propose également une méthodologie d'évaluation avec le paradigme *"Accuracy and Robustness (AnR)"*, utilisé pour l'analyse des résultats obtenus lors de nos expérimentations. Ces résultats ont montré que la méthode proposée est efficace et a un potentiel important pour la classification et la détection des malwares. Les principales contributions incluent donc la proposition du paradigme AnR pour une méthodologie d'évaluation plus fiable, l'introduction d'ECDGs pour une représentation plus compacte des graphes d'appels, et la définition de la fonction de similarité (σ^{ECDG}),

avérée efficace, précise et robuste pour comparer des ECDGs.

Enfin, dans un troisième temps, nous nous sommes focalisés sur le partitionnement des logiciels, en concevant une méthode qui ne restreint pas la combinaison des caractéristiques syntaxiques et comportementales. Elle peut permettre également une montée en charge conséquente des analyses en pratique. Pour cela, nous avons élaboré une nouvelle stratégie de traitement basée sur des techniques de partitionnement d'ensemble, combinées avec la représentation vectorielle des données extraites des logiciels.

Les méthodes traditionnelles basées sur des signatures sont sujettes à des techniques anti-analyse efficaces pour échapper à la détection. Pour pallier à cette insuffisance, il est nécessaire d'améliorer l'efficacité de l'apprentissage automatique dans l'analyse des logiciels malveillants. Pour ce faire, il faut résoudre plusieurs défis majeurs, notamment la sélection des caractéristiques, leur représentation et l'exactitude des algorithmes d'apprentissage automatique. Dans ce contexte, l'approche de sélection des caractéristiques hybrides basée sur l'analyse dynamique et statique est prometteuse. Elle diversifie les outils et les techniques utilisés pour extraire les caractéristiques, tout en fournissant plusieurs vues du jeu de données qui peuvent être interprétées comme des partitions imparfaites convergentes vers une vraie partition, une fois combinées.

En résumé, cette troisième partie décrit la conception d'un cadre de classification des malwares appelé EMB-DUET. Ce cadre est capable de traiter des caractéristiques hétérogènes, ce qui permet d'améliorer les performances globales. Pour améliorer l'efficacité de l'apprentissage automatique, le cadre utilise la technique de plongement numérique (*numerical embedding*) pour transformer les vues non structurées en vecteurs numériques, pouvant être efficacement analysés par différents algorithmes. Les méthodes de clustering par ensemble (*ensemble clustering*) sont utilisées pour consolider toutes les vues individuelles en un seul clustering. Notre étude décrit également l'impact et les effets de différents hyperparamètres impliqués dans EMB-DUET et la vectorisation des ECDGs basée sur la technique de régression SVM.

Les contributions principales de cette partie sont donc la proposition d'EMB-DUET comme cadre de classification de données, ainsi que la vérification de l'exactitude et de la robustesse offertes par l'agrégation des différentes vues de données (à savoir, les strings, les n-grammes et les ECDGs). EMB-DUET s'avère capable de traiter des caractéristiques hétérogènes en s'appuyant sur le plongement numérique pour améliorer les performances globales. En outre, cette étude fournit des directives pour le réglage approprié des hyperparamètres impliqués dans EMB-DUET, ainsi que pour l'apprentissage de la représentation vectorielle des

ECDGs avec la régression SVM.

En résumé, nos principales contributions sont les suivantes :

— Élaboration d'une ontologie réaliste pour la caractérisation des instances de programmes malveillants.

— Discussion des implications épistémologiques et des limites de l'utilisation de l'apprentissage automatique dans la classification des malwares.

— Analyse de l'effet des différents paramètres heuristiques sur les solveurs SMT dans l'utilisation de l'exécution symbolique pour tracer les appels de programmes pour la construction des ECDGs.

— Étude de l'efficacité, exactitude et robustesse des ECDGs en tant que représentation singulière (*birthmark*) pour la structure des programmes informatiques.

— Proposition d'EMB-DUET en tant que cadre polyvalent de classification des malwares, utilisant des techniques de plongement numérique pour le traitement efficace de grands corpus de données.

En conclusion, la thèse étudie la problématique de la classification des malwares et s'attaque au scénario moderne des malwares multiples et hautement complexes en s'appuyant sur des méthodes d'apprentissage automatique. La thèse examine également les questions ontologiques et épistémologiques liées à la recherche sur les malwares et à l'apprentissage automatique. Notre étude est donc menée en trois étapes qui suivent l'enchainement basique des étapes de l'apprentissage automatique: i) [*extraction des données*] l'amélioration des outils pour l'analyse des malwares, ii) [*transformation des données*] l'étude de la représentation comportementale des codes binaires et iii) [*analyse des données*] la proposition d'un framework polyvalent appelé EMB-DUET.

Les résultats de nos expériences ont montré que notre méthode de classification de logiciels malveillants était efficace pour gérer des ensembles de données complexes et multifacettes. Les contributions de cette thèse ont ainsi permis de proposer une approche innovante et efficace pour la classification des malwares dans un contexte de haute complexité.

TABLE OF CONTENTS

Introduction **15**

Context & Motivation . . . 15
Problem Statement . . . 19
Research Objectives . . . 22
Research Questions . . . 22
Contributions . . . 23
Thesis Outline . . . 23

1 Epistemological View on Malware Research **25**

1.1 Introduction . . . 25
1.2 An Ontology of Malware . . . 26
1.2.1 Historical Approach . . . 27
1.2.2 Analytical Approach . . . 34
1.3 An Epistemology of Malware Research . . . 44
1.3.1 Background . . . 45
1.3.2 Epistemological Matters of Cybersecurity . . . 50
1.4 An Axiology . . . 54
1.5 Conclusion . . . 56

2 General Background **57**

2.1 Introduction . . . 57
2.2 Machine Learning . . . 57
2.2.1 Definitions and Terminology . . . 58
2.2.2 Supervised Learning . . . 64
2.2.3 Unsupervised Learning . . . 74
2.3 Malware Analysis . . . 87
2.3.1 Binary Analysis . . . 88
2.3.2 Malware Analysis Primitives . . . 98
2.4 Discussion . . . 106

2.5 Conclusion . . . 107

3 Call Tracing with Symbolic Execution 111
3.1 Introduction . . . 111
3.1.1 Research Questions . . . 112
3.1.2 Chapter Outline . . . 113
3.2 Background . . . 113
3.2.1 Symbolic Execution . . . 113
3.2.2 Frequent Subgraph Mining (FSM) . . . 122
3.3 Related Works . . . 123
3.4 Methodology . . . 127
3.4.1 SMT Optimization . . . 129
3.4.2 Enhanced Symbolic Execution . . . 130
3.4.3 Impact on Malware Classification . . . 132
3.5 Experiments . . . 135
3.5.1 Implementation Setup . . . 136
3.5.2 Optimizing z3 for symbolic execution of binary files . . . 139
3.5.3 Overall Toolchain Optimization . . . 143
3.5.4 Impact of graph and execution heuristics . . . 146
3.6 Discussion . . . 147
3.6.1 Threats to Validity . . . 150
3.7 Conclusion . . . 151

4 Structural-Based Binary Code Similarity 153
4.1 Introduction . . . 153
4.1.1 Research Questions . . . 154
4.1.2 Chapter Outline . . . 154
4.2 Background . . . 155
4.2.1 Notations and Definitions . . . 155
4.2.2 Binary Code Similarity . . . 155
4.2.3 Hybrid Clustering . . . 157
4.3 Related Works . . . 159
4.3.1 Call Graphs for Binary Code Similarity . . . 159
4.4 Methodology . . . 163
4.4.1 External Calls Dependency Graph (ECDG) . . . 163

4.4.2 Similarity Function (σ^{ECDG}) . . . 164
4.4.3 Accuracy-and-Robustness (AnR) Paradigm . . . 165
4.5 Experiments . . . 168
4.5.1 Dataset . . . 168
4.5.2 Benchmark: Framework . . . 169
4.5.3 Benchmark: σ^{ECDG} vs. *radiff2* . . . 173
4.5.4 Parametrization: NEF Selection . . . 175
4.5.5 Accuracy-and-Robustness (AnR) Analysis . . . 176
4.5.6 Prototype Analysis . . . 180
4.6 Discussion . . . 182
4.6.1 Threats to Validity . . . 185
4.7 Conclusion . . . 186

5 EMB-DUET: Multi-Feature Clustering Based on Numerical Embedding 189
5.1 Introduction . . . 189
5.1.1 Research Questions . . . 190
5.1.2 Chapter Outline . . . 191
5.2 Background . . . 191
5.2.1 Support-Vector Machines (SVMs) . . . 191
5.2.2 Embedding Learning . . . 195
5.2.3 Clustering Ensembles . . . 201
5.3 Related Works . . . 204
5.3.1 Hybrid Features for Malware Analysis . . . 204
5.3.2 Numerical Embedding for Malware Analysis . . . 207
5.4 Methodology . . . 208
5.4.1 EMB-DUET . . . 208
5.4.2 σ^{ECDG} -learning with SVM Regression . . . 210
5.5 Experiments . . . 212
5.5.1 Implementation Setup . . . 212
5.5.2 Dataset . . . 216
5.5.3 σ^{ECDG} -Learning with SVM Regression . . . 218
5.5.4 EMB-DUET . . . 227
5.6 Discussion . . . 252
5.6.1 Threats to Validity . . . 255
5.7 Conclusion . . . 256

Conclusion **259**
5.7.1 Future Work 262

Bibliography **269**

6 Appendix **301**
6.1 Basic Formal Ontology (BFO) 301
6.2 Logistic Equation and the Sigmoid Function 303
6.3 Density-Based Clustering Methods 307
6.4 Mutual Information 308
6.5 PE Headers 309
6.6 Call Tracing with Symbolic Execution 310
6.6.1 Supplementary analysis 310
6.6.2 Evaluation Dataset 331
6.7 Structural-Based Binary Code Similarity 334
6.7.1 NEF Selection: Detailed Analysis 334
6.8 EMB-DUET: Multi-Feature Clustering Based on Numerical Embedding 344
6.8.1 SVM 344
6.8.2 EMB-DUET 345

INTRODUCTION

Context & Motivation

The eminence of computer systems in our society is patent. The presence of information technology in our society is ubiquitous and in the course of the last decades it has revolutionized the ways in which individuals communicate, learn, work, do shopping, travel, and so forth.

The dawn of this technological revolution can be traced back to the works on computability theory started by Alan Turing in the 1930's, passing through the construction of the first general-purposed machines in the 1940's, the creation of the first programming languages in the 1950's and the prototyping of the modern computer in the 1960's. As of the 1970's —with the invention of *floppy disks* and the *Ethernet*, the release of the first video-game (Atari) and the foundation of technology companies, like Microsoft and Apple, that popularized the use of computer technology —information technology ceases being a subject circumscribed to academic and military research to start gaining a foothold in society.

The 1980's marks the advent of personal computers; the 1990's the creation of the world wide web and the first uses of website in everyday life (like Google Search launched in 1998); the 2000's the popularization of the Internet with the expansion of social medias; and the 2010's the progress of cloud computing and the widespread use of smartphone and other connected devices. The bottom line is that we have been witnessing progressive waves of a digital revolution, which is becoming increasingly accelerated and pervasive in society over time.

The International Telecommunication Union (ITU) [1] estimates that the number of individuals who have used the Internet attained 4.9 billion people in 2021 [132], which corresponds to 62.5% of the world population. According to the World Economic Forum's (WEF) 2020 Global Risk Report [305], there are one million additional people joining the Internet every day. Due to COVID, in the last few years the increase of internet users soared —which is referred as the "COVID connectivity boost". ITU estimates that the COVID pandemic brought an increment of 782 million Internet users in a period of just two years, which accounts for

1. The United Nations specialized agency for information and communication technologies.

the largest annual connectivity growth rates ever recorded.

The report of the United Nations (UN) Secretary-General's High-level Panel on Digital Cooperation [210] recommends that by 2030, "every adult should have affordable access to digital networks, as well as digitally-enabled financial and health services", as digital technologies are deemed imperative to fulfill the Sustainable Development Goals (SDGs) endorsed by 193 UN member states in 2015. In Europe, the European Comission (EC) established as connectivity objective that for all European households, rural or urban, they must have access to an internet connectivity of at least 100 Mbps, upgradable to Gigabit speed [59].

On top of that, new technologies such as the 5G network and the Internet of Things (IoT) [2] usher in yet another technological disruption.

Cisco estimates [51] that the number of connected devices will grow from 18.4B in 2018 to 29.3B in 2023, which corresponds to 3.6 devices per capita in 2023. The profile of these devices is also evolving: in 2018, only 33% (6.1B) of all networked devices were IoT devices [3], while 27% (4.9B) were smartphones and 12% (2.2B) were PCs and Tablets; by 2023, it is estimated that 50% (14.7B) of the networked devices will be IoT devices, while 23% (6.7B) will be smartphones and only 7% (2.1B) will be PCs and Tablets. As for the profile of the network connection, Cisco estimates that by 2023, 45% of all networked devices will be connected through mobile network, whereas the remainder will be connected through wires or Wi-Fi; 5G will account for 10.6% of total mobile connections by 2023, compared to 0% in 2018.

These new technologies are bringing more convergence of the digital and physical worlds, as IoT devices have been integrated into everyday objects (e.g. clothing, manufactured products, foodstuffs, medicines, etc). McKinsey Global Institute estimates that by 2030, "IoT could enable $5.5 trillion to $12.6 trillion in value globally, including the value captured by consumers and customers of IoT products and services" [49]; from which, the largest amount of potential economic is in the factory setting ($\sim$ 26%) —which includes standardized production environments in manufacturing, hospitals, and other areas —and the second largest is in the human-health setting ($\sim 10\% - 14\%$).

The projected scenario is a world in which computer technology is ever more integrated in everyday life, ever more ubiquitous, and performing critical tasks that have been traditionally accomplished offline. Therefore, "global security and stability are increasingly dependent on digital security" [210].

Near the end of the first decade of the years 2000, global leaders in the public and pri-

2. Roughly, the IoT consists of sensors and actuators that are connected by networks to computing systems.
3. Cisco's report refers to *Machine-to-Machine (M2M) modules*.

vate sectors engaged in a broad range of initiatives and directives to secure the cyberspace. The WEF's 2010 Global Risk Report [299] included "critical information systems and cyber-vulnerability" in the section of *risks to keep on the radar*, as they were still perceived by experts as relatively low in terms of likelihood and severity, and among the least interconnected risks. Yet, the WEF 2010 report called leaders to give greater attention to cybersecurity foreseeing that "the increasing complexity and rapid development of dynamic systems and networks, the sophistication of changing threats and the presence of intrinsic vulnerabilities present demanding challenges to the information society".

Notwithstanding the efforts to secure the cyberspace, numerous cyber incidents loomed large in the course of the last decade. Some of the major security breaches include the PlayStation Network outage in 2011 [50], in which the personal data of 77M accounts were compromised and the services of PlayStation 3 and PlayStation Portable were temporarily down; the Yahoo! data breach [106] in 2012, which leaked the credential of almost half million user accounts; the Target [62] and Home Depot [164] security breaches in 2013 and 2014 —where around 40M and 56M credit card data were respectively stolen —that were widely reported in the press and had heavy consequences in their business; the Sony Pictures hack, in which the attackers released confidential data from the studio containing information about employees and their families, emails between employees, information about executive salaries at the company, copies of then-unreleased Sony films, plans for future Sony films, scripts for certain films, and more; the theft of $81 million from the Bangladesh Central Bank in 2016 by the manipulation of the SWIFT global payments network [324].

Another facet of cyber security which is also on the rise is *cyberwarfare*. Sophisticated cyber attacks are prepared and perpetrated by nation-sponsored actors for different purpose, such as spying and sabotage.

The first time that "a known cyber attack had coincided with a shooting war" was during the in Russo-Georgian War in 2008, when a unremitting campaign of distributed denial-of-service attacks (DDoS) and defacement attacks targeted numerous South Ossetian, Georgian, Russian and Azerbaijani organisation [178]. Amidst the conflicts between Russia and Ukraine, major cyber incidents were also recorded, as the cyber attack against the Ukrainian power grid on 23 December, 2015, which resulted in power outages for roughly 230K consumers [88]; or the recent attacks against Ukranian government sites in the prelude of the 2022 Russian invasion of Ukraine [219].

The first publicly known cyberweapon to target industrial control systems (a.k.a. SCADA systems) was Stuxnet. It was uncovered in 2010, targeting the Iran's nuclear programme,

including the uranium enrichment centrifuges at the Natanz facility [87]. It was a highly sophisticated piece of software, reported to be developed under an American-Israeli cooperation [249]. Stuxnet was a real eye-opener in terms of the potential peril that cyberthreats can pose to the real world.

Equally highly sophisticated cyberweapons have already fallen into the wrong hands, as consequence of operational mistakes made by national agencies. WannaCry, which broke out in May 2017, used cyberweapons developed by the US National Security Agency (NSA) that were stolen by a hacker crew called Shadow Brokers [35]; it led to a cyberattack with unprecedented scale, which impacted over 10K organizations and 200K individuals in over 150 countries [168]. Other cyberattacks like NoPetya, which caused disruption around the world and infected companies in 64 countries, were also made possible by the leak of state-sponsored cyberweapons [297].

The safeguard of political elections is another consequential domain that has been targeted by cyberthreats. An incipient episode was the DDoS atttacks ahead of the 2010 Myanmar general election against Burma's main Internet provider, the Ministry of Post and Telecommunication. These attacks succeeded in flooding the network links, which teared down the Internet of the country and restricted the flow of information over the election period [36].

During the 2008 US presidential elections, the campaigns of Barack Obama and John McCain were hacked by allegedly Chinese units. On May 29, 2009, the US President Obama referred to the attacks at an event to announce the plans for securing US digital future: "Hackers gained access to emails and a range of campaign files, from policy position papers to travel plans". Officials and former campaign officials acknowledged that the security breach was far more serious than has been publicly known, involving the potential compromise of a large number of internal files [130].

The 2016 US presidential elections were also target of cyberattacks by foreign actors. According to Intelligence Community Assessment (ICA) [127], Russians were directly ordered by the Russian President Vladimir Putin to perpetrate the attack. However, according to CIA this time the goal was to actively intervene in the election, rather than just undermine confidence in the US electoral system [82]. Thousands of hacked emails from the Democratic National Committee (DNC) and others, including Hillary Clinton's campaign chairman, were leaked through WikiLeaks. According to ICA, Russians also collected information from Republic-affiliated targets, but they did not conduct an equivalent disclosure campaign.

The 2017 French presidential elections were also target of a similar cyberattack. Two days

before the scheduled vote, just hours before the election media blackout[4], more than 20K e-mails related to the campaign of Emmanuel Macron were disclosed on WikiLeaks. This e-mail leak spread swiftly under the hashtag *#MacronLeaks* on Twitter and Facebook.

As illustrated by the aforementioned episodes, cyberthreats have been expanding, while becoming more targeted and sophisticated. In the view of this evolution, *cyberattacks* (and alike, e.g. "*data fraud or theft*") were listed in the *Top 5 Risks by Likelihood* seven times in five WEF's Global Reports from 2012 to 2019 [300–304] —both *cyberattacks* and *data fraud or theft* were in the top 5 risks in 2018 and 2019 [306, 307]. In the last two editions of report, the methodology has evolved to include the *Global Risks Horizons*, which considers the risks according to different time frames: in both cases *Cybersecurity Failure* is the top technological short-risk ($0-2$ years), being 4th in 2021 and 7th in 2022 when all risks are considered.

Cybersecurity is the leading technological issue at the center of concerns for a safer world, which has even become a new *domain of war*. In May 2011, the US International Strategy for Cyberspace [122] was released, being announced as the first time that the US has laid out an approach that unifies their engagement with international partners on the **full range of cyber issues**. As deterrence measure, it was declared in this document that "when warranted, the United States will respond to hostile acts in cyberspace as we would to any other threat to our country". In July 2011, the US Department of Defense (DoD) declared the cyberspace as a new operational domain in addition to land, sea, air and space [70]. In the same vein, in 2014 NATO endorsed an Enhanced Cyber Defence Policy, which established that cybersecurity became part of the Alliance's core task of collective defense [201]; in 2016, NATO recognized cyberspace as a new operational domain [202].

Within this context, this thesis addresses the problem of *malware classification*, which is an important component of a complete strategy for the security of digital systems.

Problem Statement

In this thesis we tackle the issue of cybersecurity via Data-Driven Malware Classification Assisted by Machine Learning Methods.

Malware is a prevalent cyberthreat that put the digital security in jeopardy worldwide. In 2019 the number of known malware exceeded the mark of one billion instances, by May 2022 this number already surpasses 1.35B instances, and over 450K new malware instances are

4. Period of time when candidates and the media are prevented from commenting on campaign activities.

discovered every day [280]. Moreover, the complexity of malware also evolves over time with new obfuscation techniques and new attack methods. This combination of factors poses new challenges for the protection of computer systems against malware.

Historically, malware analysis has heavily resorted to human action to manually create signatures that have been used to detect and classify malware. This tradition comes from the dawn of the anti-malware industry at the end of the 1980's, when the computational power available to fight malware was much more limited than today, but the number of samples to process was just as small —in 2008, Dr. Jan Hruska (co-founder of Sophos) refers to "tens of viruses per month" in the 1980's [123].

Initially, the objective of malware classification was closely connected to malware identification, which was used in the lineage of malware families. However, with the emergence of techniques such as polymorphism, which produces many different-looking versions of the same software components, the number of new potential malware instances to analyze every day skyrocketed —in 2015, Microsoft refers to "tens of millions of daily data points to be analyzed as potential malware" [187]. For this reason, malware classification gained a key role in the workflow of anti-malware systems, being principally used to break down large amounts of data into subgroups of manageable sizes, thus optimizing the use of human expertise in the overall analysis.

There has been a remarkable evolution in the workflows of signature creation over the last decades: Anti-malware companies and the practitioner community have developed many (proprietary and public domain) languages to support the creation of signatures; analysis frameworks have integrated pipelines to bolster cooperation between human analysts and machines, using the prior constituted knowledge database in the analysis of new coming malware and to implement large scale QA procedures to validate newly created signatures; security products (with virtually no exception) have integrated the support to share and fetch new signatures; the community has developed (open and closed) platforms to support large scale sharing and reuse of signatures.

Yet, the procedure to create and test new signatures is inherently very costly and time consuming; signatures are barely able to keep the pace with the current state of affairs and they are certainly unfit to cope with future cyber threat scenarii. Due to shortcomings of the signature-based strategy, over the last decade there has been a great effort to find viable alternatives.

The solution to this plight relies not only on widely automating malware analysis, but also on making the end-to-end classification analysis as autonomous as possible. Toward

this goal, *machine learning*-based approaches stand out as a promising alternative, as they have been increasingly used to analyze large and heterogeneous corpora of data, like those in the malware analysis scenario. Therefore, in our work we consider a malware classification framework that relies on machine learning to optimize the handling of large malware corpora.

Our Approach

Our strategy targets the identification of inherent characteristics that allow to measure the resemblance of different software instances. This allows to optimize the analysis of large data corpora, since the analysis of an individual instance —which may well involve costly human expertise —can be propagated to similar instances with basis on the inherent characteristics identified by the classification framework, thereby producing an application effect.

Withal, this plan entails intricate issues. *How to get a description that suffices to represent inherent characteristics of malware? How to measure the resemblance of malware instances based on their inherent characteristics? How to evaluate the success of a malware classification framework?* These are some of the challenging questions that this thesis addresses.

Orthogonally to these issues, we have to overcome manifold practical hurdles. A satisfactory framework for malware classification should provide high quality analysis, being able to produce results with high accuracy, resist to the profusion of anti-analysis techniques frequently employed by malware and be scalable, so as to concretely withstand the surfeit of malware in current and future cyber threat landscapes.

To address the multiaxial issues involved in the creation of a successful malware classification framework, we take an accruing approach. First, we focus on the conceptual basis of our work in order to define our axiology, which stipulates the guiding principle of the methodologies developed in this thesis. Then we concentrate on each stage of our malware classification workflow, which includes the enhancement of tooling for malware analysis, the improvement of software characteristics representations and the design of alternative analysis methodologies that can provide the required accuracy, robustness and scalability.

Research Objectives

This thesis addresses the problem of malware classification taking an approach in which human intervention is spared as much as possible. We steer clear of subjectivity inherent to human analysis by designing malware classification solely based on data obtained directly from malware analysis, thus taking a *data-driven approach*.

Our objective is to enhance the automation of malware analysis and to straighten the use of machine learning methods with aim to autonomously spot and reveal hidden commonalities within large data corpora. Improving the automation of malware analysis involves the development of new tools and methods for extracting inherent characteristics of software instances; in its turn, improving the autonomy of *malware classification* requires to restrain it to data obtained through direct analysis only, without the handicap of trusting indirect information exterior to the boundaries of the system.

This multifaceted desideratum entails not only technical challenges but also acute epistemological concerns. Although machine learning has been proving to be a top-notch alternative for such complex scenarios, leveraging it to pull off malware analysis is far from being straightforward. We endeavor to improve the application of machine learning in the field of malware analysis —though we do not intend to improve the state-of-the-art of the machine learning field itself. Therefore, we dedicate special attention to methodological aspects that impact the setup of our experiments as well as the evaluation of the results.

Research Questions

In this thesis we take an accruing approach. To reflect this plan, we organize our research based on top-level research questions that outline each stage of our workflow for *malware classification*, which comprehends *data extraction* → *data transformation* → *data analysis*.

Our top-level research questions are:

TOP-RQ1 *How to improve the arsenal of techniques currently used in malware analysis, in particular for the analysis of binary files?*

TOP-RQ2 *How to compute the similarity of unknown programs with high accuracy while being friendly to search and clustering algorithms for malware analysis?*

TOP-RQ3 *How to design a data-driven clustering framework that is accurate, robust and scalable?*

Contributions

As main contributions of this thesis we:

— review epistemological matters that severely impact the use of machine learning for malware analysis; thanks to this review we propose a realist ontology of malware and we settle our axiology [chapter 1 (page 25)].

— study how to optimize the setup of symbolic execution in the context of malware analysis [chapter 3 (page 111)];

— study a graph-based representation to characterize the behavior of computer programs and propose a related similarity function that is accurate and robust [chapter 4 (page 153)];

— propose an evaluation paradigm for the evaluation of frameworks for malware classification [chapter 4 (page 153)];

— propose a data-driven clustering approach that can work with multiple heterogeneous features, which is thus prone to high accuracy and robustness, while being inherently scalable [chapter 5 (page 189)].

As positive externalities, the development of this thesis also promoted contributions on the applied aspects necessary to set the study up. This includes:

— the development of an analysis tool for call tracing based on symbolic execution [section 3.5.1 (page 136)];

— the optimization of a graph mining algorithm implementation [section 3.5.1 (page 137)];

— the development of a docker-based container orchestration to scale up malware analysis [section 3.5.1 (page 138)];

— the integration into our analysis framework of publicly available external tools [section 5.5.1 (page 212)].

In all these cases, our experiments produced benchmarks that contribute to improve the state-of-the-art of malware research.

Thesis Outline

This thesis is structured in two segments. Initially, in chapters 1 (page 25) and 2 (page 57) we unravel the theoretical foundation that support our research. Then, in chapters 3 (page 111), 4

(page 153) and 5 (page 189), we tackle the nub of the thesis, which contains our main contributions.

More specifically, this thesis is organized as follows:

— chapter 1 (page 25) draws a general overview on epistemological issues related to malware research. Our intent is to clearly establish our axiology, which guides the remainder of our study;

— chapter 2 (page 57) contains background material on machine learning and malware analysis;

— chapter 3 (page 111) presents our study on the use and optimization of symbolic execution for call tracing in malware analysis;

— chapter 4 (page 153) targets the behavioral representation of computer programs in the form of graphs and studies our proposition for an associated similarity function;

— chapter 5 (page 189) tackles the issues of multi-feature analysis and scalability in malware clustering;

— Finally, the last chapter (page 259) concludes this thesis and presents possible future perspectives.

CHAPTER 1

EPISTEMOLOGICAL VIEW ON MALWARE RESEARCH

1.1 Introduction

Malware research is the field that studies subjects related to malware analysis, which is a discipline that analyzes computer programs and other artifacts in order to assess the degree of danger that they carry to users and systems.

This rather generic definition derives from the broad spectrum of computer threats that can be subject to malware analysis. Even the term malware —portmanteau for *mal*icious soft*ware* —is an "umbrella term" defined to accommodate all different definitions of malicious programs (e.g. computer virus, worm, trojan horse, etc).

The lack of an agreed taxonomy establishing a common understanding of terms and definitions (i.e. a *domain ontology*) is an old problem in the field. To illustrate it, we can cite the Virus Bulletin editorial of December 1989 [79] where the authors raised the lack of consensus to define a "*terminology for defining different types of computer virus*" nor the existence of any "*standard nomenclature to identify specific viruses*". This still remains an open problem in the community to date.

This ontological gap goes against the current trend in malware research, which focuses on machine learning to aid analysis. How can we teach machines about things that we did not figure out yet? How can we evaluate results if we do not know the answers?

In part, this challenge derives from idiosyncrasy of the malware analysis field, which is highly dynamic and strongly influenced by the industry, thus less keen to *epistemological* thinking and more driven by problem solving (i.e. *Episteme* versus *Techne*).

In this chapter, our motivation is to closely review how knowledge is created in the field and how it evolves afterwards (i.e. the structures of knowledge). We endeavor to lay out an epistemological view on malware research, which will enable us to determine our research paradigm more precisely. This research paradigm will then ground our methodolo-

gies throughout our study.

We organize this section in three parts: an ontology (section 1.2, page 26), an epistemology (section 1.3, page 44) and an anxiology (section 1.4, page 54). This organization intends to shed light on underlying assumptions taken by research works, but that are quite often implicitly rather than explicitly defined. Each of these three parts views the research field from a different angle, where we are interested in answering the following questions:

— **"What does exist (in the malware research field) ?" [Ontology]**

— **"What can we know? How do we know what we claim to know?" [Epistemology]**

— **"What do we value to guide our research?" [Axiology]**

1.2 An Ontology of Malware

> *The variables of quantification, 'something', 'nothing' , 'everything', range over our whole ontology, whatever it may be; and we are convicted of a particular ontological presupposition if, and only if, the alleged presuppositum has to be reckoned among the entities over which our variables range in order to render one of our affirmations true.*
>
> Willard Van Orman Quine (1980). "From a Logical Point of View", p.13, Harvard University Press.

Ontology contemplates the description of existence. In Philosophy, the term was often synonym of *metaphysics*, referring to subjects that are "beyond Physics" [1]. Philosophically, the term *ontology* expresses the reflection about what *might exist*. In the scientific context —often referred as "*domain ontology*" —it describes the fundamental analysis that seeks to identify key concepts in the study of a subject. In the domain of computer security, ontologies are for the most part "taxonomies, definitions and collections of the varieties of flaws, prophylactics and preventatives" [253].

1. *Physics* was better understood as a synonym for what we know as *Science* today.

A good starting point to initiate an ontological analysis of the malware research field is the simple concept of *malware*. Interestingly, the term malware postdates the existence of the field. Many sources attribute the creation of this term "malware" to Yisrael Radai in 1990 [81], however it is hard to find evidences online to confirm that nowadays. Nonetheless, the very first time the term appeared in a Virus Bulletin edition was in march 1992 [78].

We adopt two approaches to establish an ontology of the malware research field: historical and analytical. The former identifies key concepts that emerged organically in the field in the course of its history, whereas the latter builds on existing ontologies of computation systems to extend them in the context of malware research.

1.2.1 Historical Approach

This section provides an overview of history-oriented ontologies of malware proposed in the literature. As a foundation to appreciate these ontological proposals, we begin with a brief historical introduction that intend to highlight the challenges entailed by this approach. Therefore, beyond the genuine intention of recollecting a (brief) timeline of major events in malware history, we are specially interested in understanding how new concepts and taxonomies emerged and how they evolved over time.

A Brief History of Malware

An inchoate idea of a malicious program has been around since at least 1949, when John Von Neumann presented the conception of *self-reproducing automaton* in a series of lectures [295]. This notion of self-reproducibility is invariably found in the early ages of malware history, because of the inherent need of computer threats to spread [2].

The first documented program to exploit the ability of self-replication was dubbed the *Creeper*, in 1971. The *Creeper* was written by Bob Thomas at BBN as an experiment, where the program was able to self-replicate onto other computers connected to the local network and to trigger the execution of these copies. As a result of its execution, the "infected" system printed the message `"I'M THE CREEPER : CATCH ME IF YOU CAN"`. Despite not attempting to cause any harm to the system, the *Creeper* is broadly considered as the first ***computer worm***. Interestingly though, the term *worm* actually postdates the *Creeper*; it originally comes from a John Brunner's 1975 science-fiction novel called The Shockwave Rider and was adopted by John Shoch and Jon Hupp at Xerox PARC in 1979 [260].

2. Since the advent of *Advanced Persistence Threats* (APTs) [101] this requirement is no longer imperative.

In 1984, Fred Cohen defined the concept of a ***computer virus*** as a program capable of attaching itself to other programs (i.e. "infect") causing them to become viruses as well [54]. In this paper, Cohen mounted experiments to validate in practice the implementation of a real computer virus, to observe it and to propose preventive measures and countermeasures against viral infection. Cohen also provided several interesting results such as the *undecidability proof of a universal virus detector*.

In addition, Cohen's paper mentioned the Xerox worm program and the potential danger of using computer viruses in conjunction with a ***"Trojan horse"*** that could clearly cause widespread denial of services and/or unauthorized manipulation of data. It is interesting to notice that the term *Trojan Horse*, which is still highly used —in its short form, i.e. *trojan* —was already present at the time of Cohen's paper.

[Computer] Trojan horses were (semi-)formally defined by Ken Thompson in his 1983 Turing Award acceptance lecture "Reflections on Trusting Trust" [281]. Thompson described a compiler that deliberately introduces bugs during the compilation process, denoting the addition of unexpected and concealed code in programs [3]. Thompson acknowledged the Air Force Data Service Center (AFDSC) report released in 1973 for describing —hence conceiving —the possibility of a *trojan horse* on early implementation of Multics. It is also possible to find a prior usage of the term *trojan horse* in a 1971 Bell Labs' manual [156].

At the time, there was already a heed about what is currently known as ***backdoor*** (then referred as *"trap door"*). In 1967, H.E. Petersen and R. Turn discussed about privacy issues concerning the (back then) recent advances in computer systems allowing multi-user support [216]. They defined *trap doors* as entry points planted in the system and conceived the possibility of an active infiltrator that knows about them (e.g. "unscrupulous programmers or maintenance engineers") or who probes for it. Despite the long-lasting concerns toward this threat, backdoor installers became widely adopted by malware writers only starting from late 1990's, after the disclosure of *Back Orifice* at Def Con 6 [64] in 1998.

With the advent of personal computers as commercial success in the 1980's, malware started to expand fast. The first malware that appeared *in the wild* (i.e. not as part of a lab experiment) was the *Elk Cloner*, written around 1982. It was an instance of what is currently known as ***boot sector virus*** (or even a primitive version of a ***bootkit***), because it modified the boot sector of floppy disks to install a copy of its payload. Computer booted from an infected floppy disk would load the virus in the memory, which would await until an uninfected disk be inserted into the computer in order to copy itself in the disk, thus spreading the infection

3. This strategy is comparable to the current *software supply chain attack (e.g.* 2019 *Barium's attack [323]).*

from disk to disk. Despite targeting the boot sector of floppy disks, the *Elk Cloner* did not damage any system file; instead it displayed a little poem as a prank.

The first computer virus to target IBM PCs became know by the name of *Brain*, because it deliberately included the following signature message in the program:

```
Welcome to the Dungeon ©1986 Basit & Amjads (pvt).  BRAIN COMPUTER
SERVICES 730 NIZAM BLOCK ALLAMA IQBAL TOWN LAHORE-PAKISTAN PHONE:
430791,443248,280530.
```

Similarly to the *Elk Cloner*, the *Brain* virus also affected the boot sector of floppy disks to insert a self-replication copy on them. *Brain* is yet another example of non-destructive *computer virus*; the authors of *Brain*, the brothers Amjad Farooq Alvi and Basit Farooq Alvi, told in an interview [311] that they acted on an experimental motivation to find out how far and quick this virus could spread.

The first Internet malware became known by the name of *"Morris worm"*, since it was written by graduate student Robert Morris at the Cornell University. It spread out in November 1988, exploiting several vulnerabilities: cracking of password files, abuse of the debug option in the Unix sendmail program, buffer overflow attack on Unix finger daemon program [44]. Similarly to the *Creeper*, the *Morris worm* was not deliberately programmed for damage; again, the purpose was an intellectual and computing exercise, although it did not have any kind of consent of the potential affected parties.

The emerging trend of malware that targeted an ever larger basis of computers in the end of the 1980's, promoted the advent of anti-malware programs. *Flushot Plus* by Ross Greenberg was the first anti-malware software to be released in 1987. In 1989, John McAfee released the *VirusScan*™ program, which could detect and repair several viruses at one shot [247]. Other companies like Sophos, Avira and F-Secure were also founded before the end of the decade.

The concept of ***rootkit*** appeared in the beginning of the 1990's. These are programs developed to conceal the presence of their payload, avoiding analysis and detection while maintaining privileged access to a target system [143]. The first rootkit (is believed to have) appeared in 1990, written by Lane Davis and Riley Dake for the Sun operating system [34] —hence the name combining *"root"* + *"kit"*. The first public rootkit for Windows appeared only in 1999, written by Greg Hoglund and called *NTRootkit* (as it targeted the WindowsNT operating system).

During the 1990's the awareness about malware grew considerably. For instance, *Michelangelo* became the first computer virus to attain worldwide attention in 1992. This virus, which

targeted DOS systems, was designed to infect and remain dormant until March 6 (the birthday of the Michelangelo painter, hence the name), date on which the virus overwrites critical system data, including the boot and file allocation table (FAT) records, rendering the disk unusable.

As of 1990's the profile of malware shifted, as they started being more frequently oriented towards profitable activities (e.g. *spamming*). It is in the 1990's that ***spyware*** —portmanteau for *"spying software"* —emerged. The origin of the term is unclear, however the first record of the term *spyware* that can be found nowadays dates from 1995 in a *Usenet* post by Roland Vossen, who was mocking the marketing strategy of Microsoft [296]. *Spyware* began to be used in the context of computer software in 1999 when Zone Labs used it in a press release for its Zone Alarm firewall product [60]. In 2000, Gibson Research launched the first anti-spyware product called *OptOut*, which described *spyware* as "any software that employs a user's Internet connection in the background (the so-called 'backchannel') without their knowledge or explicit permission".

As of the years 2000's, *spyware* gained great notoriety as the threat spread; in 2002 the Federal Trade Commission (FTC) logged up to 150K identity theft complaints and in 2004 this number grew to 250K [97]. In 2005 the FTC provided the following comprehensive definition for *spyware*: "software that aids in gathering information about a person or organization without their knowledge and that may send such information to another entity without the consumer's consent, or that asserts control over a computer without the consumer's knowledge".

Around the same time, the other prevalent threat to appear was ***Adware***, which was defined as "small pieces of code installed on the client machine whose primary purpose is to achieve highly targeted marketing by a number of methods: Displaying usually an excessive number of pop-up advertisements; Installing toolbars full of adverts; Modifying Browser settings such as the default home page favourites; Hijacking website searching and use." [259].

Due to the numerous similarities of techniques used by *adware* and *spyware* there has been a huge lack of consensus about what distinguishes one malware type from the other, which was subject of intense discussions in the first decade of 2000. Another debate that unfolded around *adware* was whether it should be classified as malware or simply as *potentially unwanted program (PUP)*.

Ransomware is yet another example of computer threat, which demands a ransom to undo the damage caused after hijacking personal data through the use of encryption or blocking the normal use of system. The former, which demands a ransom to decrypt the hi-

jacked data, is known as *crypto-ransomware*, and the latter, which demands a ransom to to release the system, is known as *locker-ransomware*. The *crypto-ransomware* was presented by A. Young and M. Yung at the 1996 IEEE Security and Privacy conference [321] decades before it becomes a major widespread threat in the wild.

Despite of being currently a major threat, the first documented ransomware dates back 1989. It became known as *AIDS* (a.k.a *PC Cyborg*) because it was present in a floppy disk with the inscription `"AIDS Information Introductory Diskette"`. When the malware was executed, it would hide all folders in the system and encrypt the files on the `C:` drive. Then, it would display a ransom message demanding $189 to be directed to the *PC Cyborg Corporation*, otherwise the compromised systems would not be restored.

Crypto-ransomware "reemerged" with *GPcoder* in 2005 after a lull of 15 years. The attack included the encryption of many files of predetermined extensions; in each directory including an encrypted file, *GPcoder* would leave a ransom note with an e-mail address to communicate with the malware authors and eventually get the corresponding decoder [102]. Since then, *crypto-ransomware* became a major threat specially in the second decade of 2000's after the advent of *cryptocurrencies*, which can anonymize wallets to receive payments (e.g. *bitcoin*).

So far we named but a few concepts (often associated with malware types) that have emerged throughout history. Yet, there are major inconsistencies with respect to the characteristics taken into account depending on the definition. For example, *trojans* and *backdoors* are very different from *worms* and *viruses*: *worms* and *viruses* are essentially defined by their propagation techniques; *trojans* bear upon some concealed functionality in a program (notwithstanding which functionality it might be); *backdoors* are specifically related to the concealment of entry points in the system and the possibility of exploiting them.

Furthermore, by the definition of *trojan* sketched above, we note that the frontier distinguishing malware from ordinary software quickly becomes flimsy, due to the potential subjectivity involved in the definition of what an "expected functionality" may be. The same subjectivity judgement blurs the distinction between *spyware* and *adware*, or even whether *adware* are malware or *PUP*.

On top of that, in the course of the last decade we have seen a great evolution in the complexity of malware. For instance, *Stuxnet* (2010) was a modular malware that had *worm* capabilities, while targeting SCADA systems acting as a *rootkit*; *Mirai* (2016) was a *botnet* that targeted IoT devices to install *backdoors* and remotely control these devices in order to perpetrate *DDoS attacks* against other systems on the Internet; *WannaCry* (2017) displayed

worm capabilities in the propagation phase, exploiting vulnerable systems on the network, and executed a *ransomware* payload on the vulnerable systems.

As we have seen through the aforementioned examples, concepts and taxonomy widely adopted by practitioners in the malware analysis field appear and evolve organically, following the sequence of events that eventuate in the field —or even from popular culture, as we have seen terms like *worm* being borrowed from sci-fi novels. This constant adaptation is a major challenge for the establishment of a historical-oriented ontology for malware research. For instance, in 2006 Giri *et al* wrote: "Self-replicating malware can use file infection or mass mailing to spread. Ransomware falls in the category of Trojans that are non-self-replicating malware and that use different methods for transmitting themselves." [102]; however, after *WannaCry*, this distinction is no longer true.

History-Oriented Ontologies of Malware

We consider as history-oriented an ontology whose constituent concepts (or entities) are by-product of the organic evolution of malware analysis throughout its history. Several ontologies concerning malware can be found in the literature, and almost invariably they can be categorized as history-oriented.

Swimmer was one of the first to attempt formally defining an ontology of malware [276]. Swimmer described a malware class hierarchy that reflected the types of malware as existing at the time. Swimmer also provided descriptions to identify characteristics that one expects to find in malware, mapping them to corresponding types of malware. Finally, Swimmer expressed his ontology using the Web Ontology Language (OWL) [181] so as to make it public available for universal usage, specially for alerts, malware descriptions and intrusion detection response systems. Figure 1.1 shows the malware class hierarchy class as defined by Swimmer [276].

Mundie and McIntire sought to build an ontology for malware (analysis) to tackle the lack of common vocabulary and shared concepts. They acknowledged that "nowhere in cybersecurity community is the lack of a common vocabulary, and the problems it causes, more apparent than in malware analysis" [199]. To address this problem, they established a methodology for mining malware-related terms in a database containing 10 years of emails, so as to build a controlled vocabulary of ~270 malware analysis terms —which clearly follows a history-oriented approach. Based on the mined terms, Mundie and McIntire provided a taxonomy and constructed two ontologies in OWL: a static ontology and an intentional ontology, respectively defined as "an ontology that describes static aspects of the world, i.e.,

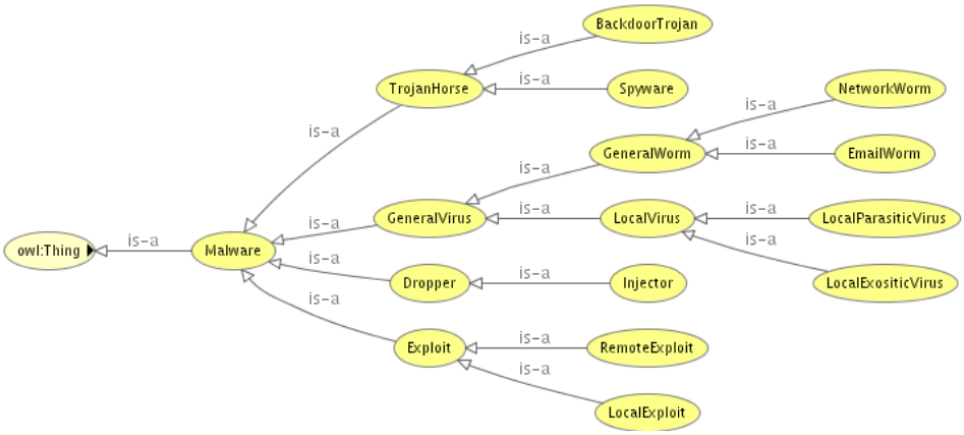


Figure 1.1 – Swimmer's ontology of malware classes

what things exist, their attributes and relationships" and "an ontology that encompasses the world of things agents believe in, want, prove, disprove, or argue about".

Gregio *et al* pointed the obsolescence of prior ontologies: "Current efforts are based on an obsolete hierarchy of malware classes that defines a malware family by one single prevalent behavior (e.g., viruses infect other files, worms spread and exploit remote systems autonomously, Trojan horses disguise themselves as benign programs, and so on)" [110]. They proposed to move away from the identification of inherent characteristic as a sort of definition of malware and focus attention on behaviors, in order to define an ontology that addresses behavioral aspects and activities performed by malicious programs. However, the authors also sidestepped the more deep ontological issue of clearly defining the nature of malware by only dealing with "suspicious activities" as subject of their ontology.

Xia *et al* addressed the gap between malware ontology and behaviors in a practical case [313]. They took a subset of Swimmer' malware ontology [276] and defined a set of behaviors to find correspondences between malware classes and behaviors using several mining algorithm. Nevertheless, the subset of malware classes used was still very reduced (i.e. only five types) and therefore meager to provide an extensive description that could accomplish the practical use. Furthermore, the detection method as designed suffers from the limitation of malware classes as ontological entities, which are *far from adequate to address the issue of modern malware*[4], as asserted by Gregio *et al* [110], Orbst *et al* [207] and others.

MALOnt [230] takes an approach that is similar to the one by Mundie and McIntire [199]. However, instead of mining email, Rastogi *et al* analyze a corpus of threat reports to extract

4. As previously discussed, malware classes may become inaccurate or obsolete due to the evolution of malware (e.g. the definition of *ransomware* by Giri *et al* [102])

the data elements that underpin their ontology, which comprises entities and relationships in the form of knowledge graphs. Once again, the approach is guided by the organic development of the field.

In addition to self-proclaimed ontologies, several standards, taxonomies, and languages involving malware have been proposed for practical reasons (e.g. data exchange). It is the case of OpenIOC [255], MAEC [176] and others. They can provide useful information and thoughts about malware description and outline some ontology of malware, despite not being their main intent.

1.2.2 Analytical Approach

Alternatively to the historical-based approach, which deals with concepts that came about organically, it is possible to conceive an ontology of malware entirely through reasoning. By construction, this method is likely to devise a simpler and more coherent ontology.

Ontologies of Software

Before delving into the discussion about malware ontology, let us first review the existent ontologies of software, since malware, by definition, is a specific kind of software (i.e. a "malicious kind").

In turn, the ontologies of software are subsumed in the ontology of computational systems. Therefore, we start by presenting the two main approaches pursued by the philosophy of computer science [7]:

— a first view that considers a **conceptual ontology**, which breaks down computation systems on the basis of a hierarchy of levels of abstraction;

— a second view that considers a **realistic ontology**, which analyze computational systems as a composition of two distinct ontological entities (i.e. *software* and *hardware*)

Method of Abstraction Abstraction is in the core of sciences such as mathematics or computer science. Sciences benefit largely from abstractions when analyzing complex systems by abstracting away finer details of these systems into some "higher level" description that contains only the most relevant attributes. Despite widely applied in sciences, historically this approach has found little application in philosophy: the formalization of levels of abstraction (i.e. "the method of abstraction") as a conceptual and phenomenological framework for analysis was only recently proposed by Floridi and Sanders [91].

This approach formalizes the representation of an object of study (referred as *system*) as a set of *observables*, which are in turn interpreted from *typed variables* together with statements of what features of the system they represent. The set of observables used to describe a system is called *Level of Abstraction (LoA*).

For instance (as presented in [91]), when analyzing the taste of wines, one can describe them in terms of attributes that commonly appear on 'tasting sheets': 'nose' (representing bouquet), 'legs' or 'tears' (viscosity), 'robe' (peripheral colour), 'colour', 'clarity', 'sweetness', 'acidity', 'fruit', 'tannicity', 'length' and so on; each attribute corresponds to an *observable* that is described by some determined typed value, and the set of all values (i.e. observables) forms a "tasting LoA". If the purpose is the analysis of the market of wine, instead of adopting the "tasting LoA", one would be rather interest in a "purchasing LoA" consisting of observables like 'maker', 'region', 'vintage', 'supplier', 'quantity', 'price', and so on.

The method of abstraction is in fact an epistemological levelism method; however, whenever applied it requires an "ontological commitment", meaning that the theory commits itself ontologically by opting for a specific LoA [91]. Still, one must not assume that LoA has the cogency of a universal ontological theory [254] [5], in the sense pursued by some branches of natural sciences, i.e. a mereological theory with a lowest level on which subsequent levels piles up (e.g. as the theory of "reductive levels" defended in [209]). Yet, for some anthropogenic systems (e.g. computer systems) the method of abstraction can perfectly underpin ontological theories.

Primiero was the first to devise an ontology of computer systems in terms of LoA [222]. The main interest is how Primiero ponders upon the ontological and epistemological relationships inferred (or implied) at the interfacing of each one of the following layers: (i) Intention, (ii) Specification, (iii) Algorithm, (iv) High-level programming language instructions, (v) Assembly/machine code operations, and (vi) Execution.

The overall LoAs of computer systems as proposed by Primiero [222] are depicted in figure 1.2. Primiero defends that computer systems are built from an *intention* that aims at a *problem*, which *is reflected by* a task *fulfilled* by an algorithm, and so forth. This interplay goes up to the physical level, keeping corresponding stances of epistemic interpretation at each stage (e.g. "electrical charge" → "action"). According to this view, no LoA taken in isolation is able to define what a computational system is (nor to determine how to distinguish software from hardware); the computational systems are instead defined by the whole ab-

5. In the sense of "metaphysics"; not to confuse with the broader meaning of "ontology" that has been in use here (e.g. domain ontology).

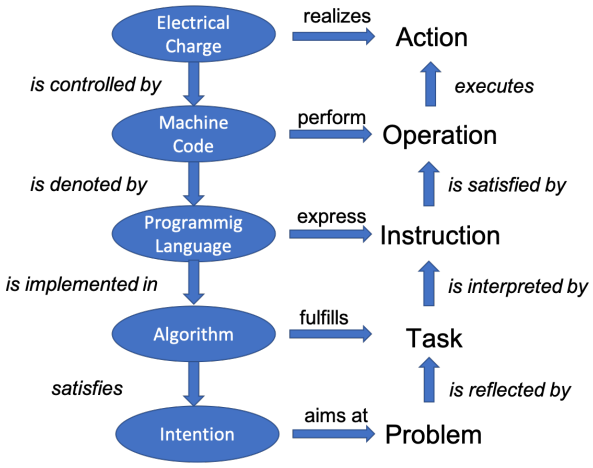


Figure 1.2 – Primiero's ontology of computer systems [222].

straction hierarchy.

Describing in detail Primiero's ontology of computer systems goes beyond our scope and intentions. However, as these LoAs follow the lines of textbook abstraction layers of computer organization, it is possible to point a shortcoming of this ontology: the computation model that guides this ontology is oversimplified as if all computer programs ran on bare metal machines —it does not consider other types of execution such as interpreted code, intermediate representations, virtual machines, hardware emulation, etc.

Yet, it is important to notice that the *Intention* is a constituent part of the ontology of computer system. As a matter of fact, the purpose-oriented aspect of computer systems (expressed here by the stance of *Intention*) —further discussed below —is constantly subsumed in the ontology of computer systems.

Hardware and Software This approach regards algorithms, source codes, and programs as belonging to the category of abstract entities (i.e. *software*), while microprocessors, hard drives, and computing machines fall in the category of physical entities (i.e. *hardware*). This view is, however, not unanimous and even the mere differentiation between the concepts of hardware and software is still in debate.

Moor seems to be the first to propose an ontology of computing systems [196]. Moor starts by giving some examples of programs that contradict commonsense about "hardware" as defined in a dictionary entry (i.e. "the physical units making up a computer system (Chandor [1970], p. 179)"). According to Moor, programming of early digital systems was

commonly done by plugging in wires and toggling switches, thus being physical/tangible [6]. For Moor, a more accurate definition for computer programs would be "a set of instructions which a computer can follow (or at least there is an acknowledged effective procedure for putting them into a form which the computer can follow) to perform an activity.", thus following that computer programs can be understood on the physical level as well as the symbolic level. Therefore, to Moor, the dichotomy of software/hardware was only pragmatic, but not ontological.

Tanenbaum agrees in essence with Moor's view. He sees the structure of computer organizations in terms of layers [7]. By noting that "programs written in a computer's true machine language (level 1) can be directly executed by the computer's electronic circuits (level 0), without any intervening interpreters or translators" and that "any instruction executed by the hardware can also be simulated in software", Tanenbaum concludes that "hardware and software are logically equivalent" [279].

Suber's essay [274] goes further, stating that hardware is part of the software ontology. Suber argues that software (or program, interchangeably) can be defined as "*pattern* that is readable and executable by a machine", where pattern takes a "broad sense to signify any definite structure, not in the narrow sense that requires some recurrence, regularity, or symmetry". Suber then analyzes the concepts of machine *readability* and *executability*, and argues that any pattern $\mathscr{P}$ can fulfill both properties, because even if $\mathscr{P}$ is seen as noise for a machine $\mathscr{M}$ and language [8] $\mathscr{L}$, there exists a machine $\mathscr{M}'$ and language $\mathscr{L}'$ for which $\mathscr{P}$ is meaningful, regardless whether their existence is currently unknown. Then, Suber defines the *freedom of interpretation* as the "the flexibility to read intrinsically unmeaning formal patterns as holding a given meaning" and the *freedom of formalization* as the "the complementary flexibility to create a formalism or code to express a given meaning". These definitions open up an unlimited freedom of means for conceiving machines and languages, thus following that in principle "everything is software", given that any concrete object displays patterns. The immediate corollary is that "hardware, in short, is also software".

Duncan agrees only partially with Tanenbaum's views and finds Moor's and Suber's positions implausible [76]. Duncan acknowledges that hardware and software are logically equivalent, but he diverges from Tanenbaum's claim that "the boundary between hardware and software is arbitrary and constantly changing.";

6. In this sense, it is possible increment the list of examples with other technologies such as punched cards or hardware description languages (HDL) and programmable logic devices (PLD).
7. In the lines of the hierarchy of levels of abstraction.
8. Maybe "grammar" would be a more appropriate term.

Duncan's critics to Moor's argument consider the proposed ontology too limited, as it consists of only three main *"kinds of thing"*: the symbolic level of computer programs, the physical level of computer programs, and the activities related to each level. This is the reason why Moor is not able to find any significant ontological differences between software and hardware.

As for Suber's ontology of software, Duncan summarizes it with the following syllogistic argument:

(P1) Software is a pattern that can be read and executed.
(P2) A pattern can be read and executed if it can in principle satisfy the physical and grammatical conditions of readability and the requirement of executability.
(P3) All patterns can satisfy the physical and grammatical conditions of readability and the requirement of executability.
(P4) All concrete objects display patterns.
(C1) All concrete objects can satisfy the physical and grammatical conditions of readability and the requirement of executability.
(C2) Therefore, all concrete objects are software.
(P5) Hardware is a concrete object.
(C3) Therefore, hardware is software.

Duncan then points that there is a flaw in the logical sequence (P3) → (P4) → (C1). According to Duncan, the premise that "all concrete objects display patterns" does not allows one to conclude that "all concrete objects are patterns". Rather, the conclusion of Suber's argument should have been that "all concrete objects *display* software patterns". Duncan also provides the following example redolent of absurdity:

(P1) Software is a pattern that can be read and executed.
(P6) A peanut butter sandwich is a concrete object.
(C4) Therefore, a peanut butter sandwich is software (from C2).
(C5) Therefore,a peanut butter sandwich is a pattern that can be read and executed (from P1 and C4).

Duncan argues that even if one could hold that (C4) is in fact valid, despite sounding implausible, this would raise the question about "how to distinguish the various types of software in the world". Therefore, to Duncan, Suber's ontology of software is to little or no avail in practice.

Towards the formulation of an ontology of software, Duncan takes up on Turner descrip-

tion of computation artifact[9] (e.g. computer program) as *specifications* [285]. For Turner, "programming and specification are not easily separable activities" inasmuch as programming is not an aimless activity and "any articulation of its goals constitutes a specification" —even if the description of the goals is created *post hoc* and quite informally. Nonetheless, Duncan points the lack of a detailed ontological theory distinguishing software and hardware as a shortcoming of Turner's work.

Turner's conceptualization of specification says that "something is a specification when it is given *correctness jurisdiction* over an artifact". In other words, a specification has to provide a "criterion of *correctness* or *malfunction* for purported artifacts". Turner notes that such analysis of specification is an idealization, meaning that it should not be taken as an exact reflection of practice; in practice not every software project will evolve from prescribed specifications, however the purpose-oriented nature (or *teleological function*) of a software project delimits a sort of specification —as illustrated by Turner: "For example is there a specification for a search engine? Yes. It has to be a search engine not a language translator. It may be vague, but it is still a specification".

To formulate an ontology of software, Duncan analyzes the notion of *correctness jurisdiction* through the lens of the Basic Formal Ontology (BFO) [12] [10]. Duncan maps Turner's *correctness jurisdiction* to the BFO's definition of *role*; in particular, it is defined a ***normative role***[11], due to its "authority to govern" (of software over hardware [12]). Duncan argues that the notion of *correctness jurisdiction* as BFO's role category is suitable because "an entity can lose its *correctness jurisdiction* without being physically changed". For example, a machine that fails to execute the instructions encoded in a software is defective; however, a software that is found to contain mistakes (i.e. *bugs*) no longer implies the machine to be defective, but instead it loses its *correctness jurisdiction* over the expected behavior of the machine.

Duncan thus proposes an ontology of software, capable of distinguishing *software program* from *computer hardware*, as follows: "software program: a *specification* that consists of one or more programming language instructions and whose concretization is embodied by an artifact that is designed so that a physical machine may read the concretized instruc-

9. Artifact can be defined as "a product of human workmanship", in contrast to a "natural object".

10. For the sake of contextualization, we provide a brief introduction on BFO in the appendix section 6.1.

11. There is no specification of role types in the BFO hierarchy; therefore, *normative role* is not a proper BFO type. Here, Duncan uses the word *normative* to emphasize the quality of "correctness jurisdiction" for the BFO concept of *role*.

12. Duncan argument about "normative role" goes beyond "the authority to govern how a physical device is constructed or behaves" (here referred as *software over hardware*), but delineating all details of this argument goes beyond the scope of this thesis.

tions" [76].

We note that this ontology neatly depends on Turner's conceptualization of *specification* —not only "software program" is defined as a kind of specification *per se*, but also a *programming language* is defined as a *specification* that provides syntactic rules for expressions and semantics to determine computation operations that result from the execution of such expressions.

In a previous work, Duncan proposed an ontology of software, which held a more straightforward bind with BFO and was also capable of differentiating hardware and software [75]. This version of Duncan's ontology follows the steps detailed below:

Is software a continuant or an occurent entity?

Observing that software is an entity that maintains its identity through time, Duncan concludes that software is a *continuant entity*.

Is software an independent continuant or a dependent continuant?

Duncan defines hardware as an *independent continuant* and software as a *dependent continuant*, inasmuch as software bears on hardware to be realizable [13].

Is software a generically dependent continuant or a specific dependent continuant?

Duncan splits two aspects embodied in software: the encoding facet (at any level, e.g. source code), which is a *generically dependent continuant* as it can be copied and exists in multiple places, and the **teleological function** of the software (referred as *computational function*), which is a *specific dependent continuant*.

Which kind of specific dependent continuant is a computational function?

Duncan observes that "any instance of this *[computational] function* is actualized in the running of some particular computational process, and the particular computational process could not exist without the instance of a piece of computing hardware on which the process runs", concluding that computational function is therefore a realizable entity. In particular, Duncan describes *computational function* under the BFO's concept of *function*, since it fulfills some essentially end-directed activity, which is defined (explicitly or not) by the programmer who realizes (i.e. implements) it into some software program (i.e. encoding).

Duncan thus provides the following ontological definitions for software:

13. There is an acknowledged lack of clarity on this argument in the text as Duncan goes over it too quickly.

— **piece of computing hardware**: An *independent continuant* that is intentionally designed for the actualization of one or more computational functions.

— **computational function**: A *realizable entity* that inheres in one or more pieces of computing hardware. The actualization or manifestation of a *computational function* is an essentially end-directed activity in virtue of the kind(s) of context(s) that the computing hardware is made for.

— **software application**: A *generically dependent continuant* that encodes a representation of one or more *computational functions* (realizable functions).

The search for distinguishing hardware and software has set the ground for a dialectics that highlights the essence of software; in particular, the duality of *computational function* and *software application* is a noteworthy outcome that can have practical effects in the development of methods for software analysis.

An Ontology of Malware

Here we move forward with the discussion about malware ontology. There exist scant propositions of malware ontologies that follow an analytical approach.

The appreciation of malware analysis (and malware classification) through the lens of the philosophy of technology is very recent, notwithstanding the profound conceptual issues that malware analysis entails. First we briefly present some contributions of philosophy of technology on the subject of malware ontology, then we discuss issues related to malware ontology with basis on the concepts previously presented.

Malware as Malfunction-Inducing Artifacts Primiero *et al* provide definitions for malware and malware types based on existing malware taxonomy [223]. They do not have the ambition of reformulating the ontology of malware from the ground up, instead they aim to obtain a general, language-independent functional description of malware. They note that malware categorization through specific languages (along the lines of the historical approach seen above) is "extensive and it offers a detailed identification of all properties of malware artifacts", which is the reason why they cannot explain what makes certain software to be malware.

The central concept in Primiero's ontological and functional analysis of malware is software *malfunction*. Floridi *et al* developed the notion of malfunctioning software based on

the view that software are human-made and teleological artifacts, that can therefore malfunction [92].

Malfunctioning occurs when the artifact does not do what it is supposed to do. Floridi *et al* distinguish two kinds of malfunction: a negative malfunction, referred as *dysfunction*, when an "artifact instance [14] either does not (sometimes) or cannot (ever) do what it is supposed to do"; and a positive malfunction, referred as *misfunction*, when an "artifact instance may do what it is supposed to do, but it also yields some unintended and undesirable effect(s), at least occasionally".

Primiero *et al* consider these definitions in the presence of *malfunctioning inducing software*, which they proposes as a definition for malware. Starting from a pragmatic taxonomy that classifies malware into four ground types, they recast these definitions in the form of conceptual formulations.

For instance, the definition of type 0 malware is recast from "an attack of the system limited to monitoring activities and possibly data leaks, without inducing any interruption of the system's functionalities" to a malware that incurs in "no compromise of target system functionalities (no side effects, no dysfunctioning) but production of additional, unintended functionalities (misfunctioning)".

Detailing Primiero's ontology of malware goes beyond the scope of this thesis. Yet, once again the basis for his ontology is oversimplified as it considers only software that "enacts the exploitation and installation phases to attack the system"; existing malware types can nonetheless have other purposes that are not reached by this definition (e.g. ransomware, adware). Yet, creating a correspondence between malware and malfunctioning artifact is a major contribution of this work.

A Realistic Ontology of Malware Seemingly, no previous work in literature attempted to propose a malware ontology in terms of a *realistic* ontology [15].

Starting from the premise that malware must be a special type of software (i.e. a malicious type), like any regular software, malware also embody the duality of *software application* and *computational function*. The difference between malware and regular software resides precisely in the facet of *computational function*, which involves an essential end-directed activity that is *malicious* (in the case of malware). As for the *software application*

14. Floridi *et al* use the term "token" in the paper.

15. The only reference found is a project by Timothy Schuler on Ontological Engineering at University at Buffalo, where an ontology for malware classification is presented. Link to the YouTube video: https://www.youtube.com/watch?v=22J6c0sPgHw.

aspect, there is no reason to distinguish malware from regular software. Consequently, malware can be defined as a software whose the computation function fulfills a *malicious* purpose.

Software Misfunction According to the reasoning above, malware is a software whose essential end-directed activity is malicious. This definition implies the notion of *malice*, i.e. the deliberate intent of the software creator to cause harm to another party.

However a cyberattack may occur without any software purposefully created with malice. Indeed, a weak email password that is cracked by an attacker, or a backdoor that is obtained through a (legitimate) remote access software, are examples in which cyberattacks take place without involving malware. In both cases regular software are *abused* by an attacker who acts with malice.

Being an artifact, software has the inherent property of possibly *misfunctionning* (as defined by Floridi *et al* [92]). This concerns any software artifact and can be exploited in the course of a cyberattack, thus it can be claimed that any software bears a risk of being abused by a threat actor. In BFO language, this scenario can be described as a software that takes up the *role* of malware through the abuse of a misfunction in the course of a cyberattack.

As a consequence, in principle malware analysis cannot be restricted to (outright) malware only. It must instead deal with any kind of software. Beyond the detection and classification of outright malware —which is sometimes very straightforward, for cases like *ransomware*—malware analysis should embrace a more refined analysis of software in general, including those that are not malware (a.k.a. *cleanware* or *goodware*).

From a risk-based standpoint, while it is clear that malware are placed at the peak of the (computer security) risk spectrum, malware analysis should also strive to place goodware in this risk spectrum. Therefore, understanding commonalities between goodware and malware instances is essential to the success of malware analysis.

1.3 An Epistemology of Malware Research

> *One of the goals of scientific theorising is to develop concepts which are adequate to the phenomena under study. In my view, things should work the same way in epistemology. We want to know what knowledge actually amounts to, not what our folk concept of knowledge is, since, just as with our pretheoretical concept of acidity, it might contain all sorts of misunderstandings and leave out all manner of important things.*
>
> Hilary Kornblith

Epistemology comes from the Greek words *epistéme* and *logos*, where the former can be translated as *knowledge* (or "understanding", or "acquaintance") and the latter can be translated as "reason", or "study of". The goal is to understand the ways of knowledge and how we can gain knowledge. Historically, epistemology deal with the fundamental questions like [309]:

— What is knowledge, and what do we mean when we say that we *know* something?

— What is the source of knowledge, and how do we know if it is *reliable*?

— What is the scope of knowledge, and what are its *limitations*?

Here we are interest in analyzing theses questions in the context of malware research. However, before any attempt to contemplate knowledge through the prism of any epistemology of malware analysis, it is important to settle a basic understanding of these topics in a conceptual sense. Still, we do not intend to provide an extensive introduction about epistemology as treated in Philosophy; instead, we want to put forward the notions that were developed throughout centuries of debate, which can shed light on tacit assumptions presented in the methodologies of many studies. For a more comprehensive introduction, a curious lector may be interest in reading [272] and [128].

1.3.1 Background

The aforementioned fundamental epistemological questions are crucial to understand how to shape knowledge and how to wittingly apply it to tackle practical problems (e.g. malware analysis). Indeed, these are very intricate questions that can be barely untangled.

This short background on epistemology intends to provide elements to unravel some important aspects of epistemology that clue in on answers to the fundamental epistemological questions.

What is knowledge? In the study of knowledge, the very first objective is defining what type of knowledge interests epistemology. One can know how to play guitar, but not know anything about music theory or how a guitar is built; the opposite may also occur, meaning that one can know everything about music theory and guitar fabrication, but be unable to play music on a guitar. The former type of knowledge is related to procedural knowledge ('i.e. 'knowing-how"), whereas the latter is related to propositional [16] knowledge (i.e. "knowing-that"). Furthermore, "knowing-that" [17] should be distinguished from "knowledge by acquaintance" (i.e. "knowing-of") [18], where the former is expressed in a declarative sentence or an indicative proposition (e.g. "Alice knows that Bob is a musician"), and the latter is constituted by familiarity with or direct awareness of the subject (e.g. "Alice knows Bob"). Epistemology is fundamentally interested in the type of knowledge related to propositional knowledge —henceforth implied in the use of the term "knowledge".

A very influential epistemological formulation is the notion of knowledge as *justified true belief* (known as JTB analysis), around which many philosophical debates gravitate. With aim of unpacking this notion, it is defended by many theorists —with many objections in many aspects by many other theorists —that JTB analysis holds because of the following arguments:

> *The Truth Condition*: Most epistemologists consider that one can only know propositions that are true; otherwise they cannot be, or express, facts, and therefore they cannot be known. Here the notion of Truth is metaphysical, as opposed to epistemological: "truth is a matter of how things *are*, not how they can be *shown* to be" [128].

16. A proposition is a sentence capable of being true or false. For instance, the sentence "look at the sky" is not a proposition, whereas the sentence "the sky is blue" is a proposition.
17. Associated with Latin word *scientia*, that originates the French verb *savoir*.
18. Associated with Latin word *cognitio*, that originates the French verb *connaitre*.

The Belief Condition: The belief condition is slightly more controversial than the truth condition [128]. The motivation for this condition leans upon the claim that you can only know what you believe, and failing to believe in something precludes knowing it. For the JTB analysis, "belief" means *full* belief, or *outright* belief. To believe *outright* that some proposition holds it is not enough to have a pretty high confidence, it is something closer to a commitment or a being sure.

The Justification Condition: True beliefs are not enough to establish knowledge. Alice can outright believe that it is currently sunny in Copacabana, without being there nor having any weather information about this location, and it can actually be that it is sunny there at this very moment. As such, Alice's proposition is a true belief but this proposition does not qualifies as knowledge, since it is the mere outcome of a lucky guess [19]. The role of justification in the JTB analysis is to ensure that luck does not play a primary role in producing true belief. (True) knowledge in some epistemic sense must be proper or appropriate, hence *justified* [128].

Besides criticisms regarding the JTB analysis (that goes beyond our scope), a major pitfall of this approach is assessing whether these conditions are met. For instance, how can one assess that the truth condition for a given proposition is met? What is the metaphysical nature of truth? Can we ever expect to reach Truth? If not, does it preclude the creation of knowledge? In the remainder of this section we present some theories that can help to scrutinize the attainability of each condition.

Theories of Truth There are a multitude of theories addressing the "problem of truth", stated as: what truths are, and what (if anything) makes them true. The most significant ones for the contemporary philosophical literature include the *correspondence*, *coherence*, and *pragmatist* theories of truth [103]. An elementary summary of these theories is provided below:

The Correspondence Theory of Truth: An ontological thesis is at the core of this theory: a belief is true if there exists an appropriate entity —a fact —to which it corresponds. If there is no such entity, the belief is false [103]. Here, facts are composed of particulars and properties or relations or universals. More importantly, facts are entities in their own right that exist in the world. Therefore, propositions are true when they *correspond*

19. Some epistemologists consider the sole factor of "true belief" a form of *weak* knowledge.

to a (metaphysical) fact that exists in the world. The truth of the proposition "the sun shines in Copacabana" denotes that the fact <sun, Copacabana, now> currently belongs in the world, while the contrary fact <rain, Copacabana, now> does not.

The Coherence Theory of Truth: To this theory, a belief is true if and only if it is part of a coherent system of beliefs [103]. Thus, a proposition is true if it is subsumed in a belief of a system (or "significant whole"). For the coherence theory, truth is a content-to-content (or belief-to-belief) relation —which is different from the neo-classical correspondence theory, where truth is rather a content-to-world relation.

The Pragmatist Theory of Truth: A very different perspective to truth is taken by pragmatist theorists. To pragmatists, truth is what "is useful to believe and has practical values in our lives". Simply put: "truth is what works". Another important disposition of this theory is the tentative nature of truth. A frequently-quoted passage in Charles Peirce's "How to Make Our Ideas Clear" (1878) asserts [41]: *All the followers of science are fully persuaded that the processes of investigation, if only pushed far enough, will give one certain solution to every question to which they can be applied. (...) The opinion which is fated to be ultimately agreed to by all who investigate, is what we mean by the truth.*

All these theories are subject to critics. The major pitfall related to the *Correspondence Theory of Truth* relates to the critic that one cannot experience reality firsthand *as it is*; instead one always experience *through* (potentially deceptive) sensors —even with respect to the "personal experience", which is mediated by the individual's sensory organs and mind. As for the *Coherence Theory of Truth*, its main issue relates to the possibility of constructing a theory that is (intrinsically) a coherent theory, but actually biased. With respect to the *Pragmatist Theory of Truth*, the main critics is that ideas that "work" may not necessarily match with the common understanding of truth.

Justification: Internalism vs. Externalism According to the JBT analysis of knowledge, when deciding whether a proposition may be accounted as knowledge, the proposition must successfully observe the justification condition. Nonetheless, nothing is said about what makes a justification valid. What makes a belief properly justified? What types of evidences are relevant in determining justification? What counts as evidence for justifying a belief? In a nutshell, what is evidence?

Evidences are the constituent of a *knowledge base*. They can be obtained through per-

ceptual, introspective, memorial, and intuitional experiences, or resorting to reliable source, where a reliable source is one that tends to result in mostly true beliefs [272]. Toward this end, two main schools of thoughts exist with different views on what counts as evidence: *internalism* and *externalism*.

According to the internalist view, the justification has to have *access* to internal evidential reasons of a justified belief. It means that upon reflection one must be able to *recognize* the justification for holding a certain belief. According to the externalist view, justification is the result of an attitude that prevents beliefs from being accidentally true; even if one cannot recognize how the belief itself is justified. For externalists, justification is *truth-conducive*.

To illustrate the differences between the internalism and externalism views, let us imagine that two people are trying to predict the birth date of a baby. Person A believes that the delivery date will be within 130 days, based on the results of clinical examination, ultrasonography, and human chorionic gonadotropin pregnancy tests. In turn, person B believes that the delivery date will be within 130 days because it will be the first full moon after the 9th month of pregnancy. Both are justified in their beliefs: person A adheres to scientific knowledge about estimation of delivery date and the outcome of direct examination, whereas person B adheres to cultural creeds that say that full moons are auspicious to childbirth. Both may turn out to be correct if the child actually is born within 130 day. In this case, externalists (may) admit that both people *knew* that the birth date would be within 130 days, because both of them held a justified true belief. However, internalists do not accept the justification of person B for the estimation of the birth date, because there is no clear-cut causal relation between the baby in the womb and the Moon; therefore internalists would say that only person A knew the birth date of the child.

The epistemological difference between both justification is that the former one is a ***doxastic*** *justification*, whereas the latter is a ***propositional*** *justification*. They are defined as follows: "One has propositional justification when one *has justification* for belief in a proposition (i.e., when one possesses good reasons, evidence, or justification to believe a proposition). One has doxastic justification when one not only has justification to believe a proposition *but also believes* the proposition and believes it at least partly on the basis of good reasons, evidence, or justification one has." [120]. According to this definition, *doxastic justification* sets stronger requirements than *propositional justification* to be fulfilled; on the other hand, one does not always has access to internal evidential reasons of a belief, which make this requirement much harder (sometimes impossible) to be met.

Two more basic concepts are important to nuance the internalist and externalist think-

ing: evidentialism and reliabilism. Evidentialists value the possession of firsthand evidences for justifying beliefs; for instance, a "experientialist" version of evidentialism would only count firsthand experience as evidence for justification. On the contrary, reliabilists value reliable sources (e.g. experiments) for justifying beliefs, where a source is reliable just in case it tends to result in mostly true beliefs. Evidentialism is typically associated with internalism, and reliabilism with externalism [272].

Returning to the example of the childbirth date prediction, let us admit by a fact that the full moon indeed does not have any interference in the pregnancy whatsoever. This scenario boils down to the *Gettier problem*, since "neither the possession of adequate evidence, nor origination in reliable faculties, nor the conjunction of these conditions, is sufficient for ensuring that a belief is not true merely because of luck" [272]. It means that holding justified true belief may not always be sufficient to knowledge. Detailing the implications of the Gettier problem and all the theories of knowledge that emerged thereupon is beyond the scope of this thesis; yet, the JTB analysis is a good starting point to reach a better understanding of knowledge.

The Regress Problem and the Structures of Knowledge In the traditional justification method, evidences are required to hold a belief as knowledge. However, evidences are constituent of some knowledge base, thus being themselves also knowledge that must be justified. This inference relationship engenders a recursive link of reason for reason, which is known as the "regress problem".

To address the regress problem, epistemologists conceived four possible structures of knowledge:

Foundationalism: according to this theory, there are certain beliefs that are "self-evident truths", or *basic*, which are exempted from any justification. There are supplementary beliefs (i.e. *nonbasic* beliefs) that need to be justified, having a proper *basing relation* [150] in the the steps of a justification that backtracks down to basic beliefs. To foundationalists, the structure of knowledge is like a building, consisting of a superstructure of nonbasic truths that rests upon a foundation of basic truths [272], thus having a ground level where the regress problem reaches an end.

Coherentism: coherentists reject the idea that there are any basic beliefs [272]. To them, justified beliefs receive justification from other beliefs in their epistemic neighborhood. The more these beliefs dovetail in a coherent way (i.e. without contradictions), the better

justified they are. To coherentists, the structure of knowledge is like a web, where the strength of any given area depends on the strength of the surrounding areas [272], thus having loops where the regress problem gets trapped.

Infinitism: infinitists claim that that infinite evidential chains can provide justification to their members [58]. Infinitists do not consider the regress problem incompatible with the building of knowledge.

Skepticism: Skeptics reject all the other epistemological structures, which leads to the conclusion that (true) knowledge is impossible. Therefore, to skeptics it is indeed impossible to deal with the regress problem.

The most popular school of thoughts among epistemologists is the foundationalism, followed by coherentists. Infinitists appeared recently in the debate and is still in development. Skeptics is the least adopted school of thought.

This short background on epistemology provides some clues to untangle the aforementioned fundamental epistemological questions: Knowledge can be thought (despite limitations) in terms of the *JTB analysis*; the source and scope of knowledge depends on the *Structure of Knowledge* adopted, which sets the conditions and limitations for knowledge building; finally, the *reliability* of knowledge depends on the requirements for *justification*, which must be met in order to validate the constitution of knowledge. These concepts will guide our subsequent epistemological discussion about malware research.

1.3.2 Epistemological Matters of Cybersecurity

Epistemological Diagnosis of Current Malware Research In this section we elaborate a concise epistemological diagnosis of malware research. The epistemological diagnosis aims to identify how knowledge is structured in malware research [20].

Botacin *et al* define *malware research* as "a term used in the literature to describe a wide field of work that embraces multiple goals" and they provide an extensive list of research objectives in malware analysis (e.g. malware detector, family classifiers, honeypots, etc) [29]. For our diagnosis, the marker to identify the *malware researches* of interest is the use of instances of malware in some part of the experimental process. This criterion encompasses an overwhelming majority of malware-related studies —not to say all practical ones.

20. Our allusion to *malware research* is intentionally open, along the lines of several related works whose subject is the meta-study of scientific methods and methodologies existent in the literature.

Notwithstanding the methods and methodologies [21], these studies have to inevitably confront a bootstrapping problem: software instances recognized as malware —or alike, e.g. belonging to a malware family —are needed in order to build and/or evaluate a malware analysis systems.

As a rule, this problem is addressed through the prior constitution of a set of annotated software instances that are assumed to be *known malware* (or malware family, or malware type, etc), which are designated as **ground truth**.

Analyzing prior works, Kantchelian *et al* identified four main approaches to assign ground-truth labels to datasets [140]: 1) label data manually, 2) use labels from a single source, 3) use labels from a single anti-virus vendor, and 4) use labels from multiple anti-virus vendor.

In case 1, software instances are analyzed by experts in order to (manually) assign them to ground-truth categories.

In case 2, instances are taken from public repositories (e.g. Anubis, VX Heavens, Wildlist Collections and Malfease dataset [22]) that have their own methods for assigning instances to ground-truth categories [23].

In case 3, the labels come directly from the anti-malware products which, despite working as black-boxes, have technical documents alluding to a combination of machine learning algorithms and expert knowledge in the analysis [154]. Botacin *et al* summarized this case as a "combination of all aforementioned techniques [pattern matching signature, heuristics and machine learning] in their detection engines", concluding that "their detection rates and labels are biased by all these factors at the same time". [30]. Case 4 is merely an extension of case 3 [24].

Examining the aforementioned approaches through the prism of the epistemological regress problem, we almost invariably get to a situation where the human analysis process takes the lead, with few exceptions (e.g. Anubis). Even in the case of machine learning algorithms, depending on the approach taken, ground knowledge may be traced back to human analysis. The process to produce labels in which the assignment is the outcome of a firsthand human analysis —or derived through some rule derivation process (e.g. pattern matching signatures) —can be biased due to the inherent subjectivity of human analysis.

Epistemologically, when studies in malware research use ground-truth primordially de-

21. *Methodology* refers to the overarching strategy and rationale used to address a research problem, whereas *methods* refers to specific algorithms and procedures used to collect and analyze data.

22. Listed here as in the paper [140], even though some of them do not exist anymore.

23. For instance, Anubis used the clustering method proposed by Bayer et al. [22], whereas VX Heavens gradually builds an inventory of malware samples gathered from reports of security companies.

24. Case 3 can also be seen a particular case of case 2.

rived (directly or indirectly) from human analysis, the manual analysis is tacitly assumed as being *true* (along the lines of the Pragmatist Theory of Truth), product of an *outright belief* (of the analyst) and exempted from any further *justification* (along the lines of the propositional justification).

Such ground-truth is predominantly adopted by our scientific community as *basic knowledge* upon *nonbasic knowledge* (i.e. new methods and methodologies for malware analysis) are built. Thus, we can claim that *Foundationalism* is the prevalent epistemological *structure of knowledge* in malware research.

Pitfalls of externalist justification for malware research When a study takes ground-truth as an opaque value that underpins an evaluation process without carefully identifying its relationship with the observables, the study takes an *externalist* approach. For as long as the outcome of the evaluation process is considered positive —and reliable, for reliabilists —the study is considered (externally) justified.

This is a practical approach, because it is not always possible to recognize a direct link between observables and the ground-truth, while it is still possible to measure their correlation. However, this entails some hazards that must be taken into account.

For instance, in a study conducted at Harrisburg University, researchers developed a software for automated computer facial recognition that was allegedly capable of predicting whether someone is prone to be a criminal [25]. According to this research, the software could "predict if someone is a criminal based solely on a picture of their face", with "80 percent accuracy and with no racial bias". This study was heavily criticized by a coalition of 2425 professors, researchers, practitioners, and students in the fields of anthropology, sociology, computer science, law, science and technology studies, information science, mathematics (and more), who released a public letter to debunk this study [26], pointing weaknesses on the adopted premises.

Other examples of similar scientific blunder are the 2016's *"Automated Inference on Criminality Using Face Images"*[27] by Xiolin Wu and Xi Zhang at Shanghai Jiao Tong University or the 2017's *"Deep neural networks are more accurate than humans at detecting sexual orientation from facial images"*[28] by Yilun Wang and Michal Kosinski at Stanford University.

25. Link to the press release announcing this publication: https://archive.md/N1HVe#selection-1593.69-1593.197
26. Link: https://medium.com/@CoalitionForCriticalTechnology/abolish-the-techtoprisonpipeline-9b5b14366b16
27. Link: https://arxiv.org/pdf/1611.04135.pdf
28. Link: https://osf.io/zn79k/

(a) Three samples in criminal ID photo set S_c.

(b) Three samples in non-criminal ID photo set S_n

Figure 1. Sample ID photos in our data set.

Figure 1.3 – Example of flawed external justification

The former proposes an "automated inference on criminality based solely on still face images, which is free of any biases of subjective judgments of human observers", "achieving 89.51% accuracy" (according to their ground-truth) —criticized by engineers form Stanford and Google [29] —the latter uses deep neural networks to determine the subjects sexual orientation solely based on their faces, achieving accuracy rates of 91% and 83% when distinguishing sexual orientation for men and women respectively (according to their ground-truth) —which was also subject of critics [30].

All these examples fall into a practice known as *Physiognomy*, which was completely abandoned by the scientific community and is understood nowadays to be a pseudoscience (with deep racist traits) [270]. Beyond the ethical issues, note that the culprit does not reside in the adopted methods (as *Techne*), but instead in the basal knowledge on which the methodology (as *Episteme*) operates.

At least in hypothesis, some approaches used in malware analysis can potentially sustain the same malformation of *Techne* and *Episteme*, which can be very difficult to spot. Despite not engendering the same ethical implications, this circumstance must avoided (or acknowledged as a threat to validity) by methodologies adopted in malware analysis.

29. Link: https://medium.com/@blaisea/physiognomys-new-clothes-f2d4b59fdd6a
30. Link: https://qz.com/1078901/a-stanford-scientist-says-he-built-a-gaydar-using-the-lamest-ai-to-prove-a-point/

Security as Social Knowledge Ground-truth is a far-reaching problem in malware research. Even when the epistemological stance concerning ground-truth is left aside, studies struggle with issues like inconsistency (i.e.multiple ground-truth sources providing contradictory labels), instability (i.e. labels change over time), delay (i.e. labels may not be readily available when data is collected), etc.

A standpoint that helps understanding the overwhelming complexity in establishing basic knowledge (i.e. ground truth) in the field is to consider security as a *social* discipline. Schaefer firmly claims "(computer) security is social"[253], as there is no "act of nature" (e.g. law of physics, molecular binding, etc) ruling data; on the contrary, they are representation of *ideas* instilled through custom, culture and law, by education and training, hence social.

Scharfer elaborates on the essences of security; for him, security is a "risk tradeoff" between greater or lesser *control* for better or worse consequences. This view outlines the nature of security as the rationale for *control*. In computer security this control is exerted on data, therefore the rule of thumb in this case is *differentiating data* (harmful data from harmless data).

In malware analysis this reasoning seems to hold as the final goal is to differentiate programs (i.e. data) into the categories harmful (i.e. malicious) and harmless programs. We may also be interested in many other intermediate goals (e.g. variant detection, malware families, etc) but they are auxiliary to the ultimate goal of protecting computer systems, thus precluding the execution of malicious programs whenever possible.

This social perspective highlights a pitfall in malware research related to the creation of ground-truth, which excessively depends on (too few) experts that are in charge of label creation. Other strategies hinging on methods and methodologies borrowed from social sciences (e.g. internal validity, external validity, construct validity, and statistical conclusion validity) could be applied for creating ground-truth, specially when it comes to "qualitative data" related to malware.

1.4 An Axiology

In the previous sections we examined several important issues from different perspectives that underlie malware research. Our discussion about the ontology of malware provides a fundamental analysis about the nature of malware; our discussion about the epistemology of malware research aims to identify how the prior knowledge about existing cyber threats influences the construction of new knowledge about new threats, scrutinizes the pitfalls of

certain justification strategies and determines the social dimension of (computer) security.

These topics, although all-important, are almost always implicit in studies about the methodologies of malware research. We take another approach; by developing the discussion about these topics we are able to firmly ground our *research paradigm*, namely:

Axiology-1 pursuing no ontological difference between malware and software at the "realization level";

Axiology-2 being more coherent driven;

Axiology-3 being more doxastically justified;

In accordance to item **Axiology-1**, we do not seek any malware trait that is manifested at the "realization level" (e.g. a sequence of bytes that is inherently malicious). We understand that the malice in software is revealed within a context of use, which can even denature software that were not designed to be malicious —hence not malware, by definition —into acquiring the role of malware depending on this context.

However, we do assent to the idea of *software birthmarks*, which are properties of software instances which provide a unique identity to them [203]. Software birthmarks allow to distinguish software instances according to the degree of commonalities shared among them. This shapes the notion of (software) families (that can happen to be malicious or not) and precludes the practical use of "malware types" (e.g. virus, trojan, etc) that emerged organically throughout history.

While the analysis of software birthmarks can be fully automated, the assessment of malice in software (or at least the degree it can be harmful) depends on some human judgement at some point in chain inasmuch as (computer) security is a socially-rooted activity. Nonetheless, automated analysis could leverage software birthmarks to propagate former analysis of software instances onto new similar ones, thus producing an amplification effect that can optimize the use of any necessary human effort dedicated to malware analysis.

Additionally, in our research paradigm we strive to steer clear of subjectivity inherent to human analysis by following items **Axiology-2** and **Axiology-3**. The intention is not to rebuild malware analysis from scratch ignoring any kind of human analysis. Instead, we want to bootstap from established research (which is predominately foundationalist-based) toward a more coherent-based knowledge justification in which we privilege observables that can be measured and are internal to the system over external ground-truth knowledge.

We still deem valuable and necessary to count on external knowledge to establish a ground-truth. In the evaluation process, in particular, external knowledge is very important as we

only have access to a tiny part of reality through our dataset. But our aim is to build methods and methodologies that lean more and more on data obtained from direct analysis, thus emancipating our analysis framework from external dependencies.

The goal is to shift the paradigm in which knowledge is propagated as a cascade, where past information propagate onto new information, to a new paradigm where software birthmarks are used as a network that connects software instances, carrying information from one end to another.

1.5 Conclusion

In this chapter we exposed our research paradigm, which entails a broader discussion about ontology and epistemology in malware research. This is necessary because malware analysis is a field highly driven by problem-solving thinking that has been evolving organically over the last decades.

Initially we adopt a twofold approach to establish an ontology of malware, i.e. a historical one and an analytical one. The historical-based approach reviews the history of how concepts emerged in the field, and then present several ontologies that were proposed following this perspective. The quick obsolescence of historical-based ontologies is a particular hitch of this approach, which happens due to the accelerated pace in which new threats emerge and new concepts are created in response. The analytical approach works through a more perennial reasoning, building a parallel between malware and software, in view of malware as a special kind of software (i.e. a malicious kind). This approach allows to identify the teleological function inherent to software, providing a basis for a definition of malware as software that have an essential end-directed malicious function. On the other hand, this approach also reveals that malware analysis cannot be restrict to (pure) malware, since software can misfunction and thus assume the role of malware.

Then, we provide an epistemological discussion about malware research. We recognize malware research as being predominately foundationalist, and we identify some pitfalls that need to be avoided. Furthermore, we take notice of (computer) security as a socially-rooted activity, which precludes the idealistic conceptualization of a framework that would completely exempt from any human intervention.

All identified limitations mold the pieces of a puzzle that compose our research paradigm. As a result, they allowed us to set up our axiology, which guides the methods and methodologies proposed all along this thesis.

CHAPTER 2

GENERAL BACKGROUND

2.1 Introduction

This section presents the general background with fundamental topics that situates the entire body of the thesis. Here we provide a literature review broadly introducing themes that are proper to the thesis. Greater details as well as the state-of-the-art for each topic are presented in the next chapters, as they are more relevant to each contribution.

This chapter is organized in two parts (machine learning and malware analysis) reflecting the two research fields that ground our study. This general background starts with a review about machine learning (section 2.2, page 57), then adresses malware analysis (section 2.3, page 87). This order is preferable since the latter is increasingly dependent on machine learning methods.

2.2 Machine Learning

Machine Learning is a branch of artificial intelligence that emulates the process of learning, aiming to give machines (or in fact IT systems) the ability to learn from previous information. Mohri *et al* give a broad definition of machine learning as "computational methods using experience to improve performance or to make accurate predictions" [195]. This definition entails the use of past information available for the learning process, i.e. the notion of *experience*.

In the classical computational approach, tasks are carried out by algorithms designed to mechanically solve specific problems. For instance, given a graph as input to the Dijkstra's algorithm, it can output the shortest path between any two given nodes. Machine learning takes another approach, as it leverages existent data related to the targeted problem in order to autonomously figure out solutions.

According to this approach, instead of finding the best solution through a task-oriented algorithm (e.g. Dijkstra's algorithm), data about the structure of the graph can be learnt by

a machine learning algorithm, which becomes able to output a short (perhaps the shortest) path between a pair of nodes provided as input. Arthur Samuel, a computer scientist who pioneered the study of artificial intelligence, defines machine learning as "the study that gives computers the ability to learn without being explicitly programmed" [115].

Despite the lack of guarantee that this output actually is the best solution —which can be understood at first glance as a handicap —experience shows that machine learning algorithms are in practice able to converge to (quasi-)optimal solutions, by improving the quality and size of the data available during the learning process.

Typically, the information used in machine learning is supplied in the form of data that are processed by agnostic algorithms, independently of the subject represented by the data. Conceivably, raw data can be enriched with additional meta-information —most often by human experts, referenced as "expert knowledge" —that can be leveraged during the learning process.

Machine Learning is currently used as an important tool in a plethora of areas, such as natural language processing, speech recognition, computer vision, among many others. Mohri *et al* express the practical motivation for using machine learning as follows: "The main practical objectives of machine learning consist in generating accurate predictions for unseen items and of designing efficient and robust algorithms to produce these predictions, even for large-scale problems" [195].

2.2.1 Definitions and Terminology

Machine Learning has its own vocabulary, which we define below and use hereinafter:

- Sample (or Data Points or Example): unit of data used in the algorithms.
- Features: set of attributes or characteristics associated to a sample.
- Labels: meta-data assigned to the samples.
- [Machine Learning] Model: an instantiation of a [Machine Learning] algorithm.

Machine learning instantiations are stateful due to the need of maintaining internal states that are updated during the learning process, hence the importance of referring to each instantiation as an individual *model*. Note that two models of the same algorithm differ because they may undergo different learning processes (by learning from different data, by randomness of internal states, etc), therefore the results obtained may differ in use.

Classes of Learning Problems in Machine Learning

The classes of learning problems addressed by machine learning include [195]:

- Classification: category assignment to unclassified items based items on priory classified.
- Regression: prediction of new data points based on a set of known data points.
- Ranking: ordering of items according to some malleable criterion.
- Clustering: partition of items into homogeneous regions.
- Dimensionality reduction or manifold learning: transformation of a given representation of items into a lower-dimensional representation which preserves properties of the initial representation.

In this thesis we are specially interested in the problems of *classification* and *clustering*; therefore, henceforth, we implicitly assume these problems as our use cases/contexts of machine learning.

Phases of Machine Learning Algorithms

Machine learning algorithms have to 1) learn from previous experiences and 2) make predictions. Therefore, as a rule, they comprise two main routines: *fit* and *predict*. This dichotomy, associated with the fact that in general learning (i.e. *fit* routine) and prediction (i.e. *predict* routine) do not take place concomitantly, implies the notion of *phases* when working with machine learning algorithms.

In the *learning phase* (also referred as *training phase*) the fit routine is evoked with some (learning) data corpus. Depending on the usage goal, the expected results of the subsequent prediction routine for the data corpus can be already known or not.

For instance, in the case of classification models the categories associated with the learning data are already known beforehand, whereas in the case of clustering there is no prior information about the regions shaped by the data points. In the former case, during the learning phase the goal of the model is to fit its internal states according to the learning data in order to reproduce predictions corresponding to the expected categories as much as possible; in the latter, the goal of the model is to fit its internal states so as to partition the space in regions according to the learning data points.

To measure the performance of models once the learning phase is complete, the *validation phase* (also named *test phase*) often benefits from the prediction routine. In this phase,

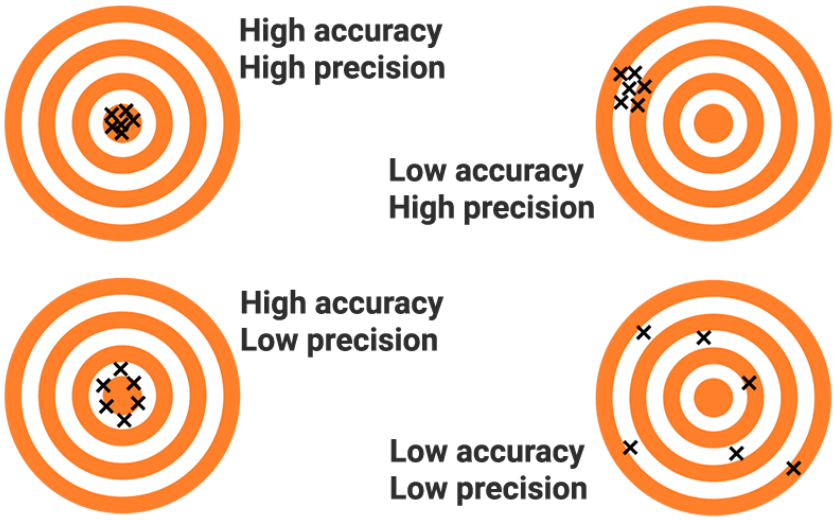


Figure 2.1 – Visualization of bias and variance errors as darts at a dartboard[2].

predictions on new data (i.e. data not used during the learning process) are made and the results are evaluated according to some *ground truth*[1].

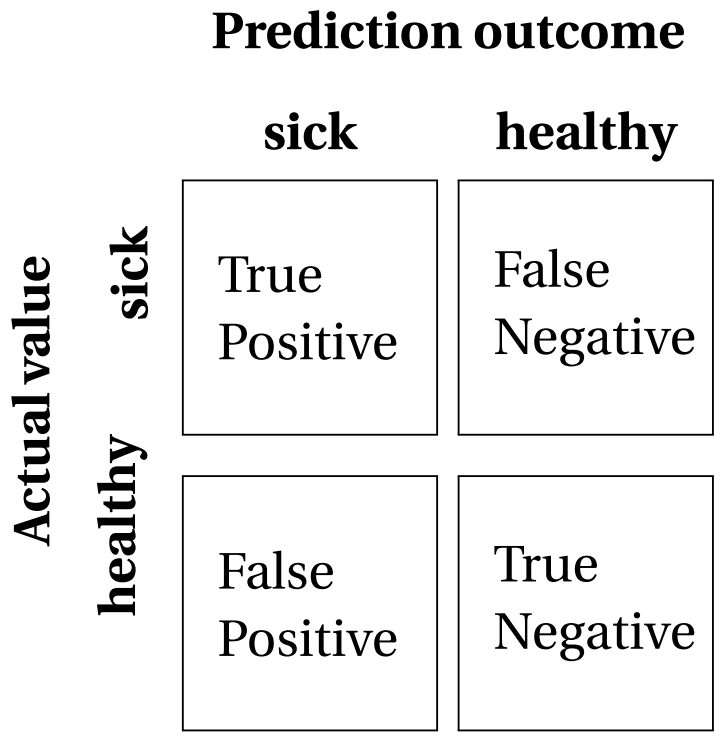
Making sense of this phase can be tricky because the expected responses for the test data are already known beforehand (i.e. in accordance with the ground truth); however, the test data is completely new to the model, since they were not used during the learning phase. The goal is, therefore, to measure how well predictions of the (trained) model can match with the ground truth.

Ground Truth-Dependent Evaluation Metrics

Several metrics commonly used when measuring models performance during the validation phase are ground truth-dependent. They allow to measure prediction errors and thus to evaluate how successful was the training of the model.

There are two fundamental causes for prediction errors: *bias* and *variance* [115]. The former refers to the type of error occurred when the predictions deviate from the expected values (i.e. ground truth); the latter refers to the type of error produced by the internal deviation of the predictions. A model that makes predictions with low bias is qualified as *accurate*, while a model with low variance is qualified as *precise*.

1. Establishing a ground truth involves epistemological issues that may be extremely challenging in practice. Refer to section 1.3.2 (page 50) for an extensive discussion of this topic in the context of cybersecurity.
2. Image published at https://www.mfg-space.com/wp-content/uploads/4-situations-of-your-CNC-machined-parts.jpg (accessed October 2022)



Several metrics are commonly used to measure the performance of models. A simple applied example is screening studies. When assessing a prediction model used to diagnose whether patients have a particular disease, we basically find the following scenarios: when the model correctly predicts that a sick person is sick, this prediction is a *true positive*; when the model correctly predicts that a healthy person is healthy, this prediction is a *true negative*; when the model wrongly predicts that a healthy person is sick, this prediction is a *false positive*; when the model wrongly predicts that a sick person is healthy, this prediction is a *false negative*.

With these four outcomes it is possible to measure the *accuracy*, *precision*, and *recall* of the model. For a set of N predictions, let TP designate the number of true positives, TN the number of true negatives, FP the number of false positives, and FN the number of false negatives. The model *accuracy (ACC)* is computed as follows:

$$ACC = \frac{TP + TN}{N} \tag{2.1}$$

In general, producing correct predictions (i.e. accuracy) is a primary goal of machine learning models. However, from equation 2.1 we observe that different models can attain the same level of accuracy in very different ways: for instance, a model which is good at

predicting that a patient is healthy but bad in predicting that a patient is sick —i.e. good TP and bad TN —may attain the same accuracy of a model which is bad at predicting that a patient is healthy but good in predicting that a patient is sick —i.e. bad TP and good TN.

Therefore, it is necessary to consider the *sensitivity* and the *specificity* of the model to better understand how true positives and false negatives interplay. The model *sensitivity* (or *recall* (R) or TPR, i.e. true positive rate) evaluates the rate of true positive predictions among the totality of positive predictions, whereas the model *specificity* (or *selectivity* (S) or TNR, i.e. true negative rate) evaluates the rate of true negatives predictions among the totality of negative predictions.

$$R = TPR = \frac{TP}{TP + FN} \qquad (2.2) \qquad S = TNR = \frac{TN}{TN + FP} \qquad (2.3)$$

We notice that the *specificity* of the model increases inasmuch as the rate of negative predictions grows. However, the number of negative predictions can be increased at the expense of producing false negatives —in the extreme case of a model that always outputs a negative prediction, the selectivity attains $S = 1$. On the other hand, *sensitivity* increases inasmuch as the rate of positive predictions grows, therefore *specificity* and *sensitivity* are mutually complementary measures.

Another supplementary notion is the *precision* (P) of the model, which measures the rate of correct positive predictions and is computed as follows:

$$P = \frac{TP}{TP + FP} \qquad (2.4)$$

Precision measures how good the positive predictions of a model are, while recall measures how good the model is in not missing any positive predictions. Both values can be combined in a unique score like the *F-score*, computed as the harmonic mean of precision and recall:

$$F = \frac{1}{\frac{1}{2}P^{-1} + \frac{1}{2}R^{-1}} = \frac{2}{P^{-1} + R^{-1}} = 2\frac{P \cdot R}{P + R} \qquad (2.5)$$

A more general definition of F-score is the F_β*-score*, which introduces a factor β to counterbalance the importance of precision or recall in the final value. It is defined as follows [3]:

3. The reason for using β^2 instead of simply β is because this parameter is derived from Van Rijsbergen's effectiveness measure which attaches β times as much importance to recall as precision, i.e. $\frac{\partial F_\beta}{\partial P} = \frac{\partial F_\beta}{\partial R}$.

$$F_\beta = \frac{1}{\frac{1}{\beta^2+1}P^{-1} + \frac{\beta^2}{\beta^2+1}R^{-1}} = (1+\beta^2)\frac{P \cdot R}{\beta^2 P + R} \tag{2.6}$$

It is trivial to notice that standard F-score is the special case of F_β-score in which $\beta = 1$.

Learning Paradigms

Three major learning paradigms are used in machine learning:

- *Supervised Learning* in which examples are provided along with labels at the learning phase. The fitting routine hinges on the classification of known examples to adjust internal parameters of the model in order to set up the prediction routine for the classification of new (unseen) examples.
- *Unsupervised Learning* in which examples do not have ready available labels at the learning phase. The fitting routine look for affinities between data points across the whole set of examples, creating groups of similar objects accordingly (referred as *clusters*) [4].
- *Reinforcement Learning* in which agents [5] deduct how to perform a certain action through a series of trial and error that receive rewards and punishments every so often.

Examples of applications of these paradigms include, respectively, algorithms that:

— can learn how to identify cats and dogs on pictures given a set of pictures (of cats and dogs) and their corresponding classes (i.e. labels indicating whether a given picture contains a cat or a dog). After the training phase, the model becomes able to predict the classes of new pictures (i.e. never before seen by the model);

— form clusters in which consumers with similar buying habits are put together according to their transaction history on a shopping website. These clusters allow to constitute a set of consumer profiles that buy on the shopping website.

— can train agents to play Atari games [193] through a numerous repetition of game playing while adapting strategies according to the result of each game instance —i.e. a reward whenever the strategy leads to a victory or a punishment whenever it leads to a loss.

4. Dimensionality reduction methods are another example of unsupervised learning, however we do not detail them here since they are not used in the remainder of the thesis.

5. In the context of reinforcement learning taxonomy, *agents* is the taxonomy equivalent to *models*.

In the context of this thesis, **we only use supervised learning and unsupervised learning** and, therefore, we only detail these two learning paradigms henceforth.

2.2.2 Supervised Learning

The main goals as well as the overall working principles of supervised learning were discussed above. In this section we provide a more detailed presentation of this topic, which includes the description of several important methods and the discussion of methodological concerns related to supervised learning.

Supervised Learning Methods

Here we provide a brief introduction of different classification methods used in supervised learning: Logistic Regression, Naive Bayes, Decision Trees, k-Nearest Neighbors (k-NN) and Neural Networks. The purpose here is to present some of the main strategies used in supervised machine learning, starting from methods originated from traditional statistics (e.g. Logistic Regression, Naive bayes), which fall under supervised machine learning as they leverage training data to fit a prediction routine, up to more recent methods (e.g. neural networks), which were developed for the purpose of supervised machine learning.

Method 1: Logistic Regression The logistic regression is used to solve the problem of binary classification. During the learning phase, the logistic regression method fits a *sigmoid function*[6] (σ) to the set of training data, hence the name *logistic regression*.

The learning routine is given as input a set of training data $\{X_i\}$ whose labels belong to one of two (mutual exclusive) classes: $\{X_{1,j}\}$ and $\{X_{0,k}\}$, where $\{X_{1,j}\} \cup \{X_{0,k}\} = \{X_i\}$ and $\{X_{1,j}\} \cap \{X_{0,k}\} = \emptyset$. In the statistical notation it means that $P(X_i) = 1$ if it belongs to the reference class (say, $X_i \in \{X_{1,j}\}$) or $P(X_i) = 0$, otherwise (i.e. $X_i \in \{X_{0,k}\}$).

The learning algorithm uses *logit* (the inverse function of the sigmoid function) to re-shift the coordinates system in such a way that the values of the sigmoid become represented by a line as shown in figure 2.2.

Then, the best fitting line is found through an iterative procedure that maximizes its total likelihood. Figure 2.3 shows how to obtain the values of the logit projection Y_i for the training data points. Since the computation of the likelihood depends on the class of the data point,

6. Details about the sigmoid function and some of its properties are provided in the appendix section 6.2 (page 303).

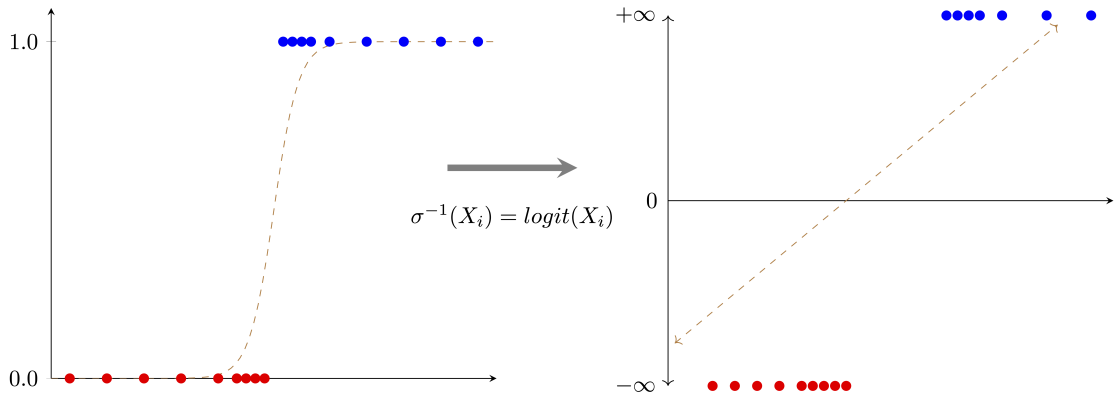


Figure 2.2 – Coordinate re-scaling using the *logit* function

the training data $\{X_i\}$ is split in two, for data points that do (in blue) and do not (in red) belong to the reference class (i.e. respectively $\{X_{1,j}\}$ and $\{X_{0,k}\}$).

The computation of likelihood of a data point then goes as follows:

$$P(X_i) = \begin{cases} \sigma(Y_{1,j}) & \ni X_{1,j} \rightarrow X_i, \text{if } X_i \in \{X_{1,j}\} \\ 1-\sigma(Y_{0,k}) & \ni X_{0,k} \rightarrow X_i, \text{otherwise} \end{cases}$$

Finally, the total likelihood of a candidate sigmoid curve is computed as $\sum P(X_i), \forall \{X_i\}$. The sigmoid curve that provides the maximum total likelihood (σ_β) is used in the prediction routine to output whether an unseen data point (Z) belongs to the reference class (i.e $\sigma_\beta(Z) \geq 0.5$) or not (i.e $\sigma_\beta(Z) < 0.5$).

Logistic regression is a *discriminative classifier*, because it directly computes the probability of a data point belonging to a class, i.e. the *posterior probability*. The predicted class ($\hat{c}$) is the one which maximizes $\sigma_{c,\beta}$ [7], i.e. $\hat{c} = \underset{c \in C}{\arg\max}\, \sigma_{c,\beta}(Z) = \underset{c \in C}{\arg\max}\, P(c|Z)$.

Method 2: Naive Bayes Naive Bayes is another method for binary classification. This method is based on the Bayes' Theorem, which we recall below:

Theorem 1 (Bayes theorem) *The posterior probability $P(H|E)$ of an hypothesis H being true given an evidence E, can be computed from the prior probability of hypothesis being true $P(H)$, the probability of seeing the evidence $P(E)$ and the likelihood of the evidence given the hypoth-*

7. The notation denotes a multiclass generalization, where C is the set of classes and $\sigma_{c,\beta}$ is the logistic regression for the class $c \in C$.

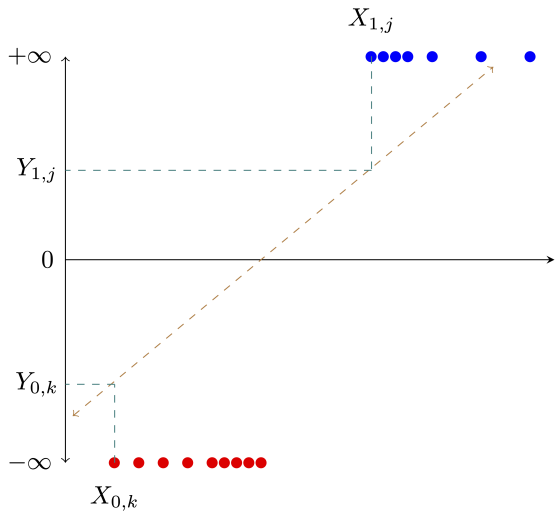


Figure 2.3 – Logit projects of two points of different classes

esis $P(E|H)$ *as follows*

$$P(H|E) = \frac{P(H) \cdot P(E|H)}{P(E)}$$

In this case, the classifier also selects the class which maximizes the posterior probability given a data point (Z). However, instead of directly computing the posterior probability of each class, this method used the Bayes' Theorem to switch to the analysis of *prior probabilities* and *likelihoods*.

$$\hat{c} = \underset{c \in C}{\arg\max}\, P(c|Z) = \underset{c \in C}{\arg\max}\, \frac{P(Z|c) \cdot P(c)}{P(Z)} \leftarrow \boxed{\text{Bayes' Theorem}}$$

Since the $P(Z)$ is constant independently of the class $c \in C$, the prediction of $\hat{c}$ can be simplified to:

$$\hat{c} = \underset{c \in C}{\arg\max}\, P(Z|c) \cdot P(c)$$

where $P(Z|c)$ is the likelihood of seeing Z in c and $P(c)$ is the probability of a random data point belonging to the class c.

Expanding the representation of Z in the form of a feature vector $Z = \{z_1, \cdots, z_n\}$, we have:

$$\hat{c} = \underset{c \in C}{\arg\max}\, P(z_1, \cdots, z_n|c) \cdot P(c)$$

The naive assumption assumes that Z features are independent, and therefore $P(z_1, \cdots, z_n|c) = P(z_1|c) \cdots P(z_n|c)$. This simplified further the computation of the prediction to:

$$\hat{c} = \arg\max_{c \in C} P(c) \prod_{i \in \{1, \cdots, n\}} P(z_i|c)$$

In Naive Bayes, the learning routine uses the training data to set up the probabilistic models that are used to compute each factor of $P(Z|c)$ and $P(c)$. This can be as simple as a frequency model, where:

$$P(z_i|c) = \frac{|\{z_i \in x, x \in c\}|}{|c|}$$
$$P(c) = \frac{|c|}{\sum_{c_i \in C} |c_i|}$$

Naive Bayes is a *generative classifier*, because it estimates the *prior probability* of $P(c)$ and the *likelihood* of $P(Z|c)$ for each class to select the one that maximizes the posterior probability of $\hat{c}$, instead of directly predicting the posterior probability for each class. Discussing limitations and optimizations of the Naive Bayes method is beyond the scope of this thesis.

Method 3: Decision Trees Decision Tree is a popular method used in multiclass classification. The classifier is organized in the form of a tree whose internal nodes (i.e. nodes that have children) represent classification criteria and leaves represent the prediction classes that fulfill all criteria accumulated in the path traversed all the way up to the root node.

Figure 2.4 shows a generic decision tree which includes four criterion with two conditions each (represented by $C_{i,j}$). If all criterion conditions are mutually exclusive and collectively exhaustive, the overall decision tree will also be so.

Given an example to be classified and conditions $\{C_{1,2}, C_{2,II}\}$, the prediction is `Class II`. Similarly, given conditions $\{C_{1,3}, C_{3,4}, C_{4,II}\}$ the prediction is `Class II`.

The learning routine is used to build the decision tree that better fits the training data by reducing the overall *impurity* or *entropy*. These measures are most commonly computed over a random variable Y which takes values $\{y_1, \cdots, y_n\}$, as follows:

$$Gini(Y) = 1 - \sum_i P(y_i)^2 \qquad \text{(Gini Impurity)}$$

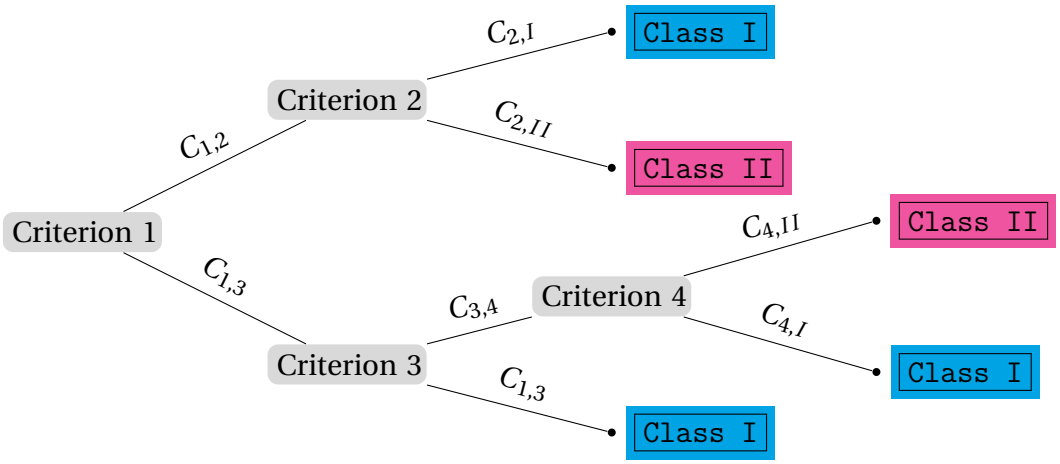


Figure 2.4 – Depiction of a generic decision tree.

$$H(Y) = -\sum_i P(y_i) log_2(P(y_i)) \qquad \text{(Entropy)}$$

A common fitting method starts by choosing the criterion that produces the minimal score (e.g. gini impurity or entropy) to be the root of the decision tree. Then, it gradually insert new levels in the tree by including new criteria such that the score of the overall tree is minimized as much as possible. For each path, if its score cannot be minimized by the inclusion of new criteria or if there is no criterion left to be included, the node splitting terminates. The final decision tree is the one for which there is no longer any node splitting to be done.

Method 4: k-Nearest Neighbors k-Nearest Neighbors (a.k.a. k-NN) is another method used for multiclass classification. Unlike the previous methods, k-NN requires the notion of distance between data points, thereby engendering the notion of coordinate points in a hyperspace.

During the learning phase, when parameter k is known the k-NN classifier does not need any kind of fitting, except on account of optimization. The training data contains a set of points $\{P_i\}$ and their corresponding classes, to which it is possible to create a mapping such that $\mathscr{C}(P_i) = c_i \in C$.

To predict the class of a unclassified data point P_x the k-NN classifier first finds the k closest (neighbor) points $\{P_{N_1}, \cdots, P_{N_K}\}$ to P_x, i.e. $d(P_i, P_x) \leq d(P_j, P_x), \forall\, i \in \{N_1, \cdots, N_K\} \wedge j \notin \{N_1, \cdots, N_K\}$. Then the prediction class is defined as the one which is the most present among

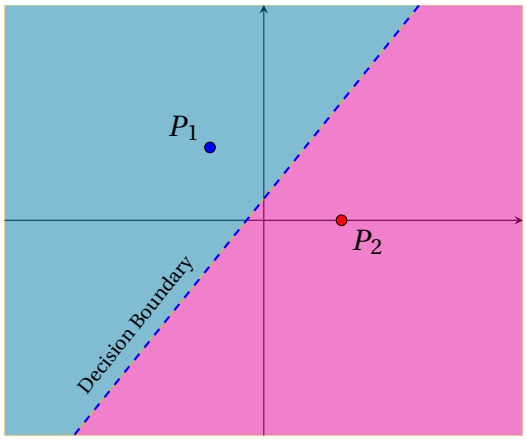


Figure 2.5 – Simple example of 1-NN with two points and their classification regions.

the nearest neighbors $\{P_{N_1}, \cdots, P_{N_K}\}$ of P_x, i.e.:

$$\hat{c} = \underset{c_i \in C}{\arg\max} |\{\mathscr{C}(P_i) = c_i, i \in \{N_1, \ldots N_K\}\}|$$

Since the prediction class depends exclusively on the location of a point in the hyperplane, it is possible to define regions corresponding to the class predictions. Figure 2.5 depicts the simplest case, where $k = 1$ and the training data contains only $\{P_1, P_2\}$. The frontier that separates two regions is called *decision boundary*, since the prediction changes when the boundary is crossed.

In practice, most of the times the optimal value for k is unknown, therefore the learning routine can test different values and choose the best one according to some given criteria. Further discussing special cases (e.g. tie in the number of leading classes among the k nearest neighbors), limitations and optimizations of the k-NN method is beyond the scope of this thesis.

Method 5: Neural Networks Neural Networks (NN) refer to models used in different supervised learning applications (especially for multiclass classification), which are built from special mathematical functions (referred as *artificial neuron*) that mimic the operating mode of biological neurons.

An artificial neuron follows the subdivision of a biological neuron, which comprises three main parts:

— the *dendrites* (a.k.a *dendritic tree*), which allow the cell to receive signals from the upstream neuron and transmit it down to the body of the neuron;

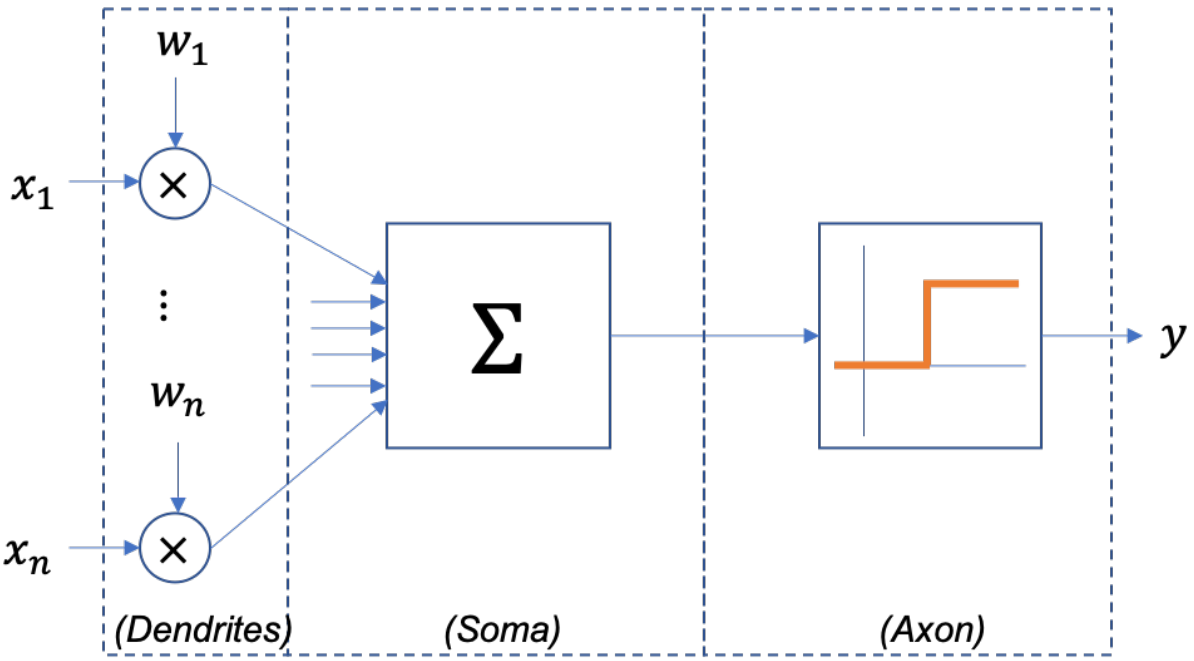


Figure 2.6 – Mathematical model of a neuron (a.k.a *artificial neuron*).

— the *soma*, which combines all the signals transmitted by the dendrites;

— and the *axon*, which is activated once a certain signal potential is reached, firing the transmission of a downstream signal.

Figure 2.6 shows the mathematical model of a neuron, and the respective subdivisions of a neuron cell that each part represents. The input values $x_1, \cdots, x_n \in \{0,1\}$ (that models the signals received by the *dendrites*) are multiplied by weights $w_1, \cdots, w_m$ (that model the synaptic connection between neurons), then they go through an adder (that models the *soma*) and finally the output value y is activated (i.e. $y = 1$) if the accumulated value goes beyond a threshold (τ), otherwise nothing happens (i.e. $y = 0$).

Basically, the neuron is activated if $\sum_{i=1}^{m} w_i x_i > \tau$. It is possible to simplify the mathematical notation to remove the use of an explicit variable for the threshold by setting $x_0 = -1$ and $w_0 = \tau$, i.e.:

$$y = \varphi(\sum_{i=0}^{m} w_i x_i)$$

where φ is known as the *activation function*, which is "activated" (i.e. outputs 1) if the input value is positive [8] . Various activation functions can be chosen, depending on the properties

8. The magnitude of the output for the activation function in the immediate neighborhood around 0 varies

sought in the application.

A Neural Network is composed of multiple neurons that can be interconnected in the most different ways, which is referred as the *architecture* of the neural network. They form an approximation function that fits an unknown function f given a limited set of images (Y) and preimages (X) thereof, i.e. $X = \{x^{[1]}, \cdots, x^{[m]}\}$ and $Y = \{f(x^{[1]}), \cdots, f(x^{[m]})\}$.

To measure how well the approximation succeeded in fitting the underlying function f (i.e. the learning performance) it is necessary to define an *objective function* that undergoes an optimization process for maximizing (or minimizing) it; the objective function is called *performance function* ($\mathscr{P}$) if the learning goal is to maximize it, otherwise it is called *loss function* ($\mathscr{L}$).

The *optimization function* depends on the prediction output of the neural network and the true value (i.e. $\hat{Y}$ and Y), while the prediction itself depends on X and on the internal set of weights of the neural network (W). Therefore the loss function $\mathscr{L}$ and the performance function $\mathscr{P}$ are both functions of X, W and Y.

A common method to minimize the *loss function* is the *Gradient Descent* —correspondingly, the analogous method for *performance function* maximization is called *Gradient Ascent*. It uses the geometric fact that the *gradient* of function (F) points towards its steepest descent, if F is continuous and convex. Thus, in order to iteratively progress towards a local minimum of F, the *gradient descent* computes:

$$a_{n+1} = a_n - \gamma \nabla F(a_n) \qquad \text{(Gradient Descent)}$$

where γ is the *learning rate*. For an adequate value of γ, the computation of the gradient descent results in a monotonic sequence such that $F(a_0) \geq F(a_1) \geq F(a_2) \geq \cdots$.

In the context of neural networks, gradient descent can be used in the training process to minimize the *loss function*. Each iteration of the learning process (referred as *epoch*) computes the gradient of the loss function with respect to W so as to find the set of values that minimizes the prediction errors. This part of the training method is usually referred as *back-propagation*, because the prediction errors obtained with the values W of the i-th epoch are used in the computation of new values for W used in the $i+1$-th epoch, i.e.:

$$W_{[i+1]} = W_{[i]} + \gamma \nabla_W \mathscr{L}(X, W_{[i]}; Y).$$

depending on its particular formulation.

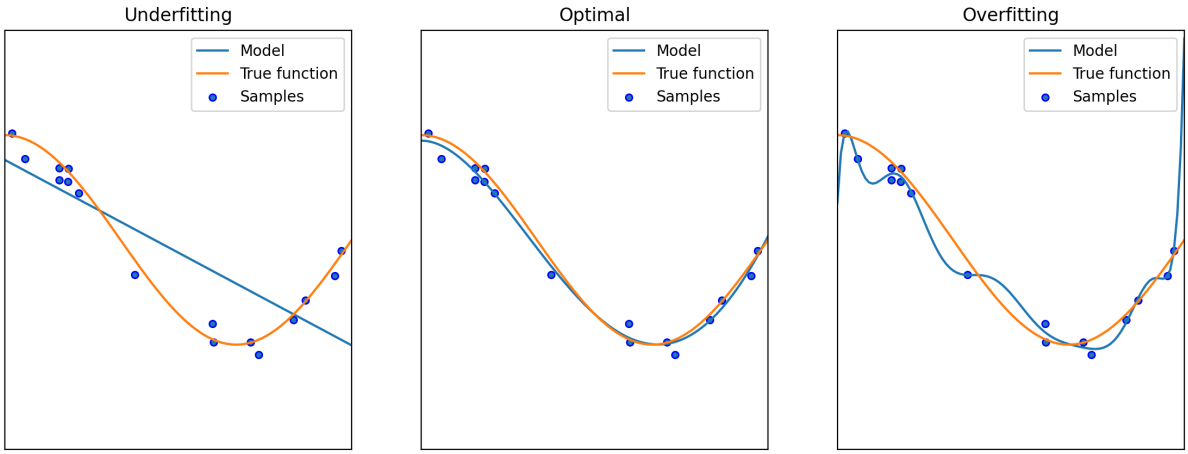


Figure 2.7 – Examples of underfitting, overfitting and optimal fitting.

Methodological Concerns of Supervised Learning

Some issues in supervised learning are not particular to any singular method, they are instead inherent to the bound between data and algorithm and thus being latent to any chosen method. This section discusses the issues of overfitting, selection bias and cross-validation, as they are pertinent to any supervised learning method.

Underfitting & Overfitting When a model is trained the goal is to set up its internal states so as to produce trustworthy predictions. Therefore, it is expected that a successfully trained model is able to 1) reproduce predictions of known data points —i.e. produce predictions that do not deviate too much from the expected results —and 2) produce good predictions for unknown data points, thus allowing the learning data to be generalized to unknown data.

Depending on the inner workings of the model, the trained model may come short in reproducing predictions of known data or in producing good predictions for unknown data (or both).

When the former case occurs the model *underfits* the data, since it fails to adequately capture the underlying structure of the data. When the latter case occurs but the model is nevertheless able to reproduce predictions of known data points (used in the training process), the model *overfits* data. Overfitting occurs because the model is overly sensible to residual variations (i.e. noise), thus it captures every slight deviation.

Figure 2.7 provides a visual representation of underfitting (on the left) and overfitting (on the right). For comparison, the figure also displays an optimal fitting (in the middle). It is possible to see that the underfitted model is unable to make correct predictions thus incur-

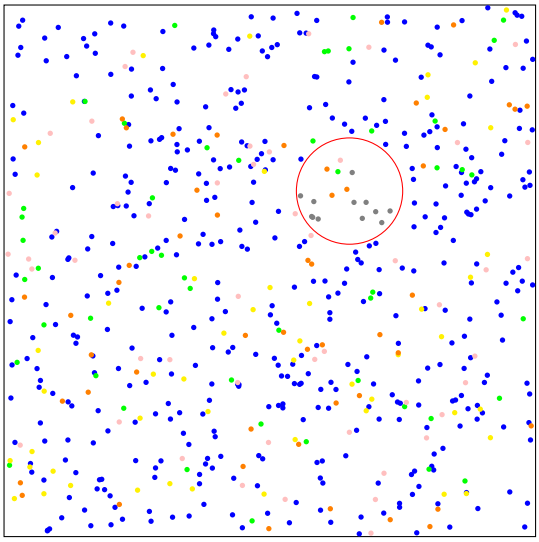


Figure 2.8 – Visual representation of selection bias.

ring a high *bias* error. The overfitted model has lower bias error, however it is excessively flexible in fitting the prediction function to the learning samples. Accordingly, intervals that lack learning data produce poor predictions. Therefore, overfitted models generally suffer from high *variance* error and cannot be generalizable to unknown data.

Selection bias Selection bias occurs when the learning data are not representative of the phenomena intended to be learnt and generalized. Since the data do not represent the general case, it carries an inherent bias error that spoils any attempt of learning from it.

Figure 2.8 shows the representation of a selection bias, where the population is misrepresented by the data points selected inside the red circle. Despite the majority of blue points in the overall population, the most frequent color within in the selected population is gray.

Cross-validation The effectiveness of supervised learning substantially depends on the quality of the learning data. However, in practice data are often not available in the quantity and quality required to properly train the models. To cope with this problem, it is common to validate models using resampling methods. One of these methods is cross-validation.

Cross-validation is carried out in multiple rounds where the entire data corpus available is partitioned into a learning and a validation sampling of the dataset. As the same data point can fall in the learning sampling in one round and the validation sampling in another round, there is no clear-cut distinction between learning and validation data when cross-validation is used.

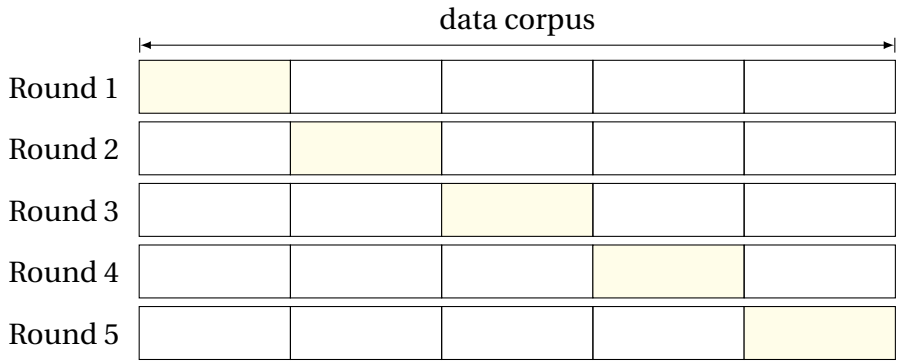


Figure 2.9 – Example of 5-fold cross validation.

In each round, the model is trained from scratch using the round's learning sampling and then evaluated with the round's validation sampling. Since for a given round, none of the data in the validation sampling is used for learning, this avoids problems related to overfitting. Furthermore, since the selection of the learning and validation sampling is done randomly, problems related to selection bias are avoided. Once all the rounds are completed, the results of all evaluations are averaged in order to produce a final overall evaluation result.

A popular method is the *k-fold cross validation*. In this method, the data corpus is split in k chunks and a total of k training and evaluation rounds take place. At each round, one of the chunks is chosen as validation dataset, with the condition that it was not previously chosen, and the other $k-1$ chunks are used to train the model. At the end, the k evaluation results are averaged to produce a single final result. Figure 2.9 shows an example of 5-fold cross validation.

2.2.3 Unsupervised Learning

The main goals and overall working principles of unsupervised learning were previously discussed; here we provide a more detailed presentation of this topic including the description of important methods and a methodological discussion.

Distance Metrics

All methods presented here are based on some metric which measures the similarity or dissimilarity between data points. This is actually necessary for the vast majority of existing clustering methods. The chosen distance metric largely influences the final result of the clustering methods.

The distance between x and y is denoted by $d(x, y)$ and must obey the fundamental properties of metric spaces, i.e.:

— $d(x, y) = 0 \iff x = y$

— $d(x, y) = d(y, x)$

— $d(x, z) \leq d(x, y) + d(y, z)$

Minkwoski Metric Given two n-dimensional data points $x = (x_1, \cdots, x_n)$ and $y = (y_1, \cdots, y_n)$, the Minkowski metric is defined as [237]:

$$d(x, y) = (\sum_i |x_i - y_i|^p)^{1/p} \qquad \text{(Minkowski metric)}$$

where $p \in \mathbb{N}^+$ is the order of the metric.

The most commonly used metrics are the Euclidean distance ($p = 2$), the Manhattan distance ($p = 1$) and the Chebyshev distance ($p \to \infty$).

Methods of Unsupervised Learning

Here we provide a brief introduction of different clustering methods used for unsupervised learning.

Hierarchical Methods Hierarchical clustering methods operate in multiple iterations in which the elements are joined together —or split apart —forming new cluster configurations with one less —or one additional —cluster at each round, hence the hierarchical notion. The goal is to follow a greedy strategy in which the cluster configuration of the n-th iteration is evolved into the optimal configuration of the $n+1$-th iteration through the merging of two clusters, or the splitting of a single cluster in two.

There are two main approaches for the hierarchical method: *agglomerative cluster*, which operates from the bottom up; and the *divisive cluster*, which operated from the top down.

The **agglomerative** approach starts by assigning each item to a cluster of a single element each (also known as *singleton*). It then proceeds by reducing the number of clusters by one at each iteration and by merging together the two most similar clusters.

Thereby, at each iteration a new hierarchical level is created. This continues as long as no cluster overruns the cut-off condition set as parameter —this value can be provided in terms of distance between elements of a cluster. Figure 2.10 shows a dendogram with the cut-off parameter set to 0.7, which results in two clusters (i.e. $\{a, b\}, \{c, d, e\}$).

The **divisive** approach is similar to the agglomerative, however instead of starting with individual clusters, it starts with all elements as part of a unique cluster. Each iteration then creates a new hierarchical level by selecting the pair of elements with highest dissimilarity (say x_1 and x_2) when considering the most dissimilar pair of element in each cluster; the cluster containing x_1 and x_2 is thus split into two new respective clusters, and the remaining elements are moved into the closest one of the new clusters (i.e. considering their distance wrt. x_1 or x_2).

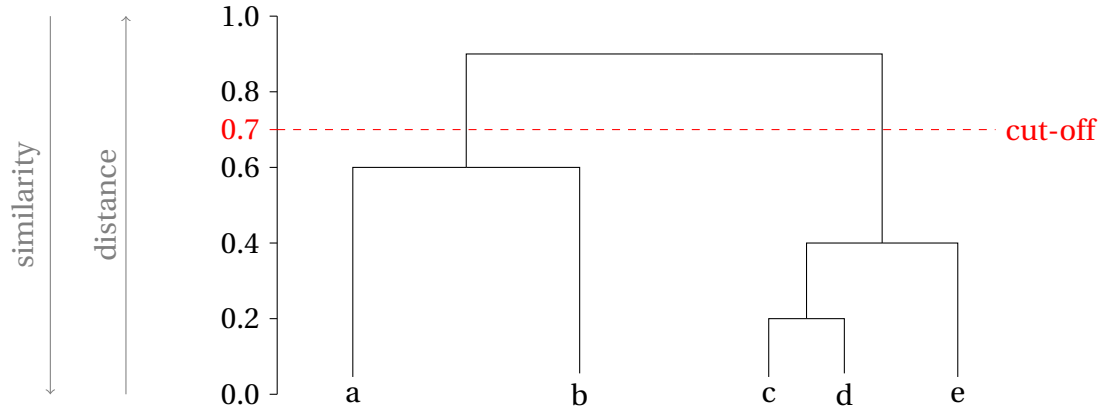


Figure 2.10 – Example of a dendogram with cut-off set to 0.3.

The measure of similarity or dissimilarity used in the merging/division process follows the chosen *linkage* method. The *single-link* clustering considers the distance between two clusters as the shortest distance between any two elements of both clusters. The *complete-link* considers the longest distance between any two members. The *average-link* considers the average of all distances between the elements of each cluster.

The main drawbacks of single-link and average-link are, respectively:

— Merging (or not splitting) two clusters due to outliers that are incidentally close (this is known as the *chaining effect* [237])

— Splitting of elongated clusters or merging of small clusters that surround elongated clusters.

In general, complete-linkage produces more compact clusters, whereas single-linkage produces more versatile clusters. Another general drawback of hierarchical methods is their complexity, which is too costly for medium and large datasets. For a set of n elements in total, the time and memory complexity of the standard agglomerative clustering are $\mathcal{O}(n)$ and $\Omega(n^2)$, respectively. The time complexity of the divisive clustering is $\mathcal{O}(2^n)$ due to an

exhaustive search of the most dissimilar element for each iteration, which is overwhelmingly costly.

Partitioning Methods Partitioning methods arrange a set of elements into a pre-determined number of clusters. The goal is to maximize the similarity of elements within the same cluster, while being constraint to a pre-set number of clusters. The partition methods algorithms generally follow an error minimization strategy, in which the distances between the elements and the cluster representative points are computed to be as close as possible.

The most popular partition method is **K-Means**, where K represents the number of clusters pre-set by the user. For K-Means, a cluster representative point is its centroid, computed as the mean coordinate of all the elements in the cluster, i.e.:

$$\mu_i = \frac{1}{|C_i|} \sum_j c_{i,j} \qquad \text{(Centroid of i-th cluster)}$$

where cluster $C_i = \{c_{i,0}, \cdots, c_{i,n}\}$.

In K-Means the goal is to find the optimal coordinates of K centroids. Once computed the optimal centroids, the K clusters are formed by assigning each element to the nearest cluster —measured as the distance between the element and each cluster centroid.

To find the optimal centroids, the naive version of K-means starts by randomly selecting an initial position for K reference centroids and then iterates through the following loop:

1. Assign each element to the cluster corresponding to the nearest centroid;
2. Recompute the centroid of each cluster, using the new computed values as the reference centroids in the next iteration.

In general, this loop iterates until the values of the reference centroids converge, i.e. when the new centroids recomputed in a new iteration are the same as the ones in the previous iteration. Other stop conditions can also be used, such as a pre-defined value for the number of iterations.

The time complexity of K-Means is $\mathcal{O}(T \cdot K \cdot N \cdot M)$ [237], where T is number of loop iterations, N is the total number of elements and M is the number of features per element. Given the quick convergence of K-Means in practice, its complexity is roughly linear with respect to N. Efficiency is thus one of the main advantages of K-Means.

The two main drawbacks of K-Means are the requirement of pre-selecting a fixed number of clusters and its sensitivity to the selection of the initial reference centroids, which can

often converge to local rather than global minima. Furthermore, K-Means is also sensitive to noisy data and outliers.

To cope with K-Means weaknesses many options for optimization exist, such as the Forgy initialization [117] and K-Means++ [13], which target the initial centroid selection, to mention just a few. There are also modifications of K-Means that use other representative values for each cluster instead of the centroids, such as the Partition Around Medoids (PAM), which uses medoids [9] to represent clusters, or the K-Medians, which uses the median of the elements instead of the mean as centroid.

Density Models Density models analyze the surrounding regions of each element to identify whether the point has many neighbors within this nearby region, thus denoting that the element is located in a high density region, or whether it is the opposite, indicating that the element is located in a low density region. Elements belonging to the same high-density regions are included in the same cluster.

A popular density-based clustering algorithm is DBSCAN (Density-Based Spatial Clustering of Applications with Noise). The algorithm is given the parameters ϵ and *MinPts* that specify the radius of the nearby region and the minimal threshold for the number of points within a region, and it classifies the points $\{P_i\}$ as *core points*, *(density)-reachable* and *outliers* (or *noise*) as follows:

— P_i is a *core point* if its nearby region of radius ϵ contains at least *MinPts* points (including P_i).

— P_i is *directly reachable* from P_j if $d(P_i, P_j) \leq \epsilon$.

— P_i is *reachable* from P_j if for every point in $\mathscr{R} = \{P_k\} \cup \{P_i, P_j\}$ there is another directly reachable point in $\mathscr{R}$.

— P_i is a *border point* if it is reachable from a core point but is not itself a core point.

— P_i is a *noise* point if it is neither a *core point* nor a *reachable* points.

Figure 2.11 shows the analysis of p_c (in blue), where the directly reachable points $P_{DR} = \{p_i : d(p_i, p_c) \leq \epsilon\}$ (in red) are within the circle of radius ϵ (in blue) centered in p_c and the remaining reachable points $P_R = \{p_i : \exists p_j \in \{p_c\} \cup P_{DR} \ni d(p_i, p_j) \leq \epsilon\}$ (in green) are within the circles of radius ϵ centered in P_{DR} (in dashed red lines). Noise points with respect to p_c, i.e. $P_N = \{p_i : \nexists p_j \in \{p_c\} \cup P_{DR} \cup P_R \ni d(p_i, p_j) \leq \epsilon\}$, are represented in gray color. If *MinPts*

9. The element of the cluster whose sum of dissimilarities to all other the elements in the cluster is minimal.

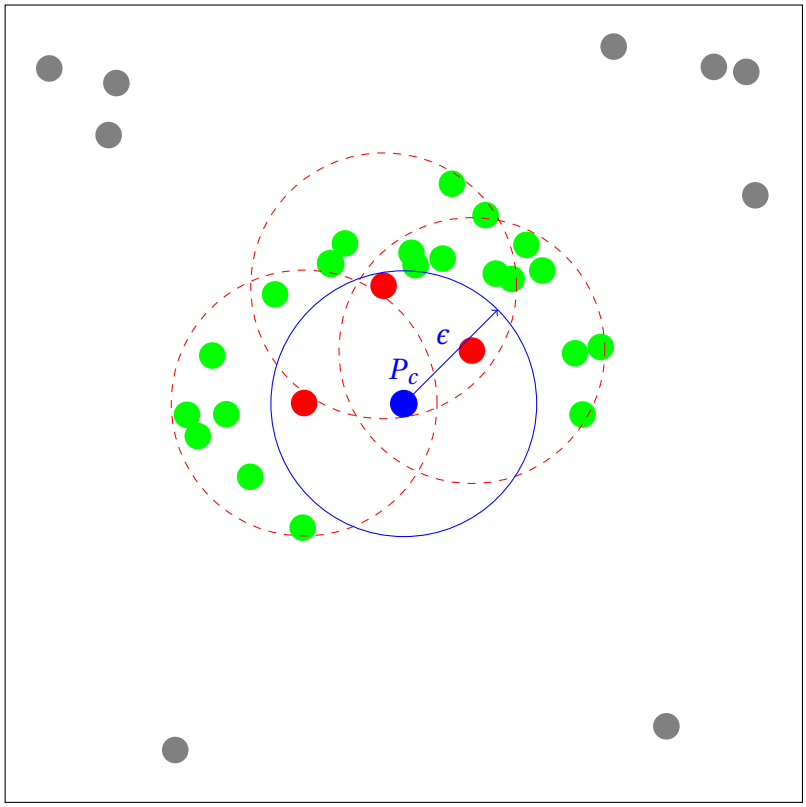


Figure 2.11 – DBSCAN (reference: blue point): directly reachable points in red and (inderectly) reachable points in green.

≤ 4, then p_c is classified as a core point, which also implies that the colored (i.e. not gray) points form a cluster.

The algorithm starts with the analysis of an arbitrary point to verify whether this point is a core point. If enough points are in its ϵ-neighborhood, a new cluster is started and all reachable points are included in this cluster; otherwise the point is tentatively marked as noise, as it can later be reachable by some cluster. The process continues until all density-connected points (hence clusters, depending on *MinPts*) are found. The pseudocode of DBSCAN is provided in algorithm 1 (page 307).

The complexity for the non-optimized implementation of DBSCAN is $\mathcal{O}(n^2)$, because all pairwise distances between the points have to be computed. If a distance matrix is used —which takes $(n^2-n)/2$ distance computations to be built —DBSCAN runs in constant time (i.e. $\mathcal{O}(1)$), but takes $\mathcal{O}(n^2)$ of memory, whereas the basic implementation takes only $\mathcal{O}(n)$ of memory. It is possible to speed-up the computation using an indexing structure to find the neighbors of a point in $\mathcal{O}(logn)$, which results in an overall complexity of $\mathcal{O}(nlogn)$ for

DBSCAN.

The advantages of DBSCAN include:

- no need for specifying the number of clusters in advance;
- ability to find arbitrarily-shaped clusters;
- notion of noise points;
- robustness to outliers.

On the other hand, DBSCAN is non-deterministic because any border point that is reachable from multiple clusters is assigned to the first one formed during the execution the algorithm, which follows a random order. Furthermore, having a single value for ϵ may not be an optimal fit depending on the profile of the dataset to be clustered.

DBSCAN is very sensitive to parameter settings. To reduce this sensitivity, OPTICS (Ordering Points To Identify the Clustering Structure) was proposed [9]. Instead of operating on a prefixed value for ϵ to form clusters as DBSCAN does, OPTICS uses ϵ as a "generating distance" to look for higher density regions and orders the points in such a way that the value for a cut-off parameter ϵ' ($0 \leq \epsilon' \leq \epsilon$) can be chosen to form clusters *a posteriori*.

To help defining OPTICS, let's define function $\mathcal{N}_\epsilon(p)$, which returns all neighbors points of p within distance ϵ, and function $\mathcal{M}_\epsilon(p,i)$, which given a point p returns the distance of its i-th closest point within the ϵ-neighborhood. OPTICS defines the concepts of *core distance* ($\mathscr{D}_{core}$) and *reachability distance* ($\mathscr{D}_{reach}$) as follows:

$$\mathscr{D}_{core}^{\epsilon,MinPts-1}(p) = \begin{cases} \emptyset & \text{, if } |\mathcal{N}_\epsilon(p)| < MinPts \\ \mathcal{M}_\epsilon(p, MinPts) & \text{, otherwise} \end{cases}$$

$$\mathscr{D}_{reach}^{\epsilon,MinPts}(p_1,p_2) = \begin{cases} \emptyset & \text{, if } |\mathcal{N}_\epsilon(p)| < MinPts \\ \max\left(\mathscr{D}_{core}^{\epsilon,MinPts}(p_1), d(p_1,p_2)\right) & \text{, otherwise} \end{cases}$$

To establish an ordering for the points, OPTICS processes each of them, finding their core-distance and their reachability-distance. For this, the main loop of the OPTICS algorithm proceeds as follows:

1. randomly visit a unprocessed point and verify whether it is a core point. If it is not a core point, it is marked as processed and the main loop restart;
2. iteratively collect density-reachable points in the ϵ-neighborhood, mark them as processed and insert them in a seed-list;

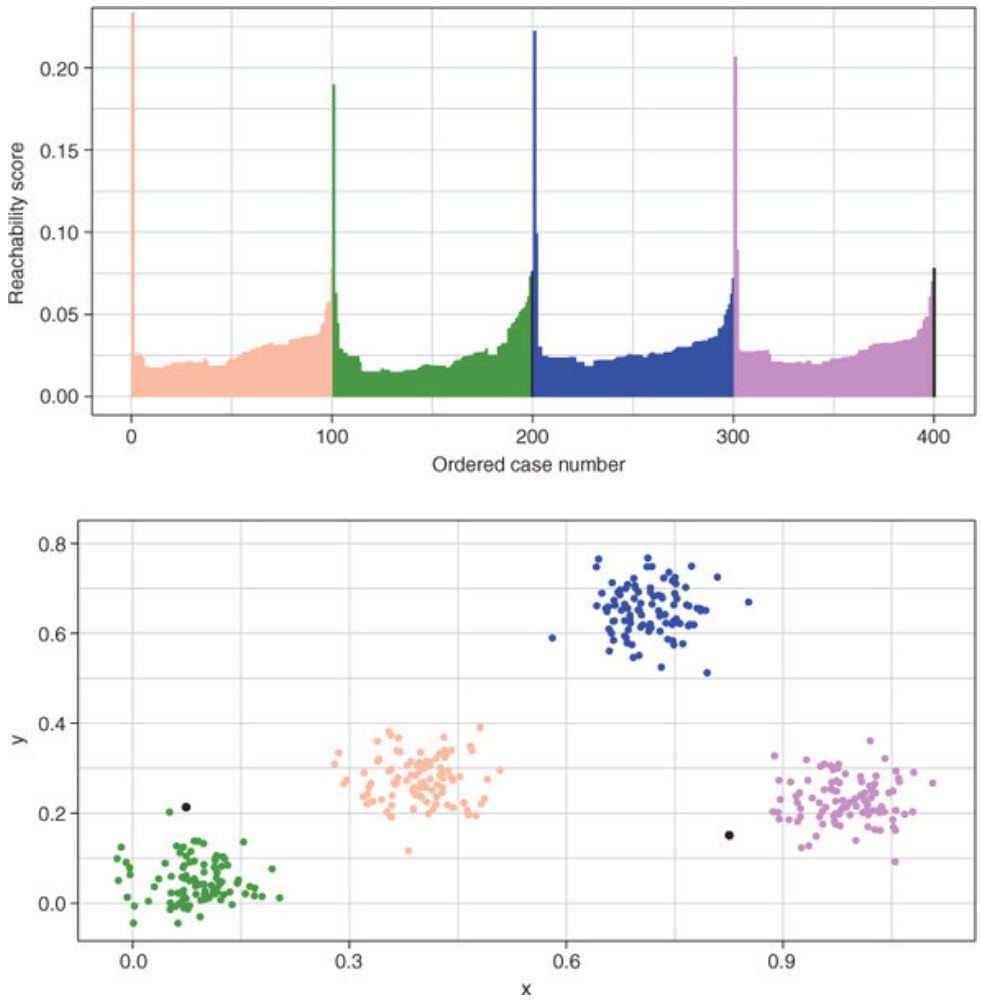


Figure 2.12 – Example of a reachability plot and the corresponding clusters [10].

3. sort the points in the seed-list according to their reachability-distance to the closest core point;

Once completed the main loop, the order as well as a minimum reachability-distance for all the points are available. This allows drawing a reachability plot, where the y-axis contains values for the minimum reachability-distances and the x-axis simply contains the order of the points. Figure 2.12 shows an example of a reachability plot.

The final clusters can be formed by choosing any value for ϵ' $(0 \leq \epsilon' \leq \epsilon)$ and checking which points mutually have minimum reachability-distances below this threshold. By construction, these points are together "in the valleys" of the reachability graph, as their are part

10. Image published at https://drek4537l1klr.cloudfront.net/rhys/Figures/fig18-6_alt.jpg (accessed October 2022)

of the same seed-list and therefore density-reachable.

Although being a little less time-efficient than DBSCAN, the complexity of OPTICS does not change. Both algorithms are heavily dominated by the runtime of the ϵ-neighborhood scanning. The original paper reports a ratio of 1.6x the runtime of DBSCAN for OPTICS [9].

Evaluation of Unsupervised Learning

Unsupervised learning aims to analyze data based on their inner structure, which allows to reveal patterns that were not previously known. To validate the outcome of the learning, independently of the method chosen, it is required to evaluate the quality of the final clustering. This allows not only to measure the performance of a clustering instance, but also to compare different clusterings that use different methods, thus providing a means to chose the best method according to the topology of the data.

When the unsupervised learning analysis is supplemented with a test data for which the expected results (i.e. ground truth) are known, it is possible to measure scores that are ground-truth dependent —including the scores seen above in section 2.2.1 (page 60). Other evaluation methods take into account only the inner structure of the data; thus not requiring any ground-truth knowledge and being ready applicable to any kind of data partitioning.

External Evaluation For a given set of points $\Pi = \{p_1, \cdots, p_N\}$, the external evaluation requires the knowledge of a ground-truth for the clustered data. Notating this reference "golden standard" classes (i.e. ground truth) as $\mathscr{C} = \{C_1, \cdots, C_n\}$ —such that $\bigcup \mathscr{C} = \Pi$ and $C_i \cap C_j = \emptyset$ for $i \neq j$ —and the partitions obtained from clustering as $\mathscr{P} = \{P_1, \cdots, P_{n'}\}$ —such that $\bigcup \mathscr{P} = \Pi$ and $P_i \cap P_j = \emptyset$ for $i \neq j$ —we define some metrics used for external evaluation. For this, we define the class matching of a cluster $C_i \in \mathscr{C}$ as:

$$\mathscr{M}(\mathscr{P}, C_i) = \{|P_j \cap C_i|, \forall\, j \in \{1, \cdots, n'\} \wedge P_j \in \mathscr{P}\}$$

Two commonly used external metrics to evaluate a clustering are *purity* and *entropy*, defined as follows [327]:

$$P(\mathscr{C}, \mathscr{P}) = \frac{1}{|\Pi|} \sum_{C_i \in \mathscr{C}} \max(\mathscr{M}(\mathscr{P}, C_i)) \qquad \text{(Purity)}$$

$$H(\mathscr{C}, \mathscr{P}) = \sum_{C_i \in \mathscr{C}} \frac{|C_i|}{|\Pi|} H(C_i, \mathscr{P}) \qquad \text{(Entropy)}$$

where the term $H(C_i, \mathscr{P})$ corresponds to the entropy of an individual cluster, which is computed as follows:

$$H(C_i, \mathscr{P}) = -\sum_{P_j \in \mathscr{P}} \frac{|C_i \cap P_j|}{|C_i|} log_2 \left(\frac{|C_i \cap P_j|}{|C_i|} \right) \qquad \text{(Entropy of a single cluster)}$$

This entropy measure corresponds precisely to the entropy above (equation Entropy, page 68), where the probability terms $P(y_i)$ are replaced by the frequency of each class in the cluster (i.e. $|C_i \cap P_j|/|C_i|$).

Hypothetically, the quality of the clustering improves with the increment of the purity value. As for the entropy, clusterings with the lower entropy values are (hypothetically) better. The pitfall of these metrics is the fact that larger numbers of clusters facilitate having a high scores for purity and a low scores for entropy, as demonstrated by Uddin *et al.* [288].

A complementary notion related to entropy is the **mutual information** (MI), which measures the mutual dependence between two random variables. It is originated from the probability theory, but it can also be applied in clustering evaluation.

The MI quantifies the "amount of information" obtained about one random variable when the other random variable is observed. Therefore, for two random variables X and Y, the mutual information is defined as follows:

$$I(X,Y) = H(X) - H(X|Y) = H(X) + H(Y) - H(X,Y) \qquad \text{(MI of random variables)}$$

This definition can be expanded [11] to :

$$I(X,Y) = \sum_{x \in X, y \in Y} p(x,y)\ log_2 \left(\frac{p(x,y)}{p(x)p(y)} \right) \qquad \text{(MI of random variables)}$$

In the context of clustering evaluation, the same concept can be applied. In this case, the MI measures how much information the reference classes $\mathscr{C}$ and the clustering partitioning $\mathscr{P}$ have in common. Their MI is computed as follows:

$$I(\mathscr{C}, \mathscr{P}) = \sum_{C \in \mathscr{C}, P \in \mathscr{P}} \frac{|C \cap P|}{|\Pi|}\ log_2 \left(|\Pi| \frac{|C \cap P|}{|C|\ |P|} \right) \qquad \text{(MI - Clustering)}$$

It is possible to normalize the MI value so as to limit the range within the interval from 0 to 1, which is referred as the **normalized mutual information** (NMI). The most used normalization was proposed by Kvalseth [153] and it consists in dividing $I(x,Y)$ by $1/2(H(X) +$

11. Details in appendix section 6.4 (page 308).

$H(Y))$ [12], i.e.:

$$NMI(X,Y) = \frac{2 \cdot I(X,Y)}{H(X) + H(Y)} \qquad \text{(Normalized Mutual Information)}$$

Vihn *et al.* demonstrated that the measures of MI and NMI are biased towards the number of clusters, i.e. increasing the number of clusters results in better scores for MI and NMI. Therefore they propose a variation of the MI called **adjusted mutual information** (AMI) as follows:

$$AMI(X,Y) = \frac{I(X,Y) - \mathbb{E}\{I(X,Y)\}}{\max\{H(X),H(Y)\} - \mathbb{E}\{I(X,Y)\}} \qquad \text{(Adjusted Mutual Information)}$$

where $\mathbb{E}\{I(X,Y)\}$ is the expected value of $I(X,Y)$. The AMI is in the range from 0 to 1, being equal 1 when the two clusterings are identical, and close to zero in case both clusterings are purely random. However, the AMI is not a metric as it does not satisfy the required properties (see section 2.2.3, page 74).

The **V-measure** is another popular entropy-based measure used for external evaluation [239]. Rosenberg and Hirschberg inspired from the F-score (equation 2.6, page 63) to propose the V-measure as the harmonic means of two complementary measures, i.e. *homogeneity* (h) and *completeness* (c):

$$V_\beta = \frac{1}{\left(\frac{\beta}{1+\beta}\right) c^{-1} + \left(\frac{1}{1+\beta}\right) h^{-1}} = \frac{(1+\beta)hc}{\beta h + c} \qquad \text{(V-score)}$$

where β is a weighted factor similar to the one in F_β, which gives more importance to completeness when $\beta > 1$ and more importance to homogeneity when $\beta < 1$ [13].

The homogeneity score measures the class distribution within each cluster, providing higher score when clusters contain elements of less classes —ideally a single class, where the homogeneity is maximal (= 1).

The authors consider how close a given clustering is to this ideal by examining the conditional entropy of the class distribution given the proposed clustering. In the perfectly homogeneous case, $H(\mathscr{C}|\mathscr{P}) = 0$. In an imperfect situation, the value is normalize by the maximum reduction in entropy the clustering information could provide, i.e. $H(\mathscr{C})$. Therefore the homogeneity is defined as:

12. Other variants also exist, refer to [293] for more information on about NMI.
13. In contrast to the definition of F_β, Rosenberg and Hirschberg do not take into account the Van Rijsbergen's effectiveness measure, therefore the weighted factor is simply β instead of β^2.

$$h(\mathscr{C},\mathscr{P}) = \begin{cases} 1 & \text{, if } H(\mathscr{C},\mathscr{P}) = 0 \\ 1 - \frac{H(\mathscr{C}|\mathscr{P})}{H(\mathscr{C})} & \text{, otherwise} \end{cases} \qquad \text{(homegeneity)}$$

The completeness score is symmetrical to the homogeneity score, but it considers the class distribution over all the clusters, providing higher score when the elements of the same class are member of less clusters —ideally a single cluster, where the completeness is maximal (= 1). Through a similar reasoning as above, the completeness score is defined as:

$$c(\mathscr{C},\mathscr{P}) = \begin{cases} 1 & \text{, if } H(\mathscr{C},\mathscr{P}) = 0 \\ 1 - \frac{H(\mathscr{P}|\mathscr{C})}{H(\mathscr{P})} & \text{, otherwise} \end{cases} \qquad \text{(completeness)}$$

Internal Evaluation For a given set of points $\Pi = \{p_1, \cdots, p_N\}$ and a partitioning $\mathscr{P} = \{P_1, \cdots, P_n\}$ —such that $\bigcup \mathscr{P} = \Pi$ and $P_i \cap P_j = \emptyset$ for $i \neq j$ —the internal evaluation relies only on the information contained in the data about the internal structure of the partitioning.

The basic drive for internal measures is to have elements of the same cluster as similar as possible, while having elements of different clusters as distinct as possible. To this end, internal measures are often based on two criteria: *cohesion* and *separation*.

Cohesion measures how compact are the elements within a cluster. It is often based on the variance measures of the elements, where lower variance indicates better compactness. Another possibility is to base this measure on distances (maximum/average pairwise or maximum/average center-based distances).

Separation measures how distinct (or well-separated) is a cluster from another. It is often based on pairwise distances between cluster centers or pairwise minimum distances between elements in different clusters. This measure can also be based on the cluster density.

One of the first measures that combine both criteria was proposed by Joseph Dunn in 1974 and it is currently known as **Dunn's index** [77]. It is defined as the ratio of the smallest distance between clusters (i.e. separation) and the maximum cluster diameter (i.e. cohesion), being computed as follows:

$$DI(\mathscr{P}) = \frac{\min\limits_{i \leq i < j \leq n} \delta(P_i, P_j)}{\max\limits_{1 \leq k \leq m} \Delta(P_k)} \qquad \text{(Dunn's index)}$$

where the distance between clusters $\delta(P_i, P_j)$ and the diameter of the cluster $\Delta(P_k)$ can be computed in different ways. The original proposition considers:

$$\delta(P_i, P_j) = \min_{p_i \in P_i, p_j \in P_j} d(p_i, p_j) \tag{2.7}$$

$$\Delta(P_k) = \max_{p_i, p_j \in P_k} d(p_i, p_j) \tag{2.8}$$

For $DI \to 0$, it follows that $\Delta(P_k) >> \delta(P_i, P_j)$. It means that smaller values of DI correspond to compact clusters and well-separated clusters. The weaknesses of Dunn's index are its time complexity and its sensitivity to outliers, as they can increase the diameter of clusters [116].

A popular measure for internal evaluation, which is more robust than Dunn's index, is the cluster **silhouette** [242]. It is a distance-based that ranges from -1 to $+1$, where higher values indicate higher similarity among elements of the same cluster (i.e. cohesion) and higher distinction with elements of other clusters (i.e. separation). The computation of the silhouette of a point $p_i \in \Pi$ such that $p_i \in P_\alpha$ ($\alpha \in \{1, \cdots, n\}$) is composed of two factors, i.e the average dissimilarity of P_i to all other points in P_j and the minimum average dissimilarity of p_i and $p \in \Pi \wedge p \notin P_j$, as follows:

$$a(p_i) = \frac{1}{|P_\alpha| - 1} \sum_{p_j \in P_\alpha} d(p_i, p_j) \qquad \text{(avg. dissimilarity within cluster)}$$

$$b(p_i) = \min_{P_j \in \mathscr{P} \setminus P_\alpha} \frac{1}{|P_\alpha|} \sum_{p_j \in P_j} d(p_i, p_j) \qquad \text{(min avg. dissimilarity with other clusters)}$$

A graphical representation of both factors is provided in figure 2.13. Once these factors have been obtained, the silhouette of point p_i is defined as follows:

$$s(p_i) = \begin{cases} \frac{b(p_i) - a(p_i)}{\max\{a(i), b(i)\}} & \text{, if } \max\{a(i), b(i)\} > 0 \\ 0 & \text{, otherwise} \end{cases} \qquad \text{(silhouette)}$$

The factor $a(p_i)$ is not defined for singletons, therefore in this case the value for $s(p_i)$ is defined as 0. For $s(p_i) \to 1$, it follows that $b(p_i) >> a(p_i)$; likewise, for $s(p_i) \to -1$, it follows $a(p_i) >> b(p_i)$. The former case indicates that the cluster is compact and far from other clusters, whereas the latter indicates that the distances of elements in the same cluster are way bigger than the distance with points of another cluster.

It is possible to measure the silhouette score of a cluster by taking the mean silhouette

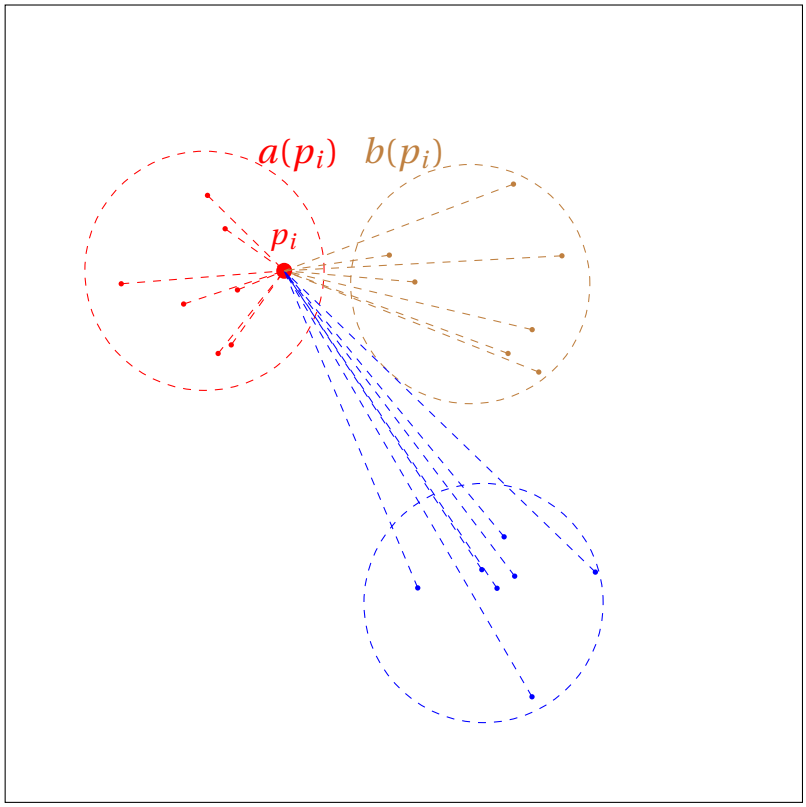


Figure 2.13 – Silhouette of point p_i: depiction of factors $a(p_i)$ and $b(p_i)$.

value of the points that it contains. The silhouette coefficient is defined as the mean silhouette over all elements.

2.3 Malware Analysis

Malware can have many forms, including malicious scripts, fileless malware, to name but a few. However, **in this thesis we are interested only in malware in the form of binary executable files** (as depicted in figure 2.14). Therefore we start this section with a review of topics related to the analysis of binary files —insofar as it is useful in malware analysis —before providing a literature review on topics of malware analysis and introducing the concept of malware analysis primitives.

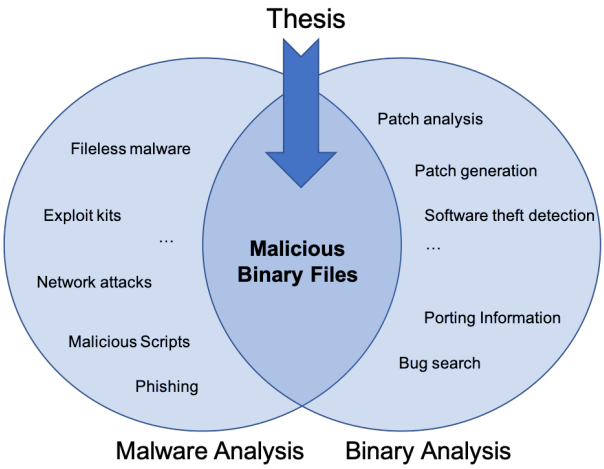


Figure 2.14 – Thesis scope: Malware Analysis of Binary Files

2.3.1 Binary Analysis

Binary analysis refers to the analysis of *machine code* that can be executed by a CPU, given the right architecture and external dependencies (e.g. operating system, DLL, etc). In the overwhelming majority of the cases, the analysis has access only to the malware in the form of a (executable) binary file without the corresponding source code, hence the interest for binary analysis.

Syntax and Semantics

When analyzing machine code (or any code) two dimensions can be to considered: syntax and semantics. Syntax concerns the rules guiding the formation of the code, which includes the available instructions and their grammatical relations. Semantics concerns the meaning of a sequence of instructions —more precisely its *teleological function*, as discussed in section 1.2.2 (page 34) —often simply referred as the *functionality*.

Computer programs (be they malware or not) have to adhere to the syntax rules of the target platform to execute. A central task in malware analysis involves the analysis of unknown programs (typically through reverse-engineering) in order to evaluate their level of maliciousness.

There is an intrinsic bound between syntax and semantics, as two program instances with similar syntax are likely to have similar semantics. Nonetheless, two program instances with very different syntax can still have the same semantics —which is a fact largely exploited by malware authors so as to preclude analysis and keep their programs undetectable.

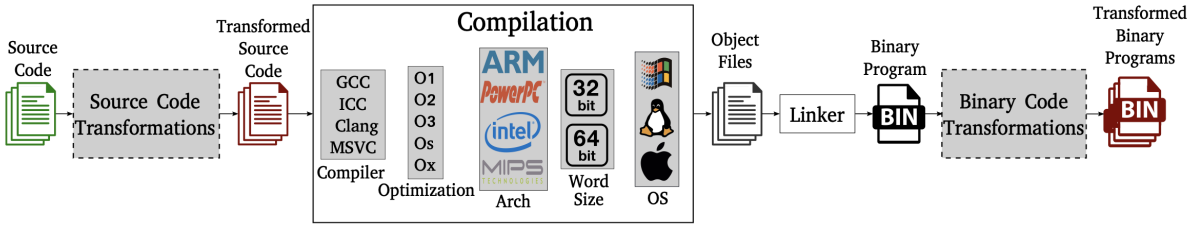


Figure 2.15 – Extended compilation process as depicted in [119]

Syntax is stiff, thus easy to analyze (specially by computers). Therefore syntax-based analysis is generally efficient and precise. However, as adversaries can circumvent this analysis through semantics-preserving transformations (i.e., not changing the program functionality), the Achilles heel of this approach is its robustness.

Semantics is flexible, therefore harder to model and analyze. Semantic-based analysis is generally less efficient and more prone to higher variance; however it is harder circumvent, hence more robust.

Compilation Process

As a general rule, machine code is produced by a *compilation process* to which a *source code* is given as input, and an object file is generated as output. Different object files are then linked together to form the binary (executable) program.

Ul Haq and Caballero detail the transformation of source code into machine code in their *extended compilation process* [119]. This extended definition is convenient because it considers semantics-preserving transformations along the compilation, which are not very usual in regular compilation processes but are indeed very frequent for malware.

Semantics-preserving transformations for malware are typically *obfuscations* intended to hamper the reverse-engineering analysis of the binary code. They can take place before compilation, altering the source code to produce a new transformed (e.g. obfuscated) version of the original source code, or after, altering the binary program to produce a new transformed (e.g. obfuscated) version of the original program. Source code transformations are platform-safe but dependent on the programming language of the source code, whereas binary code transformations are independent from the programming language but specific to the target platform.

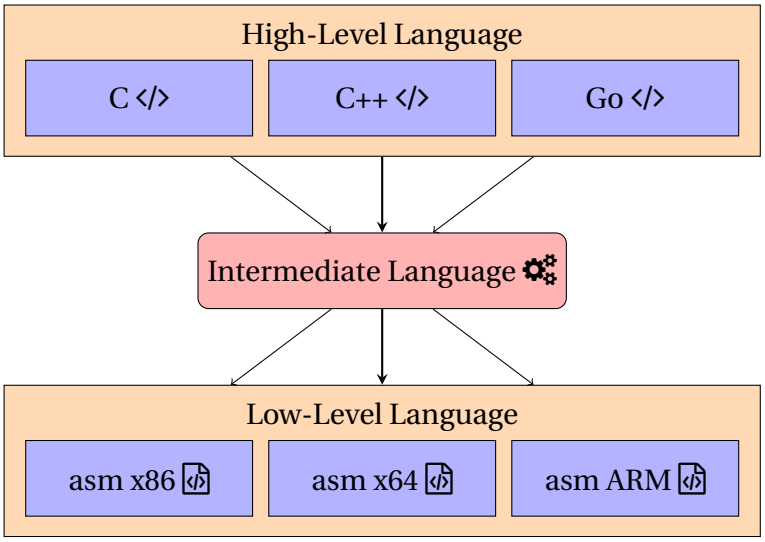


Figure 2.16 – Stages of a typical compilation process (i.e. three-stage compiler).

Intermediate Language (IL) In general, the compilation process does not transform high-level code directly into low-level code. Instead, the compiler first transforms high-level code into an *intermediate representation* (e.g. AST, intermediate language), which is then transformed into low-level code (as shown in figure 2.16) that is translated into machine code.

By using an intermediate representation, the compiler can attain a separation of concerns: the compiler front-end checks the syntax and semantics of the high-level language, whereas the compiler back-end optimizes and produces the code of the specific targeted CPU architecture. This allows to simplify optimization at each stage of the compilation process.

Intermediate representations also facilitate the portability of source codes on multiple platforms, since they breaks the one-to-one dependency between high-level and low-level languages —otherwise, for a given language, a full native compiler would be necessary for each targeted CPU architecture.

The overwhelming majority of programming frameworks, including the main ones, benefit from intermediate languages: Microsoft's *Common Intermediate Language* (CIL) is used to support dozens of languages (including C#, C++, PowerShell) on multiple architectures; the very popular GNU Compiler Collection (GCC) [269], which supports multiple languages such as C, C++, Objective-C and Go, using GENERIC [186] as its intermediate language; Clang, which is also a very popular front-end and is the main competitor of GCC [146], uses LLVM intermediate language to support C, C++ and Objective-C compilation [159].

Static Analysis

Static analysis refers to methods used to analyze programs without requiring their execution. This analysis methodology leverages the definition of syntactic rules to scan programs so as to extract descriptive features.

For instance, to run on Windows an executable file must be in the *Portable Executable* (PE) file format. The layout of a PE file comprises a number of headers and sections that are read by the Windows loader when the file is mapped into memory. Figure 6.2 (page 309) shows the layout of different headers comprised in a PE file.

By simply parsing the sequence of bytes in the binary file, a static analyzer can find out that it is a PE file because it must start by the signature bytes (i.e. magic number) 'MZ' (`0x4D 0x5A`) and have a pointer at the offset `0x3C` that points to the beginning of the COFF header, which starts with the magic number 'PE\0\0' (`0x50 0x45 0x00 0x00`) —otherwise the Windows loader does not map the file into memory. Likewise, the static analyzer can obtain a lot of other information contained in the header, such as:

— the target CPU architecture (e.g. x64, x86, ARM, etc) [field: `Machine`];

— the compilation timestamp [field: `TimeDateStamp`];

— whether the debug information was stripped [field: `IMAGE_FILE_DEBUG_STRIPPED`];

— whether it is a system file or a user program [field: `IMAGE_FILE_SYSTEM`];

— exported functions [export table];

— imported functions [import table];

— *etc.*

However, some fields in the header do not impact the correct execution of the file and therefore they can be modified by malware authors to mislead future analysis. For example, the field TimeDateStamp can reveal when the malware was generated, however it can be altered by the author to contain a fake timestamp; in some cases, the compiler sets a fixed timestamp to all generated files (e.g. Borland Delphi and its timestamp `1992-06-19 22:22:17 (Friday)`), which can unwittingly provide indicators about the language and compiler used.

Another very useful method of static analysis is (machine) code disassembling, in which the instructions and identifiable data structures are parsed from the machine code of a program. The goal is to translate the sequences of bytes into (meaningful) syntactic units of the *assembly language* (asm) [71].

It is also possible to attempt to reconstruct the (high-level) source code directly from the machine code, a process known as *decompilation* (as it undoes the compilation process). This process is way harder than disassembling because many details concerning the abstractions of the high-level language (e.g. variables, arrays, objects, functions, etc.) are lost in the compilation process, whereas disassembling is about straightforward code translation (from machine code to asm). Figure 2.17 shows the transformations of a "hello word" code written in C.

Figure 2.17 – [De]compilation and [de]assembling of a "hello word" code in C.

A well-established representation that facilitates the analysis of the execution flow of a program is the *control-flow graph* (CFG) [4]. It is a directed-graph representation of the program in which nodes correspond to *basic blocks* and edges correspond to deviation in the control flow; basic blocks are straight-line sequences of instructions that do not contain any jumps, with exception of the last instruction, and in which the first instruction is targeted by another deviation (e.g. jump, function call) in the control flow.

Figure 2.18 shows two disassembling representations of the same binary code. The left-hand side presents a plain textual representation in asm using *objdump* [26], whereas the right-hand side presents a CFG representation using *IDA Free* [231].

Disassembling and decompilation are operations tightly linked to the syntax of the analyzed (machine) code. However, as they are related to the instructions of the code, they can also capture traits of the code's semantics (which is not the case for all static analysis methods, e.g. file format analysis).

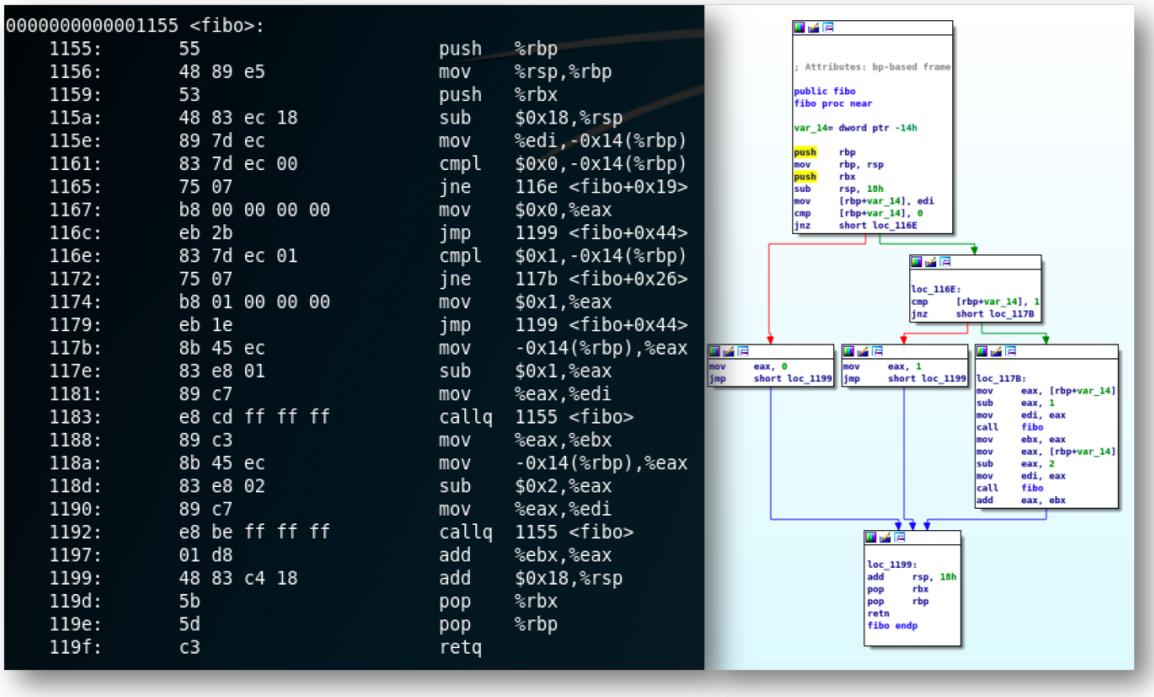


Figure 2.18 – Disassembling of a binary code: plain asm (left) and CFG (right).

Dynamic Analysis

Dynamic analysis refers to methods used to analyze programs requiring their execution. This analysis observes the program during its execution, taking into account how the executable code evolves as it runs and how it interacts with other components of the system (e.g. memory, file system, etc).

Standard *debugging* is a basic example of analysis method that can be considered as part of the dynamic analysis. It consists of an interactive method intermediated by a debugger that interacts with the operating system and alters parts of the code on the fly to inspect and control the execution of the code.

Figure 2.19 shows the screen of a debugging process using the *GNU Debugger* (GDB) [224]. It lists the data contained in each register, the disassembling of the code located in the surroundings of the address pointed by the instruction pointer (`RIP`) and the content located in the stack of the process. It is also possible to note that the execution of the code was held at the address whose tag is `<main+4>` and that the debugger awaits an input command.

Other tools such as *ProcMon* [188] interact with the operating system —in this case, through a Windows device driver [245] —to collect file system, registry and process/thread

Figure 2.19 – Disassembling of a binary code: plain asm (left) and CFG (right).

activity in real time. Figure 2.20 shows the dashboard of ProcMon with a list of activities (referred as *events*) that were captured by the tool. Figure 2.21 shows the usage of ProcMon to capture details of running processes.

Other tools such as the *Wireshark* [57] use libraries for capturing network packets (libpcap on Unix and Npcap on Windows) in order to intercept (i.e. "sniff") the network traffic. For instance, Wireshark allows to perform deep inspection of hundreds of protocols, capture and analyze live network streams, and has many other useful capabilities for networking analysis. Figure 2.22 shows the Wireshark dashboard with a filter to only display TCP packages.

There are a lot of different dynamic analysis methods and tools that can be useful for

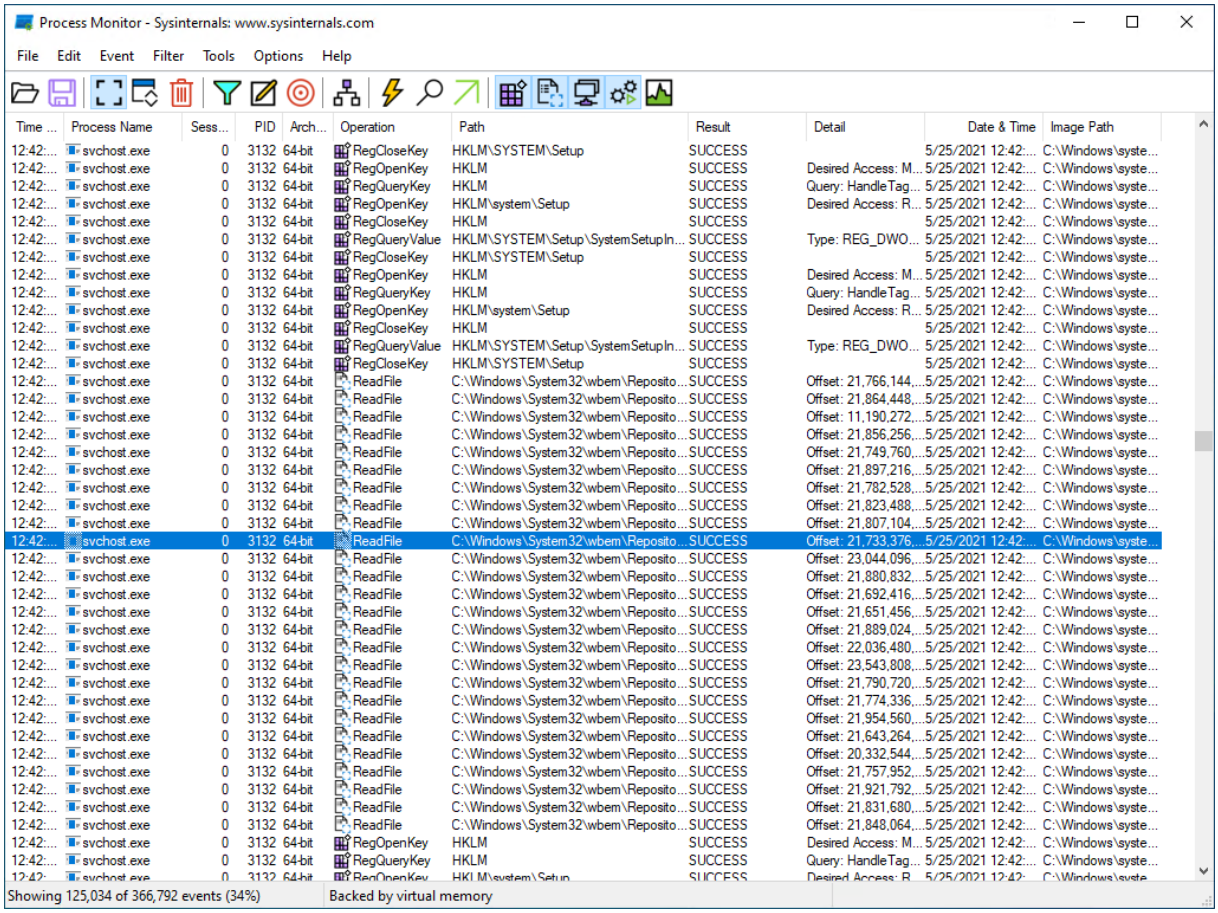


Figure 2.20 – Dashboard of ProcMon showing a list of intercepted events.

malware analysis. They can target a specific piece of code (e.g. debugging), or the interactions of running processes and the operating system (e.g. file system monitoring, process monitoring, network sniffing, etc).

Isolation (sandbox) Independently of the chosen method, dynamic analysis unfolds as the code runs; when it comes to malware analysis, running unknown code can put the system in jeopardy. Therefore, a major concern in dynamic analysis of malware is *isolation*, i.e. scope limitation of processes that run on the system. This scope can be related to network, file system, process rights and interactions, etc.

The scope within the boundaries imposed by a security mechanism is usually referred as *a sandbox*. The most common isolation strategy involves the virtualization of a guest VM on which the unknown binary codes are executed and the data generated during the execution are collected for analysis. While this setup is the popular, it is important to note that the

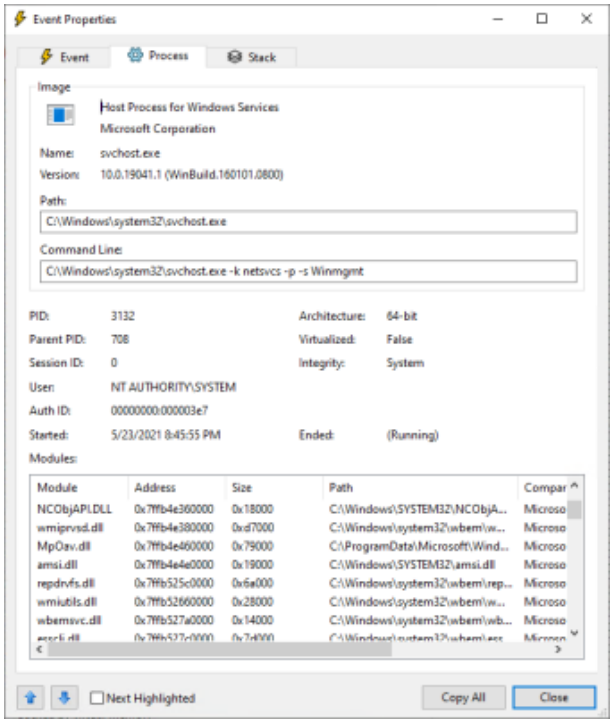


Figure 2.21 – Details of process captured by ProcMon including image path, command line, user and session ID.

concept of *sandbox* extends beyond this particular case.

Anti-Analysis Techniques

Anti-analysis techniques includes a set of different techniques intended to thwart or delay the reverse engineering of a given piece of code or to identify whether it is running inside a sandbox. In turn, anti-reverse engineering techniques include techniques for anti-disassembly, anti-debugging and various code obfuscation techniques. Malware leverage these techniques to remain as undetectable for as much time as possible.

A very common technique used by the vast majority of malware in the wild is code obfuscation through *packing*, intending to hinder static analysis. In its simplest form, packing transforms the binary code of a program into a new representation in which the original content is either compressed or encrypted and this process is reversed by a small unpacking routine that reverses this transformation in runtime. Figure 2.23 shows the transformation scheme of a simple packer. For more details about runtime packers refer to the work of Ugarte *et al* [290].

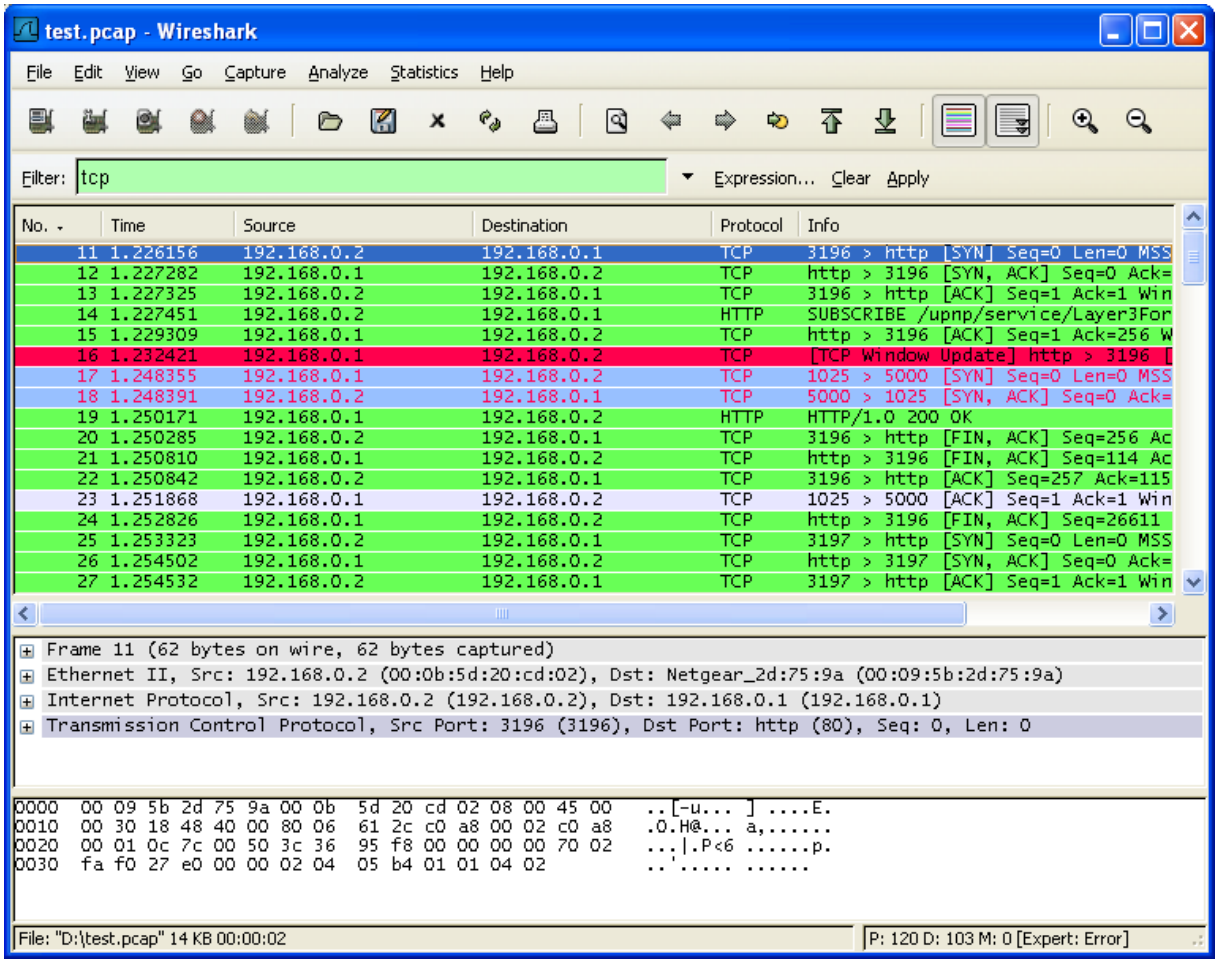


Figure 2.22 – Wireshark dashboard with the inspection of TCP network packages.

Code disassembling is targeted by some techniques like data and instruction interleaving, instructions reordering, abnormal conditions (e.g. `JMP` with the same target), opaque predicates, dead code insertion, etc.

Other techniques target dynamic analysis. This is the case for anti-debugging techniques, that attempt to detect whether a program execution is being inspected by a debugger. It can be done in many different ways, such as using the operating system API to check whether the running process is being traced (e.g. *IsDebuggerPresent*, *CheckRemoteDebuggerPresent*, *NtQueryInformationProcess*, *OutputDebugString* of the Windows API); checking the data structure of the running process to identify data flagged when a debugger in in use (e.g. *BeingDebugged*, *ProcessHeap*, *NTGlobalFlag* of the Window's Process Environment Block); checking indicators on the system (e.g. debuggers default installation paths, residue in live memory

14. Image published at https://www.joesecurity.org/blog/8506317946374998489 (accessed October 2022)

such as window name, etc); scanning the code of the process in memory to discover modifications made by a debugger (e.g. the existence of `INT` instructions); checking the time difference between two instructions, which is supposed to be smaller than a threshold otherwise it indicates the the code might be under debugging. When a debugger is detected, the malware can decide to stop its execution or to modify its behavior skipping the execution of malicious segments of the code.

Similarly, a malware can attempt to identify whether it is running on a virtualized environment to control its behaviour. Anti-VM techniques include the verification of indicators of virtualization on the system (e.g. known files and folder created by expansion package such as VMware Tools, registry keys and values, standardized values for MAC address, residues in live memory, etc) and presence of return value of instructions that differ on virtualized environments (e.g. `SIDT`, `SGDT`, `SLDT`, `CPUID`, etc).

2.3.2 Malware Analysis Primitives

A plethora of malware analysis frameworks have been proposed. In general, however, they are presented as full-fledged systems designed to fulfill specific objectives (e.g. malware detection, malware clustering) without following well-defined standards as to the fundamental building blocks of malware analysis. This hinders re-use and impairs analysis and comparison of components conceived in different works.

In this section our goal is to identify and introduce an organization related to malware analysis primitives that allows to describe most (ideally all) malware analysis frameworks. These primitives are abstract methods that provide a high level description of the tasks generally performed by malware analysis frameworks and that correspond to their most basic building blocks.

For this, we inspire from the purpose-oriented structure proposed in the survey by Ucci *et al* [287]. The authors define the following major objectives —*malware detection*, *malware similarity analysis* and *malware category detection* —as follows:

> **Malware detection** aims to identify traits in programs that allow to assess whether a given sample is malicious. Depending on the context, this assessment can target a given malware family or some specific characteristic (e.g. if the file has a ZIP format, etc). Historically, this was the first major objective of malware analysis and it still stands as the most prevalent objective in malware research.
>
> **Malware similarity analysis** is a broad topic, which concentrates on the similarity/dissimilarity across sets of program instances. Ucci *et al* [287] split this topic

in four objectives: *variants detection*, *families detection*, *similarities detection* and *differences detection*.

Malware category detection aims to assign categories (e.g. spyware, ransomware, etc) to samples. However, albeit defined in the survey by Ucci *et al* [287], we do not further detail this topic because it conflicts with our axiology[15] (**Axiology-1**, page 55). Malware category presupposes the existence of ontological traits that go beyond straightforward lineage (e.g. code reuse), since instances of the same category do not necessarily belong to the same family (otherwise categories would be equivalent to families). Furthermore, if an instance would belong to such category, this would implicate that such instance is a malware. However, according to our axiology, there are no ontological differentiation of malware and software at the *realization level*, which makes us reject this research path.

Both malware detection and malware similarity analysis are based on features generally extracted through either static or dynamic analysis, or a combination of both. Therefore, before detailing each topic related to the malware analysis objectives, we provide an overview of features frequently used for malware detection and similarity analysis as well as their usage in a typical malware analysis workflow as highlighted by Ucci *et al* [287], namely:

Byte sequences target the byte-level content of the machine code. This feature type includes chunks of bytes of fixed or variable length, sliding window of n bytes (known as n-grams), matching a given encoding pattern (i.e. ASCII strings). This feature type is widely studied and used in practice, since it is very versatile and can be easily extracted using static analysis.

Opcodes targets machine-level operations contained in a binary file (program). Since the sample behaviour is determined by its instructions, the opcodes can to some extent represent the sample semantics. Opcode frequency is one of the most commonly used feature type [287].

File characteristics target certain information such as the header data that can be statically parsed from known structures. These information provides a large set of valuable information such as sections, imports, symbols, compilers, etc.

API and System Calls target the calls invoked during the sample execution, which can include external DLLs, calls to the operating system's API or direct system calls. They can be extracted through either static or dynamic analysis, but the latter is considered more trustworthy as information about function imports and system calls

15. **Axiology-1**: *pursuing no ontological difference between malware and software at the "realization level";*

can be easily obfuscated to counter static analysis. Similarly to opcodes, APIs and system calls are related to the semantics of the sample —on a higher level though.

Filesystem Activity targets the operations executed by a sample to interact with the filesystem. In Windows it includes the information about the *Windows Registry*. By monitoring these activities it is possible to detect attempts to gain persistence in the system.

Network Activity targets the data exchanged over the network. The contacted addresses and the content of the network traffic can unveil communications with external controllers that participate in a cyberattack.

These features extracted from the sample are then transformed in order to be analyzed. For malware detection, the goal is to identify the feature components that can better distinguish a given sample from the universe of all samples —a comparison of type 1-to-N. For similarity analysis, the goal is to identify features that allow to distinguish and compare sets of samples —comparisons of type 1-to-1, N-to-1 and N-to-N.

Figure 2.24 shows a typical malware analysis workflow that applies to both malware detection and malware similarity analysis. It summarizes the general analysis process, in which the workflow starts with the analysis of unknown samples in order to extract raw features that are then transformed and analyzed. Additionally, the information obtained from the feature analysis can be fed back into the workflow in order to enrich the information contained in the raw features with extra metadata about the sample under analysis. This feedback loop can take place in the form of malware signatures, malware classifiers, and others.

Next, we detail the practical objectives —malware detection and malware similarity analysis —and then we introduce a semi-formal description of malware analysis primitives.

Malware Detection

The main objective of *malware detection* is to provide methods that allow to identify malicious traits in unknown programs. Incidentally, these methods can be used to identify whether some given sample belongs to a specific malware family or whether it matches a certain profile (e.g. being a PE file, being a ZIP file, etc).

The main approach used in practice is signature-based detection, as it pioneered the advent of malware detection and it has greatly evolved ever since. Recently, machine learning has led to the development of several new approaches for detecting malware, reflecting its increasing use in this area —refer to the survey by Singh and Sigh which includes many different malware detection schemes (18 schemes based on syntactic signatures and 22 schemes

based on behavioral signatures) [263].

We now elaborate on signature-based detection —in particular, syntactic signatures with pattern matching —since it is the most fundamental and dominant method used in practice. Moreover, we use signature-based detection later on as benchmark for our proposed methods.

Signature-based detection Signatures refer to distinctive information corresponding to some set of samples that allows detecting them. Features extracted (and transformed) from one or multiple binary files are used to create a signature, which can be compared to features from an unclassified binary to determine whether it matches the signature.

The simplest kind of signatures are *syntactic signatures* [262, 277] whose detection rules are based on the binary file syntactic properties (like *length*, *entropy*, *number of sections*, or presence of certain *strings*). Alternatively, *behavioral signatures* [262, 277] can be created from the binary file behavioral properties, like *file operations*, *registry operations*, *network operations*, *API calls*, etc.

Syntactic Signature (Pattern Matching) Syntactic signatures are used to classify binaries by looking at particular patterns in their code. Due to the simplicity of syntactic pattern matching, these techniques tend to be very effective and efficient in practice. Nonetheless, it is not trivial to create and maintain syntactic signatures that are concomitantly specific enough to match only to the desired samples, but not so specific that they would match only very few instances of close variants. Furthermore, a signature may stop matching a certain file if it changes just marginally (maybe even by only a single byte), which is considered to be one of the main drawbacks of this method.

The principle of operation of syntactic signatures is to identify and record matching rules for sequences of bytes that describe proprieties such as file checksum [2], type, API calls [129], etc. Despite the limitations of syntactic signatures [220] —notably against obfuscation methods like packing —static syntactic signatures are largely employed in malware analysis. For example, ClamAV [52] allows using syntactic signature in the YARA [225] format for protection against malicious files, and VirusTotal [294] provides an interface that takes YARA signatures to lookup matching files across its whole database. Here we present the principles of syntactic pattern matching, providing three practical examples with different tools.

PEiD [16] is an old well-known tool used to detect PE malware, packers, and compilers.

16. Link: https://www.aldeid.com/wiki/PEiD

Although the original developers have discontinued this tools, its user community continues to use and update it frequently due to its simple design and effectiveness.

PEiD defines an underlying grammar for the creation of new matching rules. The inclusion of new rules can target a new malware, packer, or compiler without requiring updating the tool. As an example of a rule, we display the signatures for matching .NET objects below:

```
1  [.NET DLL -> Microsoft]
2  signature = 00 00 00 00 00 00 00 00 5F 43 6F 72 44 6C 6C
3  4D 61 69 6E 00 6D 73 63 6F 72 65 65 2E 64 6C 6C 00 00 ??
4  00 00 FF 25
5  ep_only = false
6
7  [.NET executable -> Microsoft]
8  signature = 00 00 00 00 00 00 00 00 5F 43 6F 72 45 78 65
9  4D 61 69 6E 00 6D 73 63 6F 72 65 65 2E 64 6C 6C 00 00 00
10 00 00 FF 25
11 ep_only = false
12
13 [.NET executable]
14 signature = FF 25 00 20 40 00 00 00 00 00 00 00 00 00 00
15 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
16 00 00 00 00 00 00 00 00 00 00 00 00 00 00
17 ep_only = true
```

Each rule starts with a string identifier between square brackets, which is displayed to the user when the rule is matched. The lines starting with the word `signature` (i.e. lines 2, 8 and 14) indicate the beginning of a signature definition as a byte array, where the marker "??" is used to match any byte. Finally, the `ep_only` line indicates whether the rule is expected to match only the bytes at the binary entry point or anywhere in the file.

The main limitation of PEiD is the lack of expressiveness in its grammar rules. For instance, the most recent PEiD database [17] uses 829 rules (out of 3714) just to define version 1.25 of the VMProtect [18] virtualization packer.

Another tool used for pattern matching on files is **DIE** [19], which stands for "Detect It Easy". DIE supports a JavaScript-like scripting language for creating signatures, which provides more expressiveness than PEiD, specially due to the support of matching conditions.

17. Link: https://handlers.sans.org/jclausing/userdb.txt
18. Link: http://vmpsoft.com/
19. Link: https://github.com/horsicq/Detect-It-Easy

An example of a DIE rule is shown below:

```
// DIE's signature file

init("protector","PE Intro");

function detect(bShowType,bShowVersion,bShowOptions)
{
    if(PE.compareEP("8B04249C60E8........5D81ED........
                    80BD..........0F8548"))
    {
        sVersion="1.0";
        bDetected=1;
    }

    return result(bShowType,bShowVersion,bShowOptions);
}
```

The rule matches files protected with PE Intro, which is detected by an expected sequence of bytes at the entry point. The rule starts by declaring a new signature at "`init`" and then by proving a description of the rule in the "`detect`" function.

Like PEiD, DIE has a simple flag (`PE.compareEP`) determining whether to look for byte arrays at the entry point. DIE uses "`.`" as wildcards to match any byte. Rule matching is indicated by the variable `bDetected`, which is set to 1. DIE also supports more sophisticated rules that depend on multiple conditions and code reuse as in the rule below.

```
// DIE's signature file

includeScript("rar");

function detect(bShowType,bShowVersion,bShowOptions)
{
    detect_RAR(1,bShowOptions);
    return result(bShowType,bShowVersion,bShowOptions);
}
```

The main drawbacks of DIE are its verbose syntax and the need for an ad-hoc script for each rule; on top of that, DIE lacks detailed documentation and advanced features such as annotations or modularity.

The current de facto standard grammar/tool for the creation of syntactic signature is **Yara** [319]. Yara rules are defined using a JSON-like format, which provides higher expressiveness than PEiD and DIE rules.

Yara syntax defines three main fields: *meta*, which contains metadata that help identify the rule; *strings*, which contains a list of hexadecimal strings, text strings and regular expressions; *condition*, which defines boolean expressions over the elements present in the *strings* section.

```
rule silent_banker : banker
{
    meta:
        description = "This␣is␣just␣an␣example"
        threat_level = 3
        in_the_wild = true
    strings:
        $a = {6A 40 68 00 30 00 00 6A 14 8D 91}
        $b = {8D 4D B0 2B C1 83 C0 27 99 6A 4E 59 F7 F9}
        $c = "UVODFRYSIHLNWPEJXQZAKCBGMT"
    condition:
        $a or $b or $c
}
```

The rule above defines a Yara signature that matches files that contain any of the listed strings. Whenever a signature is matched, the tool outputs the rule name (i.e. *silent_banker*). On the same first line and separated by a colon, the string *banker* identifies the rule tag.

Yara supports many useful features that simplify the writing of signatures, such as modifier marks for strings (e.g. *ascii*, *wide*, *xor*), positional matching (address or offset), file size condition, extension modules (e.g. *PE*, *ELF*, *Cuckoo*), etc. Furthermore, Yara is a lightweight and fast tool that is available on multiple platforms (i.e. Windows, Linux and MacOSX) and can be accessed through interfaces such as CLI and Python scripts.

(Malware) Similarity Analysis

Similarity analysis is a broad topic that concentrates on the analysis of similarity/dissimilarity across groups of software instances, which includes applications such as *variants detection*, *families detection*, *similarities detection* and *differences detection*, defined as follows:

Variants detection aims to develop methods that allow to recognize that an unknown sample is actually a variant of a known malware.

Family detection aims to develop methods that given an unknown sample allows to identify families the sample is likely to belong to.

Similarity detection consists in identifying common parts or any kind of commonalities in multiple samples.

Differences detection is in many aspect similar to similarity detection, but it targets differences instead of commonalities.

The features that dominate this axis of malware analysis are based on APIs and System Calls as well as behavioral features such as file operations, registry operations and network operations. All works but one analyzed in the survey by Ucci *et al* [287] are based on collections of APIs and System Calls.

As for the feature analysis, *variants detection* includes but is not limited to schemes based on Hidden Markov Models, rule-based classifiers, variants of the decision tree algorithm, clustering with locality sensitive hashing and DBSCAN clustering [287]. Family detection includes but is not limited to schemes based on variants of graph matching, logistic regression, variants of the decision tree algorithm, K-NN, SVM, neural networks, K-Means and K-Medoids algorithms, prototype-based clustering and hierarchical clustering [287]. Similarity detection includes but is not limited to schemes based on logistic regression, SVM, clustering with locality-sensitive hashing, prototype-based clustering and hierarchical clustering [287]. Differences detection includes but is not limited to schemes based on rule-based classifier, logistic regression, variants of the decision tree algorithm, K-NN, clustering with locality-sensitive hashing and prototype-based clustering.

Malware Analysis Primitives

Here we propose *primitives* that are frequently found in malware analysis and describe them in a semi-formal notation. They can be seen as basic building blocks to design malware analysis frameworks without dealing with implementation details. Our intention is to **clearly define a taxonomy of malware analysis** that we use henceforth in the thesis. As to the best of our knowledge such formulation is the first-of-its-kind, thus it figures as contribution of the thesis.

Let $\mathfrak{S}$ represent the set of [transformed features extracted from] *samples* that are subject to the analysis of a malware analysis framework. Let $\sim$ denote an equivalence relation on $\mathfrak{S}$. We define as *family* ($\mathfrak{F}_\sim$) the equivalence class $\mathfrak{F}_\sim = [s]_\sim$, where $[s]_\sim := \{s' \in \mathfrak{S} : s' \sim s\}$. If s is considered malicious, then $\mathfrak{F}_\sim$ is a *malware family*.

For a given $s, s_1, s_2 \in \mathfrak{S}$ and a set of family $\{F_i\}$ such that $\bigcup\{F \in \{F_i\}\} = \mathfrak{S}$, we define the following malware analysis primitives:

Detection

$$\mathscr{F}_{Detection}(s,F) = \begin{cases} 1 & \text{, if } s \in F \\ 0 & \text{, otherwise} \end{cases} \quad \text{(Detection)}$$

where $F \in (F_i)$.

In the case of *malware detection*, F corresponds to a family that includes all malware instances of $\mathfrak{S}$.

Similarity

$$\mathscr{F}_{Sim}(s_1,s_2) \rightarrow \sigma \in [0,1] \quad \text{(Similarity)}$$

where σ is proportional to the similarity between s_1 and s_2; $\sigma = 0$ denotes total dissimilarity, whereas $\sigma = 1$ denotes total similarity between s_1 and s_2.

Search

$$\mathscr{F}_{Search}(s) \rightarrow \{F, \varnothing\} \quad \text{(Search)}$$

where $F = \{s' \in \mathfrak{S} | s' \sim s\}$. If no such family exists in (F_i), then $\mathscr{F}_{Search}$ outputs $\varnothing$.

Classification

$$\mathscr{F}_{Class}(s,\{F_i\}) \rightarrow \{i, \varnothing\} \quad \text{(Classification)}$$

where i is the index of (F_i) such that $\forall F_i = \{s' \in \mathfrak{S}, s' \sim s\}$.

If no such family exists in (F_i), then $\mathscr{F}_{Class}$ outputs $\varnothing$.

Clustering

$$\mathscr{F}_{Clust}(\mathfrak{S}) \rightarrow \{P_i\} \quad \text{(Clustering)}$$

where $\{P_i\}$ is the set of disjoint partitions of $\mathfrak{S}$ such that $\bigcup\{P \in \{P_i\}\} = \mathfrak{S}$.

2.4 Discussion

A key challenge currently in malware analysis is to respond to the quick expansion of malware in terms of prevalence and complexity, while traditional human-centered methodologies in use cannot evolve at the same pace. This situation imposes the coalescence of traditional and machine-centered methods, hence the ever growing influence of machine learning in malware analysis.

Considering figure 2.24 (page 109), with exception of the binary analysis phase, the depicted workflow could represent the phases of a standard machine learning analysis. Yet, the adversarial model of malware analysis entails major challenges in feature extraction, as malware creators use multiple techniques intending to mislead analyses, which makes the application of machine learning methods particularly intricate in this case.

Furthermore, there are severe ontological and epistemological matters concerning malware (extensively discussed in chapter 1, page 25) that affect the usage of machine learning in malware analysis. Humans can easily identify cats and dogs on pictures; however, when it comes to malware classification, the categorization can become much more blurry —the so-called "*gray zone*", as it the case with PUPs (c.f. section 1.2.1, page 27) —thus having a direct impact not only on supervised models, but also on any kind of external evaluation of unsupervised models.

Another aspect that emerges from the combination of both research fields is (to some extent) the lack of congruence of some concepts and terminologies. For instance, the concept of "classification" in machine learning is closer to the concept of "detection" than "classification" itself in malware analysis. Such discrepancies can be source of misunderstanding when communicating about these fields. In this thesis we privilege the terminology issued from the malware analysis field, as defined in section 2.3 (page 87) and semi-formalized in section 2.3.2 (page 105).

2.5 Conclusion

In this section we presented the general, contextual background of the thesis. We started with a presentation of topics related to machine learning, including definitions and terminology as well as topics of supervised and unsupervised learning. Then we presented topics of malware analysis, with special focus on malware in the form of binary files, before discussing some issues related to the combination of both fields.

Now that the thesis background is presented, we will move to the nub of the thesis where our main contributions will be presented.

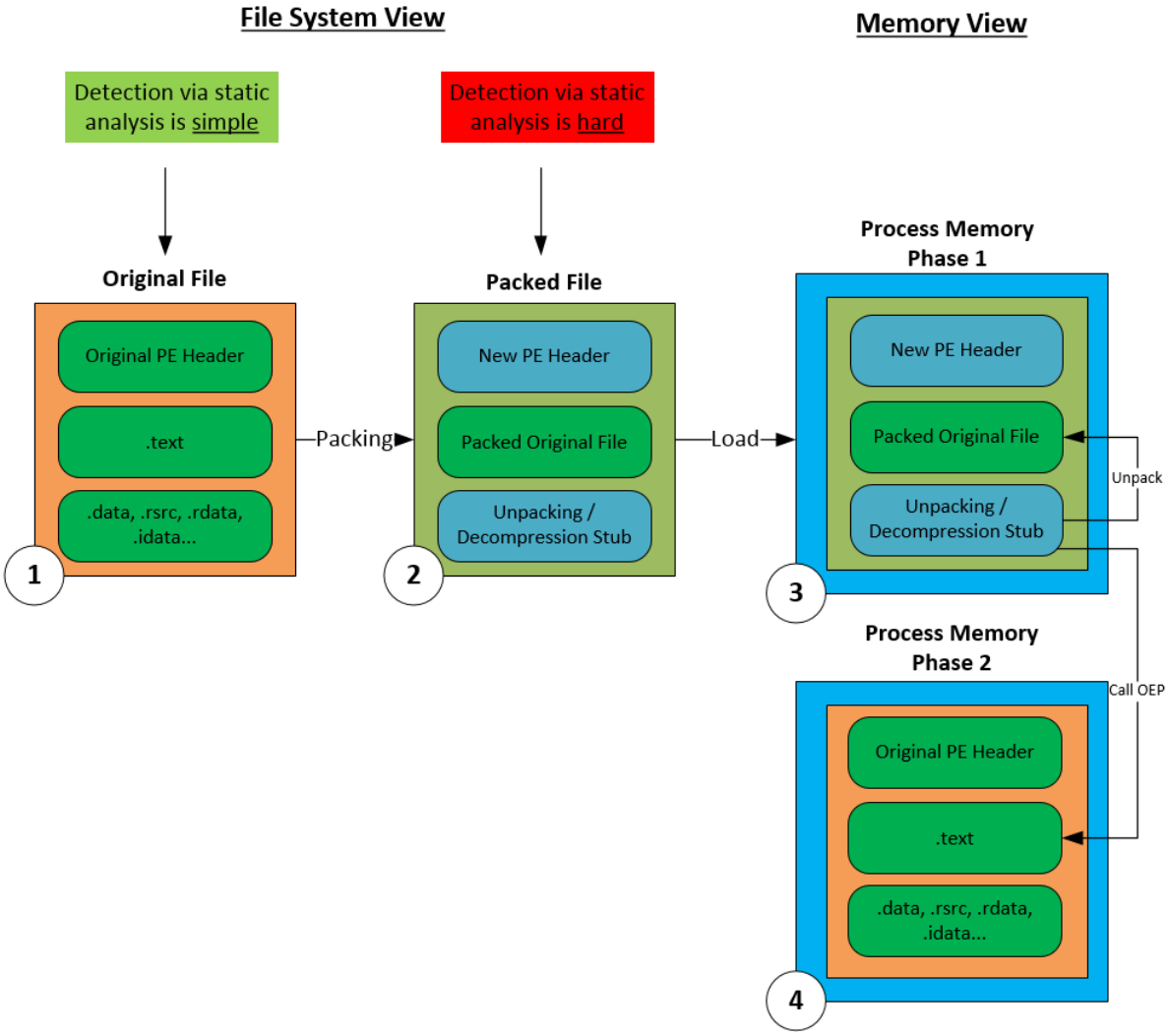


Figure 2.23 – Transformation scheme of a simple packer (file system view and memory view) [14].

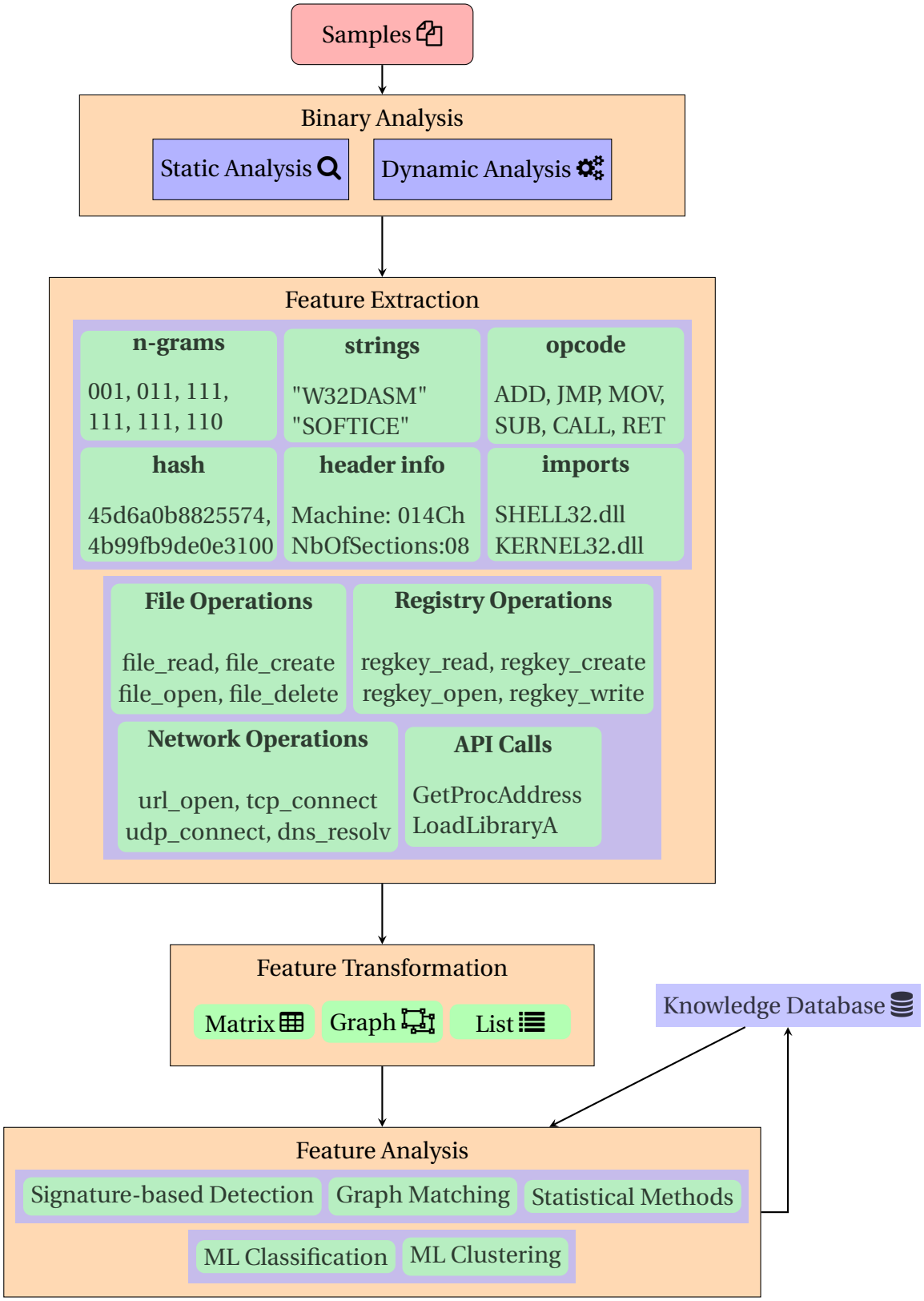


Figure 2.24 – Typical malware analysis workflow.

CHAPTER 3

CALL TRACING WITH SYMBOLIC EXECUTION

3.1 Introduction

Typically, the analysis of (malware) programs follows the dynamic and/or static analysis approaches. In dynamic analysis, programs are executed in an isolated environment with aim to understand their behavior. In static analysis, the program structures are parsed to extract data of interest without executing it.

The main limitation of dynamic analysis is code coverage, because the correct execution of the program strongly depends on the context provided by the analysis environment (e.g. sandbox), which makes it possible to finish the analysis without ever going over the most important segments of the program code. In malware analysis this problem is aggravated by the fact that malware often employ various techniques to hinder dynamic analysis.

Static analysis has a complete view over the binary code of the analyzed programs, but not executing them poses severe limitations to the analysis when the programs implement runtime modifications in their code or in any descriptive data. This is a major issue for malware analysis since malware very commonly use evasion techniques to thwart static analysis (e.g. packing, virtualization, etc), as discussed in section 2.3.1 (page 88).

As an alternative to cope with the limitations of dynamic and static analysis, we consider the use of *symbolic execution*. Symbolic execution is a method for program analysis that lies between static and dynamic analysis, and which *emulates* the execution of binary code while keeping a fine control over all elements of this analysis. Symbolic execution has been increasingly used as technique of *Formal Methods* to analyzed the correctness, reliability and security of software systems.

Tools relying on these approaches include compilers [53, 61], code analyzers [11, 32, 38, 136, 171], disassemblers, binary analysis engines [261], to name but a few. These tools combine formal methods, which aim to provide complete and correct coverage, with heuristics

that try to optimize and find approximate solutions when formal methods alone are too expensive or intractable. In practice these tools often come with hundreds or thousands of configuration and tuning options that can be used to adjust their behavior to try and improve their overall effectiveness. However, since many rely on other tools to solve sub-problems or are applied in many scenarios, this configuration is frequently left to the default one or is set up with a suboptimal setting.

This chapter considers the problem of program analysis (with aim to call tracing) assisted by symbolic execution, particularly in the context of malware analysis, which is a challenging scenario due to its peculiar adversary model. Despite the potential advantages of symbolic execution, this technique is still seldom used for malware analysis. Therefore our main goal in this chapter is to evaluate symbolic execution and improve its use in our scenario.

Our main analysis targets the machine code of binary files as presented in figure 2.14 (page 88). We are particularly interested in accurately determining the calls made during the execution (either symbolic or not), especially the calls that may be initially obfuscated in the code at rest (i.e. before execution). We assess various heuristics that can be used to guide and optimize the symbolic execution for program analysis (in general).

As main contributions, we:

— assess the performance of different tactics (or a set of tactics combined by means of tacticals) on all types of angr requests [addresses **RQ3.1**];

— improve prioritization of state exploration by comparing heuristics that target different weak spots of symbolic execution [addresses **RQ3.2**];

— evaluate a big set of parameters according to their impact on malware classification and their mutual correlation to better understand the heuristics around angr that are more likely to succeed in our scenario [addresses **RQ3.3**];

3.1.1 Research Questions

This chapter addresses the top-level research question **TOP-RQ1**: *How to improve the arsenal of techniques currently used in malware analysis, in particular for the analysis of binary files?*

To this end, we leverage symbolic execution as a promising technique. We guide our study pursuing the following subordinate research questions:

RQ3.1 *How to tune the SMT solver to improve performance of symbolic execution?*

RQ3.2 *How to effectively apply heuristics that reduce the problem of state explosion in symbolic execution?*

RQ3.3 *What is the impact of enhancing symbolic execution for binary analysis in a malware classification scenario?*

3.1.2 Chapter Outline

The remainder of this chapter is organized as follows:

— Section 3.2 (page 113) presents the specific background of this chapter;

— Section 3.3 (page 123) presents the related works;

— Section 3.4 (page 127) presents our methodology, which details our approach for SMT optimization, improvement of symbolic execution and the impacts on malware classification;

— Section 3.5 (page 135) presents the experimental setup and the evaluation results.

— Section 3.6 (page 147) discusses results.

— Section 3.7 (page 151) concludes.

3.2 Background

This section presents the background related to the topics developed in this chapter. In section 3.2.1 (page 113) we introduce the topic of symbolic execution and in section 3.2.2 (page 122) we present the background on Frequent Subgraph Mining (FSM).

3.2.1 Symbolic Execution

Symbolic execution was initially proposed in the mid-70s by James C. King from the IBM Research Center [145] and has been traditionally considered as a static analysis technique [89]. However, in the early 2000s, Patrice Godefroid *et al* [104] combined the dynamic executions of programs with a parallel symbolic execution for the discovery of alternative execution paths. This approach is known as Dynamic Symbolic Execution (DSE) and is considered as a dynamic analysis technique [89].

Newer approaches use different strategies to “execute” the code, ranging from code interpretation (s.l. static) to binary instrumentation (s.l. dynamic). Consequently, symbolic

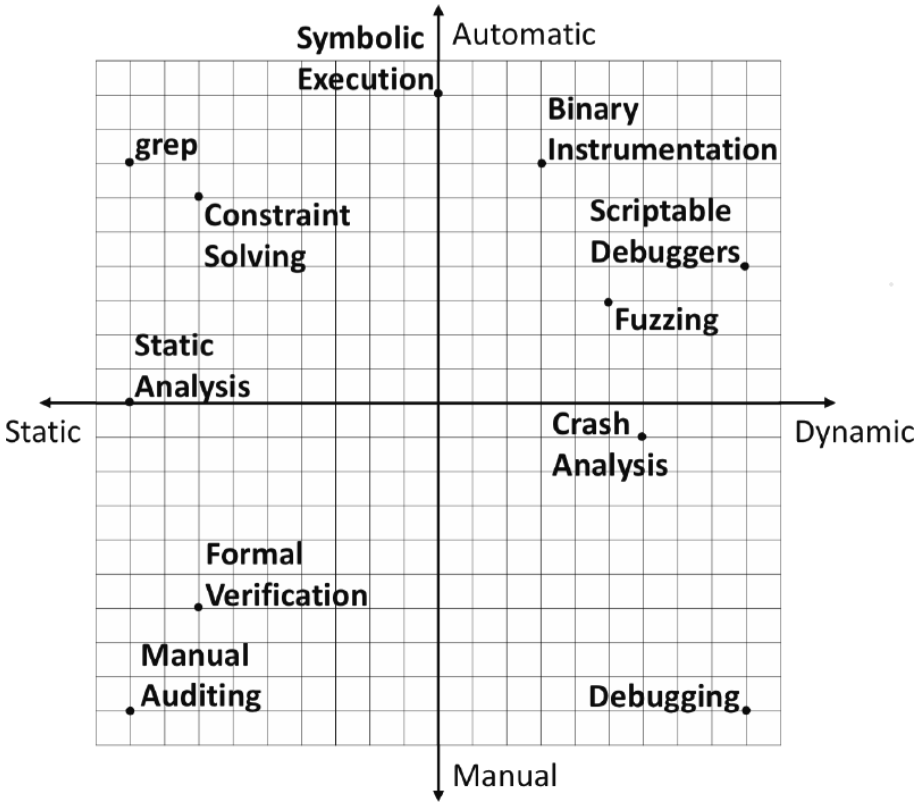


Figure 3.1 – Program analysis approaches as depicted by Julian Cohen at BlackHat 2014 [55].

execution can be seen as a mixed approach that combines the strengths of static and dynamic techniques. Figures 3.1 shows a visual arrangement of different program analysis approaches presented by Julian Cohen at BlackHat 2014 placing symbolic execution at the center of the static-dynamic axis [55].

Principles

In a standard execution, at any given time the values of variables have only a unique value. This is for instance the case of registers in a CPU. This execution model is referred as *concrete execution*, as variables are assigned with *concrete* values that change only when some —direct (e.g. `MOV`) or indirect (e.g. `ADD`) —assignment instruction is executed.

In symbolic execution, variables are *symbolic*, meaning that they are able to represent sets of possible concrete values. In contrast with concrete execution, symbolic variables accumulate constraints on the possible values for the variables as the code executes instead of representing a single value.

In symbolic execution, conditional statements are evaluated not just as *True*/*False* as

with concrete execution, but as *satisfiable* or *unsatisfiable*. This means that, given a symbolic variable, both conditions of a statement can be satisfiable ("*True*") (in opposition to unsatisfiable ("*False*")) at the same time. Due to that, the symbolic analysis can simultaneously take both execution paths of a branching, while the concrete execution can only take one of them. Consequently, while concrete execution is able to traverse only one execution trace at a time, symbolic execution can traverse multiple execution traces simultaneously.

To illustrate the difference between concrete and symbolic execution, we provide the following toy example.

```
1  #include<stdio.h>
2  /*
3          Trial and error to find a solution for x^2 - 3x - 4 = 0 for x >= 0
4  */
5  int main() {
6          int x;
7          printf("Let's try to solve x^2 - 3x - 4 = 0 for x > 0.\n");
8          printf("Enter a value for x: ");
9          scanf("%5d", &x);
10         if(x * x - 3 * x - 4 == 0) {
11                 if(x >= 0)
12                         printf("%d is a positive root.\n", x);
13                 else
14                         printf("x has to be positive.\n");
15         }
16         else
17                 printf("%d is not a root.\n", x);
18 }
```

Figure 3.2 – Sample of a code for a trial and error solution of a quadratic equation.

In this example, x is taken as a user input [line 9]; then x is tested whether it is a non-negative root of the equation $x^2 - 3x - 4$. If x satisfies both conditions [lines 10 and 11], then the `printf` [line 12] is reached. The concrete execution of this example is straightforward: a value is assigned to x and the code checks whether it is a non-negative root of the equation. It is very efficient to test whether the input satisfies both conditions, however it is not equally as straightforward to find a value for x which is a non-negative root of the equation. The traces of execution for x= -1 and x= 0 are displayed in Fig. 3.3.

In contrast, it is possible to use symbolic execution to systematically explore all execution traces. With symbolic execution, x is assigned with a symbolic variable which accumulates constraints along the different execution paths. The roots of $x^2 - 3x - 4 = 0$ are x= -1 and x= 4, therefore it follows:

— Line 11 is reached only if x is constrained to one of the root values (i.e. -1 or 4). In any other case, the execution will reach line 17.

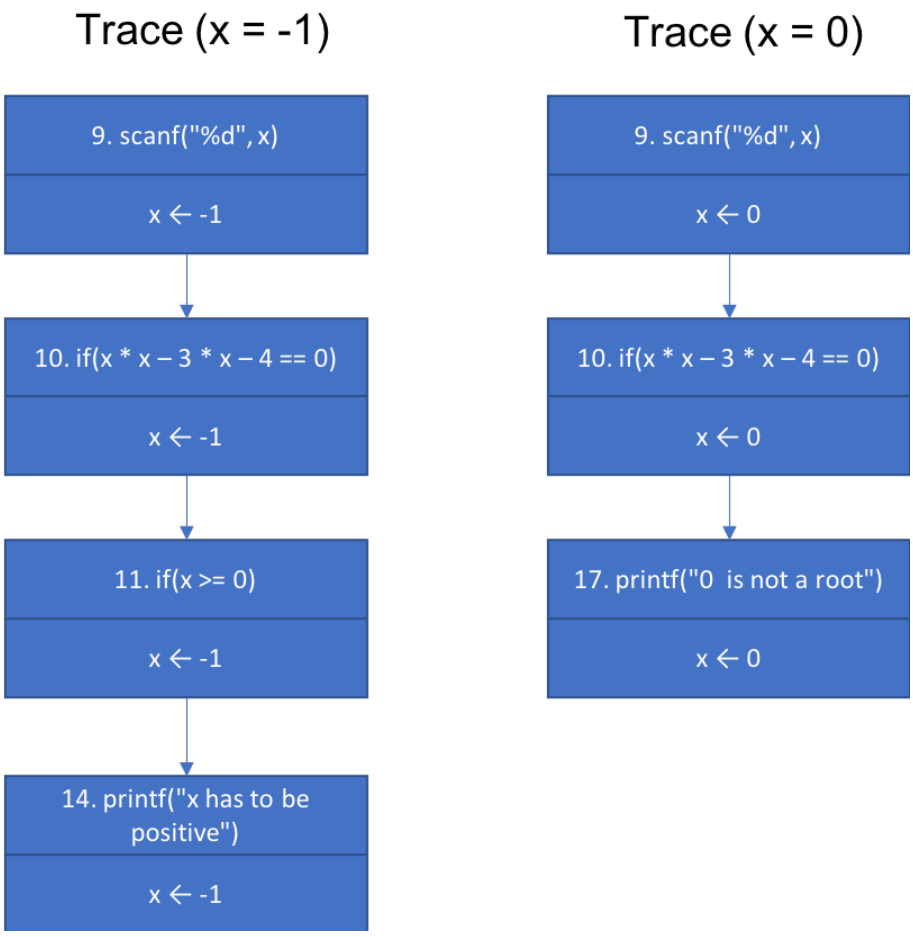


Figure 3.3 – Example of concrete traces of trial and error root solution.

— Line 12 is reached only if x is constrained to root values [line 10] and if it is positive [line 11], otherwise line 14 is reached.

Therefore the execution reaches line 12 only if x= 4. Fig. 3.4 depicts how symbolic execution proceeds, showing its constraints on the symbolic variables for each execution path, thus building a tree of execution traces, which extends the single execution trace of the concrete execution.

In a nutshell, symbolic execution traverses the code considering the register values as symbolic variables instead of concrete values. Thus, all possible execution paths are —*theoretically* —taken into account and the control flow of code (represented by the CFG) can be fully covered. When the execution encounters some branching condition, the current state of the execution is replicated into two new states along with the constraints on the set of symbolic variables corresponding to each path. These constraints create decision prob-

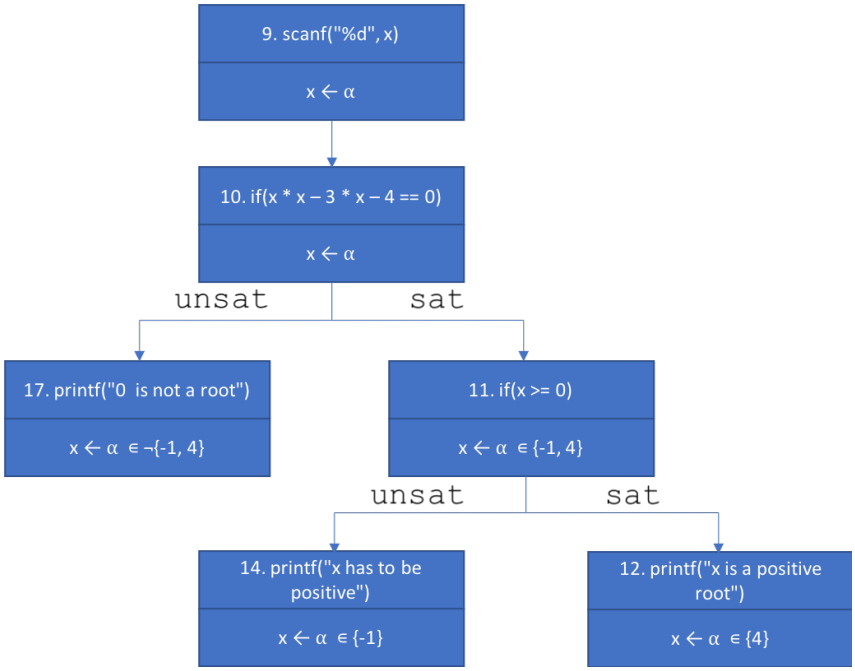


Figure 3.4 – Example of symbolic tracing of trial and error root solution.

lems that can be computed —for testing their satisfiability under the given set of conditions —with the aid of theorem solvers. For more details on symbolic execution, refer to the survey of Baldoni et al. [17].

Satisfiability Modulo Theories (SMT)

Satisfiability Modulo Theories (SMT) address the problem of determining whether a mathematical formula is satisfiable (i.e. if there is an appropriate set of values for the variables that can make a given formula true). It generalizes the boolean satisfiability problem (known as *SAT*), which determines the satisfiability of Boolean formulas, to more complex formulas that can include real numbers, integers, as well as various data structures such as lists, arrays, bit vectors and strings.

Deciding the satisfiability of formulas can be very hard. SAT was the first combinatorial problem shown to be NP-complete [105]. Furthermore, SAT is NP-hard because any problem in NP can be reduced into SAT in polynomial-time [93]; as the SMT problem includes the SAT problem, SMT is itself also NP-hard. Depending on the background theory used in SMT, it can even be undecidable [19]. Nonetheless, over the past decades SMT emerged as a problem-solving method and has been used in practice to solve problems in business and

industry.

There are dozens of tools that target the SMT problem, which as referred as *SMT solvers*. They support languages that allow to express constraints and implement algorithms that solve the satisfiability of these constraints. Apart from technical characteristic (e.g. operating system, APIs, language bindings, etc), these tools are chiefly distinguished by their built-in theories and their underlying language.

Theories commonly supported include *linear arithmetic*, *non-linear arithmetic*, *empty theory*[1], *bit-vectors* and *arrays*. For a gentle introduction on these theories, refer to the article of Barrett *et al* [21].

As a practical example, we chose to introduce the *z3 Theorem Prover* (z3). This choice is motivated by the fact that z3 is a popular well-established tool and because it is a component of our analysis framework (detailed later in section 3.5.1, page 136). z3 is a SMT solver developed by Microsoft Research [234], which supports all the aforementioned theories in addition to *datatypes*, *quantifiers* and *strings*.

Figure 3.5 depicts the overall systems diagram of z3. It shows the existing interfaces (top left), which interact with the *SMTLIB2* component. In turn, SMTLIB2 is a library that implements the SMT-LIB input/output language for SMT solvers as defined by SMT-LIB standard [20]. The formulas entered in the SMTLIB2 component are dispatched to the *solvers* in order to decide their satisfiability; possibly, these formulas can be pre-processed by the *tactics* component with the goal of simplifying them to the solvers. The *optimization* component allows to solve the satisfiability (modulo objective functions) to maximize or minimize values. The tutorial on z3 programming by Bjørner *et al* details all these components [27].

Figure 3.6 shows an example of a z3 code. It is written in SMT-LIB language, whose syntax is similar to LISP syntax —every expression is a legal *S-expression* of Common Lisp [20]. This example addresses the code illustrated in figure 3.2: initially a symbolic variable `x` is declared [line 2], then the constraints of being a root of equation $x^2 - 3x - 4 = 0$ [line 5] and a positive value [line 6] are declared; finally, the satisfiability of the constraints on `x` is verified [line 9] —which returns `sat` when the code is executed —and a representation of a model computed by the solver is provided [line 10] —which returns `(model (define-fun x () Int 4))` when command `(get-model)` is invoked.

For greater details on the theoretical aspects of SMT, refer to Fontaine's habilitation [93]. For greater details on the practical matters related to SMT-LIB, refer to its official documen-

1. Theory that deals with *uninterpreted functions*, which symbolically represent functions that are not explicitly defined.

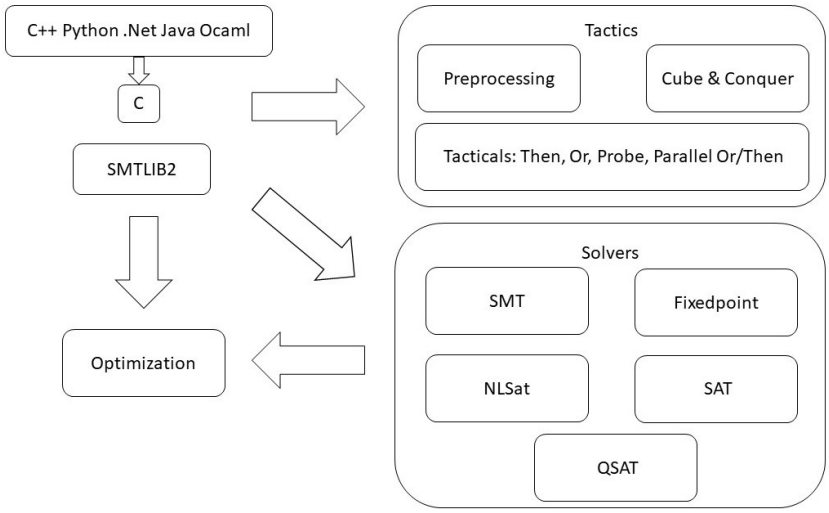


Figure 3.5 – Overall system architecture of z3 [27].

```
1  ; Variable declarations
2  (declare-fun x () Int)
3
4  ; Constraints
5  (assert (= (- (- (* x x) (* 3 x)) 4) 0))
6  (assert (> x 0))
7
8  ; Solve
9  (check-sat)
10 (get-model)
```

Figure 3.6 – Example of z3 code in SMT-LIB language.

tation [20].

angr

angr [261] is a tool for the analysis of binary files, which supports *concrete*, *symbolic* and *concolic* (i.e. a mix of concrete and symbolic) execution. angr is written in Python and is delivered as a Python package composed of the following modules (shown in figure 3.7):

— **CLE**: stands for "CLE Loads Everything" and is responsible for loading binaries and libraries.

— **Archinfo**: contains architecture-specific information.

— **PyVEX**: Python module to handle VEX[3], which is an intermediate representation that

2. Image published at https://angr.io/img/angr_diagram.png (accessed October 2022)
3. VEX is an intermediate language used in Valgrind [204].

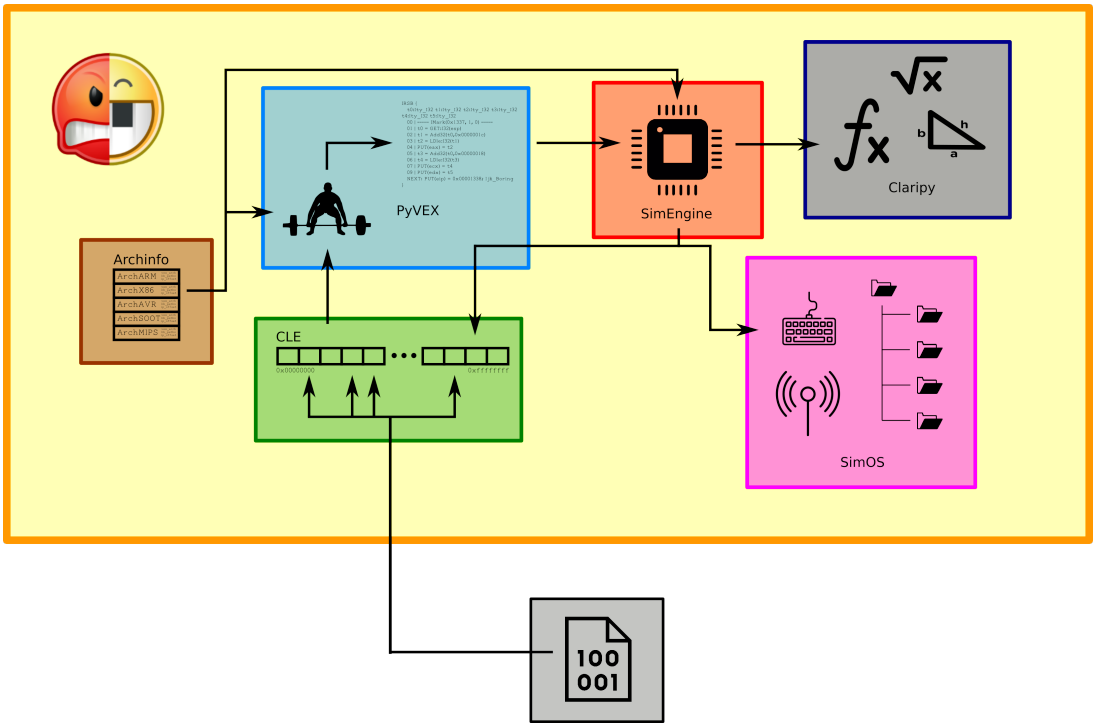


Figure 3.7 – Overall architecture of angr [2].

enables angr to analyze binaries that target different architectures.

— **Claripy**: module that interfaces with a constraint solver.

The execution starts by loading the binary, using the CLE module. For this, CLE needs information about the target architecture of the program, which is provided by Archinfo.

Once the binary is loaded, the angr SimEngine creates a new *active state* —which contains the program memory, registers, file system, and any other so-called "live data" —and starts coordinating the execution.

In its standard configuration, the analysis progresses in steps in which angr uses capstone [8] to disassemble the code in units corresponding to a *basic block*, then angr converts it into *VEX*. Figure 3.8 shows an example of a code translation transforming asm instructions into VEX *superblocks*.

angr uses VEX as intermediary representation for its SimEngine in order to support multiple architectures. By doing this, binary code of multiple architectures can be lifted to VEX without modifying the overall workings of the analysis module.

As the analysis continues, each step accumulates all the constraints defined by the current basic block in a new active state (which belongs to an *active stash*). The previous ac-

```
0x24F275:  movl -16180(%ebx,%eax,4),%eax
 1: ------ IMark(0x24F275, 7) ------
 2: t0 = Add32(Add32(GET:I32(12),# get %ebx and
      Shl32(GET:I32(0),0x2:I8)), # %eax, and
      0xFFFFC0CC:I32)            # compute addr
 3: PUT(0) = LDle:I32(t0)        # put %eax

0x24F27C:  addl %ebx,%eax
 4: ------ IMark(0x24F27C, 2) ------
 5: PUT(60) = 0x24F27C:I32       # put %eip
 6: t3 = GET:I32(0)              # get %eax
 7: t2 = GET:I32(12)             # get %ebx
 8: t1 = Add32(t3,t2)            # addl
 9: PUT(32) = 0x3:I32            # put eflags val1
10: PUT(36) = t3                 # put eflags val2
11: PUT(40) = t2                 # put eflags val3
12: PUT(44) = 0x0:I32            # put eflags val4
13: PUT(0) = t1                  # put %eax

0x24F27E:  jmp*l %eax
14: ------ IMark(0x24F27E, 2) ------
15: PUT(60) = 0x24F27E:I32       # put %eip
16: t4 = GET:I32(0)              # get %eax
17: goto {Boring} t4
```

Figure 3.8 – Example of IR code in VEX (asm x86 instructions in comments) [204].

tive state is partially saved in the program state history, before getting eventually disposed. Whenever the current state comes across a conditional jump, angr evaluates the satisfiability of all accumulated constraints according with the jump condition, proceeding as follows:

— If both possibilities (the condition and its negation) are satisfiable, angr creates two new active states and continues the analysis along each execution path.

— If only one of the conditions is satisfiable, angr creates a new active state that continues along the satisfiable execution path and a *dead-ended* state that indicates the end of the other execution path.

— If both possibilities are not satisfiable, angr created two *dead-ended* states that indicate the end of both execution paths.

Constraints are solved by SMT solvers interfaced by Claripy. The default SMT solver is Microsoft z3, however others can be plugged into angr by writing an appropriate Claripy backend.

The procedure described above emulates a bare metal environment. To support abstractions provided by an underlying operating system (such as files, network, processes and others) angr includes the SimOS module.

Limitations

The main limitation of the symbolic execution in practice is the state-space explosion (also referred as *path explosion*), which exhausts the resources available, such as memory and time. The main sources of state-space explosion are loops and function calls [17].

The ability to explore all the possible traces of a given binary is one of the main advantages of concolic execution. However, this can become an issue if the system does not have enough resources to support the analysis. This is aggravated if the analysis follows a *breadth-first search* (BFS) approach, which expands all execution paths in parallel. The main alternative is to adopt *depth-first search* (DFS), which traverses an execution path as far as possible before backtracking to the deepest unexplored branch; however, this case is particularly disturbed by loops and recursive calls.

Another critical potential weakness of symbolic execution is the constraint solving. Some malware include *opaque predicates* that are commonly used to hinder static analysis [197]. This technique adds conditional jumps that depend on many complex conditions, which always evaluates to true (or false) —thus always taking only one of the possible execution paths at runtime. As consequence, the SMT solver used in the concolic mode can struggle to solve these very complex expressions.

Finally, symbolic execution also struggles with runtime transformations of the code, such as self-modifying or just-in-time (JIT) compilation, that write the instructions in memory as the program executes. If the analysis comes across instructions that are symbolic at runtime, the concolic engine does not know the (symbolic) instruction that must be executed. One expensive solution would be to create active paths for all instructions that meet the constraints of the symbolic instruction; otherwise the symbolic execution stops or misses some potential branches [316]. A similar problem is encountered whenever the address of a jump is symbolic.

3.2.2 Frequent Subgraph Mining (FSM)

Frequent subgraph mining (FSM) is defined as finding all the subgraphs in a given graph that appear more times than a given value. More formally, we define FSM as follows:

Definition 1 (Labeled Graph) *A labelled graph can be represented as* $G(V, E, L_V, L_E, \varphi)$*, where* V *is a set of vertices,* $E \subseteq V \times V$ *is a set of edges;* L_V *and* L_E *are sets of vertex and edge labels respectively and* φ *is a label function that defines the mapping* $V \rightarrow V_L$ *and* $E \rightarrow L_E$*. G is* ***connected*** *if it contains a path for every pair of vertices in it and* ***disconnected*** *otherwise.*

Definition 2 (Subgraph) *Given two graphs* $G_1(V_1, E_1, L_{V_1}, L_{E_1}, \varphi_1)$ *and* $G_2(V_2, E_2, L_{V_2}, L_{E_2}, \varphi_2)$, G_1 *is a subgraph of* G_2, *if* G_1 *satisfies: (i)* $V_1 \subseteq V_2$, *and* $\forall v \in V_1$, $\varphi_1(v) = \varphi_2(v)$, *(ii)* $E_1 \subseteq E_2$, *and* $\forall (u,v) \in E_1$, $\varphi_1(u,v) = \varphi_2(u,v)$.

Given a graph dataset, $D = \{G_0, G_1, \ldots, G_n\}$, the support $\mathscr{S}(g)$ denotes the number of graphs (in D) in which g is a subgraph. The problem of frequent subgraph mining is to find any subgraph g s.t. $\mathscr{S}(g) \geq \tau_{min}$, where τ_{min} is a minimum support threshold.

An overview of frequent subgraph algorithms is given in the survey by Jiang *et al* [133], where gSpan —a graph-based substructure pattern mining [318] —is one of the most frequently cited FSM algorithms. gSpan builds a new lexicographic order and maps each graph to a unique minimum depth-first search code which represents a canonical label. Based on this lexicographic order, gSpan adopts the depth-first search strategy to mine frequent connected subgraphs efficiently.

In comparison with algorithms based on embedding lists, gSpan is very efficient in terms of memory usage. Experiments show that gSpan outperforms the runtime performance of related works by an order of magnitude and is capable of mining large frequent subgraphs in bigger graphs using lower support than its counterparts [318].

3.3 Related Works

To help understanding the contributions of this chapter, this section overviews some related works that adopt techniques that differ from ours but are somewhat similar in relation to the domain and settings. However, our work explores the optimization of an entire toolchain, whereas the related works only relate to individual parts/aspects of our analysis. The organization of this section aligns the related works to each component optimized in this chapter.

Symbolic Execution

Symbolic execution can be use to analyze programs in many ways including source code and binary code representations [17, 37]. However, since this thesis explores the analysis of binary codes, the remainder of this section considers only symbolic execution in this context.

Several tools can perform binary analysis using symbolic execution [42, 46, 250, 261]. Although our work uses only angr [261] for the experiments, the concepts apply to tools including Mayhem [42], S^2E [46] and Triton [250] among others. In practice, each tool has its

own optimizations and weaknesses, as well as its own ability to be (re)configured to modify its execution choices. Since the goal is to consider a holistic approach to the whole toolchain, the results can be adapted to any of these tools (and others).

Our modifications to the behavior of the symbolic execution can be easily adapted to other tools, in particular due to our focus on lightweight modifications that can yield significant improvements when considering other components and algorithms used in our toolchain. We believe that our heuristics can guide other symbolic execution experiments.

Other related works improve symbolic execution in some specific domains [83, 166, 167, 257]. In Li *et al* [167] the authors consider how to steer the symbolic execution engine to specific paths of interest. It considers the execution of sub-traces (constituted by n conditional branches) and uses statistical analysis on the sub-traces already covered to determine which execution states may steer towards the less explored paths. Others researches investigate the optimization of the SMT solver used in the symbolic execution engines [83, 166], but their focus is oriented towards the reduction of the input expressions that are treated by the SMT solver. Sen *et al* [257] consider the merging of states, which has some conceptual similarities with our trace merging strategies (discussed later in section 3.4.3, page 132); however, in their case the merging takes place during execution, which spares the creation of new auxiliary symbolic variables.

SMT Optimization

One of the main goals of this chapter is to find optimizations for the SMT solver for the specific use case of symbolic execution engine [83, 166]. There are many studies on SMT optimization [83, 114, 166, 310], however they frequently focus on solving particular problems whose the theory and expressions (handled by the SMT solver) are known beforehand.

Wintersteiger *et al* [310] optimize their SMT solver for quantified bitvector operations. Similarly, Ramirez *et al* [114] optimize for linear integer/real arithmetics. Erete and Orso [83] and Li *et al* [166] focus the problem of symbolic execution with SMT optimization, however they work with fixed constraints, thus with expressions known in advance rather than generated in runtime during exploration.

Our work approaches SMT optimization from a very different perspective. Since there is no way to know beforehand what expressions will be optimized, our approach intents to target efficiency optimization. We analyze a large set of expressions extracted from a fixed set of binary files and evaluate which expressions are the most expensive and which approaches can result in the best performance improvements.

We welcome the use of our results to guide future SMT optimizations and the proposition of other approaches that target specific SMT problems. Since we have only studied readily available tactics, we recognize that the SMT optimization community can use our approach to study this problem differently (beyond tactical analysis only) to significantly optimize an SMT solver for a symbolic execution engine.

Binary Exploration Heuristics

Several sophisticated loop un-rolling strategies proposed in literature aim to perform an informed guess when no useful information could be extracted from the loop such as the Loop-Extended Symbolic Execution (LESE) [251], the Read-Write set (RWset) [28] and the bit-precise symbolic loop mapping [314].

LESE [251] introduces symbolic variables counting the number of times each loop is executed and then links these variables with features of a known input grammar such as variable-length or repeating fields. When new execution paths are generated due to a branching condition, RWset [28] verifies the memory locations read and written by the code to determine if the remainder of the execution trace can explore new behaviors, or it prunes the execution path otherwise. The trace-based approach proposed in the bit-precise symbolic loop mapping [314] aims to identify the semantics of possible cryptographic algorithms used in obfuscated binary code.

Unlike these techniques, our approach is completely based on heuristics analysis, thus being much less computationally demanding.

Malware Classification

Malware classification is a widely explored area with many different approaches. In this chapter, we focused on malware classification based on behavioral signatures as is common to other works in the literature [24, 40, 47, 67, 72, 94, 174, 266]. Our goal is to set up a toolchain optimization to benchmark its improvements on malware classification, however due to the huge lack of standard datasets for analysis, the direct comparison of our results with other works is impaired. Therefore, this section points to the works that are somehow related to ours, but taking into account the aforementioned caveat of direct comparison.

Many works have tried to classify PE (Windows) malware by their family, however none reports classification explores the use of symbolic execution. By extracting static features (i.e., textual and binary patterns) with YARA, Sun *et al* [275] achieved an F-score of 0.936

with a random forest classifier (RF) over a dataset constituted by 12 families. Still using a RF but extracting behavioral traces through a concrete execution within the Cuckoo sandbox, Hansen *et al* [118] achieved an F-score of 0.864 on a dataset of 5 malware families. Based on an RF fed with a mix of static and dynamic features, obtained by the monitoring of the execution using the HookMe tool, Tian *et al* [282] and Islam *et al* [131] achieved an accuracy score of ~ 0.97 (misclassifications not reported).

A behavioral graph representation extracted by means of taint analysis and dynamic execution in a sandbox was also adopted by Ding *et al* [73]. In their work, the authors removed duplicate sub-graphs representing identical call sequences and grouped all the sub-graphs extracted from instances of the same malware family. The graph matching was then performed by looking for graph isomorphism. Experiments on a dataset of 6 families with about 200 samples each obtained an accuracy of about 0.96 when doing binary classification between a given family and cleanware —no multi-class classification was performed, but six independent binary classifiers were used for the results, one per family.

In Park *et al* [212] the authors considered dependency graphs and used graph similarity to classify malware. Their approach aimed to create clusters of graphs based on their distance and to compare these clusters with ground truth for classification. When tested on a dataset of 300 samples from 6 families their results were mixed with some families achieving only ~ 20% accuracy. In contrast to our work, their clustering approach required to preset the total number of clusters, which is rarely known in practice.

Said *et al* [24] used a common subgraph approach for binary classification —a single malware family and the remainder of samples being cleanware —which is similar to our approach. Their results achieved a very high accuracy score (they used $F_{0.5}$-score attaining 99.40%) after tuning parameters to their algorithms. Yet, this was only for a single family and not multi-class classification as we target in this chapter. Nonetheless, the optimizations in our work can directly applied to their approach.

Other works [144, 212, 236, 283] follow similar approaches in concept. These include using other forms of graphs to represent behavior and some form of graph similarity for clustering of common behaviors [144, 212, 283].

Rieck *et al* [236] used Support Vector Machine (SVM) on a vector which represents the behavior description obtained by running the malware on a sandboxed environment. They claimed to improve results on malware not already detected by some other system by approximately 70%, but they do not provide details about their results. Like our approach they detected malware families using behavioral signatures and machine learning techniques.

Kinable and Kostakis [144] used the clustering of call graphs to analyze malware samples. They used a dataset of 194 samples from 24 families and concluded that the K-medoid clustering is not effective for their scenario. The density-based cluster algorithm DBSCAN, not requiring any prior knowledge on the number of cluster, showed a better performance. Therefore DBSCAN was evaluated over two larger datasets respectively of 675 and 1050 samples. The results showed 29 correct clusters and 19 wrong clusters for the first dataset, and 36 correct and 14 wrong clusters for the second dataset. For both datasets a large amount of samples were not classified in any cluster (respectively 260 and 253). Unlike our approach (in the current chapter), Kinable and Kostakis adopted a unsupervised classifier, mainly focusing on the homogeneity of the clusters (i.e. samples within a cluster belonging to a single family). This choice had an impact on the number of clusters, which was significantly higher than the expected result as per the chosen ground truth.

Graph-based representations of malicious behaviors were also adopted for malware running on the Android platform. The samples were generally dynamically executed but similar classifiers were used as those adopted for classifying Windows malware, e.g. SVM [315], graph isomorphism [329], Markov Chain Monte Carlo sampling [96], graph kernel [99] and Convolution Neural Network [126]. Even if the absolute performance results of the classifiers cannot be directly compared with our work, the presence of several similarities with our behavioral analysis toolchain means that many of the optimizations and lessons learned on our work can apply in that case as well.

3.4 Methodology

This work builds a binary analysis toolchain based on symbolic execution to explore various choices in the formal approaches and heuristics used in the analysis. The goal is to better understand how to use these tools to represent malware behaviors and classify them according to this representation. This does not only provides insight on how to build more effective malware analysis engines, but also on how to tune the key components of such toolchains for the analysis of binary files.

Our approach to optimize the behavior analysis toolchain for malware analysis is sketched in a workflow in figure 3.9. Our dataset is composed of malware and cleanware that are analyzed to extract the SMT expressions that are generated during the symbolic execution, which are then fine-tuned using various tactics available in z3. The optimization of the symbolic execution is also made directly on the angr framework, with the proper evaluation of

the impact on malware classification with respect to our graph-based behavioral signatures.

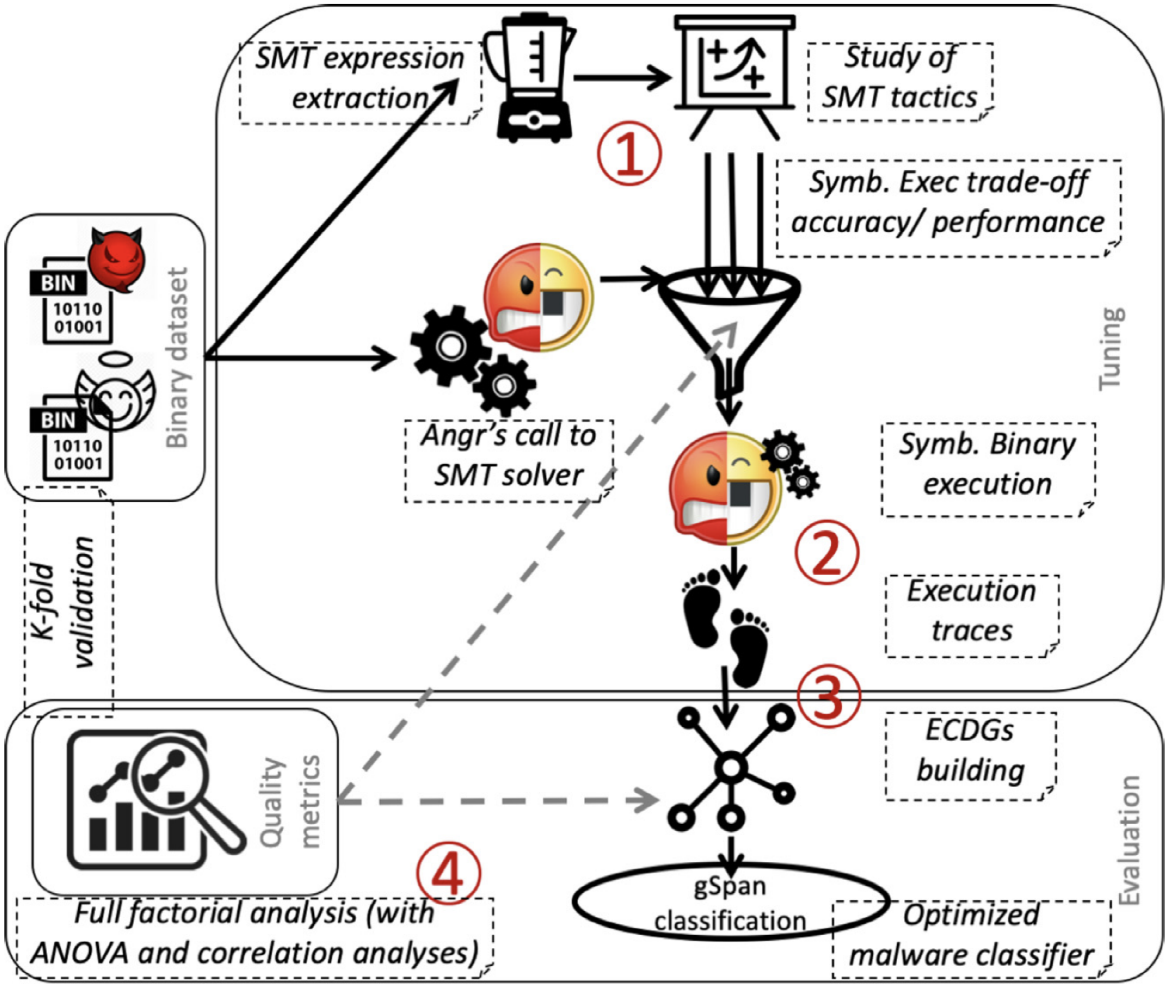


Figure 3.9 – Overall workflow of our malware analysis toolchain

More precisely, we composed a binary dataset consisting of malware and cleanware that was analyzed through a k-fold cross-validation. To optimize the SMT solver, a benchmark of SMT expressions extracted through the hooking of interactions between angr and the SMT solver (ref. 1 in figure 3.9) was built. The SMT tactics available in the z3 theorem solver was analyzed and combined to find the sweet spot between accuracy and performance, which was then integrated to our use of angr.

The implementation and optimization of binary exploration (ref. 2 in figure 3.9) as well as the heuristics for the graph generation (ref. 3 in figure 3.9) were addressed. Finally, the whole optimization of the symbolic execution, graph-building, and classification heuristics

were holistically evaluated in the context of malware detection and classification using a full factorial design of experiments (with ANOVA [268] and correlation analyses) (ref. 4 in figure 3.9).

This is a first-of-a-kind attempt to assess the impact of the different configuration options on the efficiency and efficacy of malware classification using symbolic execution. This analysis allowed us to gain insights on the interactions between the various components constituting the behavioral analysis toolchain.

3.4.1 SMT Optimization

One important efficiency concern is to remove infeasible states from angr exploration. For instance, upon a conditional jump, depending on the constraints, it may be the case that the constraint conditions may be met by only one execution path. To determine when states are infeasible, angr calls an SMT solver and prunes the execution paths that were found infeasible.

In angr this is done by using the z3 solver to answer SMT decision problems with results of: *sat* (satisfiable), *unsat* (unsatisfiable), or *unknown*. Typically an unknown result is due to z3 failing to find a sat or unsat solution in reasonable time. The following requests to the SMT solver are used in angr:

- `simplify()`: simplify constraints for a subsequent satisfiability check;
- `satisfiable()`: checks the constraints assigned to the current state;
- `batch_eval()`: evaluates a list of constraints and provides a solution;
- `max()/min()`: finds max/min value that respects the constraints.

The main limitation of symbolic execution in practice is state-space explosion, which is the root cause of problems with consumption of resources (c.f. section 3.2.1, page 122). Thus the best solution to these problems is to optimize the usage of the SMT solver in two directions: provide an answer faster, and provide a `sat` or `unsat` answer instead of *unknown*. This improves the accuracy of the exploration, and removes infeasible paths optimizing the whole exploration.

Symbolic execution faces an accuracy/performance trade-off: more resources (time, memory, etc.) potentially lead to better exploration. Thus the optimization of core components of the symbolic execution such as the utilization of the SMT solver are crucial to improve the results.

The z3 theorem prover used by angr is equipped with a few hundred parameters, grouped in modules, whose purpose is to tune z3 with the aim of speeding up the evaluation of the constraints. By using the more than 300 available parameters, a Microsoft Research team working on z3 has identified about 100 problem classes and designed optimized approaches (tactics) to tackle them. Even when the class of problem for an expression is known, a tailored tactic is not guaranteed to produce a benefit, since tactics are heuristic strategies and not proven optimizations.

Tactics tend to be highly tuned for known classes of problems but may perform poorly for new classes of problems [198]. New tactics can be easily defined with the available parameters, and also by combining other tactics. A total of eight combinators of tactics known as *tacticals* [198] are available in z3. Tacticals operate by applying a set of tactics: sequentially, in parallel, or alternatively when a prior tactic fails.

For example, to simplify constraints angr uses a customized tactic built as a sequential application of five predefined tactics provided by z3. These five tactics are: `simplify`, `propagate-ineqs`, `propagate-values`, `unit-subsume-simplify`, and `aig`, and respectively correspond to: simplification, removal of inequalities, constant propagation, simplification of unit clauses (i.e., clauses composed by a single boolean literal), and compression of Boolean formulas. When no tactic is provided (named `NO.TACTIC` hereinafter), the z3 solver will try to infer the logic the expression belongs to, and to apply the corresponding tactic (if available).

Previous works have considered optimizing the solver for a specific class of formulas e.g. quantified bitvector [310] or linear integer/real arithmetics [114]. Instead, in our work we do not have any a priori knowledge about the expressions generated during symbolic execution. It is important to note that, once instantiated, the z3 solver cannot change tactics. In the beginning of symbolic execution angr instantiates a solver that is used for all SMT requests.

Thus, we are looking for a tactic (or a set of tactics combined by means of tacticals) that is expected to work efficiently on all types of angr requests. This analysis addresses the following question **RQ3.1**: *How to tune the SMT solver to improve performance of symbolic execution?*

3.4.2 Enhanced Symbolic Execution

State-space explosion is known to be a critical issue in symbolic execution. The leading sources are loops and function calls (specially in the case of DFS exploration) or the existence of many conditional jumps in the code (specially in the case of BFS exploration).

Our call tracer includes a parameter to limit the number of execution paths explored si-

multaneously. By default, at each execution step angr considers all paths that are not finished yet, thus exploring the binary following a BFS strategy. If parameter `max-active-paths` is set, only a limited number of paths are executed while the rest await in a stash. Thus the binary is explored in a combination of BFS and DFS, which potentially controls overall memory consumption.

Due to the creation of new states on each conditional branch, loops are a particular concern for angr due to potential state explosion when loops have many iterations. To address this here we consider heuristics to detect potential state explosion in angr and reduce the number of active states. In practice, this is achieved by detecting how many times an instruction has been visited —we call this case a *concrete loop* —and if this goes over a threshold, then this active state moved to the stash. Stashed states are given low priority for angr to execute, but may be executed if no other active state is available.

We also consider the case of *symbolic loops*, in which an unconstrained variable may cause an infinite number of loops. For instance, if the code contains `while(i < x){i++}`, where `x` is an unconstrained symbolic variable, both conditions (i.e. `i < x` and `i ≥ x`) are always satisfiable. Similarly to concrete loops, when symbolic loops are detected the stashed states are given low priority for execution.

Another major issue, which is not discussed in other works (e.g. the survey of Boldoni *et al* [17]), relates to type signature and call convention. They are necessary to define the inputs and outputs of function calls and determine stack management. Whenever the analysis comes across a call to an unknown function, angr attempts to use heuristics to resolve the call, however there is a non-negligible risk of mismanaging the stack, thus engendering failures in the remainder of the analysis.

In angr, calls to external libraries can be replaced by symbolic procedures called *SimProcedures* to mitigate the problem of path explosion. In our analysis, we adopted this approach. *SimProcedures* are Python implementations of functions (i.e. procedures) that replace the execution of the code of the external library. Each *SimProcedure* requires the precise definition of the input and output parameters, i.e. the procedure *prototype*. The lacking or mismatching of these information can potentially lead the symbolic execution to a defective state. By the time of our analysis, angr had prototypes for about 100 Windows libraries (e.g. `advapi`, `glibc` and `kernel32`), accounting for about 3500 *SimProcedures*. We carried out a pre-analysis phase to extend angr with 24052 (stub) *SimProcedures* of function frequently found in our analysis with their correct call conventions and prototypes.

We also included a `step timeout` parameter, which defines a maximum time threshold

for the computation of a step during symbolic execution. This mainly corresponds to the time consumed by Z3 and therefore is indirectly related to the SMT optimization analysis.

All the parametrization, improvements and heuristics developed around angr in our call tracer intend to target **RQ3.2**: *How to effectively apply heuristics that reduce the problem of state explosion in symbolic execution?*

3.4.3 Impact on Malware Classification

Although our main focus in this chapter is to assess and improve symbolic execution for the effective extraction of execution traces, we are particularly interested in measuring the impact of these improvements on behavioral-based malware classification. To this goal, we considered External Call Dependency Graphs (ECDG) —a specialization of System Call Dependency Graphs (SCDGs) that is detailed and studied in chapter 4 (page 153) —as in the work of Said *et al* [24], which have been proved to be an effective approach to build binary behavioral signatures [24, 40, 47, 67, 72, 94, 174, 266].

We use gSpan [318] as a common subgraph mining algorithm to compute the similarity between ECDGs. The ECDGs are built from the calls traced during the symbolic execution of the binary files. The binary files present in the training set constitute the basis of the the supervised learning phase, in which ECDGs are labeled as malware (including family) or cleanware according to the ground truth.

The symbolic execution used to trace the calls targets the external calls [4] as well as their address and arguments. For the description of the binary file's behavior, we focus only on external calls and their relationships. The source address, arguments, and position in the trace allow to understand the relations between the calls and to build a directed graph to represent the behavior of the binary file.

The vertices of the ECDG represent the external calls, while the (directed) edges represent the information flows between calls (i.e., input/output from a call used as argument of another call), where the direction reflects the order of the calls in the execution trace. If there are external calls to functions with the same name but different addresses, then distinct vertices are created. If not all the external calls have interdependencies (which is a very common case), the graph will contain several subgraphs representing different sub-behaviors encoded in the binary. Furthermore, since symbolic execution can traverse multiple execution paths simultaneously, the graphs may contain disconnected components for each

4. External calls are calls to external libraries or system calls. Refer to chapter 4 (page 153) for a more detailed discussion about ECDGs.

execution path.

Since several execution traces can be generated according to the conditional branches that have been evaluated `sat` by the constraint solver, ECDGs can be built in different ways. In particular, we consider three parametric heuristics:

- *merge-calls*: define whether external calls having the same name and address are merged or not. Merging the calls has the potential of providing a higher level of abstraction over the binary behavior by considering a more compact model and simplifying the graph traversal in the machine learning classification phase. However such an abstraction could cause the loss of some details important to discriminate the malware families.
- *disjoint union/traces-merge*: each individual trace traversal obtained with the symbolic analysis generates a subgraph. The disjoint union of these graphs is then used to include all the (sub)behaviors in a single graph (as in Said *et al* [24]). The alternative is to merge prefixes of traces to connect more components reducing the overall graph size (similarly to Macedo and Touili's [174] system call dependency trees). Unlike dynamic execution, where a single execution path is traversed, the symbolic execution comprises multiple traces due to the ability of exploring multiple execution paths in parallel. The *traces-merge* option attempts to generate more connected graphs (by looking for common prefixes in the set of traces), whereas the disjoint union pursuit a different approach, i.e. treating the traces as independent executions (as if multiple runs of concrete executions have been made).
- *min-trace-size*: a minimum number of calls have to be present in a trace to be valid, thus discarding shorter traces. The rational behind such a heuristics is that very short traces could prompt symbolic explorations of bad quality e.g., due to anti-analysis or obfuscation techniques. Discarding such traces could prevent biasing the classifier over graphs that do not represent the behavior of a malware family itself but only less important code segments (e.g. obfuscated header, in the case of packed binaries).

These parametric heuristics directly affect the malware classifier because they have an impact on the set of execution traces used to build the ECDGs. The choices of these metrics address **RQ3.3**: *What is the impact of enhancing symbolic execution for binary analysis in a malware classification scenario?*

Additional parametric heuristics not considered here include `max-trace-size` and `total-size-limit`. We have not explored these two additional parametric heuristics because, according to our previous experience, both are only suitable for a scenario with resource con-

straints. Therefore, by not setting any bound to these parameters, we do not have to deal with any limiting factor on the performance evaluation of the classifier.

The following graph metrics have been computed for each set of ECDGs: number of unique system calls, number of edges, number of vertices, size of the largest weakly connected component[5], number of weakly connected components, graph density (defined as: $\frac{\#edges}{\#vertices \cdot (\#vertices - 1)}$), and number of unique edges (i.e., information flow having same source, destination and arguments).

Note that the efficiency and effectiveness of symbolic execution affect the quality of the information contained in the traces, therefore it influences the building of ECDG-based behavioral signature and the mining process. The aforementioned graph metrics are later used to form a feedback loop used for tuning the parameters of the symbolic execution, as shown in figure 3.9.

ECDG classification and evaluation

Here we present our multi-class classifier, which selects the most fitting malware family for a given binary file. Our approach exploits the mining of common behaviors for malware families. More precisely, the collection of ECDG behavioral signatures for each family of malware[6] is mined to find common subgraphs using gSpan. The largest common subgraph found in the mining process becomes the ECDG-based behavioral signature for that malware family.

To classify a new binary given its ECDG, a comparison is done using the ECDG characteristic of each malware family, i.e. a ECDG-based behavioral signature. This comparison is carried out by using gSpan to find the largest common subgraph between the sample and the family signature, i.e., how much of the malware family signature is contained in the sample.

We then compute the *similarity score* as the percentage of the malware family subgraph that is contained in the sample. A sample is then classified according to the malware family with which it has the highest similarity score, provided that this similarity is above a predefined threshold. If a sample does not meet the similarity threshold for any malware family, it is then classified as cleanware.

This approach replicates the approach used to classify a single malware family [24], but in the context of multi-class classification using graph mining, which had not been studied before.

5. Maximal subgraph in a directed graph for which there exists a path between each pair of vertices.
6. Cleanware are not considered a family since they do not have a reason to share a common behavior.

For a binary classifier (i.e., discerning malware from cleanware), it is possible to derive several metrics (c.f. section 2.2.1, page 60): (i) *recall* (sometimes also called *sensitivity*) is the number of correct positive classifications over the total number of positive cases, (ii) *precision* is the ratio of true positives over all the positive results returned by the classifier, and (iii) *accuracy* (or *trueness*) is the ratio of correct classifications over the total number of samples. We also take advantage of the *F-score* as a simple but effective metric to measure the accuracy of a binary classifier.

Since in our context we are also interested in classifying the malware family of the sample, we adopted the *micro-average F-score* which considers the counting of correct and incorrect classifications for each class independently before appropriately summing them up to compute precision and recall.

In this classification problem, the usual trade-off between sensitivity and false positives comes to light and each model designer chooses how to balance it. Nonetheless, as a guiding principle, it is worth taking into account that mis-classifications between families are less severe than mis-classifications in a binary classifier. Moreover, problems could also arise by differences in the label assignment of external models, as the same instance can be tagged with different labels that indicate contradictory malware families. For this reason, we leveraged VirusTotal [294] for identifying the samples that have a large consistency on the attribution of the malware family.

3.5 Experiments

This section presents the experimental methods used to address the research questions posed in this chapter —in line with our proposed methodology. In the following, we first describe our implementation setup (section 3.5.1), then we focus on how we optimized z3 to improve the performance of symbolic execution of binary files (section 3.5.2), then we consider potential heuristics to improve the extraction of traces with angr tuning, and finally we describe the optimization of the whole binary analysis toolchain from the symbolic execution to extract the system calls, to the ECDG building process for the classification of binaries according to ECDG-based signatures.

3.5.1 Implementation Setup

The experiments have been performed on a cluster constituted by 12 machines having four Intel CPU E5-2660 v4 @ 2.00 GHz with 132 GB of RAM each, running Ubuntu 16.04 LTS. We refer to this cluster of machines as *Madlab*.

angr-extractor

Our **code analyzer** (i.e. angr-extractor) uses angr [261] *version 7.8.8.1* with *z3 version 4.5.1.0.post2* to build a software layer (written in Python) that traces the calls invoked by the main binary object [7] to any shared object or system call (i.e. external call).

Instead of setting breakpoints at the addresses of the functions, as in traditional implementations of call tracers (e.g *ltrace* [33]), our symbolic tracer steps through code, inspecting all active states whose execution addresses are outside the main object. When these states are found, hinting that the execution of some external procedure may be ongoing, an attempt to resolve the call is performed. To do it, the current address is looked up in angr internal collectedmapping of addresses and *SimProcedures*, working as a generic hook to any external call.

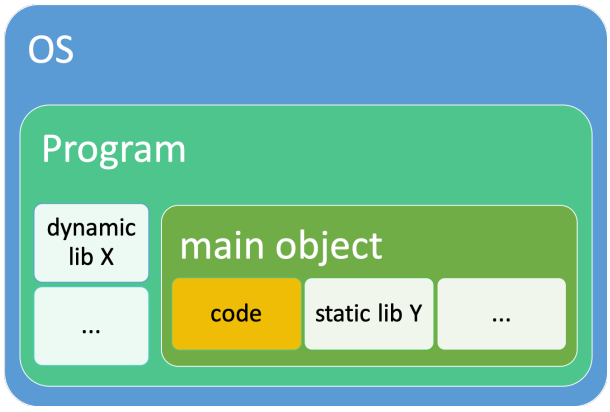


Figure 3.10 – Object code's boundaries: *main object* (us) vs. program (state-of-the-art)

To provide higher isolation for the analysis environment, we set angr not to execute code outside the main object (*auto_load_libs* option unset). Missing *SimProcedures* are replaced by stubs that only return an unconstrained symbolic variable, allowing to virtually resolve any runtime dependency. As a side effect these *stub SimProcedures* may alter the real execution of the file.

7. Memory segment corresponding to the program binary code, without considering any shared object (i.e. dynamic loaded libraries)

Additionally, notwithstanding angr fine heuristics, unknown call conventions and unknown function signatures can also incur defective symbolic execution. Therefore, we extended angr with 24052 stub *SimProcedures* of functions commonly used (along with their correct call conventions and signature definitions), which proved to be a very effective tactics to improve the quality of symbolic executions.

Our symbolic execution implementation is parameterized with `step timeout`, `max active paths`, `loop threshold`, `branching loop threshold` (for symbolic loops) and `SMT tactics` (for the underlying *z3 tactics*). Several additional tunable parameters are fixed in our experiments: the global timeout (fixed to 1h) which limits how long the angr-extractor can run before terminating and the global memory limit (fixed to 10GB) which restricts how much RAM the angr-extractor can use before stopping the creation of new active states.

ECDG-builder

The **graph builder** (i.e. ECDG-builder) builds ECDGs using the call traces generated by the angr-extractor. Each call trace contains the external calls found during code analysis, with their arguments and return value (if any).

This module was developed as an *ad-hoc Python* script. Note that symbolic execution may traverse multiple execution paths during analysis, so instead of generating a single (linked) list of calls it may produce a set of call lists. As discussed in section 3.4.3 (page 132), this opens new possibilities for the definition of call graphs, which may combine calls found in different execution traces.

Our implementation takes parameters to un/set `disjoint union`, `merge calls` and `merge trace`. When `disjoint union` is set, functions comprised in different call traces are not merged, but their interdependencies create edges between functions of all traces. When `merge calls` is set, similar functions are grouped into a single node. When `merge trace` is set, common call trace prefixes are combined into a single subtrace, thus transforming a set of call traces into a tree-like structure. All options can be combined, except `disjoint union` and `merge traces` that are mutually exclusive.

quickSpan

Our **graphs analyzer** (i.e. *quickSpan*) consists of an optimized implementation of *gSpan* [318] in C to mine common subgraphs given a set of graphs.

Our quickSpan implementation supports many tunable options for different domains

and has achieved 35-fold speedup in comparison to Yan's original implementation with multi-threading, and 6-fold speedup with a single thread. It also reduces more than 100 times the memory usage, which makes it fit to mine large graphs with a low memory footprint.

Orqal

We deployed our experiments in a distributed architecture based on Docker, as all the modules detailed above can be encapsulated in Docker containers [185]. Therefore we developed *Orqal*[8] ([OR(Q)]chestration of [AL]gorithms), which aims to optimize the use of resources on Madlab by orchestrating the different modules as individual Docker containers.

Orqal is a simple orchestrator (written in Python) to manage docker clusters, which monitors and distributes the Docker workload on different nodes of Madlab. Orqal is provisioned with a REST API that connectors can call to manage the Docker containers on the cluster (see figure 3.11) to schedule jobs[9] and retrieve data generated during the execution.

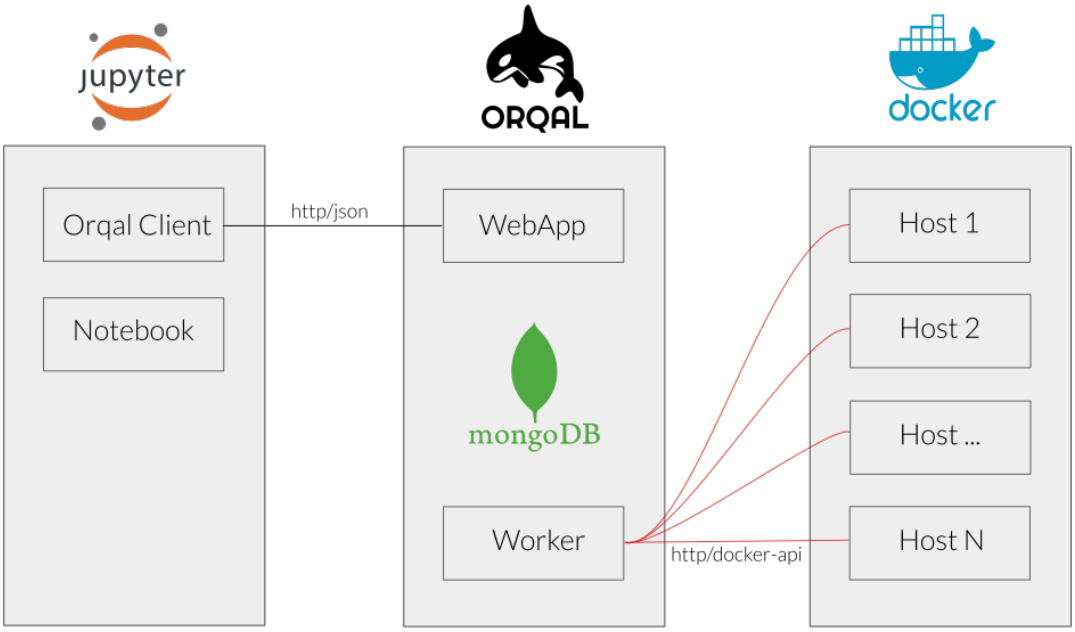


Figure 3.11 – Overall architecture of Orqal

8. Available at https://github.com/scampion/orqal.
9. We refer to each individual container run as a *job*, regardless the module in use.

We also developed a Python connector (i.e. Orqal Client [10]) that allows to manage Orqal in Python scripts (e.g. in Jupyter notebooks [147]). No special configuration is needed on the nodes of the cluster other than the Docker service running and listening to a network port. Additionally, a dashboard is provided to monitor average load per node, jobs scheduling and a redoc API.

Dataset

Two different datasets were used in the experiments to prevent any overfitting of z3 optimization with the remainder of the analysis, which includes parameter tuning of symbolic execution and evaluation of this tuning on ECDG classification.

For the first dataset we randomly select a subset of the binary files in our database to obtain a set of SMT expressions for the purpose of benchmarking. This dataset included a mix of malware collected from VirusTotal in 2018 and binary files manually obfuscated with Tigress [56], which are likely to produce complex SMT expressions.

The second dataset is composed of 8 malware families —coupon-marvel, gamemodding, installbrain, multiplug, jeefo, detroie, mira and addrop —with 25 binary files for each. The malware samples were collected by Cisco in 2018 and provided to our research as part of a scientific project. An additional family of 25 cleanware binary files, taken from a fresh installed Windows 7 Pro was also included. The list of SHA1 and family of each binary in this dataset are reported on the tables in appendix 6.6.

3.5.2 Optimizing z3 for symbolic execution of binary files

To isolate the optimizations on z3 from those that concern angr, an *ad-hoc* experiment was done to identify key tactics and tacticals. The experimental settings included a benchmark of SMT expressions extracted from symbolic executions of binary files (including both malware and cleanware). The different z3 tactics were evaluated using these expressions to identify the most promising tactics. The outcome of this analysis allowed to enhance angr symbolic analysis with the best combination of tactics in the SMT solver setup. This optimized setup was used in the remainder of our experiments, which concerns the whole analysis toolchain (c.f. section 3.5.3, page 143).

A symbolic execution was performed on the selected binaries and the expressions relayed to z3 were recorded. From these expressions, we obtained a baseline benchmark (c.f.

10. Available at https://pypi.org/project/orqal/.

table 3.1) of the number, kinds, and results of calls to be used for comparison later. We set a timeout of 2 minutes for symbolic execution, as it showed to be long enough to collect a suitable amount of expressions.

Table 3.1 – Benchmark of SMT expressions extracted through symbolic executions

	Dataset: Malware + Cleanware
# of files	906
extration from `simplify()`	54460
extration from `satisfiable()`	103881
extration from `batch_eval()`	47585
extration from `max()`	177439
extration from `min()`	112712

The expressions collected were solved by z3 with different tactics. As a result of these experiments an optimized set of tactics (combined in tacticals) was chosen to be used in the remainder of our experiments, which target the optimization of the whole toolchain. Note that finding an optimized tactic is only a partial answer to **RQ3.2** [11], as part of the answer also comes from the incorporation of these results into the analysis toolchain.

Results

To create an "optimized strategy" for the toolchain experiments, an evaluation of z3 behavior on typical angr calls was performed. The performance of z3 built-in tactics (including `NO_TACTIC`) on all the benchmark expressions was evaluated. Initially no limit was put on the resources available to solve the expressions and this led to some expressions requiring several hours for z3 to find definitive results with a given tactic. To achieve efficient symbolic execution in practice and resolve this potential time cost, a one-minute timeout was set for z3; `unknown` is returned when timeout is reached without a definitive answer (`sat` or `unsat`). The limit of one minute was chosen to allow sufficient time for z3 to attempt to solve theSMT expression, while not causing a prohibitive time cost in symbolic execution —where many calls to z3 are made for each binary, such as in the experiments considering the whole analysis toolchain in section 3.5.3.

The evaluation of each tactic on each SMT expression was repeated 30 times and the results averaged to prevent concerns from caching, scheduling, OS behavior, etc. Two metrics have been considered in the evaluation of the effectiveness of the tactics: execution time

11. **RQ3.2**: *How to effectively apply heuristics that reduce the problem of state explosion in symbolic execution?*

and ability to successfully perform a satisfiability check. The baseline considered here is the `NO.TACTIC` tactic that does not exploit any special knowledge about the SMT expression to be solved. A summary of the performance of the selected tactics over the benchmark of SMT expressions is presented in table 3.2, the best result [12] for each type of call being in bold.

Table 3.2 – Results of the selected z3 tactics of SMT expressions extracted for benchmark.

	`simplify()`		`satisfiable()`		`batch_eval()`		`max()`		`min()`	
Tactic	# solved	Exec. time	# solved	Exec. time	# solved	Exec. time	# solved	Exec. time	# solved	Exec. time
`NO.TACTIC`	54460	694.22812	103881	765.75330	47585	603.7525	177439	1764.67108	112712	1087.35926
`qfufbv_ackr`	54460	**386.65576**	103881	1139.12232	47585	449.5783	177439	1203.88404	112712	478.72028
`solve-eqs`	-	-	25136	20.14145	-	-	-	-	-	-
`fnra-nlsat`	642	90.17725	93194	78.75564	4783	196.3956	26419	189.45631	37930	199.52513
`qfnra`	642	4415.16436	93194	160.87066	4783	287.8888	26419	324.62751	37930	305.77329
`qfidl`	54460	3977.16036	103881	**671.98977**	47585	318.2233	177439	697.96655	112712	350.42974
`qflra`	54460	3968.18507	103881	2653.76949	47585	**279.9188**	177439	619.84178	112712	295.73773
`qfauflia`	54460	3954.05094	103881	1091.53502	47585	296.3621	177439	**605.63197**	112712	295.56502
`smt`	54460	3983.59492	103881	2645.59894	47585	284.2394	177439	610.80981	112712	**292.04155**
Total expressions	54460		103881		47585		177439		112712	

For each of the winning tactics from table 3.2, a deeper analysis is performed to compare the calls of each type (e.g., `simplify()`) of a single binary, in order to prevent the overfitting of the tactic with respect to the expression type.

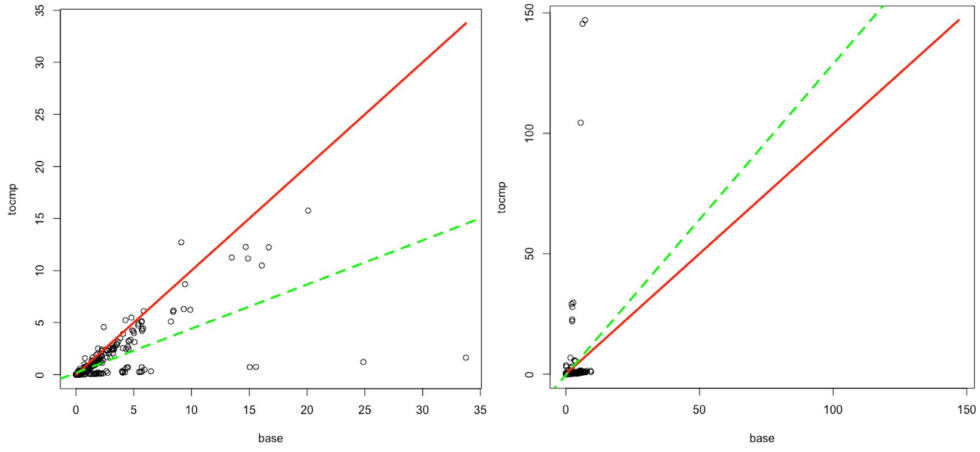
Results are presented in figure 3.12 where each plot compares the base time taken per binary (x-axis) with the time taken using the best tactic for that expression type (y-axis) as chosen from table 3.2. The x-axis is the `NO.TACTIC` and the y-axis is the compared tactic. Each data point represents the total execution time for all expressions from the corresponding function in a given binary. The dashed line in light green represents a linear regression model, such that if the green line is below the diagonal (in red), the customized tactic outperforms the default one in terms of execution time.

Figure 3.12a compares `NO.TACTIC` and `qfufbv_ackr` [13] on the execution time to solve the benchmark's SMT expressions using `simplify()` (c.f. 3.4.1, page 129). Observe that (except for very few binaries) using `qfufbv_ackr` instead of the default tactic is clearly better.

As reported in the scatter plot in figure 3.12b the `qfidl` [14] did not attained any significant improvement when considering the `satisfiable()` calls.

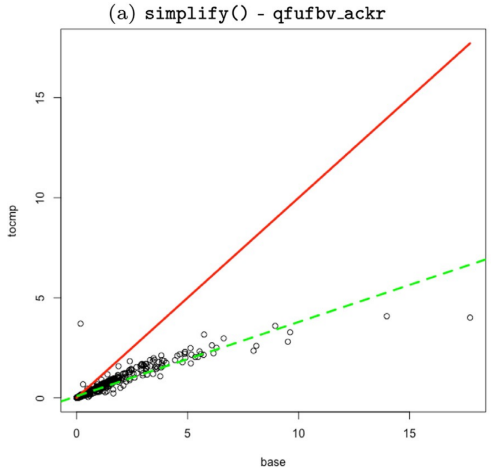
In contrast, the expressions from `batch_eval()` using the `qflra` [15] instead of the `NO.TACTIC` has a clear benefits over all the binaries, except for very few cases (see figure 3.12c).

12. Note that this is the least time spent, potentially including the timeout in case of failure to find a solution.
13. Quantified Free formulas (QF) over bitvectors (BV) with uninterpreted sort function (UF) and symbols solved with Ackermanization.
14. Boolean combinations of inequality constituted by differences between integers variables and constants.
15. QF linear real arithmetic, i.e., Boolean combinations of linear polynomials over real variables.

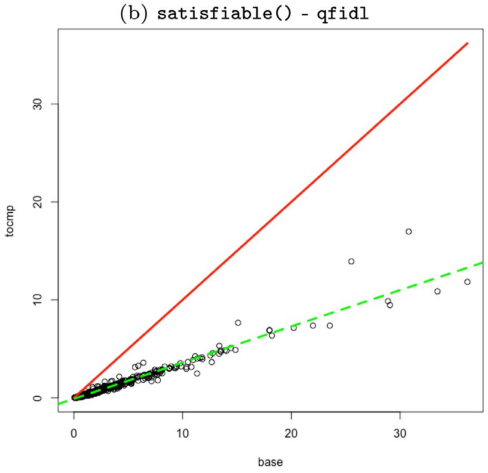


(a) `simplify() - qfufbv_ackr`

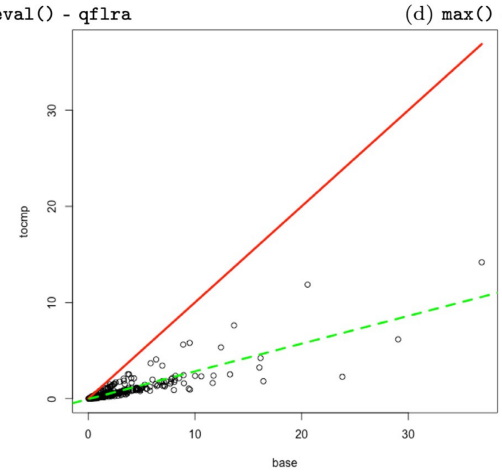
(b) `satisfiable() - qfidl`



(c) `batch_eval() - qflra`



(d) `max() - qfauflia`



(e) `min() - smt`

Figure 3.12 – Execution time (in seconds) to solve the benchmark expressions.

Table 3.2 shows that `max()` and `min()` are the most common requests and also account for most of the time spent by the solver. By selecting the right tactics, `qfauflia` [16] and `smt` [17], the default configuration could be outperformed for all the binaries as shown in the scatter plots in figure 3.12d and e.

The results in table 3.2 and in the figure 3.12 confirm that adopting heuristics as proof strategies is rarely a "one size fits all' approach' [198]. Therefore, we created a new "optimized" strategy based on the best tactics on the benchmark (`qfufbv_ackr`, `qfidl`, `qflra`, `qfauflia`, and `smt`) combined with parallel tacticals —note that while `qfidl` did not produce significant advantage on binaries for `satisfy()` as shown in figure 3.12b, we still chose to include it as there were cases where it showed a relative good performance. This "optimized strategy" was compared with the default one in the remainder of our experiments.

The analysis above addresses **RQ3.1** [18], because the usage of the z3 SMT prover can clearly be optimized by changing the tactics. And it also partially addresses **RQ3.2** [19], as a better usage of the toolchain resources can be achieved by improving the SMT tactics.

3.5.3 Overall Toolchain Optimization

Our evaluation is presented in figure 3.13, which shows the correlation between each possible pair of: heuristics for graph building and symbolic execution, ECDGs metrics and classifier's quality metrics (F-scores, accuracy and precision). For each graph metrics described in section 3.4.3 (page 132) we computed some summary statistics (total, mean, standard deviation, min, quartiles Q_1 , Q_2 , Q_3 and max). Only the correlations with significance level above 0.01 are considered here. The size of the circles and their color intensity represent how strong is the correlation (if any) between the pair of parameters indicated on the border of the matrix.

The correlation matrix shows that the F-score by malware family is negatively correlated to the presence of many unique edges and nodes, as well as to the learning time. The negative correlation of the unique edges and nodes contrasts with the positive correlation of the nodes, edges and the connected components. This could be explained by the fact that the gSpan mining overlooks the "uniqueness" of edges and nodes and obtains better performances on larger connected sub-graphs (c.f. section 3.4.3, page 132). The negative correla-

16. Closed QF linear formulas over the theory of integer arrays extended with free sort and function symbols.
17. Boolean SAT-based SMT solver.
18. **RQ3.1**: *How to tune the SMT solver to improve performance of symbolic execution?*
19. **RQ3.2**: *How to effectively apply heuristics that reduce the problem of state explosion in symbolic execution?*

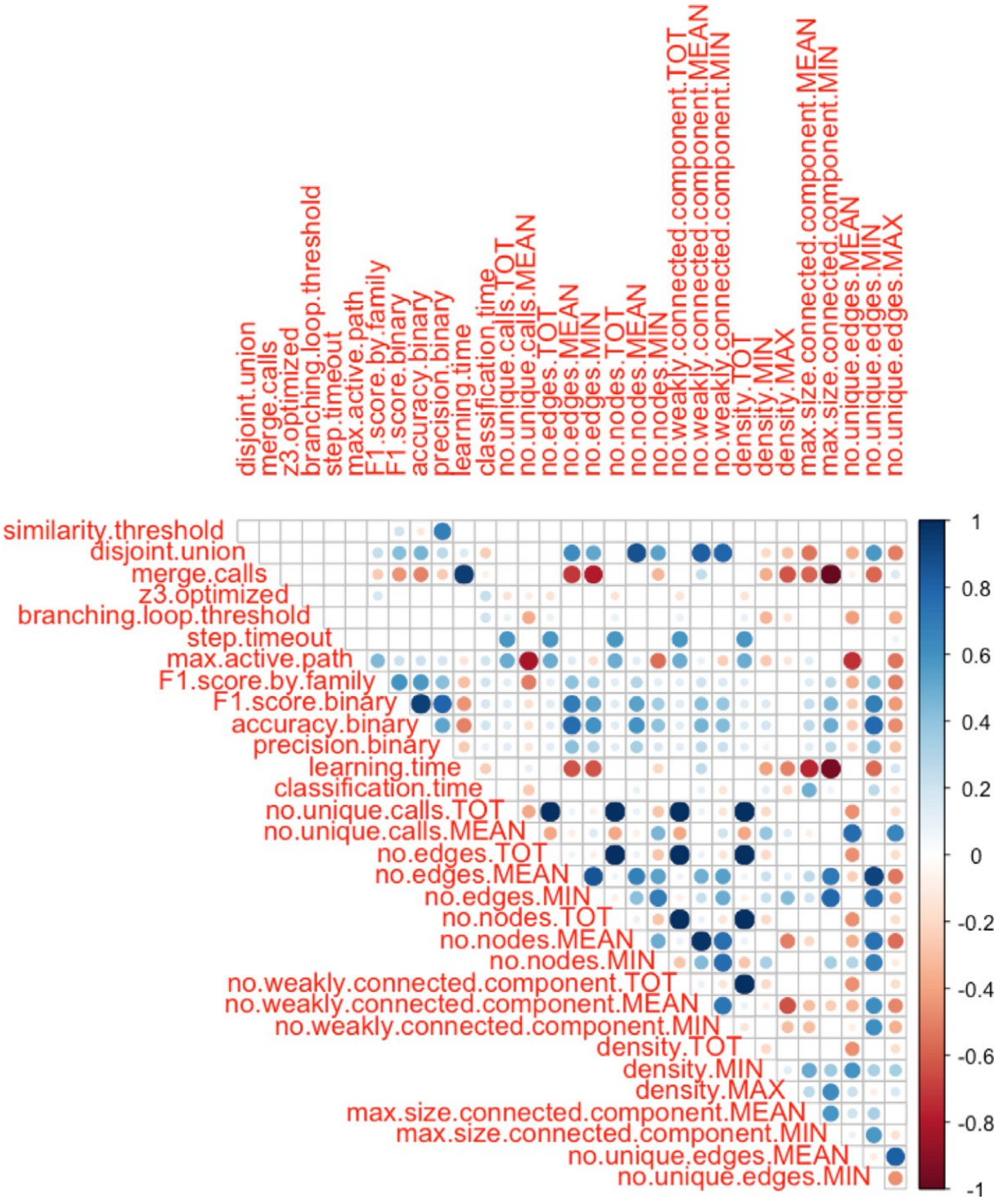


Figure 3.13 – Correlation matrix (p-*value* > 0.01) for the graph building and symbolic execution heuristics, graph metrics and performance of the classifier (some correlations have been removed from the plot for the sake of clarity).

tion for learning time is due to the fact that gSpan spends more time to mine larger graphs.

Finally, the F-score by malware family is strongly correlated with the performance of the binary classifier, the latter having even stronger correlations with the graph metrics mentioned above. The graph-building heuristic *merge calls* (described in section 3.4.3, page 132) causes a significant reduction in the number of edges and size of the connected components. Even if this simplification significantly reduces the learning time, it costs the reduction in quality of the classifier. The *disjoint union* as a trace combination heuristic results in the presence of a higher number of sub-graphs and thus requires a higher learning time. On the other hand, the presence of a higher number of connected components, as observed before, increases the F-score. The last of the considered graph-building heuristics, the *minimum trace size*, does not present any significant correlation with any of the considered performance or graph metrics (and has been therefore omitted in the correlation matrix of figure 3.13).

The most interesting correlations to address our research questions are the following: the branching loop threshold, which allows to collect more unique calls and edges, but is not reflected in the F-score (that mainly depends on the connected components as already observed) —this result addresses **RQ3.2**. Furthermore, not restricting the setting of `max-active-paths` is positively correlated with the F-score thanks to its ability to explore the CFG in BFS order. Regarding **RQ3.1** [20] our "optimized" z3 tactic has a positive correlation with the F-score (as later presented in section 3.5.4, page 146).

Therefore we can conclude that, considering the graph metrics and their impact on the F-score, the litmus test for the quality of an ECDG-based classifier is the presence and size of the connected components. This is an important and unexpected finding because, contrarily to what one might have expected, the cornerstone is not the number of (unique) calls that reflect how deep the code is explored. This could be explained considering how gSpan mining algorithm works and the adopted similarity metric based on the number of common edges between the extracted signatures and the ECDG of the sample to classify (c.f. section 3.4.3, page 132).

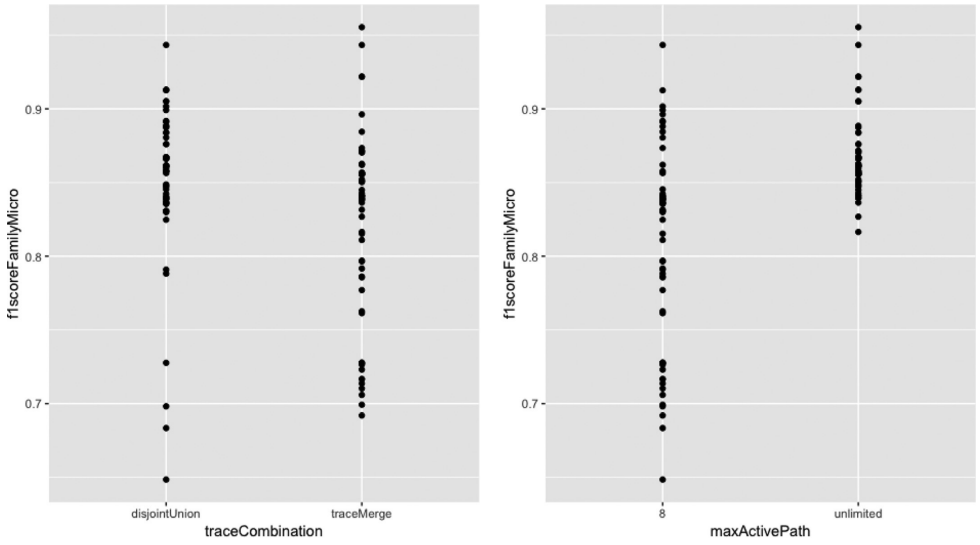
Here it is important to notice that the properties of the connected components are not under direct control for parametrization. In fact, they are the result of heuristics and techniques used during the analysis toolchain and are also influenced by the behavior of the binary itself. Even if it is not possible to identify which is the "best" size for the connected component, the appendix section 6.6 (page 310) explores the size and number of connected

20. **RQ3.1**: *How to tune the SMT solver to improve performance of symbolic execution?*

components and their relationship with the performance of the classifier.

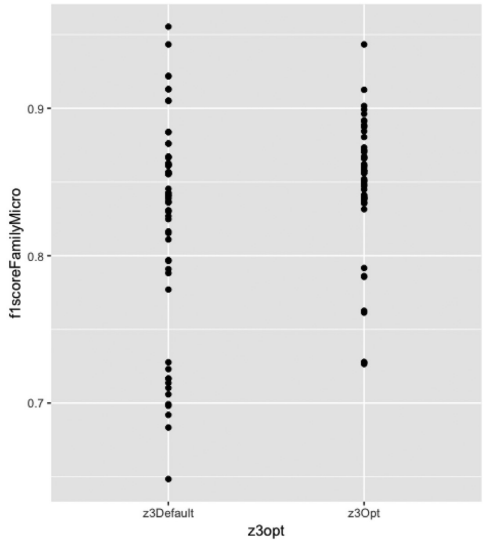
3.5.4 Impact of graph and execution heuristics

In our analysis of variance (ANOVA [268]), we first consider the main effect of each heuristic on the quality of the classifier. For the sake of conciseness, figure 3.14 shows only three of the seven factors analyzed. For the sake of completeness, the impact of all factors is reported in appendix section 6.6 (page 310).



(a) Trace combination

(b) `max active path`



(c) `z3 config`

Figure 3.14 – Main effects of the factors (x-axis) over the F-score (y-axis).

Regarding **RQ3.1**, the "optimized" tactic for z3 allows the consolidation of the classifier to an F-score above 0.825, whereas the default version has widely spread performance even reaching a very low F-score of 0.65 (see figure 3.14c).

To address **RQ3.3** [21] for the graph building heuristics, performing the disjoint union of the traces is generally preferable (see figure 3.14a), as well as to not merge the calls (not shown in the figure). Putting a limit to the minimum trace size to build a graph does not have a direct impact on the classifier. This happens because when angr is able to analyze the malware samples (see the following section 3.6), it is generally able to extract a substantial number of calls.

Limiting `max-active-paths` results in very inconsistent performance with F-score from 0.65 to 0.94. Finally, `branching-loop-threshold` and `step-timeout` show a similar behavior: when they are disabled, the performance is more stable. The overview above addresses **RQ3.3**, with significant relations between the outcomes of behavioral signatures and malware classification.

3.6 Discussion

In this section we discuss the results of the evaluation analysis taking into account the whole settings of the toolchain. During our experiments, the best classifier attained a F-score of 0.955 in the classification by malware family and 0.970 in the case of the binary classification. Based on our preliminary analysis we set a similarity threshold for gSpan of 0.7, further results supporting this are reported in appendix section 6.6 (page 310). This analysis addresses **RQ3.3** by showing that the multi-class classification based on ECDG as behavioral signatures (extracted through symbolic execution) is effective, albeit somewhat sensitive to the choices of other parameters in the experiments —note that the results here should not be considered a direct comparison with related works (c.f. section 3.3, page 123).

Our focus is the optimization of symbolic execution and heuristics evaluation. As already described in section 3.5 (page 135), the adoption of a full factorial design approach better suits our needs but is very computationally demanding. As a result, at first glance our dataset may appear small in terms of number of families and samples per family. However, we observe that symbolic execution trades more execution time for significantly more complete results, particularly on complex samples such as malware. Since we conducted

21. **RQ3.3**: *What is the impact of enhancing symbolic execution for binary analysis in a malware classification scenario?*

full factorial experiments over 128 units and 5-fold cross validation, which yields 640 classification experiments each over 200 samples. In practice this makes our experiments quite demanding; adding just 25 more samples (i.e. one family according to the family sizes in our dataset) would cost a increment of over five days of computation time. Thus, despite its seemly reduced size, the dataset is in practice larger than many works that address symbolic execution, which analyze datasets ranging from a couple of dozen to a maximum of a few hundred samples [47, 148, 192, 271, 316].

Results on a larger dataset would be ideal, however we note that using such intensive techniques on all samples is largely an academic practice. In practice, both symbolic and dynamic execution are applied only to a *subset* of the collected malware samples, while other less demanding techniques (e.g. syntactical signature matching) have not been able to provide enough confidence on the results [289].

To further complicate the direct comparison, the community lacks a common public database with a large variety of well labeled malware [98]. This precludes the establishment of a common ground truth across multiple works. Although the use of F-score is more adopted in the community (even in case of multi-class classifiers), here we preferred to use micro-average F-score (described in section 3.4.3, page 132) due to its ability to better cope with classes and provide more precise results in presence of imbalance datasets.

We adopted a linear model to study the interaction between each pair of factors. For the sake of brevity, here we only comment the significant observations of the corresponding 21 plots (reported in figures 6.5, 6.6 and 6.7 of appendix section 6.6, page 313). Trace merge combined with merge calls is too aggressive in simplifying the graphs, producing a catastrophic effect on F-score when both options are used together. This is due to the fact that both reduce the number of edges and connected components, with substantial degradation on the most important graph metrics identified in section 3.5.3 (page 143).

Although graph-building heuristic *merge calls* generally have a negative effects on F-score, the optimized version of z3 is able to mitigate this drawback. A possible explanation points to the ability of the optimized z3 of actually determining the satisfiability of SMT constraints instead of returning `unknown` due to the solver's timeout. Therefore, it can extract more edges, which connect a larger number of nodes. The effects of merging calls and branching loop threshold appear to be closely dependent, but the interaction plot of `merge-calls` and `step-timeout` shows a clear independence of these factors.

A very interesting dependency exists between the z3 solver and the limit on the number of `max-active-paths`. Limiting active paths is something usually done to partially reduce

memory consumption —so as to keep the performance degradation within tolerable boundaries —and to set the symbolic execution traversal to a configuration closer to Depth-First Search (DFS) instead of BFS. Here, we observe a significant performance decrease with the limitation of the `max-active-paths`, but our optimized version of z3 is able to almost completely remove such a negative impact on F-score —the degradation is still present but it is of about 0.01 instead of about 0.09.

Other plots provide some hints of dependencies such as: trace combination with `min-trace-size` and with `branching-loop-threshold`, `merge-calls` and `min-trace-size`, and z3 with `branching-loop` and `step-timeout`.

Finally, we consider the interactions between each possible combination of factors. The analysis shows several interaction effects asserted with a significance lower than 0.001. The Pareto chart (figure 6.4 of the appendix section 6.6, page 312) helped us to select the ones having more influence on F-score. The combination of `trace-merge`, `merge-calls`, `optimized-z3`, and unlimited `max-active-paths` has shown the most influential positive effect (above 0.3). This is followed by a negative effect of the combination of `trace-merge`, `merge-calls`, `optimized-z3`, `no-branching-loop`, and unlimited `max-active-paths`. Positive impact with a magnitude of about 0.2 on F-score are attributed to the sets of: (i) `trace-merge`, `merge-calls`, disabled `branching-loop` and unlimited `max-active-paths`, (ii) `merge-calls` and unlimited `max-active-paths`. The `merge-calls` heuristic also takes part in some of the subsequent sets having negative effects. This is exactly the opposite of what happens for the optimized z3 (which generally has a positive impact). We can thus conclude the following.

— `merge calls` is very “risky”, even if it appears in several of the best configurations. Alternatively, `disjoint union` has more constant performance than `trace merge`.

— `max-active-paths` (as expected) should be set to unlimited if there are no reasons to do otherwise.

— The `optimized z3` is effective, even though it is not the performance leader; it supports many of the other configurations that may need it to be enabled due to resource constraints (e.g., `max-active-paths`).

— `branching loop` shows very seesawing effects depending on the configuration in which it is used. This factor requires further evaluation with a higher number of levels to better understand its performance.

— `step timeout` and `min trace size` do not have a noticeable effect on F-score.

The results of the factorial experiments show that in our context tuning symbolic execu-

tion is a very complex problem and that the sparsity of effects principle, which states that the system is dominated by the effects of the main factors and low-order-factor interactions, does not hold.

3.6.1 Threats to Validity

This section discusses the validity [244] of our study according to: construct, internal, external validity and reliability.

Construct Validity: we have performed controlled experiments that allowed us to tune the environment and measure the metrics of interest. In this regard, no issue should arise from our experiential study. We remark that the presented exploratory study is more about exposing which factors are most likely to influence the outcome than on setting up a quantitative model.

Internal Validity: we performed a full factorial experiment design with the aim of getting rid of this threat to validity even if we had some *a priori* hypothesis on the most effective heuristics (e.g., the `branching-loop-threshold` and the `z3-optimization`) from previous experiments. This design choice has strictly impacted the external validity (discussed in the following). Indeed, by adopting the full factorial experiment design the number of experiments corresponds to $\#levels^{\#factor}$, where each experiment required about ~ 4 hours. Therefore adding a single factor would have meant performing 192 experiments instead of 128, hence an additional 1.5 week of computation, and adding two more families would take about 1 extra hour for each experiment. Notwithstanding, our experiment design with two levels per factor (as usually adopted in such a methodology) provides sounder results with respect to a fractional factorial design (the adoption of a response surface methodology is not feasible for timing reason). The choice of a full factorial design in place of a computationally lighter fractional one turned out to be wiser, since (as noted above) the sparsity of effects principle does not hold.

External Validity & Reliability: The choice of angr as our symbolic execution framework should not affect validity since the heuristics considered here could be implemented in other frameworks with some engineering effort (e.g., writing plugins for S2E or Triton). Datasets are a common concern in malware research. Our dataset was limited to binaries in PE format and considering 8 known malware families (and an additional group of cleanware samples) collected in 2018 (that we have not extended for experiment timing reasons as discussed above).

The labeling by antivirus is a challenge; our dataset considered samples where a very

strong consensus was shown by the antivirus engines on VirusTotal and therefore the construction of a similar dataset should be straightforward.

Another area to consider is the effect of packed binaries since the symbolic execution framework could spend time analyzing the packers (instead of the packed binary payloads) and therefore extract very similar ECDGs. In our case, a post-analysis examination with VirusTotal allowed to verify that all but one samples in the *addrop* family —which was packed with NSIS —were unpacked. Therefore it shows that, even in the case of limited symbolic execution due to packing, the rest of the analysis is still effective.

Lastly, general limitations of symbolic execution frameworks could limit the applicability of the considered malware classification approach. angr has a limited set of models for external calls and only for specific versions of libraries. Lack of information for other external calls or incorrect models due to changes in different versions of the library could prevent a complete and correct analysis. The same limits are also present in cases when particular input parameters or network resources are required to start the execution. Assuming the use of a similar set of malware families (and a family classification consistent with ours), having adopted the ANOVA test and k-fold validation obtaining evidence with a low significance level (0.001), our findings should be congruent with ones hypothetically drawn by other researchers (and thus reliable).

3.7 Conclusion

The tools used to determine software correctness, reliability, and security rely on many formal and heuristic configurations. This chapter explored the role of these in the scenario of malware program analysis, chosen as a particularly challenging form of software analysis. The results addressed three research questions, overall giving deeper insight into the usage and configuration of symbolic execution tools.

Our results here showed:

(i) how to tune an SMT solver for symbolic execution (c.f. section 3.5.2, page 139), addressing **RQ3.1** [22];

(ii) how to prioritize state exploration and parametrize heuristics to improve symbolic execution (c.f. section 3.5.3, page 143), addressing **RQ3.2** [23];

22. **RQ3.1**: *How to tune the SMT solver to improve performance of symbolic execution?*
23. **RQ3.2**: *How to effectively apply heuristics that reduce the problem of state explosion in symbolic execution?*

(iii) and that there are many positive and negative correlations between heuristics for binary analysis based on symbolic execution in a malware classification scenario (c.f. section 3.5.4, page 146), addressing **RQ3.3** [24].

In particular, the latter shows that significant improvements can be made in analysis engines and toolchains, and that components cannot be optimized in isolation, thus the optimization of the whole toolchain must be considered altogether.

24. **RQ3.3**: *What is the impact of enhancing symbolic execution for binary analysis in a malware classification scenario?*

CHAPTER 4

STRUCTURAL-BASED BINARY CODE SIMILARITY

4.1 Introduction

Historically, malware analysis has heavily resorted to manual creation of signatures for detection and classification. This human-action-based method is very costly and time consuming, thus unable to handle the exponential growth in number of unique malware instances [108]. A solution is to widely *automate* malware analysis. Towards this goal, this chapter aims to improve the computational cost of existing (and possibly future) malware *search* and malware *clustering* algorithms —although the creation of new algorithms for these problems is outside of our scope.

In this chapter we analyze *External Calls Dependency Graphs* (ECDGs) as a promising call graph representation, and a *similarity function* (σ^{ECDG}) that is reliably *accurate* and *robust*[1]. We show that they lead to an *efficient computation* of binary code similarity able to underpin the construction of frameworks for malware search and clustering.

In our experiments, σ^{ECDG} provides highly descriptive *cluster prototypes* that can help to scale up clustering, assist human-based analysis and improve classification models for malware analysis. We devote special attention to the *evaluation methodology*, an intricate issue that directly influences research validity but that is often overlooked. For this, we propose the *Accuracy and Robustness (AnR) paradigm* as guideline to create more *reliable* experiments.

As main contributions, we:

— propose the Accuracy and Robustness (AnR) paradigm for more reliable evaluation methodologies [addresses **RQ4.3**];

— propose ECDGs, as a more a compact call (dependency) graph enabling more efficient binary similarity computation [addresses **RQ4.1**];

1. Robustness relates to the ability to resist to semantic transformation.

— propose a new similarity function for ECDGs that is efficient, accurate and robust [addresses **RQ4.2**].

Our implementation also provides practical contributions, namely the practical study of symbolic execution to trace external calls, the evaluation of gSpan as a practical algorithm for sub-graph isomorphism, and the evaluation of cluster prototypes extraction to represent malware families. Our whole evaluation is based on experiments with malware samples collected in the wild from real-world dataset feeds.

4.1.1 Research Questions

We address the top-level research question **TOP-RQ2**: *How to compute the similarity of unknown programs with high accuracy while being friendly to search and clustering algorithms for malware analysis?*

For this, we break down this backbone question into the following subordinate research questions:

RQ4.1 *How to get more precise structural representations of programs wrt. state-of-the-art (with no information loss)?*

RQ4.2 *How to exploit this structural representation to define a similarity function that is friendly to binary code search/clustering schemes?*

RQ4.3 *Establishing ground truth for malware analysis is an undecidable problem, so how to evaluate this similarity function?*

4.1.2 Chapter Outline

The remainder of this chapter is organized as follows:

— Section 4.2 (page 113) presents the specific background of this chapter;

— Section 4.3 (page 159) presents the related works;

— Section 4.4 (page 163) presents our methodology, which introduces ECDGs, similarity function σ^{ECDG} and the Accuracy-and-Robustness (AnR) paradigm;

— Section 4.5 (page 168) presents the experimental setup and the evaluation results.

— Section 4.6 (page 182) discusses results.

— Section 4.7 (page 186) concludes.

4.2 Background

This section sets the notations used in this chapter (4.2.1), and recalls important notions on binary code similarity (4.2.2) and hybrid clustering (4.2.3).

4.2.1 Notations and Definitions

Definition 3 (set and multiset) *We refer as **set** a collection of elements without repetition, whereas a **multiset (or mset)** designates a collection of elements in which repetition is allowed.*

Definition 4 (cardinality) *The cardinality of a set S is notated as* $|S|$.

Let $\sim$ denote an equivalence relation on A and $x \in A$. The **equivalence class** of x is the set of all elements of A that are related to x, i.e. $[x]_\sim = \{y \in A | x \sim y\}$.

Given a set S, its **indexed family** $\mathscr{I}_S$ consists of an index set defined by a surjective function $x : \mathscr{I}_S \to S$ such that $i \to x_i = x(i), \forall i \in \{1, \cdots, |S|\}$.

A **labeled graph** is notated $G(V, E, L_V, L_E, \varphi)$, where V is a multiset of nodes (also notated as $\mathscr{N}(G)$), $E \subseteq V \times V$ is a set of edges, L_V and L_E are sets of node and edge labels respectively, and φ is a label function that defines the mapping $V \to V_L$ and $E \to L_E$.

A **directed graph** is a graph where edges E are ordered pairs of elements of V. A **directed acyclic graph** is a directed graph with no directed cycle.

Given two graphs $G_1(V_1, E_1, L_{V_1}, L_{E_1}, \varphi_1)$ and $G_2(V_2, E_2, L_{V_2}, L_{E_2}, \varphi_2)$, G_1 is **a subgraph** of G_2 if G_1 satisfies: (i) $V_1 \subseteq V_2$, and $\forall v \in V_1$, $\varphi_1(v) = \varphi_2(v)$, (ii) $E_1 \subseteq E_2$, and $\forall (u, v) \in E_1$, $\varphi_1(u, v) = \varphi_2(u, v)$. This relationship is notated $G_1 \subseteq G_2$.

G is a **connected graph** if it contains a path for every pair of vertices in it. G is **disconnected** (or **disjoint**) otherwise.

A subgraph of G is a **connected component** iff there exists a path between any pair of vertices in it. We notate the set of all **connected components** of G as $\mathscr{CC}(G)$. The **largest connected component** (in number of edges) is noted as $\mathscr{CC}_{max}(G)$.

Given the graphs G_1 and G_2, G' is a **common subgraph** of G_1 and G_2 iff $G' \subseteq G_1 \wedge G' \subseteq G_2$. We note the **largest common subgraph** (which can be disjoint) as $G_1 \cap G_2$.

More information on graphs can be found in [264].

4.2.2 Binary Code Similarity

Haq and Caballero point out three main characteristics of binary similarity approaches [119]:

- the type of comparison: identical, similar, equivalent;
- the granularity of the binary code instances (e.g., instructions, basic blocks, functions);
- the number of input instances being compared: one-to-one, one-to-many, many-to-many.

Comparison type Two (or more) instances of binary code are *identical* if they have the same syntax, i.e., the same encoding representation [119]. It is a straightforward comparison that is easy to check: the instances of binary code are either identical or they are not. Such comparison approach is very precise, however it is not robust as small —possibly frivolous —changes in the binary code (e.g. metadata as such the compilation date) result in segments of binary code that are no longer identical.

Two (or more) instances of binary code are *equivalent* if they have the same semantics, i.e., if they implement exactly the same functionality [119]. Two identical instances of binary code are unequivocally equivalent, since they impose the same *correctness jurisdiction* (c.f. section 1.2.2, page 36) over the hardware upon execution. Nonetheless, instances of binary code that are very different can be equivalent if they provide the same functionalities: *"equivalence does not care about the syntax of the binary code" [119]*. Lakhotia *et al* [157] show that proving the equivalence of arbitrary instances of binary code can be reduced to the halting problem and is therefore *undecidable*. In practice, determining binary code equivalence is a very expensive process that can only be performed for small instances of binary code [119].

Two (or more) instances of binary code can be considered *similar* if their syntax, semantics, or structure are similar [119]. Structural similarity compares graph representations of the binary code, sitting in between syntactic and semantics. The intuition is that structural representations of the binary code are derived from a syntactic parsing of binary code, but they are able to in some extent express its semantics by capturing the inner data flow dependencies of the instances of binary code. Structural similarity is robust against multiple syntactical transformation, but sensitive to transformations that change code structure such as code inlining or removal of unused function parameters [119];

Comparison granularity Instances of binary code can be compared at different granularities, the most common levels being: instruction, basic block, functions, and whole programs [119]. Possibly, approaches that target finer granularity can be combined to attain coarser granularity approaches; due to that, there is a distinction between *input granularity*, which refers to the instances of binary code that are provided as input in the comparison

approaches, and the *approach granularity*, which is the approach target similarity.

Number of inputs Approaches for measuring binary code similarity can involve different multiplicities of binary code instances. They can compare instances of binary code in one-to-one (OO), one-to-many (OM) or many-to-many (MM) relationship.

One-to-one approaches compare an original instance of binary code to a target instance of binary code; most OO approaches perform *binary code diffing* [119]. One-to-many approaches compare a *query* instance of binary code to many target instances of binary code; most OM approaches perform binary code search [119]. In contrast to OO and OM approaches, MM approaches do not distinguish between source and target instances; all input instances are considered equal and compared against each other. These approaches typically perform *binary code clustering* [119].

4.2.3 Hybrid Clustering

Traditional clustering methods follow specific approaches such as Hierachical, Partition and Density methods (c.f section 2.2.3, page 75). In contrast, hybrid methods combine different approaches with aim of optimizing their strongest points.

HDBSCAN HDBSCAN [39] has similar advantages as DBSCAN (c.f. section 2.2.3, page 78) being able to produce clusters of arbitrary shapes, different sizes and densities, while handling outliers as noise to avoid polluting the clustering model with irrelevant samples. However, HDBSCAN greatly improves on DBSCAN by adding a *hierarchical* clustering strategy; instead of computing clusters based on parametric global density (as DBSCAN does) [183], HDBSCAN obtains highly-varied cluster with different densities. Besides, HSBCAN improves on OPTICS algorithm [10], which also includes a hierarchical strategy, by tackling the limitation of finding just a single global cut/density threshold, which restrains the clustering to a *flat* partition [39].

Table 4.1 shows different strategies taken by clustering methods and how HDBSCAN differs from previous methods. Distance-based methods consider the distance between data points and clusters, whereas density-based methods look to regions surrounding data points and the number of neighbors contained therein. Flat methods provide a set of clusters without any explicit structure relating them together, whereas hierarchical methods provide a (hierarchical) inner structuring of the clusters.

Table 4.1 – Dimensions of clustering methods & methods instances

	Flat	**Hierarchical**
Distance/Parametric	K-Means	Agglomerative
Density/Non-Parametric	DBSCAN	*HDBSCAN*

Similarly to OPTICS (c.f. section 2.2.3, page 78), HDBSCAN works with the reachability distance as metric. However, instead of building a reachability plot for the choice of a cut-off value for ϵ', HDBSCAN interprets the data points as a graph where the points are vertices and the edges are the mutual reachability distances between them. The graph is extended with self-loops for all vertices having the core distance as weight and then the minimum spanning tree [107] for this graph is computed (forming the MST_{ext} as shown in figure 4.1).

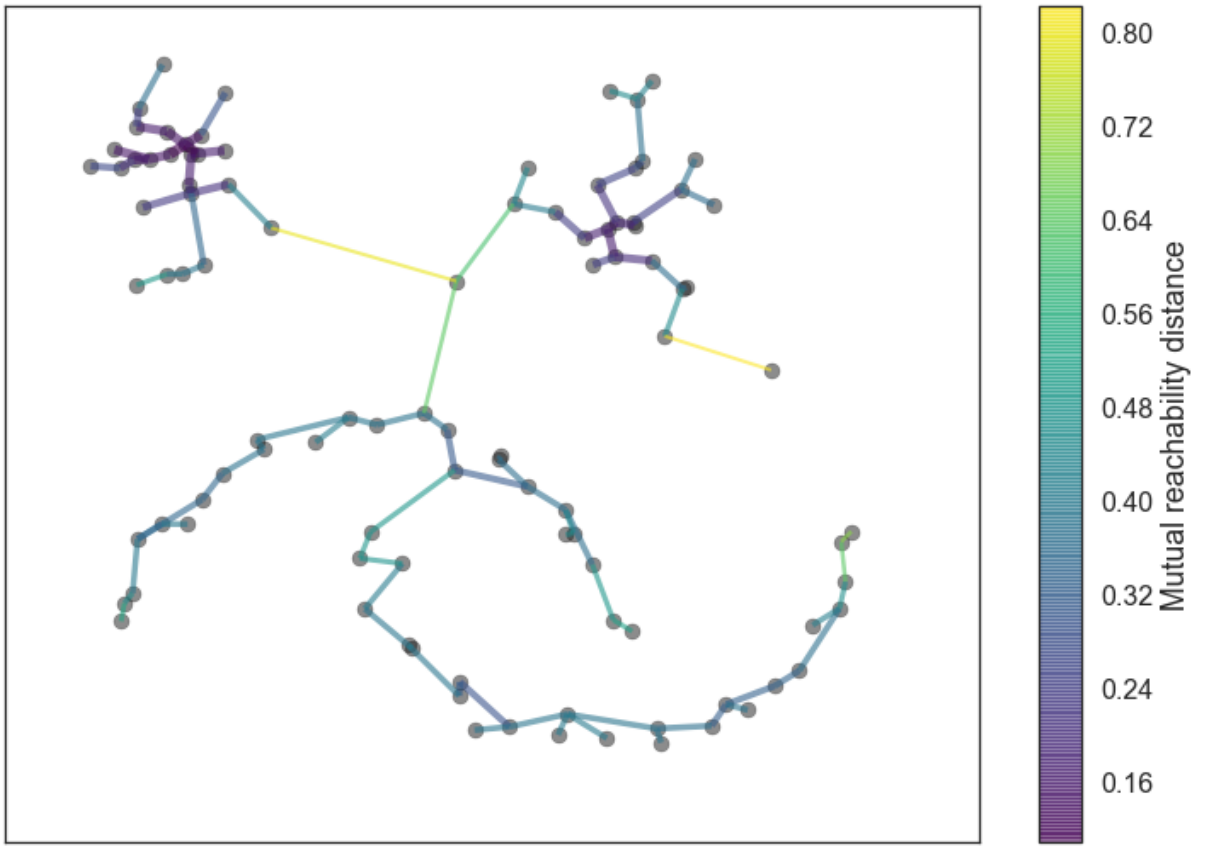


Figure 4.1 – Example of MST_{ext} (the color scale of the edges reflects the mutual reachability distance) [163]

The MST_{ext} is converted into a hierarchy of connected components, which results in

a dendrogram as depicted in figure 4.2a. The distance axis indicates the distances ($dist$) between the set of elements included in each branching of a clade[2], while λ is defined as $\lambda := \frac{1}{dist}$. This dendrogram is reinterpreted with respect to the chosen *minimum cluster size*. As long as the "splitting" (as shown in the dendrogram of figure 4.2a) of a cluster does not generate a new cluster (of the minimal size), it is simply considered that the original cluster is losing points (thus technically not splitting). This results in a condensed dendrogram of the hierarchy of connected components of MST_{ext}, as shown in figure 4.2b.

Finally, the clusters that persist for a longer λ interval are chosen, with the additional requirement that when a cluster is selected, none of its descendants can be chosen any more. The minimal value of λ for these clusters are taken, resulting in a final clustering that contains clusters of different densities. Figure 4.2b shows the selected clusters for the previous example of condensed dendrogram.

The complexity of HDBSCAN is dominated by the construction of MST_{ext}, which was initially implemented using Prim's algorithm (whose complexity is roughly $\mathcal{O}(n^2)$) [221]. The accelerated version of HDBSCAN [182], which replaces the exact computation of the minimal spanning tree for an approximation version [177], reduces the complexity to roughly $\mathcal{O}(nlog(n))$. Furthermore, further significant improvements in the accelerated HDBSCAN version made possible to obtain a Python implementation of HDBSCAN —compatible with scikit-learn [214] —that is very efficient in practice [184].

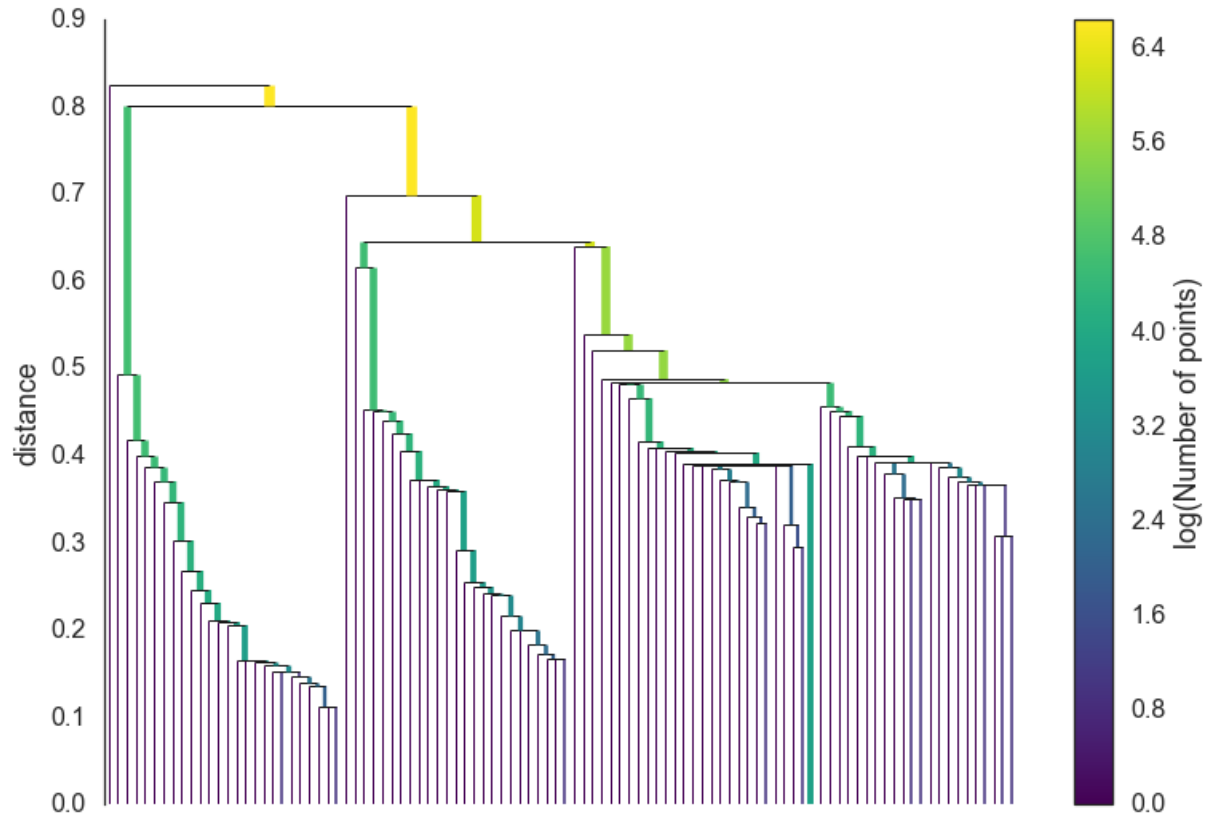
4.3 Related Works

Our focus in this chapter is on the representation of programs through *call graphs* (ECDGs), benefiting from their *structural similarities* (c.f. section 4.2.2, page 155). For this reason, this chapter entails a closer relationship to researches that study the similarity of binary codes based on structural similarities, especially in the case of call graphs.

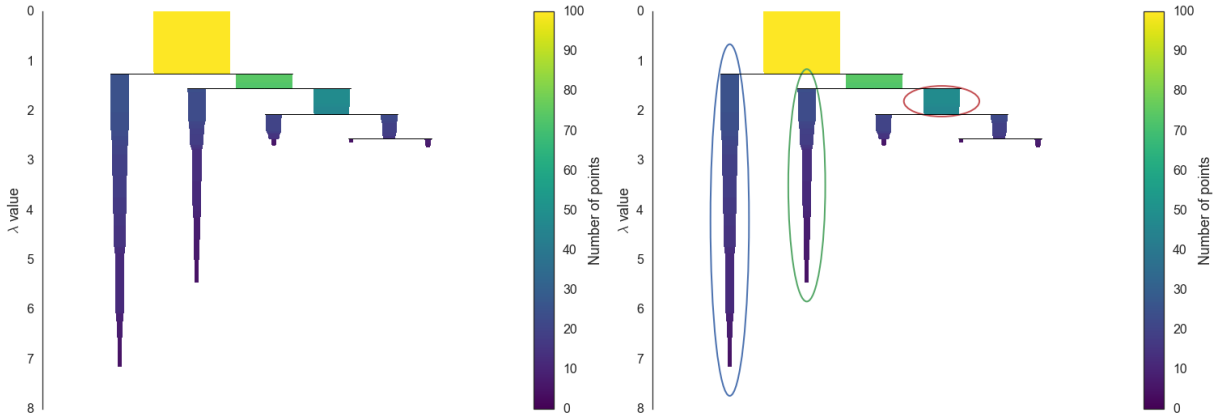
4.3.1 Call Graphs for Binary Code Similarity

The concept of call graphs dates back as early as 1979 [246], and as of 2004 [90] a plethora of studies targeting structural similarities for malware analysis have been made. Initially the focus was on the quality of disassemblers, graph formats (i.e. creating labels or grouping nodes) and graph matching algorithms, because feature extraction was done statically [152,

2. Stacked branches that break down into further smaller branches up to the leaves, where are the data elements.



(a) Conventional dendrogram



(b) Condensed dendrogram

(c) Cluster selection

Figure 4.2 – Example of dendrogram showing the hierarchy of connected components of MST_{ext} [163].

161, 258, 312]. Several *similarity functions* have been proposed and served as basis for code diffing tools, like *BinDiff* [330] and *radiff2* (used in this chapter as benchmark). The main drawback of these approaches is that they operate on hefty graph representations, that though detailed are very expensive to handle in practice, especially considering that graph matching problems fall into the NP class. Furthermore, pure static analysis is more susceptible to syntactical mutations, which can impact the result of the whole similarity analysis.

As of 2007 [16], researches related to *dependency graphs* emerge as dynamic analysis gains traction on malware analysis domain. There too, different graph formats and graph similarity functions, based on various graph matching algorithms, have been proposed [47, 80, 113, 194, 211]. In this case, the main drawback is that the tracing of calls (and OS resources) is done through filters placed as kernel modules/drivers, which is unable to distinguish traces of interest, i.e. those generated by the program main object, from spurious traces generated by shared modules (i.e. library code). This mixture in the traces muddles the subsequent analysis and, once again, produces bigger graph representations that are expensive to process.

Many different representations based on graph notations have been proposed to analyze and characterize programs (e.g. control flow graphs). They model programs as graphs and aim to examine *structural similarities* of codes, which provide analysis techniques that are likely more robust. In malware behavioral research, the most frequently adopted graph representation is in the form of *call graphs* (and derivations), which models the relationships between caller and callee procedures.

The general framework of call graph analysis comprises two main parts[80]:

— **Graph construction algorithm**, which defines the graph format and describes how the calls and any further objects of interest (e.g. OS resources) are obtained through binary analysis in order to construct the graphs.

— **Graph matching algorithm**, which describes how the graphs are compared with the others in order to produce the analysis results.

The *call graph construction* requires static or dynamic binary analysis to obtain the calls. Traditionally, this task has been fulfilled through static analysis [161, 258, 312], where a disassembly tool is used to parse binary code and identify caller and callee procedures. The main drawback is that functions imported at runtime —which is common for obfuscation methods like packing —are overlooked.

In the case of dynamic analysis, calls are intercepted and logged at execution time when the monitored/hooked functions are called. This extraction method cannot ensure total

code coverage, nonetheless it has presented positive results for detecting obfuscated malware [47, 80].

The definition of the graph format has a major impact on the quality of results obtained by the graph matching algorithm. Christodorescu defined *malspec* as a specification of malicious behavior that consists of a labeled directed graph where the nodes are (Windows) system calls and the edges represent argument dependencies between them [47]—to given an example: `RegCreateKeyA` $\xrightarrow{0\to1}$ `RegCloseKeyA` indicates the first argument of `RegCloseKeyA` is the return of `RegCreateKeyA`. Wang *et al.* [298] refer to *Malspec* as *system call dependency graph* (SCDG).

Shang et al. [258] define the *function-call graph*, which is reused later by Wu *et al.* [312]. It consists of a direct graph where the nodes are the 5-tuples defined by (function name, function type, pointer to the function first callee, pointer to the function first caller, opcode sequence) and the edges are drawn from caller to callee functions. The function type is determined according to three categories: local-subroutine, DLL function and statically-linked function.

Lee et al. [161] define the *code graph*, which consists of a direct graph whose nodes are function categories and edges are drawn from caller to callee function. The function categories encompass 128 groups that are built from the relationship of 32 kernel objects (e.g. process, memory, socket, etc.) and 4 actions of function behaviors (i.e. open, read, write and close).

Park et al. [211] define two different graphs: i) the kernel object behavioral graphs (KOBG) capturing the interactions between kernel objects that are created and used during execution; ii) the weighted common behavioral graphs (WCBG) constructed by computing the *weighted minimum common supergraph* with the KOBGs obtained from samples of the same family. Using both definitions, the paper proposes a new graph format using kernel objects as vertices and their relationships as edges. The relationships capture resources that are created/used by different system calls during execution. A final graph (the WCBG) is proposed which allows the detection by family and not only on an instance basis.

Elhadi et al. [80] combine and extend the representations of call graphs and *KOBG* to define the *API call graph*. In their proposal, vertexes can represent either system calls or kernel objects captured during the binary analysis, whereas edges are drawn according to different types of dependency relationships between nodes (i.e. sequence, data, declaration or API call).

Table 4.2 shows the different graph formats, binary analysis methods and graph matching

Table 4.2 – Malware detection based on call graphs

Graph format	Analysis	Graph matching
SCDG [47]	Dynamic	Minimal Constrast Subgraph
function-call [312]	Static	$Sim(G_1,G_2) = \frac{2E(G_1 \cap G_2)}{E(G_1)+E(G_2)}$
code graph [161]	Dynamic	$Sim(G_1,G_2) = \frac{E(G_1 \cap G_2)}{E(G_1 \cup G_2)}$
KOBG/WCBG [211]	Dynamic	Minimum Common Supergraph
API call graph [80]	Dynamic	Graph Edit Distance

algorithms chosen in several related works. The results obtained are promising though their validation is impaired by the relatively small malware datasets used [65]. Another important limitation is the need for a ground truth, which implies a very costly manual process to be accomplished in advance.

4.4 Methodology

Here we introduce *External Calls Dependency Graphs* (ECDGs) (section 4.4.1, page 163) and their associated similarity function (section 4.4.2, page 164), before introducing the *Accuracy & Robustness Paradigm* (AnR Paradigm) to establish a methodology of framework evaluation (section 4.4.3, page 165). ECDGs are compact, semantic-descriptive and robust *structural representations* of binary codes. The ECDG-associated similarity function can be used in practical search and clustering frameworks. The AnR Paradigm sets the basis for our experimental procedure.

4.4.1 External Calls Dependency Graph (ECDG)

A call graph is a direct graph whose nodes represent functions and edges represent one or more invocations of these functions [246]. An ECDG is a call graph whose nodes are restricted to *external calls*, meaning calls to external functions, i.e. system or library calls [144, 151] [3]. ECDGs are modeled in the form of a dependency graph whose edges represent shared arguments between external calls.

The ECDG definition resembles those of *malspec* [47] and *System-call Dependency Graphs (ScD-Graphs)* [205]. ECDGs may be considered as an instance of *malspec* in which the program main object code (see figure 3.10) defines the boundaries of the *trusted computing base*, instead of the whole image of program as in the original work. For *ScD-Graphs*, the

3. Unlike the references, we foresee the loading of new (external) functions at runtime.

main differences are our extended scope (library calls not being overseen), and the use of labeled edges providing higher precision to our similarity function.

Formally an **External Call Dependency Graph (ECDG)** is a directed acyclic graph $G(V, E, L_V, L_E, \varphi)$ whose node labels L_V are symbolic names of external functions, edges labels L_E are *def-use* dependencies [47] between these functions and φ is the labeling function.

The formal definition of a ECDG $\mathscr{G} = (V, E, \gamma, \rho)$ follows as in [47]:

— V is vertex-set and E is edge-set, $E \subseteq V \times V$

— γ associates vertices with symbolic functions, $\gamma : V \rightarrow \Sigma \times 2^{Vars}$

— ρ associates constraints with nodes and edges, $\rho : V \cup E \rightarrow \mathscr{L}$

In our study, a major difference in the graph specification introduced by the use of symbolic execution is the fact each analysis produced a set of traces instead of a single execution path. Therefore, the graph construction can be parameterized to merge common subtraces and/or common *external functions* (i.e. node tags). This parameter choice changes the description of the binary semantic, therefore impacting the evaluation of their structural similarities.

ECDGs are a good alternative for call graphs that incorporate all *def-use* dependencies between calls of a binary program —thus having the potential to be a precise structural representation —while getting rid of nonessential external calls, thereby making this call graph representation more compact (and therefore more suitable for graph matching algorithms). The analysis of ECDGs addresses **RQ4.1**: *How to get more precise structural representations of programs wrt. state-of-the-art (with no information loss)?*

4.4.2 Similarity Function (σ^{ECDG})

Our similarity function definition targets the *largest common connected components* of the graphs as well as the *common nodes. The common connected components capture common sub-behaviors, while the common nodes spot some degree of implementation resemblance.* Thus, our similarity function combines a *localized* behavioral view of the graphs (encompassed in the edges) with a *holistic* behavioral view (encompassed in the nodes) —malware analysis schemes which are based on features computed from plain call tracing are homologous to node-only analysis of ECDGs [3, 45, 141].

The **similarity of two call graphs** G_1 and G_2 is defined as:

$$\sigma_\alpha(G_1, G_2) = \alpha\ \sigma_{nodes}(G_1, G_2) + (1 - \alpha)\ \sigma_{edges}(G_1, G_2),$$

where $\alpha \in [0,1]$ is the **node-edge factor (*nef*)**.

The **node similarity** σ_{nodes} is defined as:

$$\sigma_{nodes}(G_1, G_2) = \frac{|\mathcal{N}(G_1) \cap \mathcal{N}(G_2)|}{min(|\mathcal{N}(G_1)|, |\mathcal{N}(G_2)|)}$$

The **edge similarity** σ_{edges} is defined as:

$$\sigma_{edges}(G_1, G_2) = \frac{|\mathcal{CC}_{max}(G_1 \cap G_2)|}{min(|\mathcal{CC}_{max}(G_1)|, |\mathcal{CC}_{max}(G_2)|)}$$

Thus, σ_α combines localized and holistic views of the graph, allowing efficient algorithms to independently compute σ_{edges} and σ_{nodes}. This would be impaired by greedier definitions that take many common disjoint components into account. We use σ^{ECDG} to denote the computation of our similarity function on ECDGs.

Since symbolic analysis can generate traces from different execution paths, it is important to balance the two-folded view (localized and holistic) in the graph similarity. For instance, if the largest execution paths of a pair of binaries are the same, independently of any major differences in shorter execution paths, they attain maximal *edge*-based similarity. However overly relying on common nodes for the similarity definition can produce higher similarity values for graphs with very different edges, thus very different call interdependencies. Either way, graphs that are in fact similar attain high similarities for both components (i.e. edge *and* node).

Our similarity function (σ^{ECDG}) is agnostic and therefore it can be used by different search and clustering schemes that are based on pairwise similarity/distances of data points. By proposing a similarity function that can prove to be efficient to compute while being accurate and robust, we address **RQ4.2**: *How to exploit this structural representation to define a similarity function that is friendly to binary code search/clustering schemes?*

4.4.3 Accuracy-and-Robustness (AnR) Paradigm

Methodologies for evaluating malware analysis framework are key to malware analysis research, however a systematic discussion about this topic is often neglected. Evaluations are based on measurements gauged in experiments that use some specific evaluation dataset; however, different datasets may bring different difficulty levels to different experiment setups. This issue becomes prominent when the evaluation methodology requires the definition of (malware) families to provide *labels* for measuring performance, due to the lack

of outright definition of these categories.

Formal studies on "computer viruses" (malware) show that defining a perfect detector is an undecidable problem [256], conflicting with the ambition of the detection primitive (c.f. section 2.3.2, page 105). The *Template Matching Problem*, which decides whether a given piece of code matches a template behavior in order to build semantic-aware detectors [48], is also *undecidable* [256]. Both results pinpoint that defining perfect (malware) families is theoretically *impossible*.

Li et al. [165] address this issue in the context of malware clustering, studying whether performance results are biased towards high accuracy depending on the methodology followed to select the ground truth. They first compare clustering results of a prior work [23] against clustering frameworks from another domain (plagiarism detection), replicating the same experiment on the same dataset. Then they redo the experiments on all different settings using a new dataset. As in the prior work [23], the ground truth for the F-measure evaluation is set by selecting only samples for which different antivirus tools agree on labeling (antivirus consensus). All frameworks attained good scores on the first dataset (F-measure from 0.943 to 0.960), but performed poorly on the new one (F-measure from 0.609 to 0.630). The authors hypothesize that this may come from the differences in the difficulty level of the datasets (i.e. easy-to-cluster vs difficult-to-cluster), pointing that the **elaboration of a sound evaluation methodology as an open problem**.

Towards this goal, **we propose the Accuracy-and-Robustness (AnR) paradigm as guideline in the evaluation of malware analysis frameworks**. AnR consists in conducting evaluation as a two-phased experiment, in which one phase assesses the *Accuracy* attained by the framework, and the other its *Robustness*.

The **accuracy phase** assesses the framework on a dataset that is likely to be *easy* to evaluate. The main goal is to verify whether the framework is able to generate results that are similar to the ground truth, the latter being assumed accurate by design. This means that the dataset composition should follow more *stringent methodologies*, which are likely to provide less intrinsic disparity between samples, e.g.:

Outsource consensus: given a sample, this strategy establishes the ground truth (malware family) by running a malware analysis on multiple anti-malware engines and selecting the label that comes out as consensus (if any). This strategy typically benefits from platforms that aggregates multiple sources of analysis, such as VirusTotal [294]. The weaknesses of this approach are the inconsistencies among analysis results and lack of

universal standards to generate labels [248]. As pointed by *Li et al.* [165], by enforcing consensus among results, the diversity representation within the dataset is drastically reduced.

Very specific string matching (VSSM) works on strings [4] that are strong indicators of a given sample and therefore can be used in very specific syntactical signatures, which incurs in extremely low false positive rates [240]. This idea is applied in YarGen [241].

Multiple stringent methodologies may be used to establish different ground truths on a same dataset, enabling cross-validation for further validation of the results. Furthermore, a balanced dataset should be privileged to improve the *significance* of the results [165].

The **robustness phase** assesses the framework on a very diversified dataset, which is able to represent a *real world scenario*. For this, the dataset should be built with *looser methodologies*, e.g.:

Manual labeling of the whole dataset, ideally following a unique guideline to assign labels to samples.

Direct outsourcing that takes as ground truth the output of one single anti-malware. This approach avoid the sample discard issue when anti-malware engines do not agree on labels in the consensus case.

Public, acknowledged signatures (PAS) employed to detect specific malware families in the wild (ITW). These rules are often tuned to avoid false positives, while remaining fairly general to maximize variant detection.

In the robustness phase it is essential to purposefully and gradually include *noise* [5] in the dataset, to measure the impact on the evaluation metrics. This acts as a control group in the evaluation. Therefore, unlike in the accuracy phase, our primarily concern is to observe the inner differences of metric values as noise is inserted, not to seek straightforward correspondence with the ground truth.

Despite the practical consequences that impact the evaluation of malware analysis frameworks, the problem addressed here has an epistemological nature. Therefore, its solutions can be to some extent elusive to put in practice. By phasing the experiments and measuring the metrics evolution as the dataset profile changes (according to some tangible heuristics), we address **RQ4.3**: *Establishing ground truth for malware analysis is an undecidable problem, so how to evaluate this similarity function?*

4. Not only "text strings", but generic sequence of bytes, much like in Yara rules.
5. Samples that do not correspond to any malware family.

Our Evaluation Methodology

We measure and analyze different partitions of our dataset, under the premise that each partition is a "corrupted" version of a true partition but that their combination converges to the truth.

Since we obtain these partitions through unsupervised learning (i.e. clustering) we hinge on external labels only for performance measurements —also under the premise that these labels represent a corrupted version of the truth.

Furthermore, to optimize the computation of clusters, we choose algorithms that can work with precomputed pairwise similarities/distances (computed as $1-\sigma^{ECDG}$). This incurs a huge initial cost (i.e. $\mathcal{O}(n^2)$) to compute all pairwise distances in order to build the similarity/distance matrix. However, since it can be reused by different algorithms and parameterizations, it fits our goal of producing different partitions of our dataset.

4.5 Experiments

This section evaluates the *accuracy* and *robustness* of ECDGs according to σ^{ECDG} with a data-driven approach that follows our AnR Paradigm (section 4.4.3). With exception of the dataset, we reuse the experimental setup of the previous chapter (see section 3.5.1). Therefore, in the remainder of this section we present the evaluation dataset (section 4.5.1), the performance benchmark of σ^{ECDG} (section 4.5.2), the benchmark comparing σ^{ECDG} and *radiff2* (section 4.5.3), the initial parametrization (section 4.5.4), the AnR experiments (section 4.5.5) and a prototype analysis (section 4.5.6).

4.5.1 Dataset

We rely on *syntactical signatures* (Yara rules) for dataset composition, because they are typically very precise, with very low false positive rates, and stable in terms of reproducibility across datasets.

To build our evaluation dataset while avoiding defective call traces (see section 3.5.1), we only selected files whose ECDGs have at least 100 edges. Thus, we selected 1,5K samples from a total of 7,306 traces organized in three groups:

- **Group I** contains 600 files matching with manually crafted Yara rules following the VSSM rationale, equally balanced into four families with no overlap between families.

- **Group II** contains 1,001 files from 16 different families defined by public, real-world Yara rules of public repositories (i.e. Yara-Rules, InQuest and McAfee ATR Team), following the PAS rationale, where families are unbalanced and may overlap.
- **Group III** contains 499 randomly chosen benign, cleanware files, with no prior knowledge about them.

Table 4.3 shows the number of samples per family and their corresponding class.

Table 4.3 – Evaluation dataset

family	**class**	**source** (*"rare string"* or *[repository] yar file*)	**#samples**
Mira	I	"Mira.h"	150
Shohdi	I	"USR_Shohdi_Photo_USR"	150
Bogy	I	"BOGY'S GAME ENGINE"	150
TwarBot	I	"TwarBot"	150
spyeye	II	[Yara-rules] MALWMiscelanea.yar	162
Wabot	II	[Yara-rules] MALW_Wabot.yar	162
IceID_Bank_trojan	II	[Yara-rules] MALW_IcedID.yar	149
shylock	II	[Yara-rules] MALW_Miscelanea.yar	109
Bublik	II	[Yara-rules] MALW_Bublik.yar	88
sakula_v1_3	II	[Yara-rules] RAT_Sakula.yar	80
Njrat	II	[Yara-rules] RAT_Njrat.yar	58
njrat1	II	[Yara-rules] RAT_Njrat.yar	58
win_exe_njRAT	II	[Yara-rules] RAT_Njrat.yar	58
Cerberus	II	[Yara-rules] RAT_Cerberus.yar	50
Glasses	II	[Yara-rules] MALW_Glasses.yar	45
Mirage_APT	II	[Yara-rules] APT_Mirage.yar	41
ClamAV_Emotet_String_Aggregate	II	[InQuest] ClamAV_Emotet_String_Aggregrate.rule	36
Monero_Mining_Detection	II	[McAfee ATR Team] MINER_Monero.yar	27
Warp	II	[Yara-rules] MALW_Warp.yar	9
Cleanware	III	-	499

4.5.2 Benchmark: Framework

Our focus in this chapter is the study of ECDGs and σ^{ECDG} ; nonetheless, generating the ECDGs for the new dataset requires to repeat part of the experimental procedure of section 3.5.3 (page 143). The difference is that previously ECDGs were computed for a smaller dataset with many different parametrizations, whereas now we use a single set of parameters —chosen from the lessons learnt in chapter 3 (page 111) —with a a bigger and more diversified dataset.

This section provides another benchmark for the symbolic execution of our toolchain. Here we use the new dataset (c.f. section 4.5.1, page 168) in complement to the one studied

in chapter 3.

This benchmark includes the total number of traces, i.e. 7,306 prior selection. The parameters for the symbolic execution with *angr-extractor* (see section 3.5.1) are the following:

- *Max memory*: 8GB
- *timeout*: 3600s
- *step timeout*: 8s
- *symbolic loop threshold*: 4
- *max active paths*: 8

The default solver for the z3 backend used by Claripy was changed to:

```
s = z3.ParOr(
    z3.Tactic(tactic='qflra', ctx=self._context),
    z3.Tactic(tactic='qfidl', ctx=self._context),
    z3.AndThen(
        z3.Tactic(tactic= 'solve-eqs',ctx=self._context),
        z3.Tactic(tactic='fail', ctx=self._context)), #from satisfiable
    z3.Tactic(tactic='qfufbv_ackr', ctx=self._context), #from AND-ing batch_eval
    z3.Tactic(tactic='qfauflia', ctx=self._context), # from max()
    z3.Tactic(tactic= 'smt',ctx=self._context), # from min()
    z3.Tactic(tactic= 'qfbv',ctx=self._context),
    z3.Tactic(tactic= 'qfaufbv',ctx=self._context)
).solver()
```

The profile of the extraction times for traces is shown in figure 4.3. The majority of the analysis take under 10 minutes [600s] to complete, averaging an extraction time of 583s. The stop causes are largely dominated (~ 93%) by the memory consumption that exceeds the predefined limit; this is a known issue with symbolic execution (as discussed in section 3.2.1, page 113). The remainder of the analysis terminates because there are no more traces to be explored (~ 7%) and in very few cases the 1h timeout was reached.

Figure 4.4 shows the distribution of maximum memory taken in the course of the symbolic execution per time of analysis. The colors show the stop cause for each execution and the radius size of each point represent the quantity of calls obtained in the given analysis run.

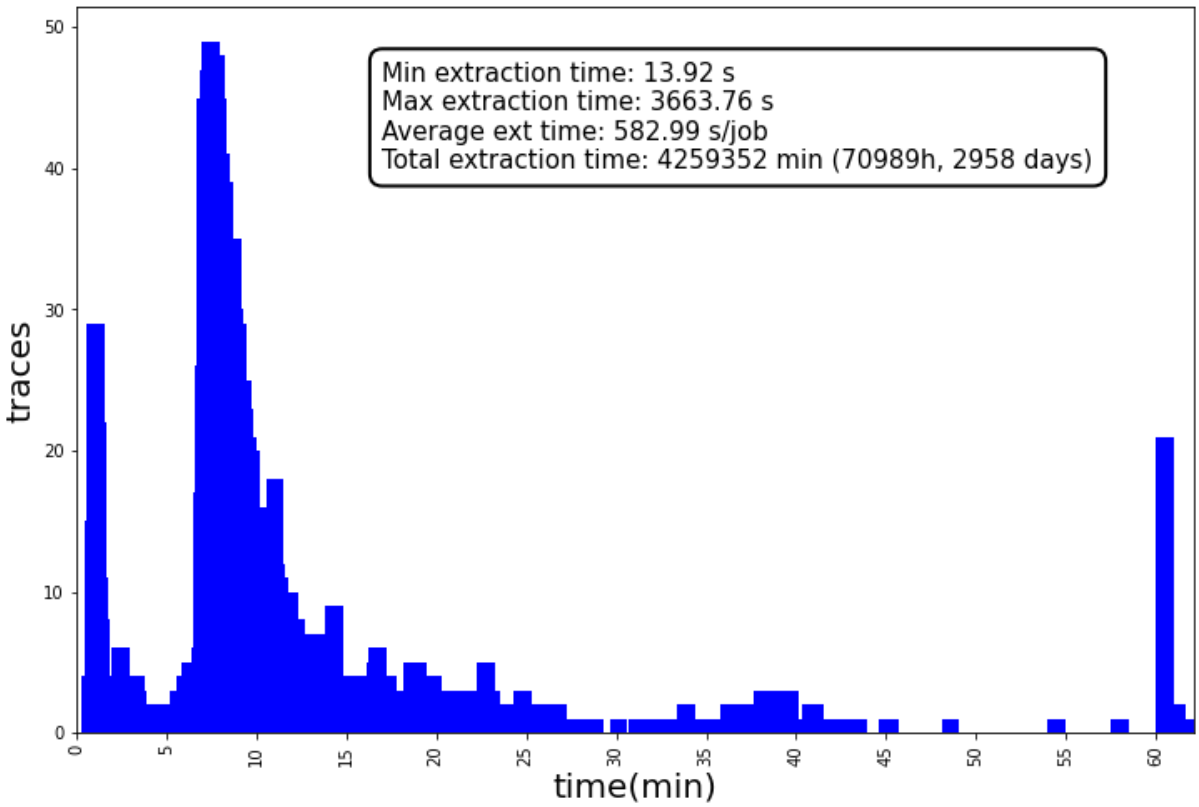


Figure 4.3 – Profile of the extraction time for all the traces

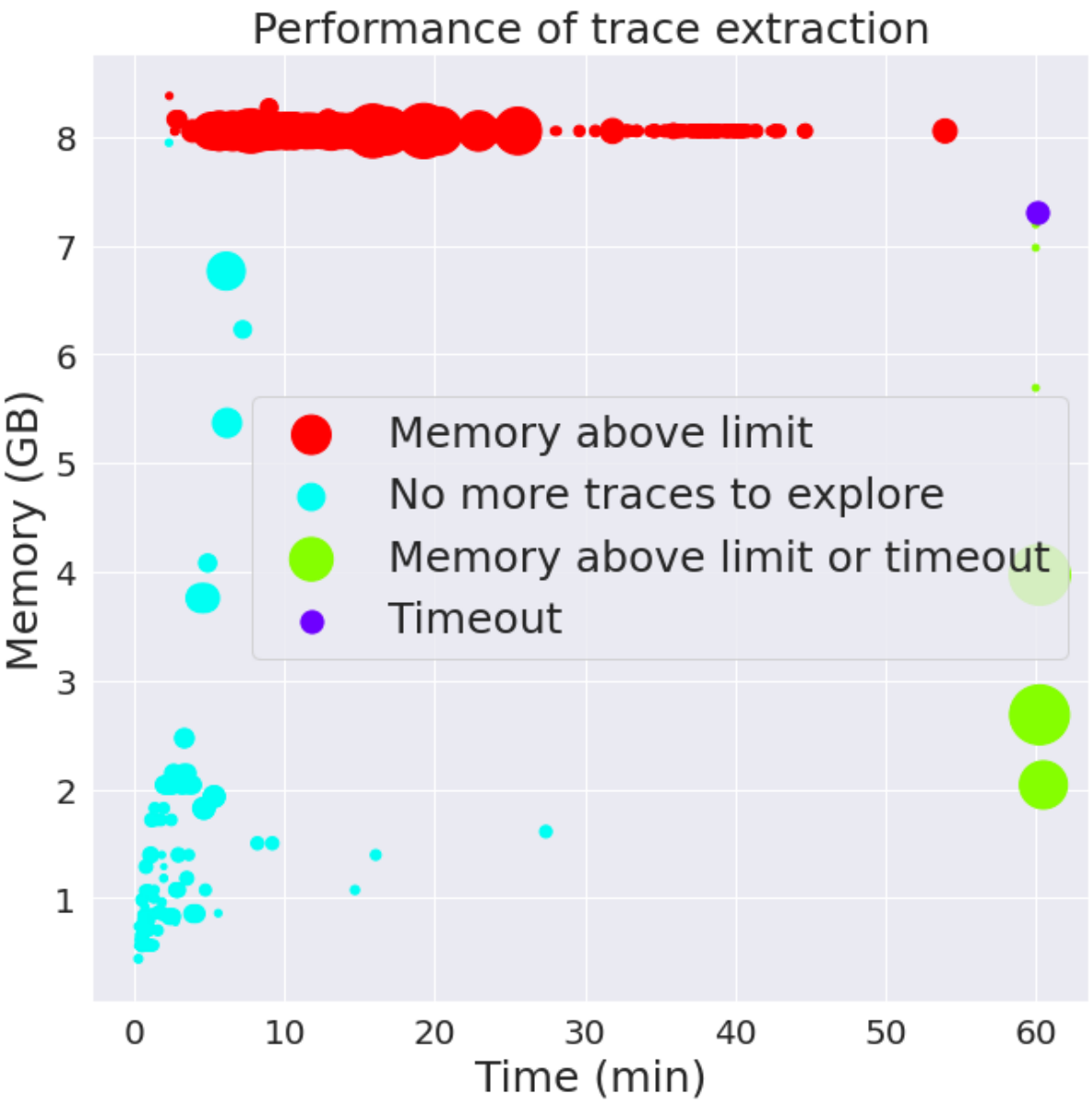


Figure 4.4 – Distribution of maximum memory vs time per analysis. The colors show the stop cause and the radius of each entry is proportional to the number of calls extracted.

4.5.3 Benchmark: σ^{ECDG} vs. *radiff2*

To evaluate the time efficiency of σ^{ECDG} in a practical scenario, we carried out a the benchmark evaluation comparing our approach against *radiff2*, an open-source tool dedicated to binary comparison in radare2 suite [228]. For this, we attempted to compute all pairwise similarities with the files of group I. However since radiff2 can take an overwhelmingly long time to compute all pairwise similarities, we stopped the process after three weeks of computation, obtaining $\sim 175k$ pairwise similarities (for each of radiff2 and σ^{ECDG}) that we named *Benchmark-DS*.

Our framework keeps a cache of the already computed ECDGs, which greatly speeds up the whole computation process. Table 4.4 notates this optimization by *cached*-(CA + GC) + σ^{ECDG}. To measure the storage cost of this trade-off, we plotted the graph for *file size* vs. *#edges* taking into account over 10K files shown in figure 4.5. The scattered points have an almost linear profile and their linear regression results in a slope of 44.81(±0.02) edges by Kb of storage, where the biggest cache file takes 113Kb for 4806 edges.

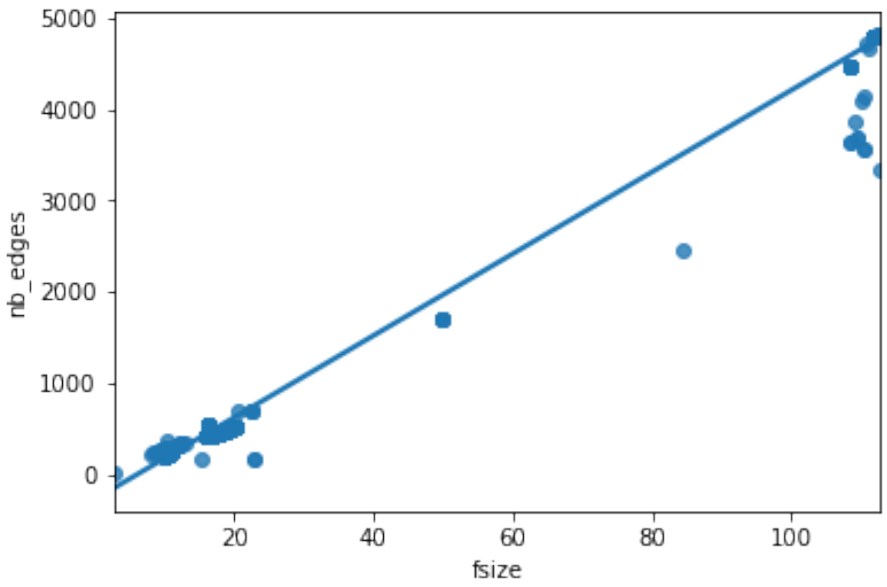


Figure 4.5 – Plot of *#edges* vs. *cache size* (KB) for 12K random files

Table 4.4 shows that σ^{ECDG} largely outperforms (in time) *radiff2*, achieving a speedup gain of 3.30x and 354.11x for the standard and cache-enhanced implementations wrt. *radiff*.

It also shows that individual jobs in the *code analysis* and *graph generation* stage are much more expensive than the *graph analysis* jobs (i.e. σ^{ECDG}). However the latter largely

Table 4.4 – Benchmark evaluation

Stage	**Dataset group**	**job time MEAN (SD, MED)**	**% common subgraph**
Code analysis (CA) Graph generation (CG)	I + II + III	640.32s (419s, 478s)	
σ^{ECDG}	I	10.27s (25s, 3ms)	54,84%
σ^{ECDG}	II	8.67s (20,6s, 4ms)	57,45%
σ^{ECDG}	II + III	7.21s (19.5s, 4ms)	51,60%
radiff2	*Benchmark-DS*	4181.35s (6580s, 2031s)	
CA + GC + σ^{ECDG}	*Benchmark-DS*	1266.78s (570s, 1104s)	
cached-(CA + GC) + σ^{ECDG}	*Benchmark-DS*	11.81s (24s, 2.5s)	

dominates the overall calculation time because it requires a quadratic number of computations (to build the similarity/distance matrix), while the former ones grow linearly with respect to the total number of instances. So our modular approach allowed us to devote more effort to the gSpan implementation, reducing memory footprint more than 100x, achieving 35x speedup with multi-threading and up to 6x speedup with a single thread wrt. original implementation [318].

To compare the similarity results of σ^{ECDG} and *radiff2*, their values were split into four categories: *strong-dissimilarity* ($\in [0, 0.25[$), *weak-dissimilarity* ($\in [0.25, 0.5[$), *weak-similarity* ($\in [0.5, 0.75[$) and *strong-similarity* ($\in [0.75, 0.1]$).

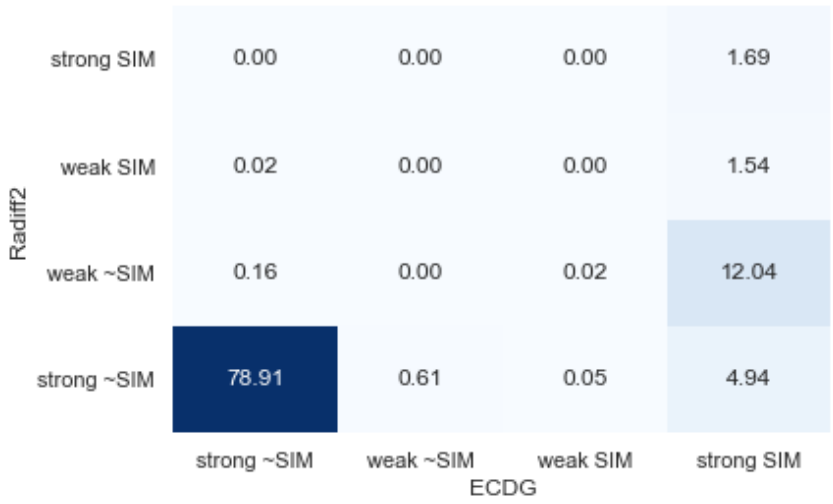


Figure 4.6 – Contingency matrix of σ^{ECDG}/*radiff2*

Figure 4.6 shows the contingency matrix of σ^{ECDG}/*radiff2*, where each cell contains the

percentage of pairs in *Benchmark-DS* whose similarity felt into weak/strong similarity/dissimilarity (notated as SIM/~SIM) for σ^{ECDG} and *radiff2*. Ideally, supposing that σ^{ECDG} and *radiff2* were both flawless, all values would be placed in the matrix main skew diagonal. However, we note that in general σ^{ECDG} is able to find stronger similarities than *radiff2* for the same pair of files. In particular, σ^{ECDG} found strong similarities where *radiff2* found weak and strong dissimilarities for 12.04% and 4.94% of the file pairs, respectively. A hypothesis to explain the differences in the similarity values obtained with σ^{ECDG} and *radiff2* is the fact that σ^{ECDG} targets semantic similarity whereas *radiff2* relies on comparisons of the programs control-flow graphs.

4.5.4 Parametrization: NEF Selection

We do an exploratory analysis to select a value for parameter *nef*, as required by the setup of σ^{ECDG} (c.f. section 4.4.2, page 164). Since *group I* contains files corresponding to the most trustworthy ground truth in the dataset, we use this group for the analysis. Furthermore, we target the *homogeneity score* as our main metric of interest.

In this exploration, we build various similarity/distance matrices containing all pairwise computations of σ^{ECDG} with different *nef* values. For each matrix we compute clustering instances using different algorithms: Agglomerative, DBSCAN, HDBSCAN and OPTICS. For algorithms that require input parameters (i.e. DBSCAN and Agglomerative), we performed a hyper-parameter tuning, which adds up to four more clustering instances for each *nef* iteration.

Figure 4.7 shows the *nef* exploration for HDBSCAN with homogeneity score as target metric —the remaining curves are shown in appendix section 6.7.1 (page 334). Overall, the profile of all metric curves for OPTICS and HDBSCAN are fairly similar, whereas HBSCAN and Agglomerative have significantly different performances depending on the metric chosen as target. In all cases, the best scores are achieved with *homogeneity* or *silhouette* as target metric, while the worst are obtained with the *completeness* score. When tuned by silhouette score, both HDBSCAN and Agglomerative clustering displayed a decay in completeness score for $nef \geq 0.5$.

As outcome of the whole exploratory analysis, we selected value 0.25 for *nef*, because it provides positive results for all metrics, in particular for the *homogeneity score*. It also takes both nodes and edges components of σ^{ECDG} into account, thus balancing between localized and holistic views of the graphs.

We present and discuss only the clustering results of HDBSCAN in the following exper-

Table 4.5 – Clusterings for *nef* = 0.25

	DBSCAN	**HDBSCAN**	**OPTICS**	**Aggl.**
#clusters	6	7	7	13
#noise	5	4	2	3
silhouette	0.969	0.955	0.956	0.978
similarity	0.946	0.954	0.949	0.991
homogeneity	0.995	0.983	0.978	0.995
completeness	0.846	0.796	0.796	0.828

iments, since the exploratory analysis shows very stable and good results for homogeneity and completeness scores.

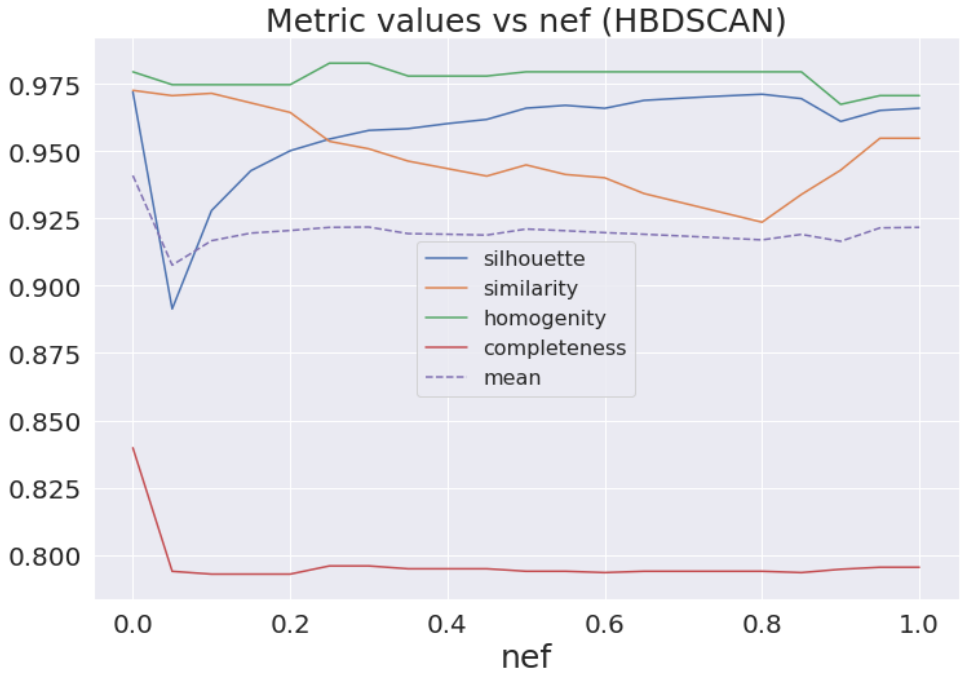


Figure 4.7 – *nef* exploration for HDBSCAN

4.5.5 Accuracy-and-Robustness (AnR) Analysis

Our experiments aim to evaluate σ^{ECDG} accuracy, robustness and practical efficiency, all prerequisites to enable functional application like malware search and malware clustering.

Accuracy phase

We evaluate σ^{ECDG} accuracy, only on files in *group I*. This phase derive directly from the *nef* selection exploratory analysis. Here we detail the corresponding experiment, i.e. clustering of *group I* files with HDBSCAN and *nef* = 0.25.

Clustering results for this phase (see also table 4.5, column HDBSCAN) are shown in table 4.6, and illustrated in the heatmap figure 4.8. They contain 8 clusters and 4 singletons, with 0.955 silhouette, 0.954 mean similarity, 0.983 homogeneity and 0.796 completeness, the latter two computed with Yara rules created for the definition of the *group I*.

Table 4.6 – *Accuracy phase* clusters

Cluster	#samples	Similarity	Yara Rule	#samples (per rule)
0	127	0.979	Shohdi	127
1	132	0.96	Shohdi TWarBot	2 130
2	132	0.998	Mira	132
3	18	1.000	Mira	18
4	6	0.998	Shohdi	6
5	150	0.994	Bogy	150
6	10	0.701	Shohdi	10
7	21	1.0	Shohdi TWarBot	1 20

The clusters are very well discriminated. Out of 600 files, only 3 samples are found in mixed clusters (according to our ground truth): two Shohdi samples in cluster #1 and one in cluster #7, both composed predominantly of TWarBot samples. *All other clusters are pure* and cluster #5 is complete, including all 150 Bogy samples.

Robustness phase

We evaluate the robustness of σ^{ECDG} using *real world* samples (*group II* files). We assess how much the clustering degrades as *noise* (*group III* files) is gradually inserted in the evaluation dataset. Table 4.7 shows the clusterings (HDBSCAN with *nef* = 0.25) of the initial state (*group II* files) and of the final state (*group II* and *group III* files).

The initial clustering results in 20 clusters with 54 singletons (figure 4.10), with scores of 0.789 silhouette, 0.974 mean similarity, 0.746 homogeneity and 0.712 completeness, where the two last metrics are computed with public Yara rules (i.e. PAS) as ground truth.

Table 4.7 – *Robustness phase* clusters: initial (*group II*) and final (*group II + group III*)

Cluster index		**Initial**				**Final**			
Initial	Final	#samples	Similarity	*Yara rule*	#samples (per rule)	#samples	Similarity	*Yara rule*	#samples (per rule)
0	0	9	0.978	ClamAV_Emotet_String_Aggregate	9	9	0.978	ClamAV_Emotet_String_Aggregate	9
4	1	10	0.951	spyeye Glasses	2 8	10	0.951	spyeye Glasses	2 8
3	2	158	0.987	Wabot	158	158	0.987	Wabot	158
-	3					5	0.98	ClamAV_Emotet_String_Aggregate spyeye Warp	1 2 2
12	4	7	1.000	spyeye	7	7	1.000	spyeye	7
13	5	19	0.993	spyeye	19	19	0.993	spyeye	19
14	6	18	0.998	spyeye	18	18	0.998	spyeye	18
9	7	23	0.995	spyeye	23	23	0.995	spyeye	23
6	8	11	0.819	spyeye	11	14	0.605	spyeye cleanware	11 3
10	9	15	0.993	spyeye	15	15	0.993	spyeye	15
11	10	6	1.000	spyeye	6	6	1.000	spyeye	6
8	11	59	0.992	ClamAV_Emotet_String_Aggregate Njrat njrat1 win_exe_njRAT	1 58 58 58	59	0.992	ClamAV_Emotet_String_Aggregate Njrat njrat1 win_exe_njRAT	1 58 58 58
2	12	10	0.995	IceID_Bank_trojan	10	10	0.995	IceID_Bank_trojan	10
-	13					24	0.997	Glasses IceID_Bank_trojan cleanware	1 2 21
1	14	187	0.997	Warp Wabot Glasses IceID_Bank_trojan sakula_v1_3 Bublik	1 4 1 26 80 88	193	0.998	Warp Wabot cleanware IceID_Bank_trojan sakula_v1_3 Bublik	1 4 10 23 80 88
7	15	128	0.994	ClamAV_Emotet_String_Aggregate Glasses Mirage_APT Cerberus spyeye	5 32 41 49 1	127	0.998	ClamAV_Emotet_String_Aggregate Glasses Mirage_APT Cerberus	5 32 41 49
-	16					5	0.934	cleanware	5
-	17					24	0.998	cleanware	24
-	18					34	0.990	cleanware	34
5	19	19	0.877	ClamAV_Emotet_String_Aggregate Glasses spyeye	5 2 12	55	0.516	ClamAV_Emotet_String_Aggregate Glasses spyeye cleanware	1 2 12 40
19	20	7	0.983	ClamAV_Emotet_String_Aggregate Monero_Mining_Detection	1 6	16	0.988	ClamAV_Emotet_String_Aggregate Monero_Mining_Detection cleanware	1 7 8
18	21	5	1.000	Monero_Mining_Detection	5	6	0.998	Monero_Mining_Detection	6
-	22					10	0.959	Warp cleanware	2 8
-	23					8	0.985	ClamAV_Emotet_String_Aggregate cleanware	3 5
-	24					12	0.867	IceID_Bank_trojan cleanware	1 11
-	25					10	0.927	IceID_Bank_trojan cleanware	1 9
17	26	133	0.995	IceID_Bank_trojan shylock	26 109	133	0.995	IceID_Bank_trojan shylock	26 109
15	27	96	0.935	spyeye Monero_Mining_Detection IceID_Bank_trojan	1 11 84	108		spyeye Monero_Mining_Detection IceID_Bank_trojan cleanware	1 11 83 13
16	28	27	0.995	Glasses spyeye ClamAV_Emotet_String_Aggregate	1 25 1	29	0.997	Glasses spyeye cleanware	1 25 3
-	29					13	0.954	cleanware	13
noise				54				340	

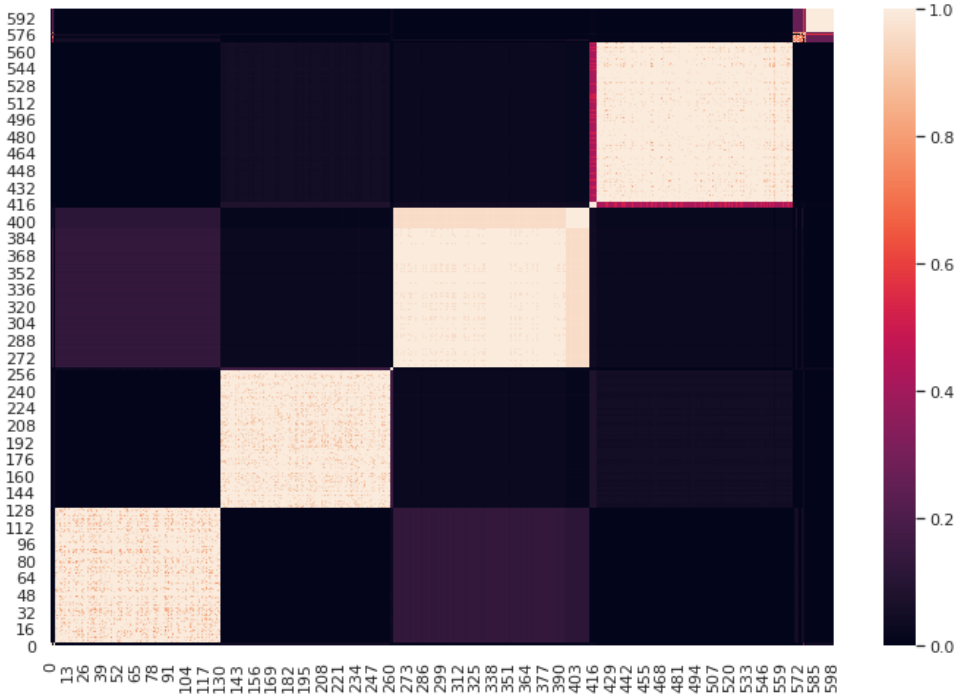


Figure 4.8 – Heatmap of clusters in the *accuracy phase*

This experiment creates a majority of pure clusters (11 of 20), especially for smaller clusters, which suggests that σ^{ECDG} can discriminate MW families at the variant level. For instance, *spyeye* produces 7 pure clusters and another one which is almost pure (25 of 27 samples). In addition, some bigger clusters are completely or almost pure. Cluster #3 is pure, with 158 *Wabot* samples and cluster #8 includes 58 *Njrat* samples from a total of 59 samples.

Cluster #1 (187 samples) has the greatest number of different families (6), dominated by *Bublik* (88 samples) and *sakula_v1_3* (80 samples). Furthermore, 13 samples are detected by both Yara rules, which suggests some commonality between both families. Similarly, cluster #17 includes all 109 *shylock* samples along with 26 *IceID_Bank_trojan*, two samples being detected by both Yara rules.

Our clustering also exposes lower quality Yara rules like *ClamAV_Emotet_String_Aggregate*, which performs poorly. Indeed, this rule was heuristically generated, as revealed by its description:“a pruned aggregate of all Emotet related strings extracted from ClamAV on 2019-07-03”. Yet, out of 36 samples detected by this rule, 9 sit together in pure cluster #0 and 13 others appear scattered in 5 other clusters. This result reinforces our claim that σ^{ECDG} is *accurate*.

To evaluate the robustness of our behavioral clustering, **we gradually introduce clean-**

ware samples of *group III* to act as noise in the dataset and we measure the disturbance introduced by this action. Figure 4.10 shows the initial clustering, before inserting cleanware samples in the dataset, and figure 4.11 shows the final clustering, after the insertion of the cleanware samples. Figure 4.9 shows the impact of *group III* files on our target clustering metrics [6].

Homogeneity slightly increases after inserting 50% benign files (1/3 of the whole final dataset). Silhouette is the most affected metric, being impacted by new, rather small, spurious clusters. The final clustering achieved scores of 0.464 silhouette, 0.950 mean similarity, 0.738 homogeneity and 0.607 completeness. The NMI measured between the initial and final clusterings was 0.974, meaning that the information loss due to insertion of noise was only marginal.

Table 4.7 displays the clusters after complete insertion of *group III* files in the final clustering. As expected, most cleanware are not included in any cluster, which increases the number of singletons (noise) from 30 in the initial clustering to 340 in the final one. Most of thee original clusters remain unaffected (grey rows in table 4.7), only 4 being altered by the inclusion of a few cleanware instances (red rows). A few clusters finish with less files or have some files replaced, which increases the clusters mean average similarity (green rows). Ten new clusters are created (yellow rows), of which four of them comprising only cleanware, and the others include a few malware samples but with relatively low average similarity.

The results confirm that σ^{ECDG} is *robust* even with highly polluted datasets.

4.5.6 Prototype Analysis

A cluster prototype is defined as *a data object that is representative of the other objects in the cluster* [278]. Here, for a given cluster, we define the *cluster prototype* $\mathscr{P}_\tau(\mathscr{C})$ as the greatest common connected subgraph with minimum support threshold τ (c.f. section 3.2.2, page 122) for the entire set of graphs in this cluster. We explore the *cluster prototypes* by incrementing τ from 0.05 to 1.0 with 0.05 steps, trying to compute prototypes for all clusters at each iteration. We repeat this with the clusterings of the *accuracy phase* and the initial clustering of the *robustness phase*.

For the clusterings of the *accuracy phase*, all support values below 1 successfully produce prototypes for *all clusters*. Only one cluster fails to provide a prototype for support value equal 1. Figure 4.12a shows an example of *cluster prototype* obtained during the *accuracy*

6. To recall the definition and use of evaluation metrics, refer to the sections 2.2.1 (page 60) and 2.2.3 (page 82).

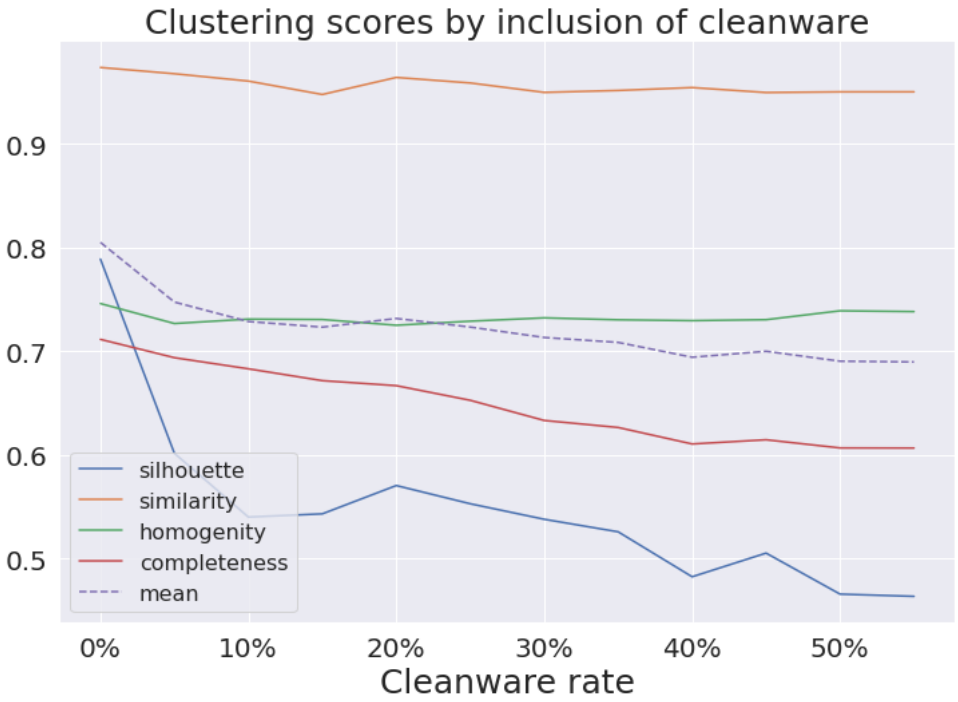


Figure 4.9 – Noise evaluation in the *robustness phase* [HDBSCAN (*nef* =0.25)]

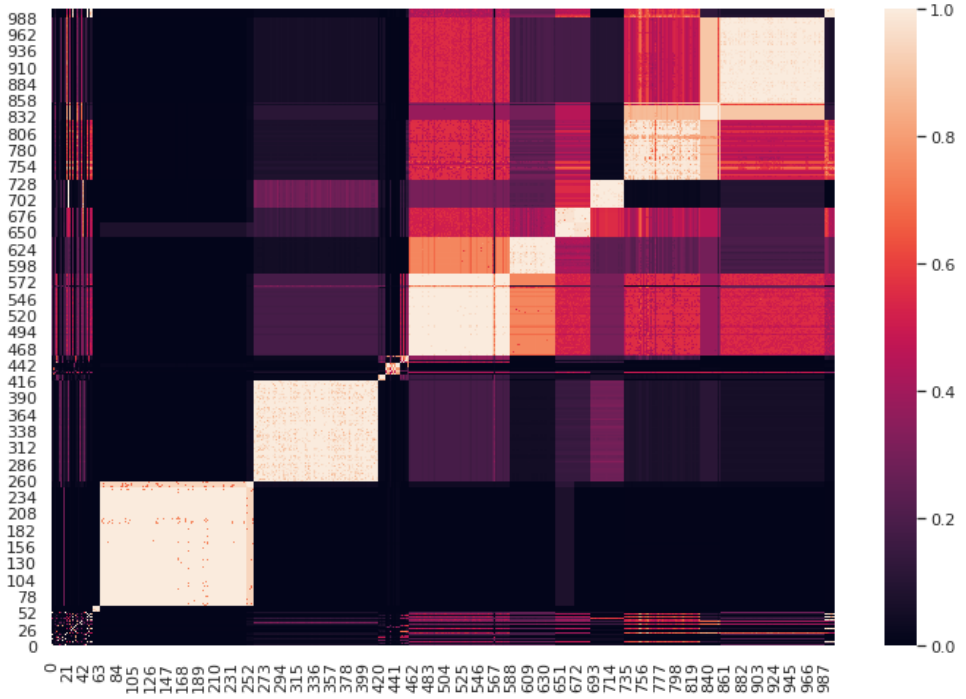


Figure 4.10 – Initial clusterings: *group II* files

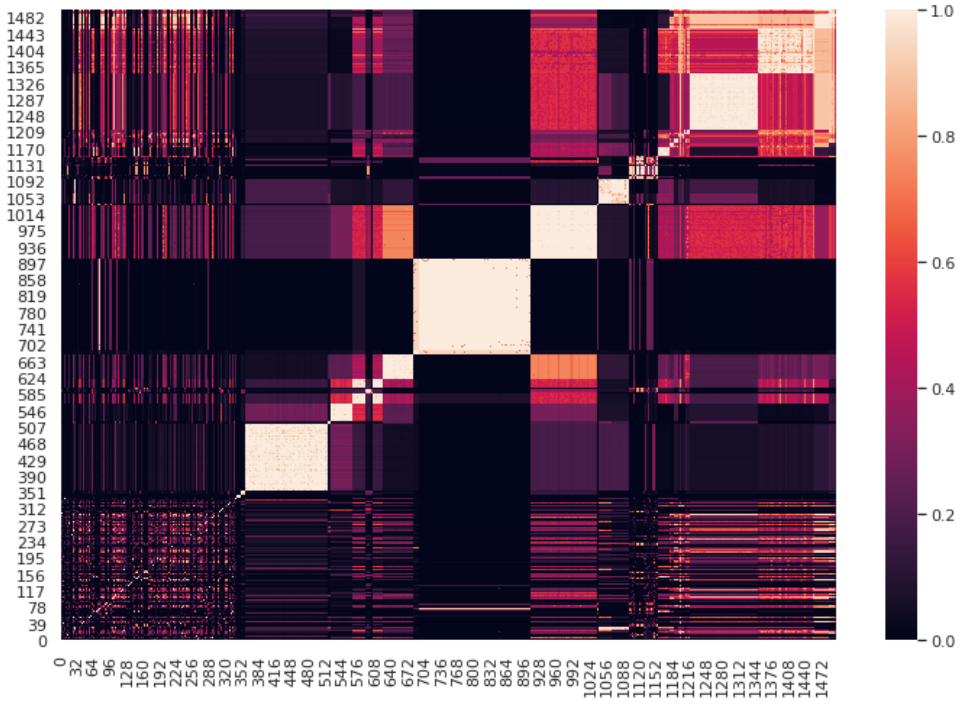


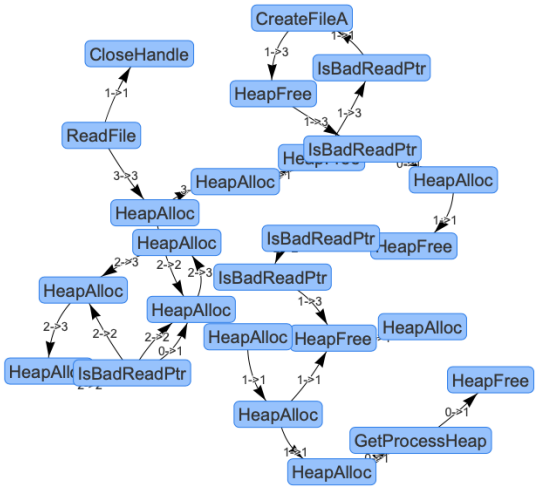
Figure 4.11 – Final clusterings: *group II* + *group III* files

phase for cluster#5, which corresponds to the *Bogy* family (cluster #5). For the initial clustering of the *robustness phase*, when *nef* ≤ 0.5 all clusters successfully produce prototypes, i.e. at least one common subgraph can be found in half of samples for each individual cluster. For greater *nef* values, the rate of prototypes decreases almost linearly. This means that the clusters have in general a core of nearly half of their samples that bear a high similarity, while the other half of the samples (i.e. less similar) gets gradually incorporated in the cluster. Figure 4.13 shows the percentage of clusters that successfully produce prototypes as function of the support.

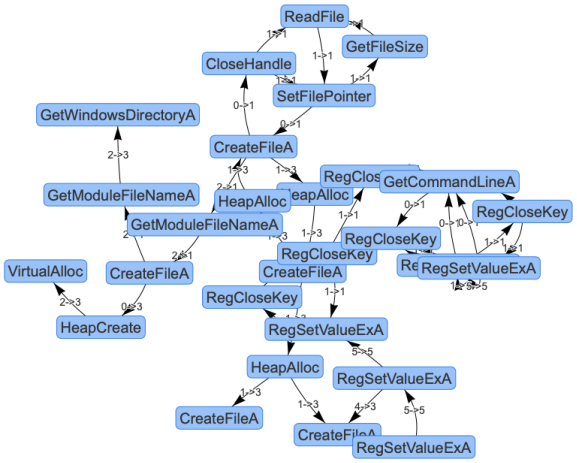
This behavior is convenient to unveil hidden similarities between malware families and variants, which would be impossible with the all-or-nothing approach of syntactical signatures. Figure 4.12b shows an example of *cluster prototype* obtained for cluster#2.

4.6 Discussion

This chapter analyzes σ^{ECDG} , which proved to be an efficient structural representation of programs that is reliably accurate and precise. We first tackled general research issues, namely the paradigm that guide our malware analysis evaluation (c.f. section 4.4.3, page 165).



(a) *accuracy phase*: cluster #5 ($\tau_{min} = 0.5$)



(b) *robustness phase*: cluster #2 ($\tau_{min} = 1$)

Figure 4.12 – Examples of clustering prototypes

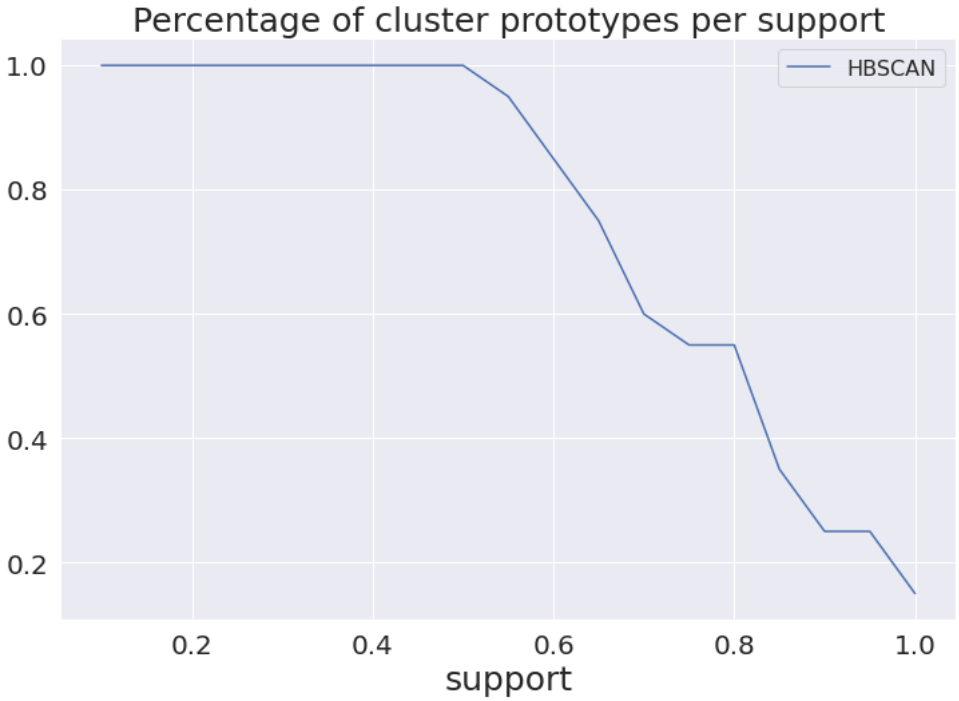


Figure 4.13 – Prototype exploration for the *robustness phase*

After showing that code similarity schemes can be extended to address search and clustering, we focused on the analysis of ECDGs and σ^{ECDG} in terms of properties desired to implement practical frameworks.

ECDGs are obtained from dependencies between external calls, therefore they are beyond the reach of any purely syntactic obfuscation (e.g. instruction replacement, opaque predicates, etc.), although syntax obfuscation can still be effective in thwarting the underlying analysis for the call tracing (e.g. our symbolic execution). Some obfuscation techniques, such as packing, can also insert new code (e.g. unpacking stub) with new external calls; however as long as this code does not interfere with the original one, the new additional calls will ensue as disjoint components added to the original ECDG, without breaking our method. To be effective against ECDG itself, the obfuscation needs to actually replace external calls or alter their argument dependencies; this procedure is much more complex than syntactic obfuscation and is not yet observed in the wild.

Furthermore, although unfit for large datasets, our evaluation clustering is able to assess the efficiency, accuracy and robustness of σ^{ECDG} . This construction can be practical with clustering strategies like those proposed by Kostakis [151] and Zhang *et al* [326], keeping the same properties verified here.

Table 4.4 shows that σ^{ECDG} calculation throughout the whole dataset has median time in the order of milliseconds, and that inclusion of noise results in lower average time to compute similarity. Indeed, gSpan typically finishes very quickly when two (or more) input graphs do not have any common subgraph. Assuming that samples of a same family represent just a small part of an entire dataset, this behavior is suitable to address the search problem, which makes σ^{ECDG} a good practical alternative for this problem.

Our prototype analysis (section 4.5.6, page 180) shows that many cluster prototypes are naturally produced, even when bigger support values are used. This is a positive consequence of using *subgraph isomorphism* as basis of σ^{ECDG} , which opens the possibility for optimization strategies like *scalable clustering* [235]. Furthermore, the cluster prototypes obtained are very descriptive (c.f. figure 4.12b), which allows assisting human analysis or training models for specific malware families.

Limitations. Our call extraction module extends angr with 24,052 new stub *SimProcedures* of commonly used functions, highly improving code coverage, hence the quality of our ECDGs. However, our experiments have limits related to state explosion in symbolic executions [17]; the main reason for job termination (93%) during *code analysis* was memory exhaustion. Additionally, the limitations of symbolic execution to handle packed binaries are

well known. To address this, we tested angr option to enable self-modifying code support [7], but this setting did not produce any noticeable gain. We noticed that packed samples often resulted in the extraction of few calls of the unpacking routines (e.g. *LoadLibraryA* and *GetProcAddress*), nonetheless several samples of the evaluation dataset were flagged by packing detection tools. Due to the 100 edges threshold (c.f. section 4.5.1, page 168) and the use of syntactical signatures, which are likely to fail with packed samples, the removal of packed binaries was not specifically handled in our work.

A possible mitigation for both issues is a smarter combination of *concrete* and *symbolic* execution (i.e. *concolic execution*) instead of pure symbolic execution. This technique should be able to bypass the unpacking routine of (simple) packers and to perform expensive analyses (i.e. CPU and memory intensive) only in a few parts of the code. To further improve the quality of call dependencies for edges in ECDGs, instead of plain comparison of (symbolic) values, a future work can include the implementation of taint analysis, where arguments and returns of functions are more finely tracked.

The computation of our similarity/distance matrix for the evaluation clearly does not scale for very big datasets. So our *evaluation* dataset for this paper is restricted in size, but this does not affect our method itself nor its practical usability. Indeed, it is only used to evaluate the accuracy and robustness of σ^{ECDG} , as well as to produce benchmarks that assess its efficiency in practice. Overall, the problem of clustering scalability (i.e. a scheme that requires asymptotically less pairwise computations) is out of the scope of this work. Nonetheless, such clustering can of course benefit from more efficient, accurate and robust pairwise similarity computations.

4.6.1 Threats to Validity

This section discusses the validity [244] of our study according to: construct, internal, external validity and reliability.

Construct Validity: we conducted a controlled experiment following the Accuracy and Robustness paradigm, which allows to assess individually and incrementally the properties of accuracy and robustness of our similarity function σ^{ECDG} . To support this evaluation, we painstakingly built our evaluation dataset leveraging a combination of heuristics that are likely to select samples in accordance with the desired profile. In this regard, no issue should arise from our experiential study.

7. Parameter that enables the emulation of code from the current state instead of the original memory, notwithstanding memory protections.

Internal Validity: our analysis follows an incremental approach, which intends to isolate the properties assessed in each evaluation phase. This diligence expressly aims to improve the internal validity of the study, therefore no issue regarding this matter should arise.

External Validity: the foundations of our work methodology are akin to other already validated in the literature —ECDG is special type of call graph, which are well-established in malware analysis, and our similarity function σ^{ECDG} is based on properties exploited by traditional methods for graphs comparison (i.e. common nodes and common subgraphs). Furthermore, our evaluation uses learning methods without any *ad-hoc* customization to our scenario, thus not affecting the generalization of these algorithms. In this sense, there should be no issue with generalization of the methodologies proposed in this study. However, the evaluation dataset can be problematic to generalization as the true profile of the population of malware and cleanware is unknown; therefore, it is impossible to know for a fact where our dataset misrepresents the true population, which could engender a selection bias issue. This is withal an unmanageable issue, which is a common concern in malware research.

Reliability: our study is based on a set of concepts and methods that are of public knowledge and well established in literature. We provide all details necessary to reproduce the analysis, and our dataset was sufficiently large to avoid the occurrence of serendipitous results (specially when considering the computation of all pairwise similarities). Furthermore, our construction uses public available Yara rules that allow to replicate the generation process of a new dataset with properties similar to ours.

4.7 Conclusion

We defined ECDG, a new call graph to optimize the representation of argument dependencies between calls, that allows concise structural representation with no information loss. It has a major positive impact on performance, because graph matching algorithms, which are very expensive, can deal with much smaller graphs.

We proposed a new similarity function σ^{ECDG} that is efficient, achieving a speedup gain of 3.30x and 354.11x for the standard and cache-enhanced implementations wrt. *radiff*. Our similarity function is also reliably *accurate* and *precise*. To support this, we built an evaluation framework to cluster samples based on σ^{ECDG} and we evaluated the results under the AnR paradigm.

The *accuracy phase* produced almost unerring results, with homogeneity score of 0.983,

which shows that our evaluation framework manages to autonomously describe malware families as accurately as a set of strict handcrafted Yara rules.

The *robustness phase* verified that the clustering was robust to noise insertion, having almost no impact on homogeneity and mean similarity, and only mild effect on completeness and silhouette scores due to creation of many singletons (c.f. figure 4.9, page 181). Indeed, the NMI score between the initial and final (highly polluted) clusterings of this phase was 0.974, indicating that information loss due to noise insertion was only marginal.

In addition, our evaluation framework produced a high rate of descriptive cluster prototypes that represent behaviors and can be used to scale up clustering, assist manual analysis or enhance classification models for malware detection.

As **main contributions of this chapter** (i) we revisited the foundations of malware analysis research, defining basic malware analysis primitives, to propose the *AnR paradigm* for reliable evaluation methodologies; (ii) we proposed ECDGs, as a compact call dependency graph that enables more efficient binary similarity computation; and (iii) we proposed a new similarity function for ECDGs that is efficient, accurate and robust.

Few other indirect contributions come from our concrete implementation, namely the study of symbolic execution to trace external calls, the evaluation of gSpan as a practical algorithm for sub-graph isomorphism, and the evaluation of cluster prototypes extraction to represent malware families.

Ultimately, our experiments show that σ^{ECDG} can reliably work as cornerstone of multiple types of frameworks, from those who autonomously produce descriptions of malware families as accurately as manually created syntactical signatures to those who target malware search and malware clustering.

CHAPTER 5

EMB-DUET: MULTI-FEATURE CLUSTERING BASED ON NUMERICAL EMBEDDING

5.1 Introduction

Historically malware analysis heavily resorted to experts who manually create signatures for malware detection and classification. These signatures very often target syntactic properties of the files like text strings or regular expressions of byte code [1], which are susceptible to anti-analysis techniques that are effective to evade detection such as obfuscation, packing, polymorphism and metamorphism. Albeit largely used in practice for threat intelligence, malware triage, or other scenarios (e.g. identifying a ransomware family), *the primary shortcoming of the signature-based methods is that they entail high precision but low recall* [1].

To work around this hindrance it is necessary to improve the efficacy of machine learning applications in malware analysis, which poses several major challenges. First, malware classification is a *undecidable problem*, which impairs the smooth usage of supervised learning as it depends on the available ground-truth. Second, the features must meaningfully describe the programs, while being at the same time accurate, robust and friendly to machine learning algorithms (e.g. feature vectors). Finally, due to the huge number of instances that need to be analyzed in practice, the whole workflow must be very scalable.

For a malware classification framework to be truly autonomous, it is ideal to take a *data-driven approach* where the analysis outcomes derive from data themselves instead of hinging on outsourced labels. Traditional signature-based methods are themselves mostly extrapolations of manually created signature, thus not scalable.

For the issue of feature selection, it can be propitious to base the classification on multi-view learning, as most studies on multi-view learning have often proven to be able to utilize unlabeled data effectively and improve classification accuracy [286]. A good alternative is

to use *hybrid features*, which are derived from a combination of dynamic and static analysis [68], because this:

(i) diversifies tools and techniques used to extract the features, thus making it much more complex for adversaries to bypass all the analysis layers;

(ii) provides multiple views of the dataset that can be interpreted as imperfect partitions which converge to the "true" one once combined.

To tackle the efficiency issue while remaining friendly to machine learning algorithms, our approach leverages the *numerical embedding* technique, which is particularly useful to analyze unstructured data that cannot be organized in predefined structures (e.g. text strings for sentiment analysis) [66]. We use numerical embedding to transform the unstructured data views into numerical vectors that can be efficiently analyzed by different algorithms, which enables to scale up clustering analysis for a large corpus of binary files.

To consolidate all individual views into a single clustering, we use methods of *ensemble clustering* [291]. They allow to improve the quality of the ensembles by combining them into a final unique clustering. Our solution blue print is similar to DUET by X. Hu and K. Shin [124], but we add a feature vectorization step —to compute numerical embedding vectors —that normalizes the input data and boosts the overall performance of the framework.

As main contributions, we:

(i) proposed EMB-DUET as a data-driven framework that is able to handle heterogeneous features, which leverages numerical embedding to boost overall performance [addresses **RQ5.1**];

(ii) verified the accuracy and robustness afforded by the aggregation of the different data views (i.e. `strings`, `n-grams` and `e-ECDGs`) [addresses **RQ5.2**];

(iii) analyzed the impact and effects of different hyperparameters involved in EMB-DUET and σ^{ECDG} -Learning with SVM Regression, so as to provide a guideline for proper parameter setting [addresses **RQ5.3**].

5.1.1 Research Questions

We address the top-level research question **TOP-RQ3**: *How to improve the arsenal of techniques currently used in malware analysis, in particular for the analysis of binary files?*

For this, we break down this backbone question into the following subordinate research questions:

RQ5.1 *How to design a data-driven clustering that can systematically work with heterogeneous features?*

RQ5.2 *Which set of features can afford accuracy and robustness to the data-driven clustering?*

RQ5.3 *How to optimize the data-driven clustering to make it scalable?*

5.1.2 Chapter Outline

The remainder of this chapter is organized as follows:

— Section 5.2 (page 191) presents the specific background of this chapter;

— Section 5.3 (page 204) presents the related works;

— Section 5.4 (page 208) presents our methodology, which introduces σ^{ECDG} -learning with SVM Regression and EMB-DUET;

— Section 5.5 (page 212) presents the experimental setup and the evaluation results.

— Section 5.6 (page 252) discusses results.

— Section 5.7 (page 256) concludes.

5.2 Background

5.2.1 Support-Vector Machines (SVMs)

Support-Vector Machines constitute a family of prediction methods that can be used for supervised classification as well as regression analysis (case referred as *support-vector regression*).

In its simplest form (i.e. as a linear classifier), SVM can be seen as an algorithm that takes a set of data points within a n-dimensions hyperspace and finds a $(n-1)$-dimensional hyperplane that separates both sides of the hyperspace. For this, the algorithm finds a function that approximates the data points within a precision margin, where misclassification on the learning data are tolerated in order to optimize the trade-off between bias and variance error.

More formally, given the training data $C_1 = \{(x_1, y_1), \cdots, (x_n, y_n)\} \subset X \times \mathbb{R}$ (data points belonging to class 1) and $C_{-1} = \{(x'_1, y'_1), \cdots, (x'_m, y'_m)\} \subset X \times \mathbb{R}$ (data points belonging to class -1), where X denotes the input space, the goal is to find $f(x)$ such that the deviation is greater than ϵ to any $y_i \in \{y_1, \cdots, y_n\} \cup \{y'_1, \cdots, y'_m\}$. In case of linear functions, $f(x)$ can be defined as:

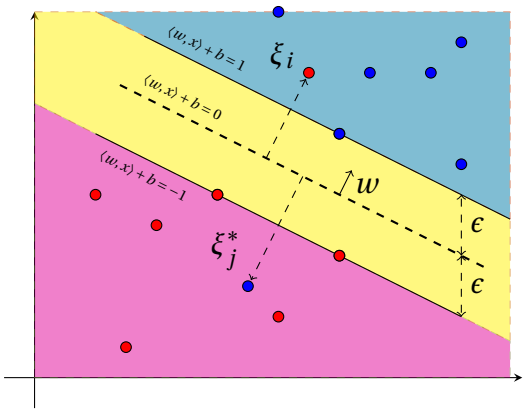


Figure 5.1 – Maximum-margin hyperplane and margins computed in SVM.

$$f(x) = \langle w, x\rangle + b, w \in X, b \in \mathbb{R}$$

where $\langle \cdot, \cdot \rangle$ denotes the dot product in X and w is a normal vector to the hyperplane. The data points can be normalized so as to have the following equation to describe the hyperplanes (as shown in figure 5.1):

$$\langle w, x\rangle + b = 1 \quad \text{(margin for class 1)}$$
$$\langle w, x\rangle + b = 0 \quad \text{(maximum-margin hyperplane)}$$
$$\langle w, x\rangle + b = -1 \quad \text{(margin for class -1)}$$

Figure 5.1 shows an example of SVM computation, where the area in blue indicates the region above the hyperplane $\langle w, x\rangle + b = 1$ (i.e. margin of class 1), the area in red indicates the region below the hyperplane $\langle w, x\rangle + b = -1$ (i.e. margin of class -1), and the area in yellow indicates the region between the margins where misclassification is tolerated (without punishment).

It is possible to demonstrate that the distance between the margins in this normalized formulation is $2/||w||$ (c.f. appendix section 6.8.1, page 344). Since the goal of SVM is to maximize the distance between the margins, the SVM problem can be rewritten as a convex optimization problem as follows:

$$\text{minimize } \frac{1}{2}||w||^2$$

$$\text{subject to} \begin{cases} y_i - \langle w, x_i \rangle - b \geq \epsilon \\ \langle w, x_i \rangle + b - y_i \geq \epsilon \end{cases}$$

If the training data (of the example) is linearly separable, then the convex optimization problem is feasible. Otherwise, the slack variables ξ_i, ξ_i^* can be introduced to cope with the optimization problem constraints, which corresponds to Vapnik's formulation of the soft margin:

$$\text{minimize } \frac{1}{2}||w||^2 + C\sum_{i=1}^{l}(\xi_i + \xi_i^*)$$

$$\text{subject to} \begin{cases} y_i - \langle w, x_i \rangle - b \geq \epsilon + \xi_i \\ \langle w, x_i \rangle + b - y_i \geq \epsilon + \xi_i^* \\ \xi_i, \xi_i^* \geq 0 \end{cases}$$

where the constant $C > 0$ determines the trade-off between the flatness of the maximum-margin hyperplane and the total deviation larger than ϵ that is tolerated.

It is possible to show that finding a solution for this optimization problem depends solely on the dot product of data point pairs [142] (which goes beyond the scope of this thesis). Therefore, given a transformation $k : X \times X \to \mathbb{R}$ whose result is equal to the inner product in another space V (not necessarily linear), it is possible compute the SVM in this new space V without explicitly recomputing the coordinates of the data points in there —which is known as the *kernel trick*, as k is often referred to as the kernel function.

More formally, given the transformation $\phi : X \to V$ of vectors in X onto V the only requirement on k for applying the kernel trick is:

$$k(x_i, x_j) = \langle \phi(x_i), \phi(x_j) \rangle_V$$

This allows the SVM algorithm to efficiently fit the maximum-margin hyperplane in a transformed feature space, which may be nonlinear and high-dimensional (as depicted in figure 5.2).

Some common kernels include:

$$k(x_i, x_j) = \langle x_i, x_j \rangle \qquad \text{Linear}$$
$$k(x_i, x_j) = (\langle x_i, x_j \rangle + r)^d \qquad \text{Polynomial}$$
$$k(x_i, x_j) = e^{-\gamma ||x_i - x_j||^2} \qquad \text{Radial Basis Function (RBF)}$$
$$k(x_i, x_j) = tanh(k\langle x_i, x_j \rangle + c) \qquad \text{Sigmoid}$$

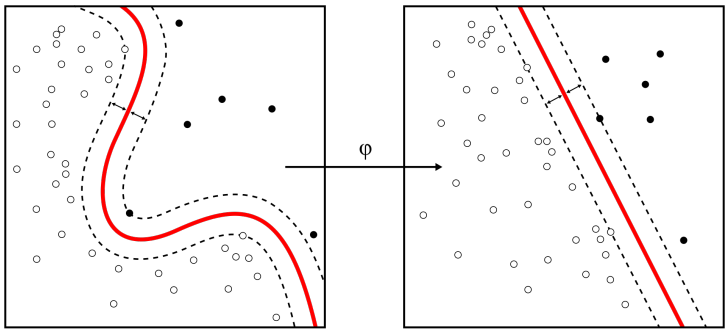


Figure 5.2 – Maximum-margin hyperplane and margins computation using the kernel trick [1].

The case of support-vector regression is almost analogous to the problem exposed above; the difference is that in classification the goal is to find the maximal margins that separate the data points (setting the maximum-margin hyperplane halfway between these margins), whereas in regression the goal is to find the maximal margins such that most of the data points are inside the region comprised between them (again, setting the maximum-margin hyperplane halfway between the margins).

Following this analogy, the support-vector regression problem can be rewritten as a convex optimization problem as follows:

$$\text{minimize } \frac{1}{2}||w||^2 + C\sum_{i=1}^{l}(\xi_i + \xi_i^*)$$
$$\text{subject to} \begin{cases} y_i - \langle w, x_i \rangle - b \leq \epsilon + \xi_i \\ \langle w, x_i \rangle + b - y_i \leq \epsilon + \xi_i^* \\ \xi_i, \xi_i^* \geq 0 \end{cases}$$

For a given data point P in X, its regression corresponds to the projection of P on the

1. Adapted from https://commons.wikimedia.org/wiki/File:Kernel_Machine.svg.

maximum-margin hyperplane defined by solving the convex optimization problem above.

Similarly to the classification case, it is possible to use the kernel trick to compute regression in a non-linear space and also to choose parameters that act on the trade-off between the flatness of the maximum-margin hyperplane and the total deviation beyond ϵ, which makes SVM a powerful regression method. For greater details on SVM regression refer to the tutorial by AJ Smola and B Schölkopf [265].

5.2.2 Embedding Learning

Numerical embedding is an essential part of modern data analysis, specially used to analyze unstructured data that cannot be organized in predefined structures, such as texts, graphs, videos, objects and chemical compounds [66]. The goal is to map unstructured data onto lower-dimensional numerical vectors that are able to retain key properties of the original data, in opposition to traditional methods (e.g. one-hot encoding), which do not inherent any relationship between the unities of the original data.

Once the numerical embedding for a given unstructured data is computed, any downstream analysis method can be applied. This is referred as the *two-stage method* and it has become a standard tool of machine learning [66].

More precisely, the idea of embedding learning is to reduce the dimension of a *predictor set* $\mathscr{S} = \{s_1, \cdots, s_n\}$ with an embedding $\mathscr{X} = \{\mathscr{X}(s)\}_{s\in\mathscr{S}}$, where $\mathscr{X} : \mathscr{S} \to \mathbb{R}^p$ and p is a tuning parameter (usually) no greater than $|\mathscr{S}|$. It means that the embedding $\mathscr{X}$ works as a digest of $\mathscr{S}$, which provides numerical vectors (in $\mathbb{R}^p$) that are friendly to machine learning methods.

We say that $\mathscr{X}$ is an ideal embedding if it can represent $\mathscr{S}$ in the learning process without information loss. In this situation, $\mathscr{X}$ is said to be a *sufficient learning-adaptive embedding*. Formally, it is defined as [66]:

Definition 5 (Sufficient learning-adaptive embedding) *An embedding $\mathscr{X} = \{\mathscr{X}(s)\}_{s\in\mathscr{S}}$ is learning-adaptive sufficient to predict the outcome of Y if the conditional distribution of Y given an embedding $\mathscr{X}(s) = \mathbf{x}$ does not depend on $S = s$, i.e. $\mathbb{P}(Y = y|S = s, \int = \mathbf{x}) = \mathbb{P}(Y = y|\int = \mathbf{x})$.*

According to this definition, learning form a sufficient embedding is equivalent to learning form the original unstructured data. In practice, the superiority in terms of empirical performance for numerical embedding for some learning tasks is widely acknowledged over the traditional approach based on manually constructed features [66].

Continuous Bag-of-Word (CBOW) & Skip-Gram

CBOW and Skip-Gram are model architectures proposed by Mikolov *et al* at Google in 2013 in the context of natural language processing (NLP) [191]. They were proposed in the wake of neural network language models based on word embedding, which substantially improved the learning generalization when compared to traditional distributed/categorical representation of language models (e.g. n-gram) [189].

Surprisingly, the learned word representation is able to capture meaningful syntactic and semantic information contained in words. For instance, denoting as v_w the vector for the word w, the distances for singular/plural relations are similar (e.g. $v_{apple} - v_{apples} \approx v_{car} - v_{cars} \approx v_{family} - v_{families}$); remarkably, the semantic constructions are also captured by the distances of this vector representation (e.g. $v_{king} - v_{man} + v_{woman} \approx v_{queen}$).

This work became a highly influential example of numerical embedding as it was the first architecture to successfully train over few hundred of millions of words with a modest dimensionality of word vectors (between 50 and 100). This was made possible thanks to the simplification of previous neural network language models (NNLM) that included non-linear hidden layers, which are powerful tools but require higher computational costs for training.

Both models exploit the principle of *collocation*, which states that related words have a natural tendency of appearing near each other in sentences (thus having a "mutual expectancy"). Thus, they use a simple feedforward NNLM (which has a single linear projection layer that is shared for all words) to predict *target* and *context* words: CBOW predicts target words based on a set of context words (referred as "bag"), whereas Skip-grams predicts context words given a target word.

For a text sentence composed of the words $w_1, w_2, \cdots, w_T$, the *training context* is defined as the set of words that surround a target (centered) word w_t, i.e. the training context of size c includes the words $\{w_{t-c}, w_{t-c+1}, \cdots, w_{t+c-1}, w_{t+c}\}$ (not including w_t). Figure 5.3 shows the model architectures for CBOW and Skip-gram (with context of size 2): the input layer takes the one-hot encoded vector as input, the hidden layer computes the corresponding embedding vector internally in order to compute the softmax unities for the output words, which are yield as prediction by the output layer.

Here we detail the Skip-gram model, as this is the model subsequently used by the next models —CBOW is similar, but with an inverted hidden layer. For a given vocabulary V and a context window of size c, the Skip-gram architecture is detailed in figure 5.4, where E is the

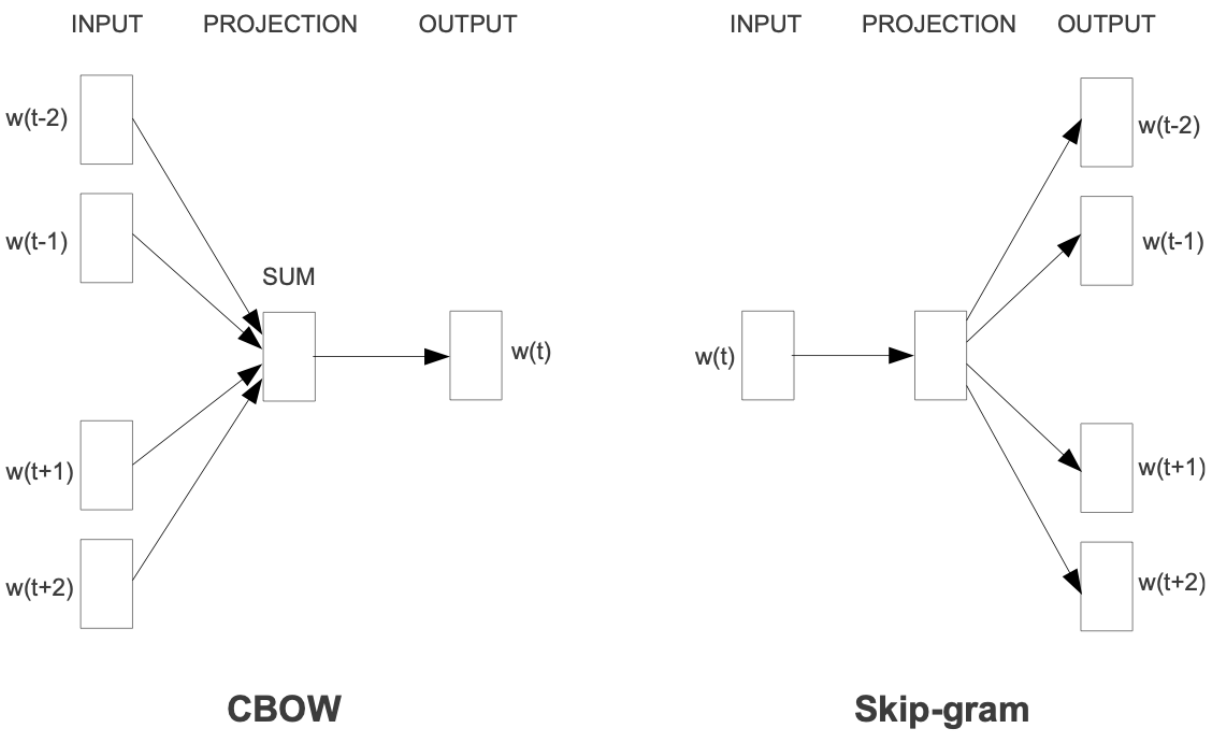


Figure 5.3 – Model architectures of CBOW and Skip-gram [191]

embedding matrix of size $|V| \times p$ [2] and W is a weighted matrix of size $p \times |V|$ used to compute the word vectors for the context words [3]. The vectors for the target word w_t and its learning context are computed as follows:

$$v_t = o_t \cdot E \qquad \text{(target word)}$$

$$v'_{t,j} = (o_j \cdot E)^T \cdot W \qquad j \in \{t-c, \cdots, t+c\}, j \neq t \qquad \text{(context words)}$$

where o_t and o_j are the one-hot encoded vectors for the target word and the context words.

Then, for each context word, the log-linear classification model softmax is used to obtain the posterior distribution of words in a vector $\hat{o}$ of size $|V|$. The training goal model is to maximize the average log probability [190], i.e. the following objective function:

2. p is the embedding degree defined as parameter, which defines the embedding $\mathscr{X} : V \rightarrow \mathbb{R}^p$
3. Two different vectors are computed for each word: one for the word when it is a target word and another when it is context word.

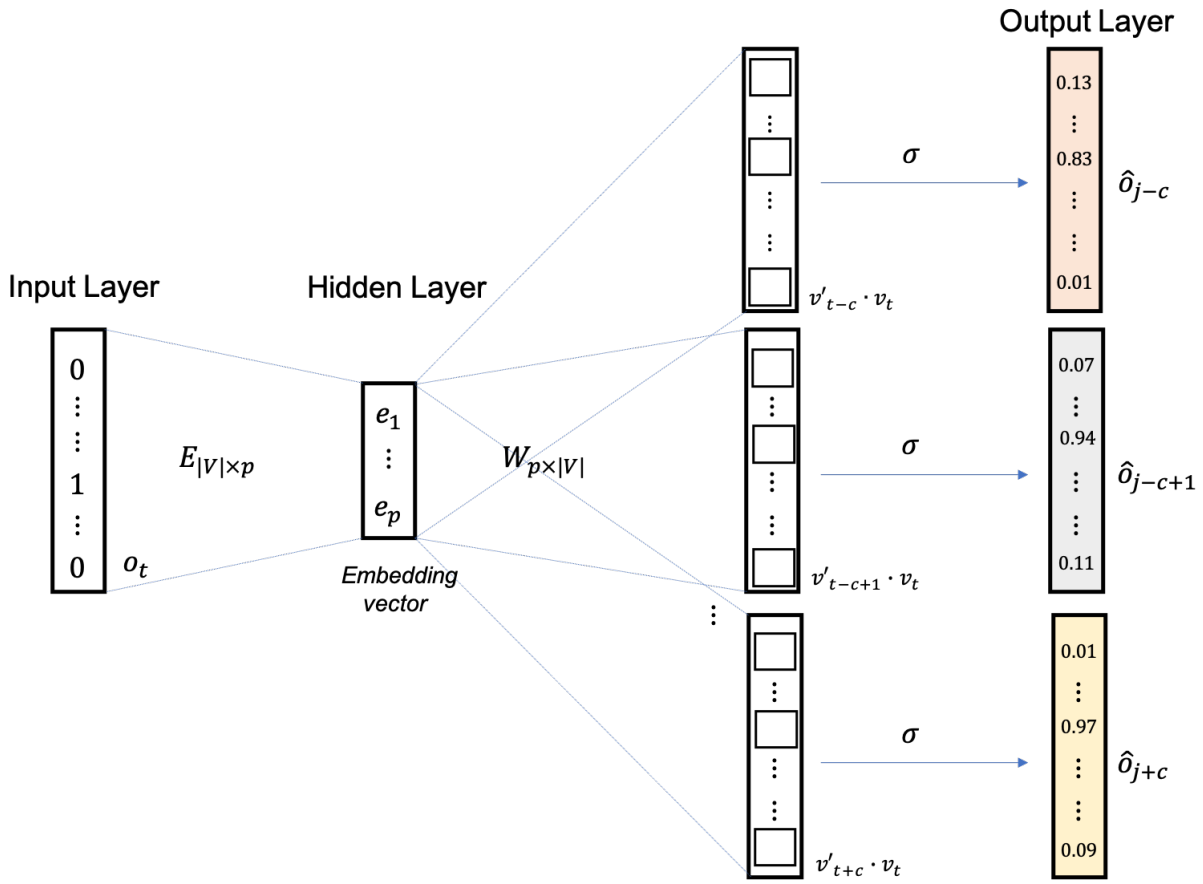


Figure 5.4 – Detailed architecture of the Skip-gram model

$$J_t(\theta) = \frac{1}{|V|} \sum_{t=1}^{|V|} \sum_{-c \le j \le c, j \neq 0} log\ p(w_{t,j} = w_{O,j} | w_t) \tag{5.1}$$

where $w_{O,j}$ is the actual word present on the j-th context position, and $p(w_{t,j} = w_{O,c} | w_t)$ is computed using the softmax probability, i.e.:

$$p(w_{t,j} = w_{O,j} | w_t) = \frac{e^{(v'^{\top}_{t,j} \cdot v_t)}}{\sum_{w=1}^{|V|} e^{(v'^{\top}_{w} \cdot v_t)}} \tag{5.2}$$

Word2Vec

Word2Vec builds on word embedding to achieve a simple, scalable and fast-to-train model. It was also proposed by Mikolov *et al* at Google in 2013 [190], basing on Skip-grams but with a new training method called *Negative Sampling*.

Negative sampling was proposed as a possible solution to simplify the maximization of the average log probability of the softmax (equations 5.1 and 5.2), because a naive classifier (e.g. logistic regression) would involve a computational cost proportional to $|V|$ (see equation 5.2). For this, the vector representations are simplified to approximately maximize the log probability of the softmax according to the following objective function:

$$J_t(\theta) = log\ \sigma(v'^{\top}_{O,j} \cdot v_t) + \sum_{k \sim P(w)} [log\ \sigma(-v'^{\top}_{k} \cdot v_t)] \tag{5.3}$$

The first term tries to maximize the log-probability for the actual words that lie in the context window, while the second term tries to iterate over random words k that (with very high probability) do not lie in the context window, aiming to minimize the log-probability of the co-occurrence. The word sampling is based on the frequency of occurrence; Mikolov *et al* propose the distribution $P(w) = U(w)^{3/4}$ —experimentally found, but not further explained —where $U(w)$ is a unigram (i.e. 1-gram) distribution.

Furthermore, to counter the imbalance between rare and frequent words, Mikolov *et al* propose a subsampling approach that discards each word w_i in the training set with probability —again experimentally found, but not further explained:

$$1 - \sqrt{\frac{t}{f(w_i)}} \tag{5.4}$$

where $f(w_i)$ is the frequency of word w_i and t is a chosen threshold (typically $\sim 10^{-5}$). This subsampling formula aggressively subsamples words whose frequency is greater than t while preserving frequencies ranking.

Word2Vec trains the Skip-gram model using standard stochastic gradient descent and backpropagation [243], boosting it with the negative sampling and subsampling techniques just presented. For more details on CBOW, Skip-gram and Word2Vec refer to Xin Rong's paper on these topics [238].

Doc2Vec

Doc2Vec is an extension of Word2Vec that works with text of variable-length (ranging from sentences to documents). This method was proposed by Q. Le and T. Mikolov at Google in 2014 [160], comprising of two models for *paragraph vectors*: a distributed memory model (PV-DM), which considers the ordering of words in a paragraph, and a distributed bag of (PV-DBOW), which ignores word ordering.

Both methods slightly modify Word2Vec to include a vector having the *paragraph id* —corresponding to the "document" terminology, as made popular by Gensim (a widely-used Python library for Machine Learning) [233]. Figure 5.5 shows the model architectures for PV-DM and PV-DBOW: the former method considers the concatenation of the paragraph vector with the word vectors to predict the target word, while the latter predicts words randomly sampled from the paragraph ignoring the context words in the input.

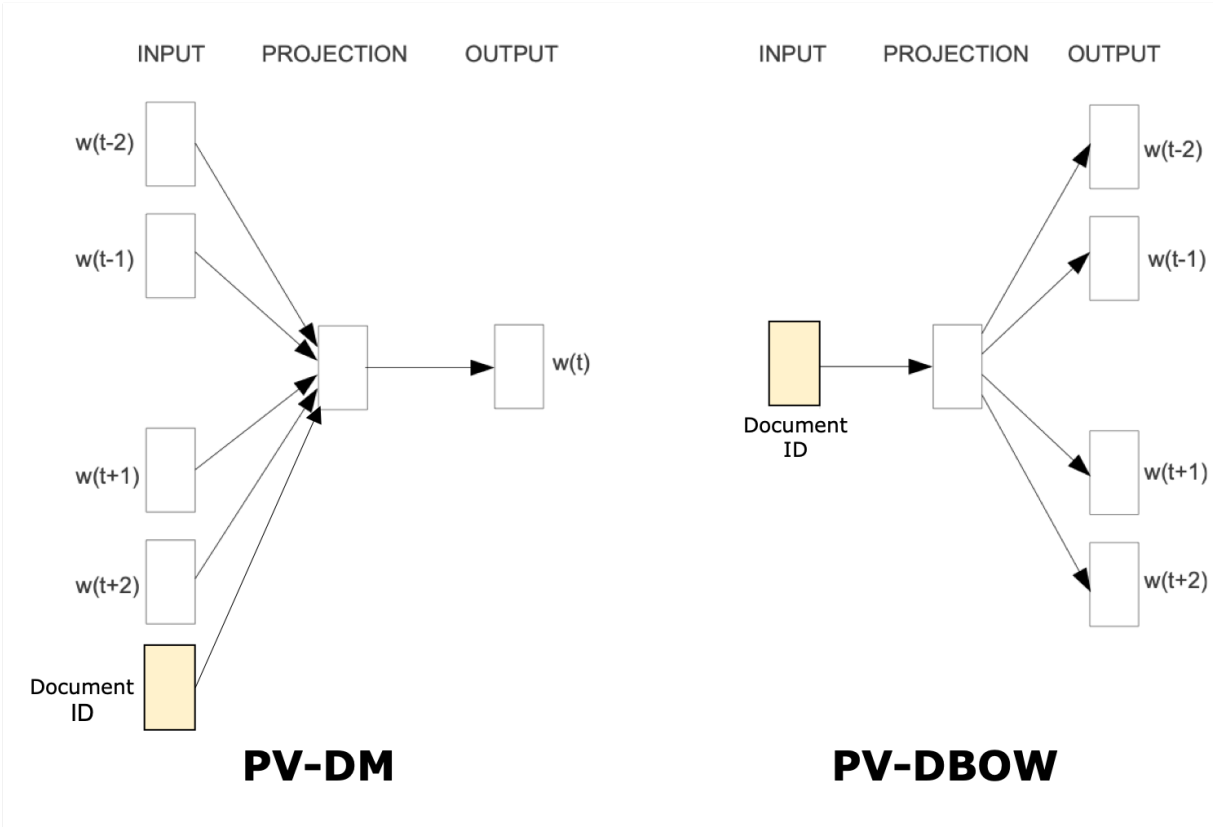


Figure 5.5 – Model architectures of PV-DM and PV-DBOW

Graph2Vec

Graph2Vec was proposed in 2017 by Narayanan *et al* as a method for numerical embedding computation of graphs inspired from Doc2Vec [200]. Graph2Vec was the first numerical embedding to learn the representations of entire graphs, instead of substructures such as nodes, paths and subgraphs.

Figure 5.6 shows the parallel between the models of Doc2Vec and Graph2Vec. In Graph2Vec graphs are considered analogical to documents Doc2Vec, but composed of rooted subgraphs

instead of words. Other graph substructures (e.g. nodes, walks, paths) could be used as atomic entities instead of rooted subgraph, however the authors point two main advantages to choosing rooted subgraphs:

- *higher order substructure*: rooted subgraphs encompass higher order neighborhoods than simpler substructures, therefore they are likely to better reflect the graph composition;
- *non-linear substructure*: rooted subgraphs capture better inherent non-linearity in graphs than linear substructures such as walks and paths.

Therefore, to train the model it is necessary to extract the rooted subgraphs and assign them with a unique label for all the rooted subgraphs in the vocabulary. This is accomplished using the Weisfeiler-Lehman (WL) relabeling strategy [74], typically used for testing graph isomorphism.

The WL relabeling strategy uses a coloring refinement method which assigns an initial color to all the nodes of the subgraph (of degree d defined as parameter) and at each iteration it aggregates a mixture of colors for each node depending on their neighbors. After n iterations (defined as parameter), the feature description for the rooted subgraphs is an ordered vector that counts the number of nodes for each color —if two rooted subgraphs are isomorphic they generate the same feature description. For greater details about Graph2Vec refer to the original paper [200].

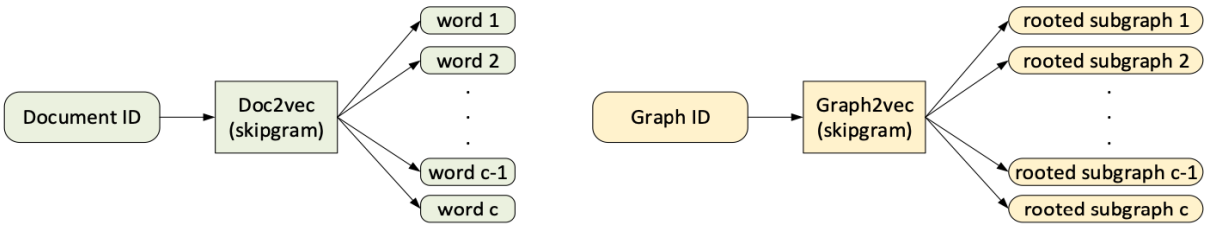


Figure 5.6 – Parallel models of Doc2Vec and Graph2Vec [200]

5.2.3 Clustering Ensembles

The most common approach for the machine learning setting assumes that features are represented in a single vector or graph space. These features are representation (i.e. *views*) of the objects (e.g. binary codes) in different feature spaces. Usually these multiple views are obtained from different vector spaces or different graph spaces or a combination of vector and graph spaces [286].

Unsupervised learning can rely on two strategies to handle data analysis with multiple views: centralized or distributed algorithms [169]. The former consolidates the multiple views into a unique representation before mining hidden patterns in the data. The latter learns hidden patterns individually from each view before learning the optimal hidden patterns of all the views.

Centralized algorithms are much more popular than distributed algorithms, despite the complexity of dealing with multiple representations that could not only be of different forms (e.g. vectors and graphs), but also have very different statistical properties [169]. Distributed algorithms have some advantages over centralized algorithms [169]:

— they do not constrain the individual (unsupervised) learning of the views, treating them as black boxes. This allows the system designer to chose the most appropriate method for each representation;

— they provide greater flexibility to the analysis system, since they do not directly operate on data.

Clustering ensembles (or *clustering aggregation*) is a method for distributed unsupervised learning. It combines multiple clustering models into a single consolidated partition without accessing neither the features nor the algorithms of the underlying partitions [273]. The goal is to produce a consolidated partition that improves the quality of individual clusters.

Cluster ensemble spun off from supervised classifiers that used a combination of different models for learning, after the success of this aggregation method to improve overall learning quality. However, in contrast to supervised learning, unsupervised learning has no clear-cut metric to optimize. Some authors tried to define a set of properties that endorse the use of clustering ensemble methods, but due to the malleable nature of unsupervised clustering no consensus emerged on this matter [291]. Some of these properties include:

- Robustness: The combination process must have better average performance than the single clustering algorithms.
- Consistency: The result of the combination should be somehow very similar to all combined single clustering algorithm results.
- Novelty: Cluster ensembles must allow finding solutions unattainable by single clustering algorithms.
- Stability: Results must have lower sensitivity to noise and outliers.

Vega-Pons and Ruiz-Shulcloper emphasize that it is not possible to claim that any partitioning of the data is (outright) *better* than another —including the *natural* organization (i.e. the ground-truth, if any) and the results obtained by a cluster ensemble [291]. In the case of clustering ensemble, it can only be ensured that the final consolidated cluster is a consensus of all the underlying partitions and thus it is expected that the consolidation process can produce a more reliable partition of data as it compensates for possible errors in the individual clusters.

The basic workflow of clustering ensemble methods consists of two steps: Generation and Consensus (c.f. figure 5.7). In the *generation* phase the set of individual partitions are computed; there are no particular constraints on how these partitions must be obtained —it may involve different algorithms, parametrizations, features and any other combination of those. In the *consensus* phase the individual partitions are consolidated in a single final clustering with the help of a *consensus function*, which is the main object of any clustering ensemble algorithm.

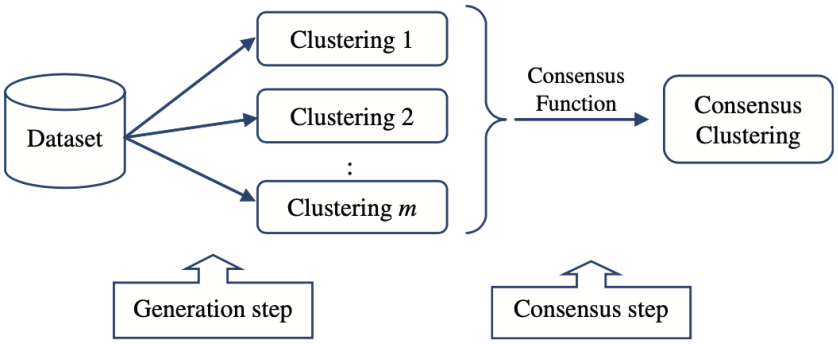


Figure 5.7 – General workflow of cluster ensemble methods [291].

There are two main approaches for designing a consensus function: *objects co-occurrence* and *median partition*. The former considers the frequency that a data point belongs to one cluster or that pairs of data points appear together in a same cluster; the latter targets the partition that maximizes the similarity with all partitions in the cluster, i.e.:

$$P^* = \underset{P \in \mathbb{P}_X}{\arg\max} \sum_{j=1}^{m} \Gamma(P, P_j)$$

where $\mathbb{P} = \{P_1, \cdots, P_m\}$ is the set of partitions (computed in the generation phase), $\mathbb{P}_X$ is the set of all possible partitions of the set of objects X (such that $\mathbb{P} \subset \mathbb{P}_X$), P^* denotes the consensus partition and Γ is a similarity measure between partitions.

Some methods used for designing consensus functions based on objects co-occurrence include *Relabeling and Voting*, *Co-association Matrix*, *Graph and Hypergraph* and several other strategies; consensus functions based on median partition include *Genetic Algorithms*, *Non-negative Matrix Factorization*, *Kernel Methods* and *Mirkin Distance*.

In this chapter we use the following consensus functions: cluster-based similarity partitioning algorithm (CSPA), Hypergraph-Partitioning Algorithm (HGPA), Meta-Clustering Algorithm (MCLA), Hybrid Bipartite Graph Formulation (HBGF), Nonnegative Matrix Factorization (NMF). For a gentle introduction about these consensus functions, refer to survey [291] by Vega-Pons and Ruiz-Shulcloper.

5.3 Related Works

In this chapter, our work explores the coalescence of *hybrid feature*, *numerical embedding* and *ensemble learning* for malware classification. For this reason, this section reviews works on malware analysis that fall in at least one of these areas of interest.

5.3.1 Hybrid Features for Malware Analysis

In malware analysis, features can be represented in many different ways (e.g. vectors having header data, strings, CFGs or ECDGs), therefore the problem of handling multi-view data arises naturally. Normally, features are derived from representative data (i.e. views) obtained from a file analysis process and are subsequently used in a data analysis step in order to fulfill the desired task (e.g. malware detection, malware clustering). These views are in general software birthmarks [4] extracted either through dynamic or static analysis methods. Anderson *et al* refer to these extractors as *data sources* [6].

We call *hybrid features* the representations that entail a composition of data views, in particular when they are obtained through a mix of static and dynamic data sources [68]. These data sources are in general respectively associated with syntactical and semantic features, although it is possible to define features composed of multiple data views obtained from a single data source (e.g. behavior profile [22]) —nonetheless they are not as relevant in our context due to their shortcomings against targeted anti-analysis techniques.

The key aspect of this composition method is the fusion step, where data views extracted from different data sources are composed to derive a (unified) hybrid feature. As many real-

4. Inherent characteristics that can be used to identify particular software [292].

world applications, the most popular approach to obtain multi-views unsupervised learning is through centralized algorithms. This includes approaches like feature concatenation [65, 95, 131, 179, 180, 194, 284], feature hashing [22, 125], Multiple Kernel Learning (MKL) [6], feature composition [84] and metric composition [86, 137, 138, 206].

Feature concatenation is the simplest and most commonly used method for feature composition. It consists in producing a multi-view data where the multiple views collected from different data sources are concatenated to produce a comprehensive description of objects. However, it can be ineffective in practice and can even result in worse clustering results than single-view clustering [328]. If the number of feature dimensions is overwhelmingly big, techniques to shrink this representation have to be applied (e.g. PCA); Dahl *et al* use random projections to map input vectors of hundreds of thousands dimensions into a feature space of few thousand dimensions [65]. Although still providing good results in practice, such features vectors are completely deprived of meaning over the original data views, which hinders their interpretation.

Feature hashing (a.k.a *hashing-trick*) works by applying a hash function to the input data views and using the output value as feature vector [308]. It is a very fast and space-efficient strategy to vectorize features, however (similarly to feature concatenation) the feature vectors fail to keep a correspondence of representation with the original data views.

Multiple Kernel Learning (MKL) explores the fact that kernels can be linearly combined to obtain new flexible ones. This allows to define individual kernels for each data view and combine them into a single consolidated kernel that takes into account all sets of data views simultaneously. Combining models in kernel machine classifiers is thereby based on combining their kernels [162]. MKL is a very flexible method that can extend SVM methods to work with heterogeneous data, but it is not compatible with many other machine learning methods that do not rely on kernels.

Feature composition refers to combination of data views of different data sources that are directly fused into a feature representation. For instance, Eskandari *et al* define a feature which combines the information gathered by static and dynamic analyses into an enhanced version of the CFG, which is in turn analyzed using different classifiers such as decision trees, naive Bayes and random forest [84]. The utility of this approach is the derivation of a feature that “naturally” combines different data views, however it is applicable only for data views that have some level of innate kinship. This approach (alone) cannot provide the flexibility necessary to benefit from very different data views, therefore it is seldom used.

Metric composition defines custom distances that work with multiple data views. This

allows to define pairwise distances that can be used by many different machine learning algorithms. The drawback of this method is its lack of flexibility, since the custom metric is defined to work for the (limited) set of selected metrics of a given analysis scenario; moreover, the topology of the feature space can become counter-intuitive, thus making it harder to interpret the results.

Alternatively, feature composition methods based on distributed algorithms exist, but they are much less popular than their centralized counterparts.

De la Rosa *et al* proposed a two-layered model composed of a meta-model and a set of underlying specialized models, which are ranked with respect to their complexity [69]. Each specialized model is trained with different features and the meta-model is trained to choose the best (and simplest) specialized model in order to predict the final class for a given input. The experiment shows that this strategy is able to speed up the overall classification, however the authors point out that it is still the most complex (and most expensive) classifier that was most frequently able to correctly assign malware into their corresponding families. This outcome could be somehow expected because there is not cooperation between the underlying specialized models to get the final classification. Therefore the meta-model can be primary viewed as a speed-up optimization, but it fails to attain an actual learning composition of the models.

Most commonly, frameworks based on distributed algorithms use standard ensemble learning. Supervised models generally use voting and stacking ensemble, which consists in a two-level model where the first level contains the specialized sub-models and the second level contains a sub-model that produces a prediction based on the result of the first-level sub-models. These are the most frequent distributed methods used (together or not) to combine heterogeneous data views in enhanced malware detection models [15, 111, 121, 149, 252, 317].

As for unsupervised models, Ye *et al* generate the final clustering from the optimal connectivity matrix, which computes the co-association matrix of the samples in each partition and use a minimum threshold (with sample-level constraints) to select the samples of the aggregation clustering [320]. X. Hu and K. Shin proposed the DUEL as a framework for combining static and dynamic features through clustering ensembles [124]; they defined cluster-quality measures and also produced the final clustering based on optimal connectivity matrix using three different algorithms. Zhang *et al* defined a mixture model scheme based on clustering ensemble, which assumes that the data views are modeled as random variables drawn from a probability distribution described as a mixture of multivariate com-

ponent densities [326].

5.3.2 Numerical Embedding for Malware Analysis

The use of numerical embedding for malware analysis is relatively recent. Inspired by concepts from natural language processing, Rieck *et al* were the first to propose the use of numerical embedding for malware analysis in 2011 [235].

They defined a special representation of behavior denoted as *malware instruction set (MIST)*, in which the n-gram decomposition are transformed into numerical vectors. Roughly, the numerical vectors belong to a vector space where each dimension is associated with one n-gram instruction. Based on this numerical embedding, machine learning algorithms are used for behavioral clustering and classification —as it is usual in machine learning with the *two-stage method* (c.f. section 5.2.2, page 195). The results obtained with this method were sound, achieving a high F-score (~ 0.95) and outperforming the then state-of-the-art framework which was based on feature hashing [22]. Nonetheless, setting n-gram instructions as orthogonal dimensions oversimplifies their geometrical representation —e.g. *why (1,0,0) and (0,1,0) would be equidistant to (1,0,0) and (0,0,1)?* —furthermore, this embedding strategy is not easily transposed to other data types that cannot be easily parsed in n-grams (e.g. graphs).

In 2016, Hashemi *et al* proposed a graph embedding method based on *power iteration*. The method targets the graph spectrum, which uses the eigenvalues and eigenvector of the adjacency matrix to deduce useful properties of the graph, in order to compute numerical embedding. In turn, power iteration is a method that given the adjacency matrix of a graph computes a vector that is a linear combination of eigenvectors proportionate to their eigenvalues. As usual, the authors use the *two-stage method* to apply supervised machine learning classifiers (KNN and SVM) for malware detection. The method achieves decent results with datasets of different sizes and the authors emphasize the low false positive ratio obtained.

As of 2018, embedding learning starts to be used in malware analysis. Awad *et al* apply Word2Vec on the opcodes extracted with binary code disassembling to obtain an embedding vector representing each instance [14]. This method was tested on ~ 11K malware instances of 9 different families using kNN-based classification, achieving a high classification scores reaching over 95%. A similar approach was adopted by R. Lu [170], but it uses a LSTM model for malware detection [100].

Other works adopt a similar methodology, using the *two-stage method* after computing the numerical embedding. Jiang *et al* used Node2Graph [109] to compute embedded vec-

tors from function-call graphs in parallel with a boolean model of Windows API calls to input them in an auto-encoder neural network for malware detection [134]. Zhang *et al* used GloVe [215], which trains a global word co-occurrence matrix and produces a word vector space model, to compute the embedded vectors and use them to train a CNN for malware detection [325]. Kale *et al* used Word2Vec on opcode sequences in conjunction with a feature vector obtained from hidden Markov models (referred as HMM2Vec), applying them in several supervised model (SVM, $k-NN$, RF and CNN) to do (supervised) malware classification [139] —S. Paul and M. Stamp followed a very similar approach [213].

5.4 Methodology

5.4.1 EMB-DUET

Our methodology improves the DUET workflow, which combines different data views (from static and dynamic data sources) using ensemble clustering, with the use of numerical embedding learning to normalize data into vectors and speed up the overall workflow. Therefore we refer to our methodology as *EMB-DUET*.

Figure 5.8 shows the general workflow of EMB-DUET. It consists of a multi-phased process in which samples are analyzed using either static and/or dynamic analysis to extract the representative data (i.e. data views), then (i) the numerical embedding vectors of the data views are computed through unsupervised learning, (ii) the individual specialized clustering (i.e. ensembles) for each embedding is computed, and finally (iii) all ensembles are consolidated into a final clustering that take all data views into account.

The embedding computation step (i) is dependent on the type of the data views (e.g. text, graph), however a large set of algorithms exist to handle most different data types. Once the embedding vectors have been computed, different clustering algorithms can be used (in step ii); eventually, this phase can use ensemble clustering to combine different clustering strategies (e.g. hierarchical, density, etc) in the process of ensemble computation.

We present our workflow as a versatile approach for malware clustering based on hybrid features. In the current work, we instantiate the workflow components as follows:

— Data views: n-grams, string and ECDGs

— Clustering: OPTICS and HDBSCAN

— Ensemble Clustering: CSPA, HGPA, MCLA, HBGF and NMF

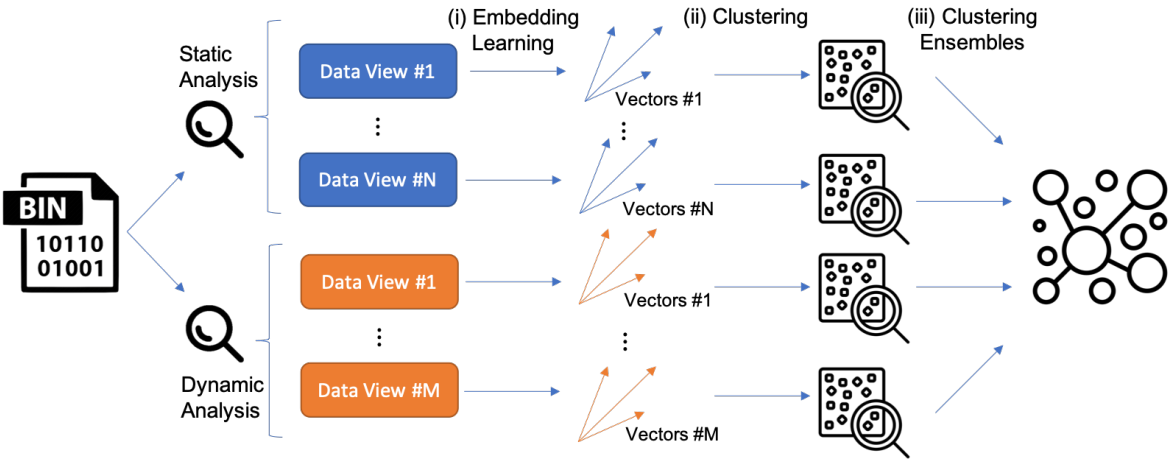


Figure 5.8 – General workflow of EMB-DUET.

Our set of data views are chosen because of their complementary characteristics. n-grams and strings bear syntactic information about the binary files (hence entailing high precision but low recall), whereas ECDG has proven to be a good alternative to represent the semantics of the binary file, providing high accuracy and robustness (c.f. chapter 4, page 153).

In order to improve the quality of ECDGs, we merge the graphs obtained through symbolic execution (c.f. *angr-extractor* in section 3.5.1, page 136) and traditional dynamic analysis using Cuckoo sandbox (detailed later in section 5.5.1, page 212) using disjoint union (c.f. section 3.4.3, page 132); thereby we follow a *feature composition* method for graph-based data view as explained above (c.f. section 5.3.1, page 204). To avoid confusion with the regular version of ECDG, which do not leverage feature composition, we refer to this setting as *enhanced ECDG* (e-ECDG). Figure 5.9 shows the feature composition of the e-ECDGs.

To compute numerical embedding vectors we choose the learning methods according to the data type that most naturally corresponds to the algorithm input data type. Therefore, we use Doc2Vec for n-grams and strings, and Graph2Vec for e-ECDGs.

The clustering workflow follows modular architecture, which is able to work with heterogeneous features and leverages ensemble clustering to consolidate all data views in a single cluster, addresses **RQ5.1** [5].

The choice of complementary features such as n-grams (syntactic), strings (syntactic) and e-ECDGs (structural/semantic) —in addition to the feature composition of the former,

5. **RQ5.1**: *How to design a data-driven clustering that can systematically work with heterogeneous features?*

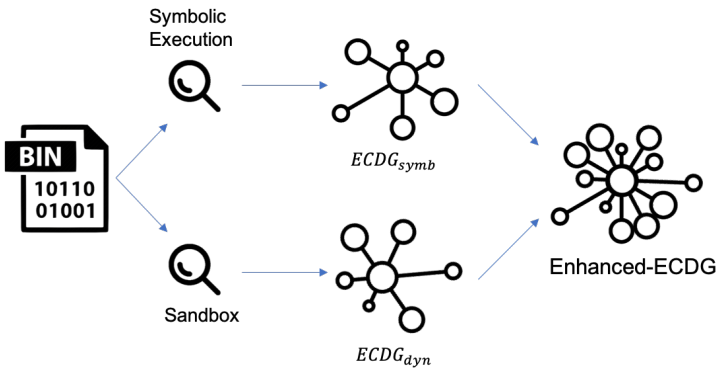


Figure 5.9 – Feature composition of the enhanced-ECDG.

which combines symbolic execution and sandboxing —targets **RQ5.2** [6].

Finally, the embedding layer (in addition to DUET) that learns vector representations for the data views, in order to normalize input data to speed up the overall workflow, addresses **RQ5.3** [7].

5.4.2 σ^{ECDG} -learning with SVM Regression

Using e-ECDG in the EMB-DUET workflow predicates on the transformation of graphs into vectors. Here we benefit from the accuracy and robustness properties previously ascertained on our behavioral graph representation (σ^{ECDG}) (c.f. chapter 4, page 153). For this, to guarantee the same properties for the vectorized instances (i.e. after the learning process), it is necessary to readjust the similarity measure of the embedded numerical vectors of e-ECDGs with respect to σ^{ECDG} .

To this end, we consider the inclusion of one additional learning layer that transposes the σ^{ECDG} -based similarities from graphs onto vectors. More specifically, this additional layer learns a SVM regression model to predict the σ^{ECDG} -similarity given a pair of vectorized e-ECDGs. Figure 5.10 shows the workflow for e-ECDG clustering based numerical embedding and the additional σ^{ECDG} -regression layer.

The σ^{ECDG} -regression model is trained with a set of e-ECDG pairs and the corresponding σ^{ECDG} -similarities. Since training dataset ($\mathcal{T}$) includes tuples that contain pairs of e-ECDGs, it grows quadratically with respect to the number of graphs in $\mathcal{G}$. However, since there is no obligation to train the SVM regression model with all pairs in $\mathcal{G} \times \mathcal{G}$, we can limite $\mathcal{T}$ to K

6. **RQ5.2**: *How to design a data-driven clustering that can systematically work with heterogeneous features?*
7. **RQ5.3**: *How to optimize the data-driven clustering to make it scalable?*

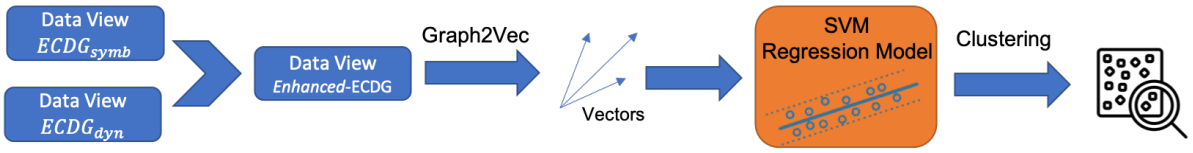


Figure 5.10 – e-ECDG clustering workflow based on numerical embedding with additional σ^{ECDG} -regression layer.

tuples, where K is a value defined as parameter. This asymmetry in the asymptotical curves actually facilitates the composition of $\mathscr{T}$.

More formally, the training data comprises a set of K tuples composed of $(G_i, G_j, \sigma^{ECDG}(G_i, G_j))$, where i, j are the indexes and G_i and G_j are graphs of e-ECDGs randomly chosen from $\mathscr{G} = \{G_1, \cdots, G_N\}$. Thus, we can notate it as $\mathscr{T} = \{(G_{i_1}, G_{j_1}, \sigma^{\text{ECDG}}(G_{i_1}, G_{j_1})), \cdots, (G_{i_K}, G_{j_K}, \sigma^{\text{ECDG}}(G_{i_K}, G_{j_K}))\}$, where $i_1, \cdots, i_K, j_1, \cdots, j_K \in_R \{1, \cdots, N\}$.

Figure 5.11 shows the training process and the use case of the SVM regression model; the elements that participate in the learning process are colored in blue, while those that participate in the use case, which rely on the trained model to predict the σ^{ECDG} -similarity of unknown ECDGs, are colored in orange —the test phase relies on the trained model as in the normal use case.

To guarantee a maximum of diversity of $\mathscr{G}$ in the composition of $\mathscr{T}$, it is possible to use round-robin tournaments generated by the circle method for selecting graphs in $\mathscr{G}$, because this generation method maximizes the carry over effect [158]. Furthermore, since the distribution of σ^{ECDG} values for graphs in $\mathscr{G}$ is not uniform (because it is more likely to find dissimilar ECDGs for a pair of graphs randomly chosen), it may be necessary to generate a superset $S_{\mathscr{T}}$ of tuples before composing $\mathscr{T}$ in accordance with the desired distribution.

Beyond $\mathscr{T}$, the success of the learning process also depends on the hyperparameter setting for the SVM regression. Altogether, the entire parametric setting encompasses a great number of variables that contribute to the regression performance; but it also provides great learning flexibility to this additional transformation layer.

This transformation layer aims to produce predictions of the σ^{ECDG} -similarity —which was demonstrated to be accurate and robust (c.f. chapter 4, page 153) —given the vectorized version of the ECDGs. Although intending to contribute to **RQ5.2** [8], the outcome of σ^{ECDG} -learning with SVM Regression is only ancillary to **TOP-RQ3**: *How to compute the similarity of unknown programs with high accuracy while being friendly to search and clustering algo-*

8. **RQ5.2**: *Which set of features can afford accuracy and robustness to the data-driven clustering?*

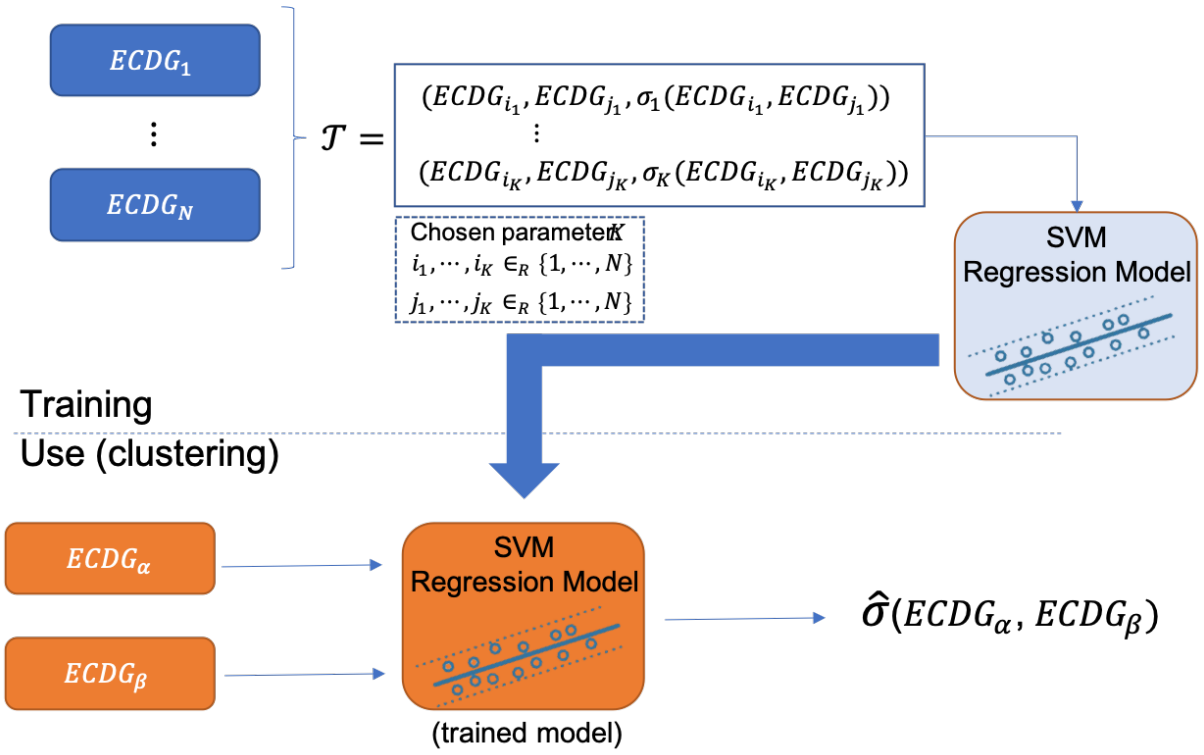


Figure 5.11 – Training process and use case of the SVM regression model.

rithms for malware analysis?

5.5 Experiments

5.5.1 Implementation Setup

ngrams-extractor

To extract the n-grams of files we extended our analysis framework with a proprietary tool written in the Go language [217]. For this, we prepared a docker image with the appropriate Orqal interface (c.f. section 3.5.1, page 138)

Cuckoo

Cuckoo [267] is a tool to automate the analysis of artifacts in the context of a sandbox[9]. Cuckoo was originally started in 2010 as an open source project in a Google Summer of Code

9. Security mechanism isolate the execution of programs (c.f. section 2.3.1).

within the Honeynet Project [10].

Cuckoo has a modular design that endows it with great flexibility and facilitates its integration in larger frameworks. Figure 5.12 shows a basic model of Cuckoo architecture, composed of a front-end (i.e. cuckoo host) and a back-end where the automated malware analysis takes place. Cuckoo host provides interfaces to interact with users as well as other systems and manages the back-end by dispatching new analyses on the sandbox machines and retrieving the generated reports.

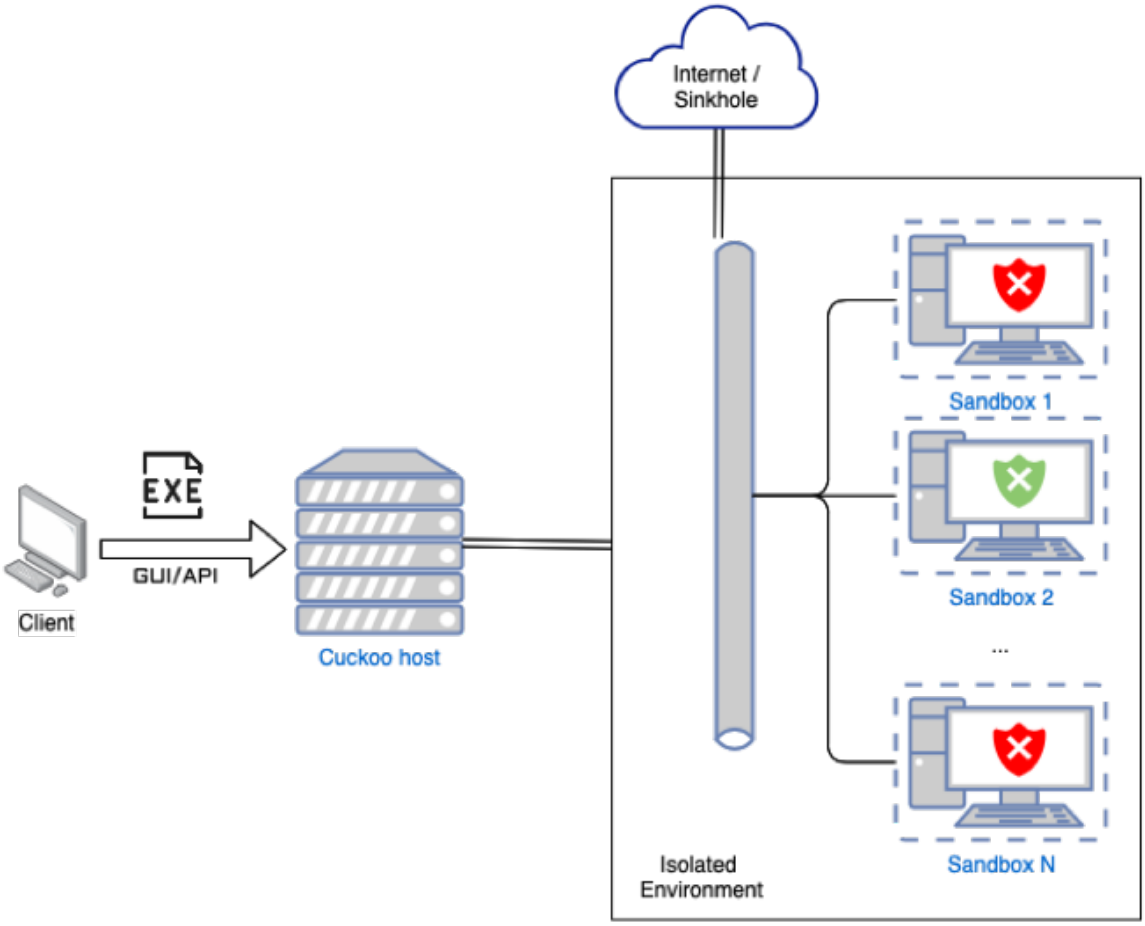


Figure 5.12 – Basic architecture model of Cuckoo.

The most basic deployment of Cuckoo consists of a single machine on which the Cuckoo Host runs as a service and the sandbox environment is deployed with virtual machines.

10. Non-profit security research organization dedicated to investigating the latest attacks and developing open source security tools to improve Internet security. Link: https://www.honeynet.org

Cuckoo has built-in integrations with several virtualization hypervisors, including VirtualBox, KVM, VMware and others. Cuckoo also supports distributed deployment on a cluster of machines, which can contain physical and virtual machines.

Figure 5.13 provides more details about each Cuckoo component. The Cuckoo Host comprises three modules (Core, Web and REST API) and the Sandbox consists of a custom analysis environment whose principal requirement is to have Cuckoo agent.py running —implementing the Cuckoo agent as a Python script enables the Cuckoo sandbox to work on different operating systems (e.g. Windows and Linux).

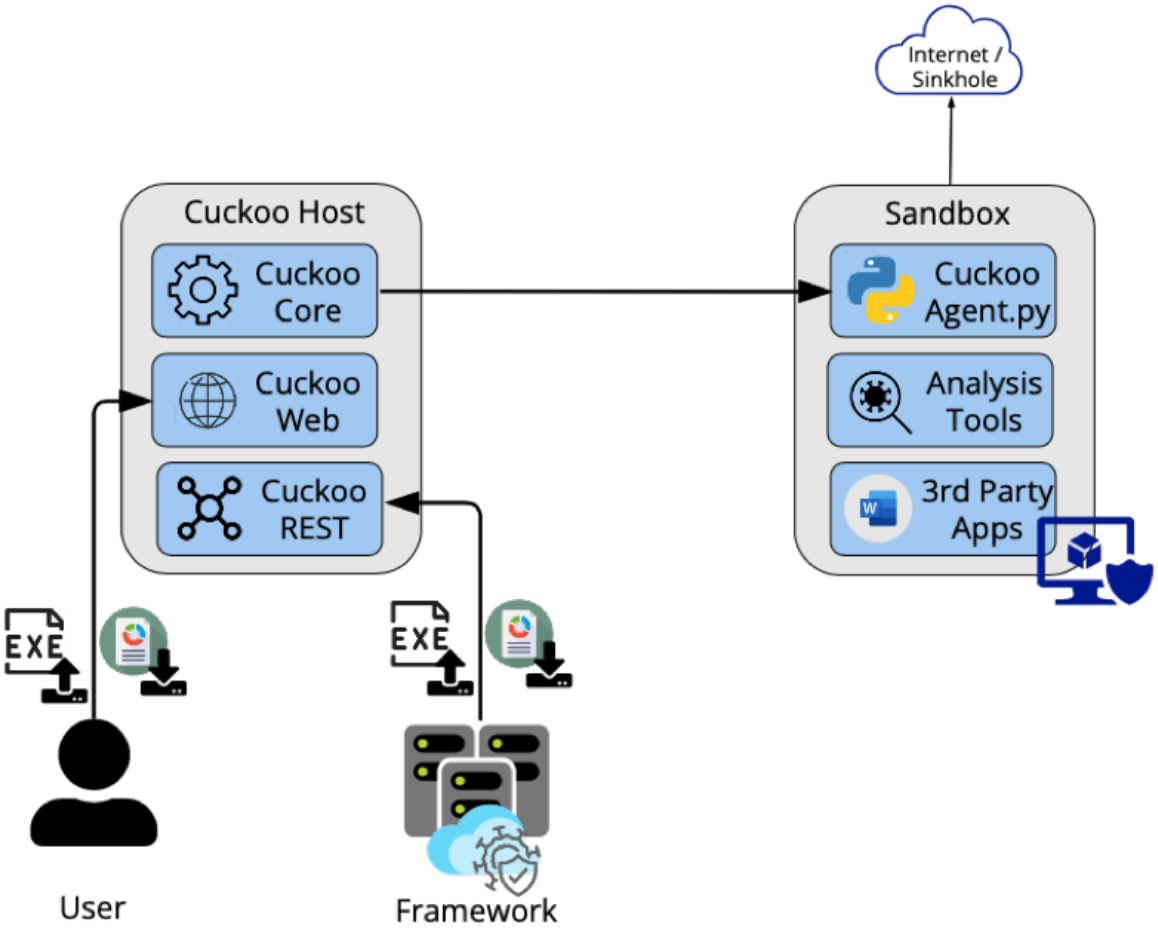


Figure 5.13 – Macroscopic system structure of Cuckoo.

Cuckoo Core is responsible for managing the analysis queue, controlling the task scheduling and launching new analysis tasks; Cuckoo Web provides a GUI that allows users to submit files to be analyzed, check the analysis dashboard and have access to reports of finished

tasks; and Cuckoo REST API provides the same functionalities as Cuckoo Web but through a REST API that is more convenient for integration with other systems.

The Cuckoo agent runs a service that listens to the network awaiting commands (with their respective files and parameters) from the Cuckoo Host to initiate a new analysis. The Cuckoo agent interacts with analysis tools that are activated during the execution of the target file; eventually, it can also interact with 3rd-party applications installed on the sandbox. When the analysis is complete, the Cuckoo agent prepares the final report and makes it ready to be fetched by the Cuckoo Host.

Cuckoo supports many different configurations and is able to integrate many other existing tools. For a complete view refer to Cuckoo official documentation [11].

Our Setup For maximum efficiency and simplicity, we deploy Cuckoo on a single machine having the following specification: 1 processor Intel Xeon W-2155 (10 cores, 13,75 MB de cache, 3,30 GHz), 128GB RAM (DDR4, UDIMM, non-ECC, 2666MHz), with Ubuntu Linux 18.04. A storage of 40TB is available over NFS where the reports are saved and we use MongoDB as database.

In our setup we installed the following modules on Cuckoo: *Pydeep* to compute the ssdeep of the submitted files; *Volatility* to analyze memory dumps obtained in the analysis; *TCPDump* to analyze the data exchanged over network during the file execution.

We integrated our Cuckoo Host with two hypervisors: QEMU-KVM and VirtualBox. We conducted a pre-analysis benchmark to compare both integration setups, which showed almost no difference in the average analysis time (~ 130s on VirtualBox vs. ~ 134s on QEMU-KVM) but better stability with QEMU-KVM. Therefore we decided to use QEMU-KVM as hypervisor to the remainder of the experiments. As for the sandboxes, we generated 30 clones of a 4GB RAM VM with a typical Windows 10 installation and a standard setup of the Cuckoo agent (i.e. without any hardening).

We also implemented a custom reporting module to generate (subset of) the ECDGs with the Windows API calls intercepted during the target file analysis. The standard Cuckoo agent setup relies on procmon [188] (c.f. section 2.3.1, page 88) to monitor filesystem activities, registry and process/thread in real time, including these information in the final report. Despite being able to extract many relevant calls —which is certainly enough for a good analysis —procmon hooks Windows at the kernel level and therefore intercepts only the calls to the Windows API; thus, our ECDG report module for Cuckoo can generate only a (reasonably

11. Link: https://cuckoo.readthedocs.io/en/latest/

good) subset of the entire ECDG.

5.5.2 Dataset

To compose our dataset we took an approach similar to the one of our preceding experiment (c.f. section 4.5.1, page 168). Again we relied on syntactical signatures due to the high precision they typically provided and also due to the stability in reproducing the experiments with other datasets. Thus we reused the set of public Yara rules used before (i.e. Yara-Rules, InQuest and McAfee ATR Team) in section 4.5.1 (page 168).

However, this time we turned to Malpedia [218] to complement our set of Yara rules and to improve the quality of our ground truth. Malpedia is a collaboration platform for curating a malware corpus, which offers "a plethora of possibilities for researchers, including using it as a *testbed for evaluations* on detection and analysis methods, *quality assurance for classification*, and contextualization of new malware". Malpedia keeps a record of curated malware families with their corresponding Yara rules. Therefore, we guided the composition of our dataset with these Malpedia Yara rules, keeping the Yara rule filename [12] as label of the malware family ground truth.

Initially we ran the Yara rules on our entire database, which comprises samples collected in previous projects, samples made available by VirusTotal for academic research and a complete dump of the MalwareBazaar database. We could identify over 100K matched files, many of which matched multiple Yara rules. Table 5.1 shows the number of files per number of Yara rules.

Table 5.1 – Number of files per number of Yara rules

#Yara Rules	1	2	3	4	5	6
#Matched files	53200	28938	4641	1287	16746	635

To simplify the remainder of our analysis we looked for a univocal mapping between samples and Yara rules. Therefore we closely checked the cases of files matching multiple Yara rules and found many redundancies.

To find such matches and combine them into a unique label, we defined the *matching graph* as a graph whose nodes represent Yara rules and edges connect Yara rules that match concomitantly at least one file. By constructing this matching graph from our set of matched

12. Henceforth we use the terms *Yara rule filename* and *Yara rule* for the sake of brevity, although technically a Yara rule file could include multiple Yara rules.

files, we found 223 isolated nodes (37.54% of the total of nodes), i.e. the files matched only a single Yara rule.

The isolated nodes correspond to our case of interest (i.e. univocal mapping) and therefore they do not need to undergo any merging strategy. Thus, for the remainder of this graph analysis, we promptly remove these nodes.

Figure 5.14 shows the matching graph (after removal of isolated nodes). We can see that besides the very dense component in the center, all other connected components are rather small. Therefore, we decided to merge them in a single label.

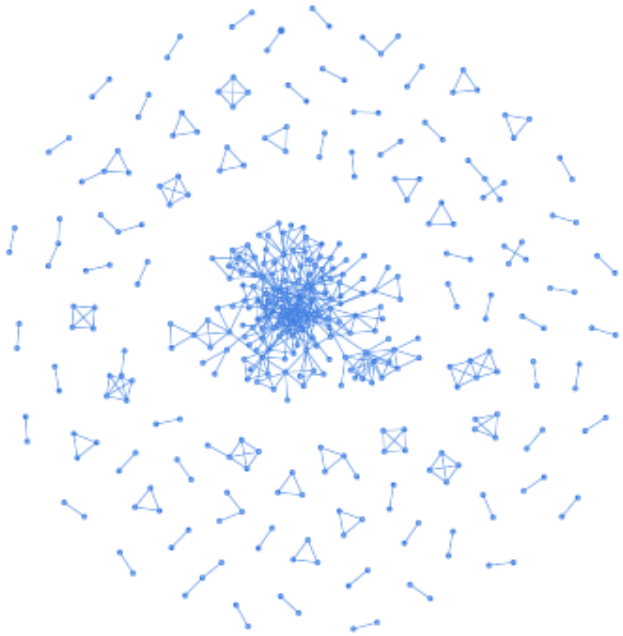


Figure 5.14 – Matching graph (after removal of isolated nodes).

For most cases the Yara rules were clearly related (e.g. `win.globeimposter_g0.yar` and `win.globeimposter_auto.yar` Yara rules). However, this merging strategy provided some surprising results. For instance, we found a connected component that included only the Yara rules for *Neutrino* [13] and *QakBot* [14], which at first sight are two very distinct threats —our final family label for such cases is composed with both families, e.g. *qakbot+neutrino*.

This merging procedure allowed us to obtain a total of 13,065 files —referred as *annotated files* or *annotated dataset* —uniquely matched with 308 Yara rules (including merging).

13. Link: https://malpedia.caad.fkie.fraunhofer.de/details/win.neutrino
14. Link: https://malpedia.caad.fkie.fraunhofer.de/details/win.qakbot

Intriguingly, 282 of these files were initially in our cleanware database, from which 168 of the cleanware files were matched with Yara rules associated with the *nymaim* [15] label. To enable the use of the AnR paradigm (c.f. section 4.4.3, page 165), our dataset was extended with non-annotated files from our cleanware database to obtain a total of 27K files [16].

5.5.3 σ^{ECDG} -Learning with SVM Regression

This section presents the experimental analysis of σ^{ECDG} -Learning with SVM Regression. Since our goal with this analysis is to readjust the similarity measure of numerical embedding vectors of e-ECDGs with respect to σ^{ECDG} as previously analyzed (c.f. section 4, page 153), we reused the former dataset (c.f. section 4.5.1, page 168) for this part of our experimental analysis.

More specifically, we sought the set of hyperparameters for Graph2Vec and the SVM Regression that elicit the best possible learning results. For this, we carried out a *random search optimization* approach [25] to explore the parametrical space as much as possible. Random search is ideal because of its asynchronicity, preemption and predisposition to cover a larger span of the hyperparameter space with less trials.

We also assessed the effects of different training and test size ratios, aiming to investigate the differences in the asymptotical curves for different sizes of the training and test datasets with respect to the total number of graphs. Our target was to find the sweet spot ratio that provides very good learning results for the least training cost.

Table 5.2 shows the parameter grids randomly explored in our analysis. We computed a total of 6,202 trial tuples consisting of $(\mathscr{P}_{emb}, \mathscr{P}_{SVM}, \mathfrak{s}_{\text{train}}, \mathfrak{s}_{\text{test}})$, where $\mathscr{P}_{emb}$ and $\mathscr{P}_{SVM}$ denote the instances of parameter sampling for Graph2Vec and SVM Regression, and $\mathfrak{s}_{\text{train}}$ and $\mathfrak{s}_{\text{test}}$ the sizes of the dataset sampling for the learning and test phases —we fixed the random seed for this sampling in order to avoid a selection bias when comparing different trials.

We defined our loss function as the mean absolute error (MAE) of the predictions, i.e.:

$$\text{MAE} = \frac{\sum_{i=1}^{\mathfrak{s}_{\text{test}}} |\hat{\sigma} - \sigma|}{\mathfrak{s}_{\text{test}}} \tag{5.5}$$

where σ is the true value of the similarity function computed directly with the graphs (c.f.

15. Link: https://malpedia.caad.fkie.fraunhofer.de/details/win.nymaim
16. More precisely, a total of 27,127 files, which corresponds approximately to the same number of annotated and non-annotated files.

Table 5.2 – Random Search: Parameter Grids for Graph2Vec and SVM Regression

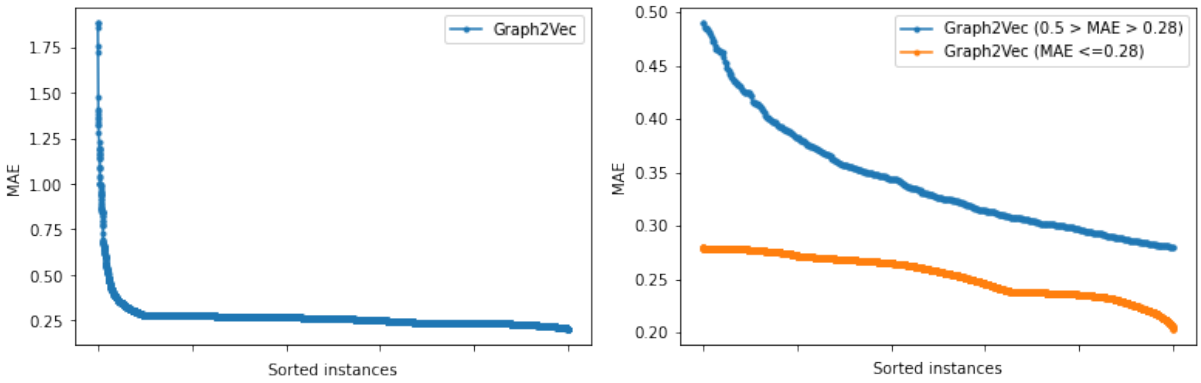
		Parameter	Range	Step
Graph2Vec		WL-iterations	$[1,128]$	$2^x, 1 \leq x \leq 7$
		Dimensions	$[128,1192]$	128
		Epochs	$[10,100]$	5
		Learning Rates	$[0.001,0.250]$	0.001
SVM Regression	Kernel: RBF	C	10^x	$x \in \{-5,-4.9,\cdots,4.9,5\}$
		γ	{scale, auto}	
	Kernel: Poly	C	10^x	$x \in \{-5,-4.9,\cdots,4.9,5\}$
		degree	$[3,20]$	1
Dataset	Training	size	$[1000,11000]$	500
	Test	size	$\{1000,5000,10000\}$	

section 4.4.2, page 164) and $\hat{\sigma}$ is the corresponding prediction provided by the SVM Regression model. Equation 5.5 implicitly assumes that the test dataset is composed of $\{(g_{1,1},g_{1,2}),\cdots,(g_{s_{\text{test}},1},g_{s}$ where both σ_i and $\hat{\sigma}_i$ are computed with $(g_{i,1},g_{i,2})$.

To help visualizing the training results for the 6,202 tuples, we elaborated a curve plot that we refer as *trend plot*; it takes the set of scores values for a given evaluation group, sorts them and presents them in a xy-plot where score values are projected on the y-axis and points are equally spaced on the x-axis (i.e. the distance between points on the x-axis is scaled with respect to the total number of points). This plot does not show the performance of individual data inputs due to the transformations in the x-axis, but it gives a quick coarse overview of overall performance.

Figure 5.15a shows the curve of MAE values corresponding to the entire set of trial tuples; the curve is composed of two segments: a steeply decreasing segment on the left side (from MAE= 1.88 up to MAE≈ 0.25), and a flat segment around MAE≈ 0.25.

Figure 5.15b shows the trend curve for MAE below 0.5 to identify the transition of the two aforementioned segments; we visually determined the threshold of MAE= 0.28 as the pivoting point of the transition —hence the splitting of the trend curve in the latter plot. The set of values with MAE≤ 0.28 comprises a total of 5,592 trial tuples, which correspond to 90.16% of the total number of tuples —the identification of this threshold is important to help setting a maximum cut-off value in the color scale of graphs whenever necessary.



(a) Entire set of trial tuples.

(b) Trial tuples with MAE < 0.5.

Figure 5.15 – Trend plot of MAE for the total set of trial tuples.

Data cuts

The learning phase aims to reduce total prediction error according to MAE (equation 5.5), which is affected by each factor randomly sampled for the trial, i.e. $(\mathscr{P}_{emb}, \mathscr{P}_{SVM}, \mathfrak{s}_{\text{train}}, \mathfrak{s}_{\text{train}})$. Therefore, for this analysis we cut the sampled data into subsets to inspect their effects on the MAE obtained. The data cuts that we considered are:

— *Embedding Hyperparametrization*: the sets of hyperparameters used in the embedding learning.

— *SVM-Kernel*: kernels used in the SVM Regression.

— *Training-Test Ratio (TTR)*: ratios between the sizes of training and test datasets (computed as $\mathfrak{s}_{\text{train}}/\mathfrak{s}_{\text{test}}$).

The sizes for the *Embedding Hyperparametrization*, *SVM-Kernel* and *Training-Test Ratio* are respectively 122, 2 (i.e. either RBF or Polynomial) and 50. We emphasize that the subsets of a given data cut are mutually exclusive, but the intersection of subsets from different data cuts are not necessarily empty.

Effects of Training-Test Ratio Figure 5.16 shows the heatmap of average MAE value with respect to the *Embedding Hyperparametrization* and *TTR* data cuts; each row in the heatmap corresponds to a unique set of hyperparameters used during the embedding learning, while each column corresponds to a *TTR* value; the red lines delineate the frames that include all embedding hyperparameter sets for each *wl-iterations*.

It is not possible to identify any noticeable pattern —besides a darker region in the center, however the darker values are also present all over the entire range of *TTR* values —we therefore conclude that the *TTR* are not a dominant factor in the SVM Regression for the chosen ranges of $\mathfrak{s}_{\text{train}}$ and $\mathfrak{s}_{\text{test}}$ in our random search.

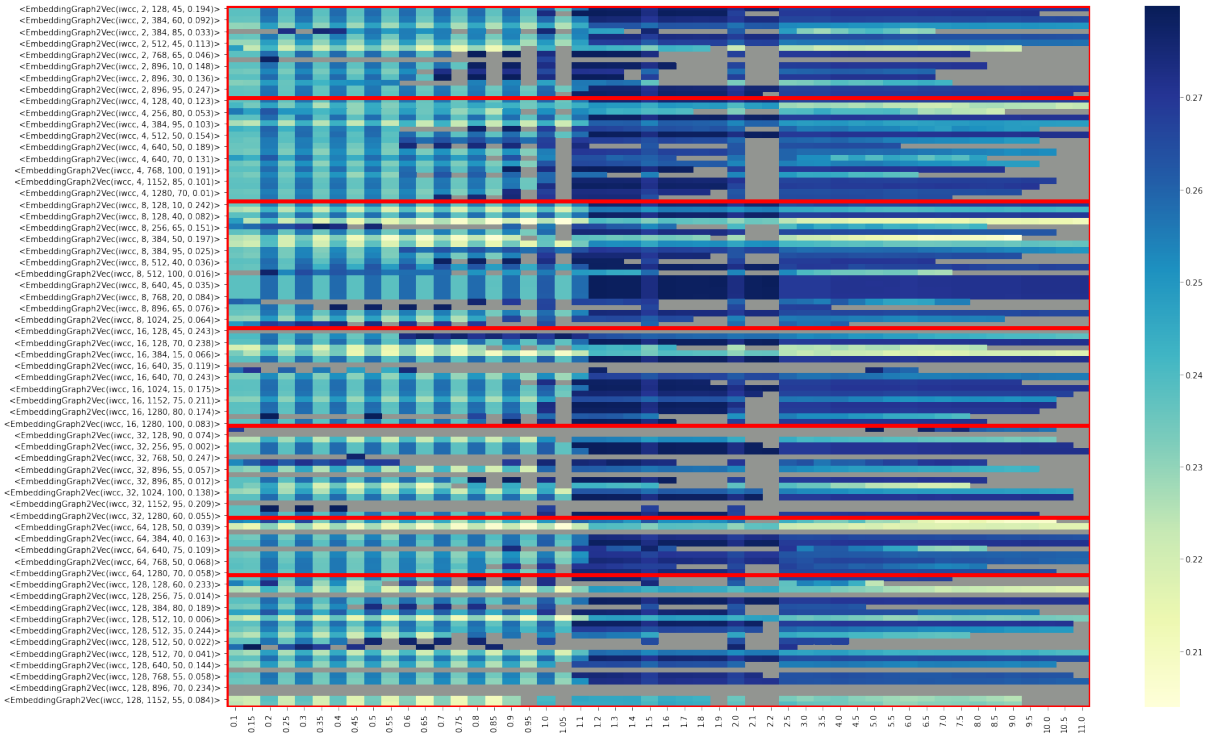


Figure 5.16 – Heatmap of average MAE with respect to the *Embedding Hyperparametrization* and *TTR* data cuts (MAE≤ 0.28).

Effects of SVM Kernel Table 5.3 shows the results of MAE per SVM kernel. We note that the average MAE for the complete set of trials for each SVM kernel deviates considerably, and that the portion of trials removed by the threshold is considerably higher for the polynomial kernel. As for the MAE below the threshold, figure 5.17 shows that RBF kernel attain in general a lower MAE than the Polynomial kernel.

Figure 5.18 shows the average MAE obtained with the polynomial kernel for different degrees. No noticeable difference appear for the different degrees; the values with higher average MAE are due to outliers, hence the bigger standard deviations for these cases. Therefore, in the remainder of the analysis we disregard the possibility of the scores for the polynomial kernel being largely impacted by the setting of the degree parameter.

Table 5.3 – Random Search: SVM Kernels

Kernel	No Threshold		Below threshold (MAE = 0.28)		
	#	Average MAE	#	% (No Threshold)	Average MAE
RBF	3307	0.260 (±0.488)	3106	93.92%	0.252 (±0.021)
Poly	2895	0.287 (±0.136)	2486	85.87%	0.256 (±0.017)

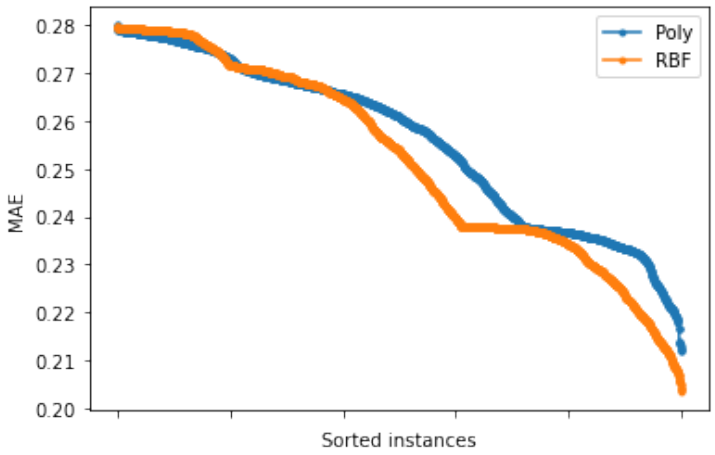


Figure 5.17 – Trend plot of MAE (below threshold) for each SVM Kernel.

To confirm the occurrence of a statistical difference between the results of both SVM kernels, we perform a two-sample independent T-Test considering the two scenarios (with and without threshold). Table 5.4 shows the value of the Percent Point Function (PPF) (for quantile equal 0.001 and freedom degree equal #RBF tuples+#Polynomial tuples−2) and the values for t-statistic and p-value (with and without mean trimming [17]) . In all scenarios the t-statistic values are considerably beyond the theoretical value from the t-distribution and the p-values are remarkably small, which confirms the statistical difference of MAE values obtained with each SVM Kernel in favor of RBF.

Table 5.4 – Random Search: Independent T-Tests of SVM Kernels

Scenario	PPF	t-statistic		p-value	
	(quantile=0.001)	no trim	trim=0.15	no trim	trim=0.15
No Threshold	−3.721	−10.075	−8.82	$1.47\ 10^{-23}$	$1.58\ 10^{-18}$
Threshold (MAE ≤ 0.28)	−3.092	−7.492	−5.642	$7.824\ 10^{-14}$	$1.805\ 10^{-8}$

17. The trimmed test corresponds to Yuen's t-test, which is recommended for long-tailed distributions or distributions contaminated with outliers.

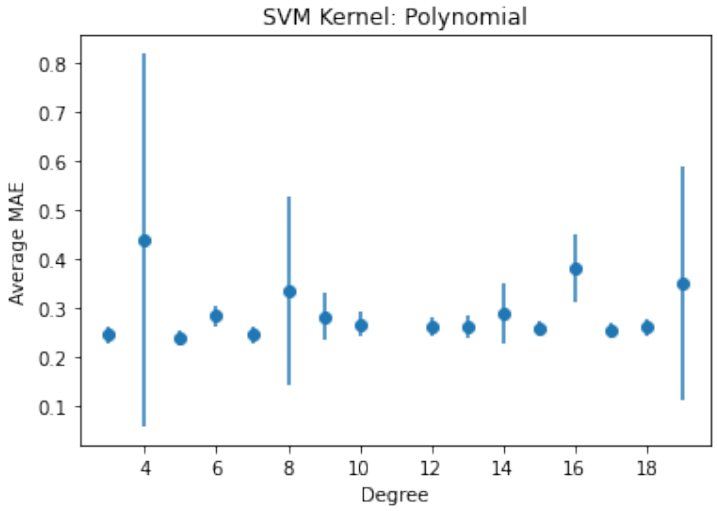
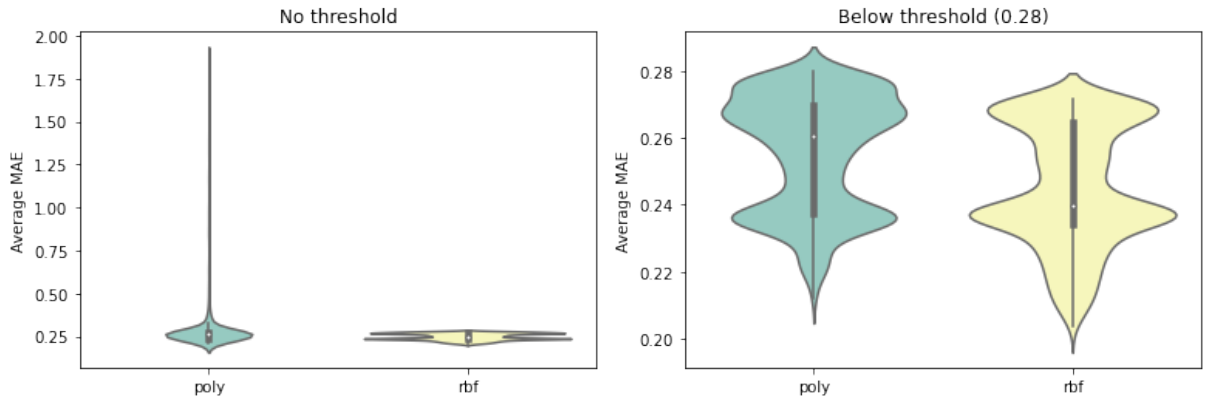


Figure 5.18 – Average MAE for polynomial kernel with different degrees.

Figure 5.19 shows two violin plots that corroborate the statistical difference of both kernels in favor of RBF (when the threshold is considered or not). Figure 5.19a shows the distributions for the scenario without threshold; we notice that Polynomial kernel generates outliers resulting in a long-tailed distribution, whereas the BRF kernel does not suffer from the same deformity. Figure 5.19b shows the distributions for the scenario with threshold equal 0.28. We see that both kernels have a larger concentration of results around Average MAE≈ 0.27 and another around Average MAE≈ 0.24; however the RBF kernel one is larger around Average MAE≈ 0.24, whereas the Polynomial kernel one is larger around Average MAE≈ 0.27. Furthermore, the BRF median is far below the Polynomial mean and the lower quartile for the RBF is smaller than the upper quartile, while in the case of the Polynomial Kernel it is the opposite.

Graph2Vec and SVM Regression with RBF Kernel Since the RBF Kernel outperforms the Polynomial Kernel, we further inspected the hyperparametrization for this case. The hyperparameters of the SVM Regression for the RBF Kernel are kernel coefficient (γ) and the regularization parameter C; the regularization parameter C determines the trade-off between the flatness of the maximum-margin hyperplane and the total deviation tolerated (c.f. section 5.2.1, page 191). The kernel coefficient (γ) for the BRF kernel (i.e. $k(x_i, x_j) = e^{-\gamma||x_i - x_j||^2}$) defines the reach of influence of $x_i \in X$ for the training set $X = \{x_0, \cdots, x_{s_{\text{train}}}\}$, which can be set to *scale* and *auto* as follows:



(a) Scenario without threshold.

(b) Scenario with threshold (MAE ≤ 0.28).

Figure 5.19 – Violin plots showing the distribution of average MAE for the Polynomial and RBF Kernels.

$$\gamma = \frac{1}{\mathfrak{s}_{\text{train}} \cdot std(X)} \qquad \text{(scale)}$$

$$\gamma = \frac{1}{\mathfrak{s}_{\text{train}}} \qquad \text{(auto)}$$

Figure 5.21a shows the heatmap of the average MAE for the embedding learning versus the kernel coefficient: the rows correspond to each hyperparameter setting of the embedding learning and the columns to the setting of the kernel coefficient (i.e. *scale* of *auto*). Only the values below the MAE threshold (= 0.28) are shown in order not to disturb the color scale, and the red horizontal lines separate each set of *wl-iterations* in the embedding hyperparameter. Although not being definitely conclusive, figure 5.21a shows that the best average MAE are obtained with the setting as *auto* for γ.

Figure 5.21b shows the heatmap of the average MAE for the embedding learning versus the regularization parameter C: the rows correspond to each hyperparameter setting of the embedding learning and the columns to the different values for C. We notice that the best average MAE are obtained with intermediary values for C (around $1 < C < 20$).

Finally, we analyzed the impact of size for the training dataset. Figure 5.20 shows the trend plot for different dataset sizes —other sizes were not included in the plot to avoid cluttering the graph. The best MAE scores are attained with the larger training dataset; in particular, there is a great difference in the minimum MAE scores obtained with training sizes equal 1000 and 10000, i.e. respectively 0.2192 and 0.2052.

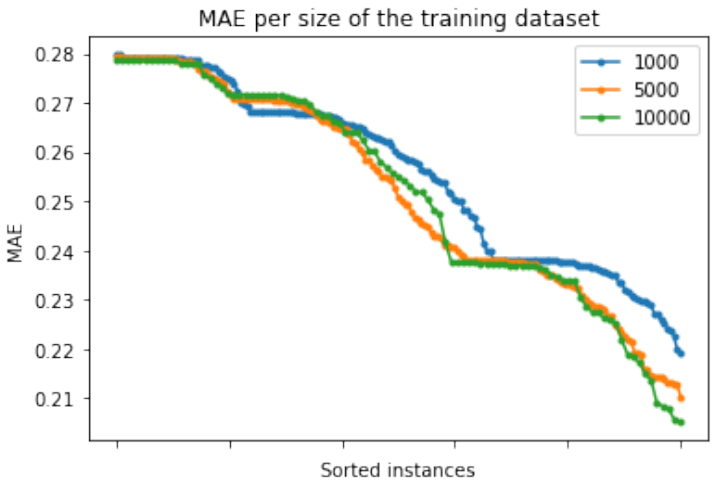
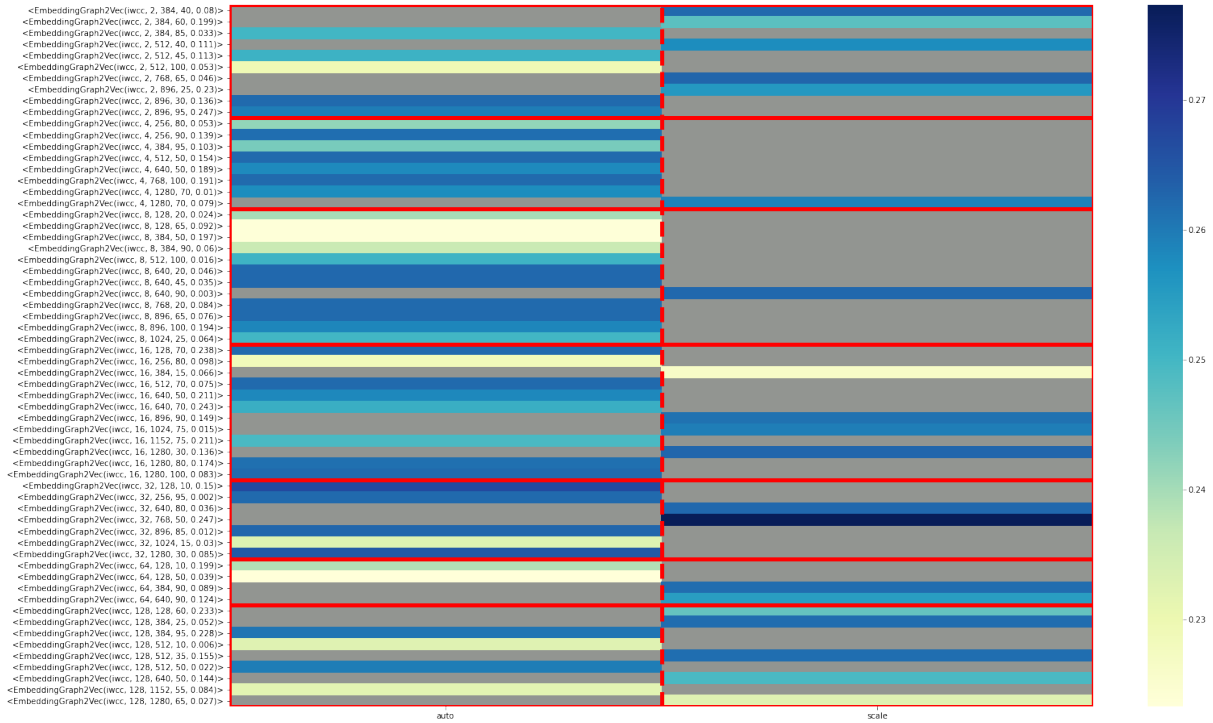
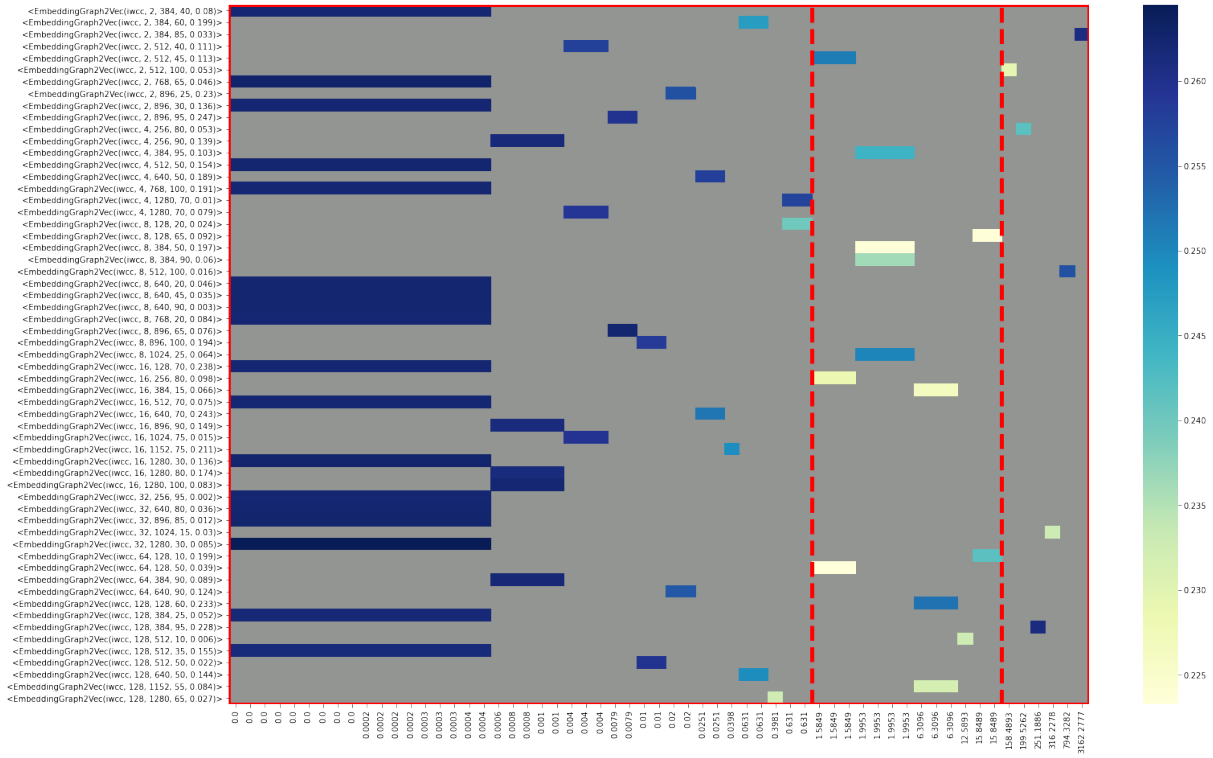


Figure 5.20 – Trend plot of MAE curves of tuples trained with RBF kernel and different training dataset sizes.



(a) SVM Setting for γ.



(b) SVM Regularization Parameter C.

Figure 5.21 – Heatmap of average MAE with respect to the hyperparameter settings for the embedding learning and the SVM Regression.

5.5.4 EMB-DUET

Evaluating EMB-DUET is challenging because of the potential combinatorial explosion of parameters when the underlying data views are taken into account (e.g. parameters for the symbolic execution as studied in chapter 3, page 111). Therefore, we focused mainly on the hyperparameters for the numerical embedding learning and clusterings to understand how they affect the final ensemble clustering.

Initially, we assessed the individual clusterings so as to identify the best strategy to select the set of hyperparameters [steps (i) and (ii) of figure 5.8], then we picked the most promising clusterings to evaluate the information gain attained with different ensemble clustering algorithms.

Generating the Data Views

Generating the data views involves the analysis of files for the extraction of "raw data", which are subsequently used to compute the numerical embedding vectors. To avoid the parametric combinatorial explosion, we set the extraction parameters used to build the data views with basis on our background expertise.

In our experiments we considered three data views: `e-ECDGs`, `strings` and `n-grams`. For the generation of the `e-ECDGs`, we obtained the call traces of the experimental dataset (c.f. section 5.5.2, page 216) using *angr-extractor* (c.f. section 3.5.1, page 136) and our custom reporting module for Cuckoo (c.f. section 5.5.1, page 212). Since the Cuckoo report already includes the strings of the analyzed file, we use them to compute the numerical embedding vectors for the strings. We obtained the n-grams from our ngrams-extractor (c.f. section 5.5.1, page 212).

We decided not to use σ^{ECDG} -Learning with SVM Regression for this experiment to avoid any error propagation, because the MAE attained is still non negligible. Therefore, the remainder of this evaluation relies on the graph similarity obtained directly with Graph2Vec, instead of σ^{ECDG} as studied in chapter 4 (pager 153).

Table 5.5 shows the set of the chosen parameters and the data extraction results. For different reasons some analyses could not provide any data after completion, therefore we do not consider them in the measurement of the average analysis runtime (for the e-ECDG generation). We also excluded the analyses that generated graphs without edges.

Table 5.5 – Data Views: Parameters and Results

Data View		Parameters	# Results	Average Time
e-ECDGs	angr-extractor	max active paths: 8 max mem: 10GB step timeout: 8s symb loop threshold: 4 global timeout: 1h	8577	$62.54s(\pm 309.26s)$
	Cuckoo	route: internet timeout: 1h	5754	$1104.75s(\pm 159.23s)$
strings		*	27004	*
n-grams		$n \in \{5,8\}$	27127	-

Exploring hyperparameters for Numerical Embedding Learning

Since we did not have any *a priori* knowledge about how the hyperparameters interplay, we took once again the *random search optimization* approach [25] to explore our space of hyperparameters.

Tables 5.6 shows the grid of hyperparameters created for the experiment. Each trial randomly selected the hyperparameters within the defined ranges, wherein the values were dispersed in intervals of fixed sizes (annotated in field *step* in the table). In the case of e-ECDGs, we consider two different step sizes: big step and small step; the former intends to cover a large span of the hyperparameter space, whereas the latter aims to catch the magnitude of effects occurred due to small changes in the hyperparameters.

Table 5.6 – Random Search: Parameter Grids for Graph2Vec and Doc2Vec

		WL-iterations	Dimensions	Epochs	Learning Rates
e-ECDGs	Range	2^x	[128, 1192]	[10, 100]	[0.01, 1.00]
	Big Step	$1 \leq x \leq 7$	128	5	0.01
	Small Step	1	5	1	0.005
strings	Range	-	[128, 1192]	[10, 50]	[0.001, 0.250]
	Step		128	5	0.001
n-grams	Range	-	[128, 1192]	[10, 50]	[0.001, 0.250]
	Step		128	5	0.001

We computed for several weeks the instances of the embedding vectors of the data views with different hyperpameters. Table 5.7 shows the summary information of these computations. For the string and ngrams data views, since the learning time was dominated by the

vocabulary building step (prior to the Skip-gram learning), we computed the vocabulary just once and kept it in cache.

Table 5.7 – Data Views: Numerical Embedding Learning

Data View	Algorithm	Vocab. size	# Trial	Average Time
`e-ECDGs`	Graph2Vec	-	723	1683.13s (±2203.01s)
`strings`	Doc2Vec	260101	1621	573.34s (±246.82s)
`n-grams-5`	Doc2Vec	10263870	37	371096.38s (±130969.74s)
`n-grams-8`		10680473	150	414628.36s (±172604.98s)

To analyze the substantial standard deviation in the runtime as well as to deepen the understanding of the impact of each hyperparameter, we plotted a scatter graph of the hyperparameters and runtime for the embedding learning of all data views (c.f. figure 5.22). For this, we rescale the range of each hyperparameter on the y-axis and mark in different colors the coordinates corresponding to (runtime, hyperparameter); therefore, for a vertical cut in the x-axis (i.e. runtime), the plot includes n points, where n is the number of hyperparameter.

The plots 5.22b and 5.22c clearly show a linear runtime growth with respect to the chosen number of *epochs* in the training, while the other factors are scattered all over the plot. We computed the linear regressions for strings and ngrams, getting the equations below (without the bias coefficient):

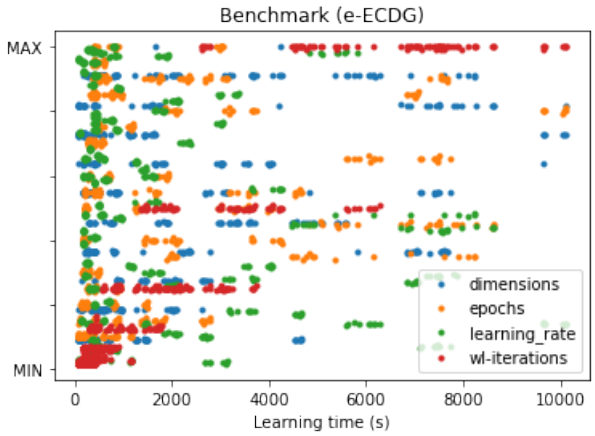
$$\hat{t}_{learn} = 4.18 + 18.73 \cdot \mathfrak{e} + 0.00 \cdot \mathfrak{d} + 15.62 \cdot \mathfrak{lr} \approx 4.18 + 18.73 \cdot \mathfrak{e} \quad \text{(strings)}$$

$$\hat{t}_{learn} = 7400.47 + 12831.94 \cdot \mathfrak{e} - 6.73 \cdot \mathfrak{d} - 45812.87 \cdot \mathfrak{lr} \approx 7400.47 + 12831.94 \cdot \mathfrak{e} \quad \text{(ngrams)}$$

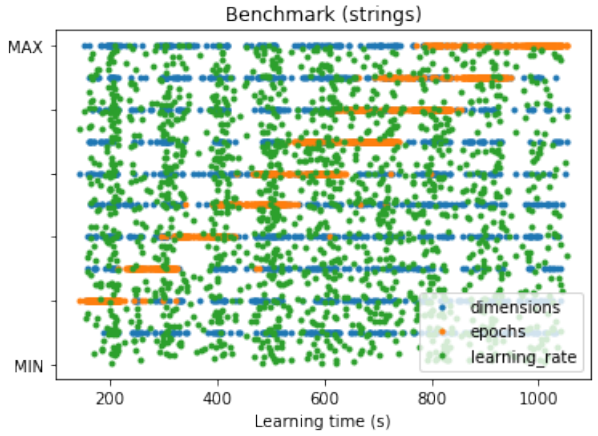
where $\mathfrak{e}$, $\mathfrak{d}$, $\mathfrak{lr}$ and $\hat{t}_{learn}$ denotes respectively the number of *epochs*, the embedding dimension, the learning rate and the prediction of learning time. Note that the coefficients for the dimensions (i.e. $\mathfrak{d}$) are negligible with respect to the others (i.e. $\mathfrak{e}$ and $\mathfrak{lr}$) and that $\mathfrak{e} \approx 10^4 \cdot \mathfrak{lr}$, therefore the runtime is roughly linear with respect to the number of *epochs*.

The coefficient of determination (R^2) for both curves are respectively 0.957 and 0.965, which means that the linear multivariate regressions fit the runtime measurements very closely. When the multivariate polynomial regression of degree 2 (or more) for the curves are computed the coefficient of determination R^2 deteriorates and leading coefficients [18] are

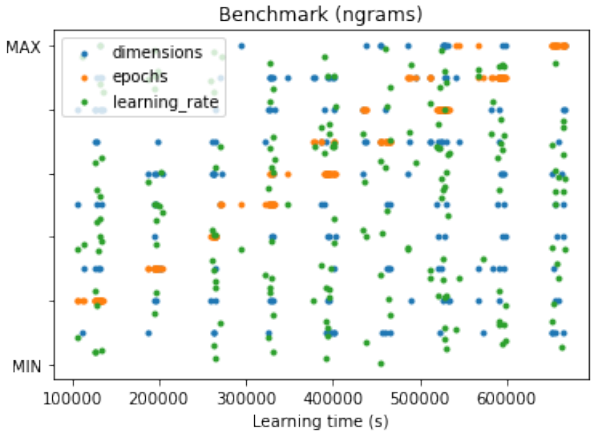
18. Coefficients of the highest monomial orders.



(a) e-ECDG



(b) strings



(c) ngrams

Figure 5.22 – Scatter plots of hyperparameters and runtime for the embedding learning

close to zero. For instance, for degree 2 the leading coefficients for the strings are {0.02, 0.00, −1.05, 0.00, −0.07, 49.03} and for the ngrams are {26.64, 0.07, −1403.36, 0.03, 104.82, 384687.62}, where the last coefficient multiplies the $\mathfrak{r}^2$ which is itself a very small number.

As for the embedding learning of the e-ECDGs, plot 5.22a shows that there is a concurrent effect of *wl-iterations* and *epochs* —where both contribute to increasing total learning time. The multivariate linear regression of learning time of e-ECDGs with respect to the hyperparameters (whose $R^2 = 0.867$) is:

$$\hat{t}_{learn} = -1059.62 + 14.56 \cdot \mathfrak{e} + 50.66 \cdot \mathfrak{wl} + 0.90 \cdot \mathfrak{d} - 588.58 \cdot \mathfrak{r} \approx -1059.618 + 14.56 \cdot \mathfrak{e} + 50.66 \cdot \mathfrak{wl}$$

where $\mathfrak{wl}$ denotes the number of WL-iterations in the Graph2Vec algorithm. The multivariate polynomial regression of degree 2 improves R^2 to 0.980 —higher degrees do not provide much better R^2 —thanks to the coefficient that combines $\mathfrak{e}$ and $\mathfrak{wl}$, while the other coefficients are negligible. Therefore, the multivariate polynomial regression of degree 2 is roughly:

$$\hat{t}_{learn} = -169.26 + 5.49 \cdot \mathfrak{e} + 26.69 \cdot \mathfrak{wl} + 0.44 \cdot \mathfrak{e} \cdot \mathfrak{wl}$$

which results in $R^2 = 0.911$.

Clustering of the Embedded Data Views

To avoid tuning additional clustering parameters, we chose to evaluate the clustering of the embedded data views only with OPTICS and HDBSCAN, as they do not require any mandatory parameter. Nonetheless, they allow to set the minimum number of points in a cluster, which we arbitrarily set to 5.

Once again, we followed the *random search optimization* approach [25] to explore the space of possible clusterings spanned by the 2531 embedded vectors of the previously computed data views. Table 5.8 shows the clustering computations benchmark, grouping results with respect to dataset composition —*annotated* takes the intersection of elements of the data view with respect to files matching a Yara rule (c.f. section 5.5.2, page 216) —and the clustering algorithm.

We note that clusters computed with HDBSCAN take much less time than OPTICS to complete, attaining an overall speedup factor of 31,1. Furthermore, clustering time is proportional to the number of vectors (equal to the number of samples) in the data views, which results in speedup factors (of HDBSCAN over OPTICS) of 12,61, 52,76 and 42,64 for the annotated dataset and 9,81, 28,71 and 25,78 for the full dataset.

Internal and External Evaluation To evaluate learning performance, we computed several scores of internal and external evaluation [19] for all clusterings. The external evaluation scores include: *silhouette score*, *Calinski-Harabasz index* and *Davies-Bouldin index*. The internal evaluation takes the annotated dataset (i.e. samples matched with our set of selected Yara rules) as ground truth and computes the following scores: *rand score*, *AMI*, *NMI*, *homogeneity score*, *completeness score*, *v-measure score* and *Fowlkes-Mallows score*.

19. For a background on metrics for internal and external evaluation, refer to section 2.2.3 (page 82) and section 2.2.1 (page 60).

Table 5.8 – Data Views: Clusterings Benchmark

	Algorithm	# Clusterings	Dataset	# Vectors	Average Time
`e-ECDGs`	HDBSCAN	154	annotated	7660	81.83s (±43.30s)
		66	full	11557	170.09s (±94.82s)
	OPTICS	21	annotated	7660	1032.39s (±47.37s)
		26	full	11557	1668.73s (±122.68s)
`strings`	HDBSCAN	304	annotated	13047	31.66s (±4.08s)
		145	full	27004	134.03s (±11.94s)
	OPTICS	36	annotated	13047	1670.40s (±17.92s)
		51	full	27004	3848.42s (±52.80s)
`n-grams`	HDBSCAN	37	annotated	13062	39.42s (±3.75s)
		17	full	27127	150.01s (±7.99s)
	OPTICS	8	annotated	13062	1680.83s (±18.41s)
		9	full	27127	3867.91s (±38.17s)

The score computations resulted in 8740 data points, which we grouped with respect to the dataset used in the clustering (i.e. *annotated* or *full*), the data view (i.e. `e-ECDGs`, `strings` and `n-grams`) and the clustering algorithm. To help visualizing scores we generated trend plots (c.f. section 5.5.3, page 218) using the data points corresponding to each evaluation group. Since the order of the points on the x-axis depends on the score value (projected on the y-axis), they vary from curve to curve. As before, these plots do not show the performance of each hyperparameter setting (due to the transformations for the x-axis), however they offer a quick and coarse overview of overall performance for each curve and compare the performances of clusterings for different data views and algorithms.

Overall Internal Validation Scores: Figure 5.23 contains all plots showing the score curves for all evaluation groups according to each internal validation metric. The plots show that:

— for any choice of dataset group (i.e. *annotated* or *full*) and data view, HBDSCAN *always* provides higher scores than OPTICS;

— the curves for annotate-`e-ECDGs` always contain the highest scoring data point;

— in general, `e-ECDGs` provides better performance than those for `strings` and `n-grams`, specially with respect to the Calinski-Harabasz and the Davies-Bouldin indexes and in the case of clustering of the full dataset;

Overall External Validation Scores: Figure 5.24 contains the main plots showing the score curves for all evaluation groups according to each external validation metric. The plots show that:

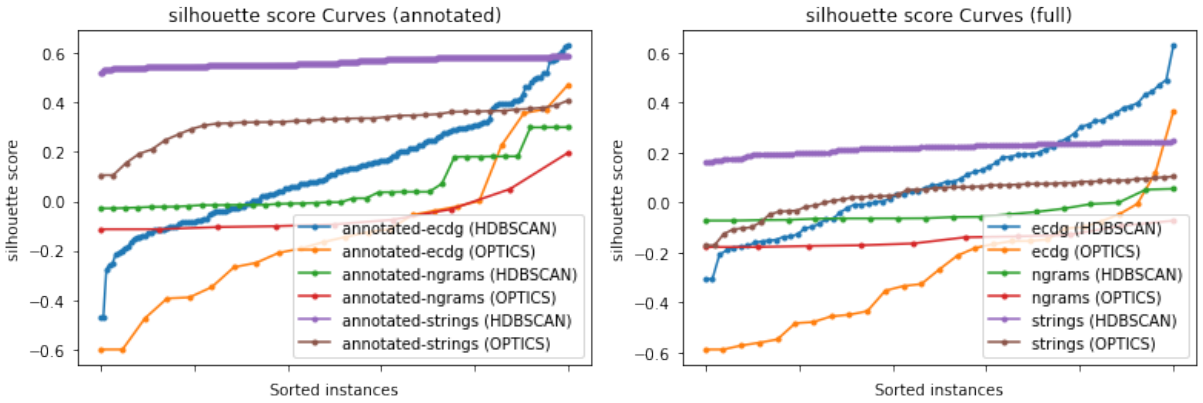
— for any choice of dataset group (i.e. *annotated* or *full*) and data view, HBDSCAN almost always provides higher scores than OPTICS —there are few exceptions, notably for lower scores of the `e-ECDGs` data view;

— generally, `strings`-based clusterings attain the best AMI score and `n-grams`-based clusterings perform considerably better than `e-ECDGs`-based clusterings;

— `n-grams` data views generate the clusterings with highest homogeneity scores;

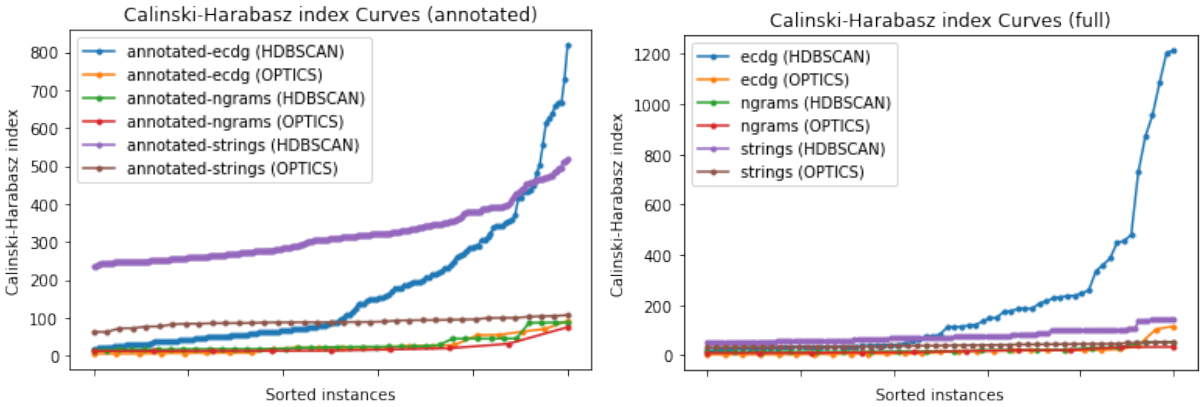
— `strings` data views generate the clusterings with highest completeness scores;

— `strings` data views clustered with HDBSCAN attain Fowlkes-Mallows scores remarkably higher than any scoring group;



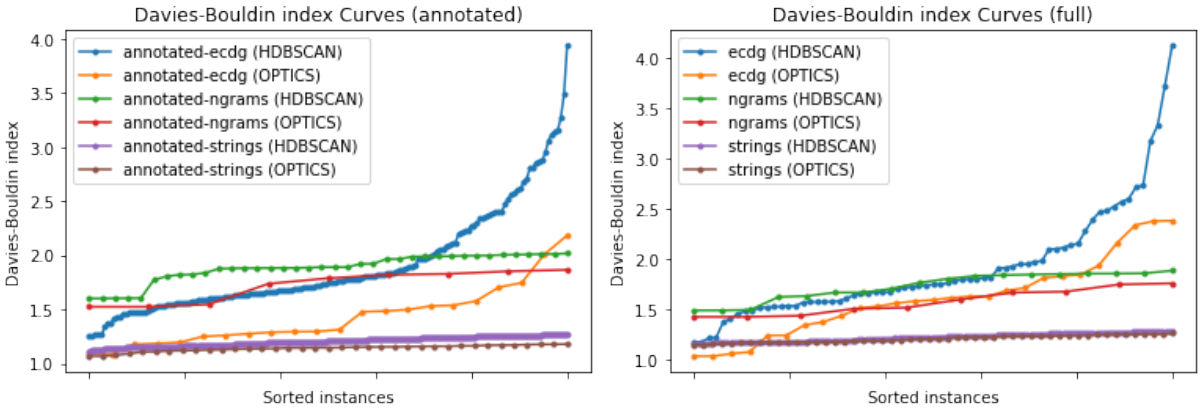
(a) Silhouette score (annotated dataset)

(b) Silhouette score (full dataset)



(c) Calinski-Harabasz index (annotated dataset)

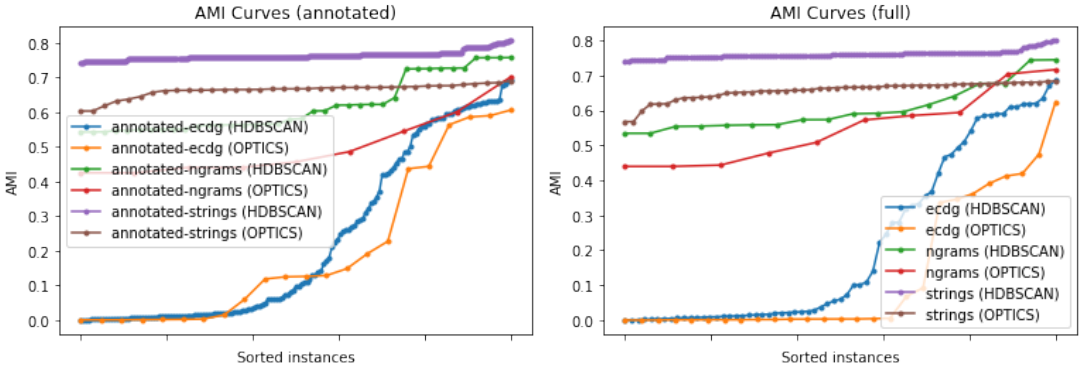
(d) Calinski-Harabasz index (full dataset)



(e) Davies-Bouldin index (annotated dataset)

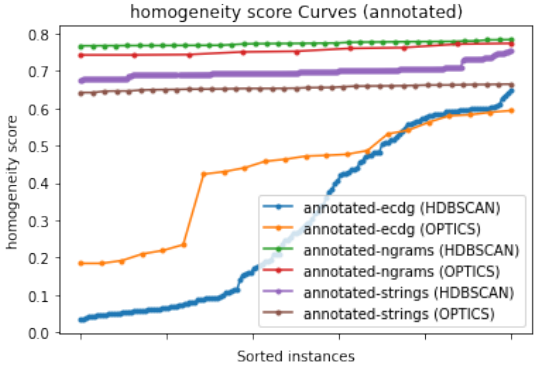
(f) Davies-Bouldin index (full dataset)

Figure 5.23 – Plots of score curves for the evaluation groups for each internal validation metric.

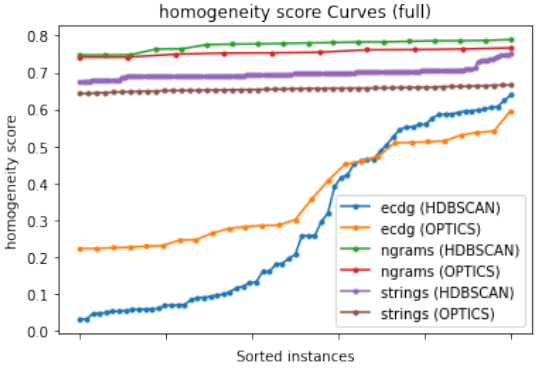


(a) AMI (annotated)

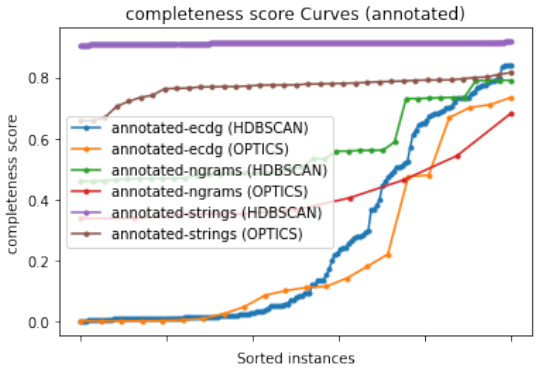
(b) AMI (full)



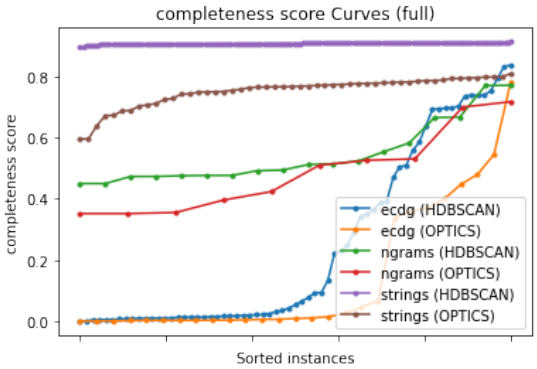
(c) Homogeneity score (annotated)



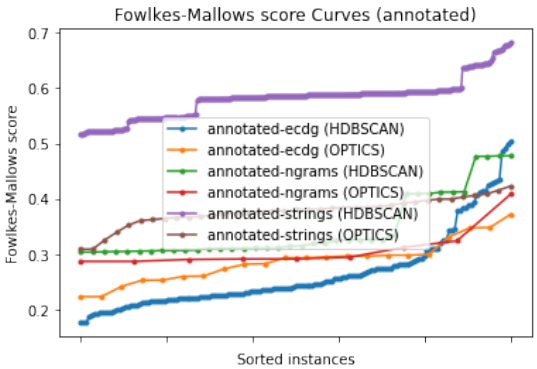
(d) Homogeneity score (full)



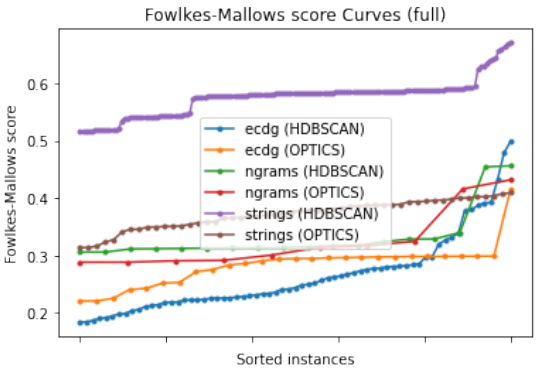
(e) Completeness score (annotated)



(f) Completeness score (full)



(g) Fowlkes-Mallows score (annotated)



(h) Fowlkes-Mallows score (full)

Figure 5.24 – Plots of score curves for the evaluation groups for some external validation metrics.

Accuracy and Robustness (AnR) We reuse the scores measured with the annotated part and the full dataset to perform a partial AnR analysis (c.f. section 4.4.3, page 165) —a detailed AnR would require to gradually insert noise in the evaluation dataset in order to gauge the impact of these changes on the chosen evaluation metrics.

For the accuracy phase, the most significant scores are the AMI and the homogeneity scores. Figure 5.25 shows a scatter plot having the data points of the different HDBSCAN clusterings with respect to their AMI and homogeneity score [20]. This figure shows that parametric settings that promote the improvement of AMI for the clusterings based on `e-ECDGs` also improve their homogeneity score; a similar effect happens with the clusterings based on `strings`, but the values for AMI and homogeneity scores are higher than those of the previous case (and their variation range is much more compact); in the case of `n-grams`, the measurements of AMI and homogeneity score are scattered within a compact region of high values.

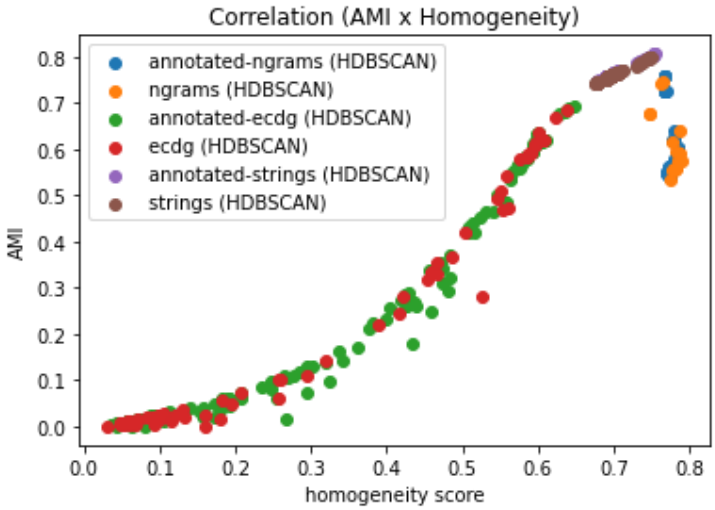


Figure 5.25 – Correlation plot (AMI x Homogeneity score)

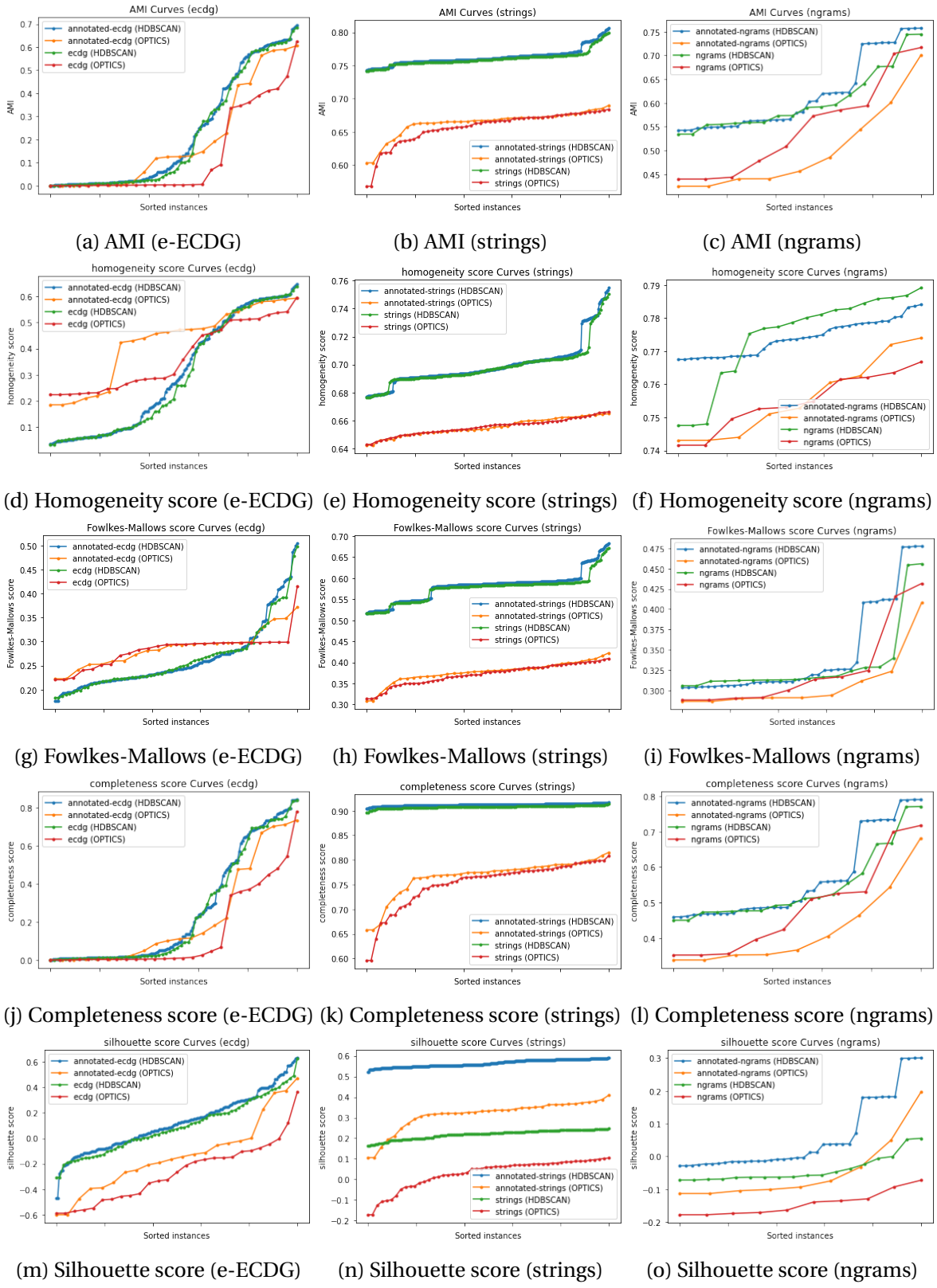
We note that as a rule clusterings based on `strings` provide a fairly high AMI (~ 0.80), while clusterings based on `n-grams` data views provide a fairly high homogeneity score (~ 0.78). Table 5.9 shows the hyperparameter settings that attained the best top 5 performances for AMI and homogeneity scores. As for clusterings based on `e-ECDGs`, they often provide lower AMI and homogeneity scores, but for the best settings these values can attain neat scores, i.e. ~ 0.7 and ~ 0.6, respectively (c.f. figures 5.26a and 5.26d).

20. The scatter plot of the OPTICS clustering displays a very different profile. For the sake of completeness, we include this plot in appendix figure 6.30.

Table 5.9 – Top-5 Accuracy scores: Embedding & Clustering settings

Score Name	Data View	Algorithm	Dataset	Embedding			Score Value
				dim	epochs	learn. rate	
AMI	strings	HDBSCAN	annotated	1152	10	0.074	0.806
	strings	HDBSCAN	annotated	1280	10	0.024	0.806
	strings	HDBSCAN	annotated	1152	10	0.023	0.804
	strings	HDBSCAN	annotated	896	10	0.009	0.804
	strings	HDBSCAN	annotated	1280	10	0.048	0.804
Homogeneity	ngrams-8	HDBSCAN	full	1280	35	0.158	0.789
	n-grams-8	HDBSCAN	full	384	20	0.059	0.787
	n-grams-5	HDBSCAN	full	1280	25	0.089	0.786
	n-grams-5	HDBSCAN	full	640	25	0.17	0.786
	n-grams-8	HDBSCAN	full	384	35	0.201	0.785

For the robustness phase, in addition to AMI and homogeneity score, we also consider the Fowlkes-Mallows index (as it is an index derived from recall and precision), completeness score and silhouette score. All curves (for all data views) are displayed in figure 5.26. We note that scores for the full dataset clusterings follow considerably closely the scores of the annotated dataset clusterings, in particular when HDBSCAN is used. Therefore we can claim that the set of embedded vectors for the chosen data views affords high robustness to the analysis.



(a) AMI (e-ECDG) (b) AMI (strings) (c) AMI (ngrams)

(d) Homogeneity score (e-ECDG) (e) Homogeneity score (strings) (f) Homogeneity score (ngrams)

(g) Fowlkes-Mallows (e-ECDG) (h) Fowlkes-Mallows (strings) (i) Fowlkes-Mallows (ngrams)

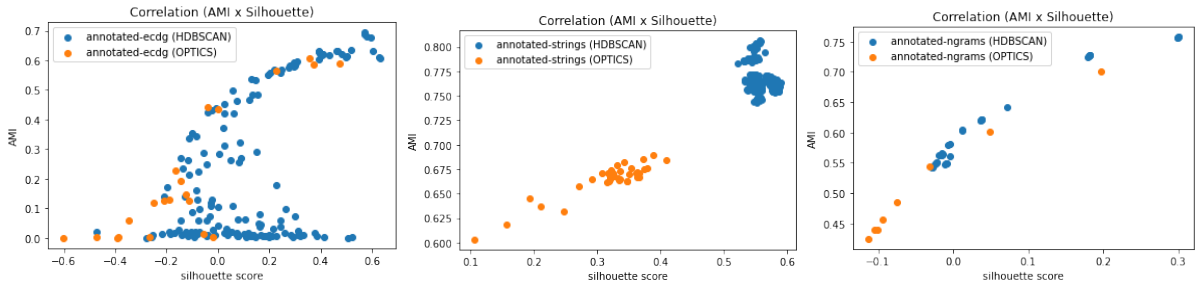
(j) Completeness score (e-ECDG) (k) Completeness score (strings) (l) Completeness score (ngrams)

(m) Silhouette score (e-ECDG) (n) Silhouette score (strings) (o) Silhouette score (ngrams)

Figure 5.26 – Plots of score curves for the AnR of evaluation groups wrt. some validation metrics.

Score Correlations We investigated the following question: *What can we infer about the external validation scores having access only to interval validation scores?* For this, we analyzed the correlations of the scores obtained with each clustering (taking into consideration only the annotated dataset). In particular, we considered the correlation between silhouette score (an internal validation score) and AMI (an external validation score).

Figure 5.27 shows the scatter plot having the data points with respect to their AMI and silhouette score in order to visualize the correlation of theses scores. Figures 5.27b and 5.27c display relation between AMI and silhouette score showing a approximate (positive) linearity —the Pearson correlation coefficient of AMI and silhouette scores for the `strings` is 0.887 (being 0.921 for OPTICS clusterings and −0.291 for HDBSCAN clusterings, however the latter is restrained to a very compact range of values), while for `n-grams` the value of the Pearson correlation coefficient is 0.964.



(a) correlation (e-ECDG) (b) correlation (strings) (c) correlation (ngrams)

Figure 5.27 – Correlation plots with AMI x Silhouette Score for all data views.

As for `e-ECDGs`, at first glance there is no patent correlation between AMI and silhouette score (c.f. figure 5.27a). However, after scrutinizing the hyperparameter settings that provided the best AMI *and* silhouette score, we find out that the *learning rate* is an important factor to improve the positive correlation of AMI and silhouette score.

Figure 5.29 shows the correlation plot of the `e-ECDGs`-based clusterings split by different values of learning rate. We plotted in figure 5.28 the curves with the values of the Pearson correlation coefficient (for AMI and silhouette score) considering only clusterings with learning rate smaller than the threshold; in our experiment, the optimal overall threshold for the learning rate is around 0.23 (Pearson correlation coefficient = 0.928).

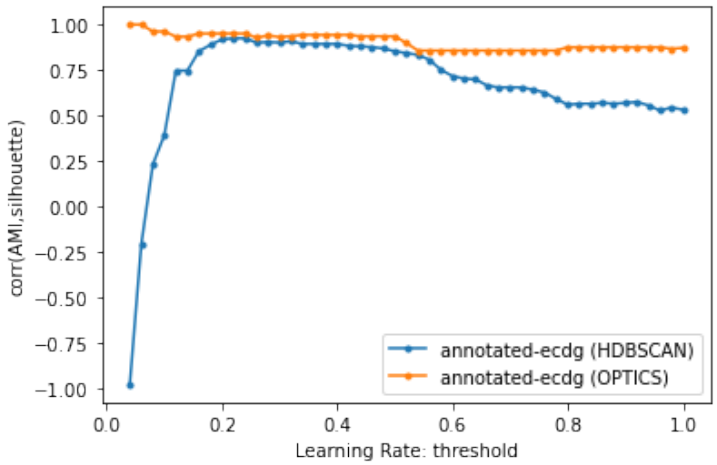
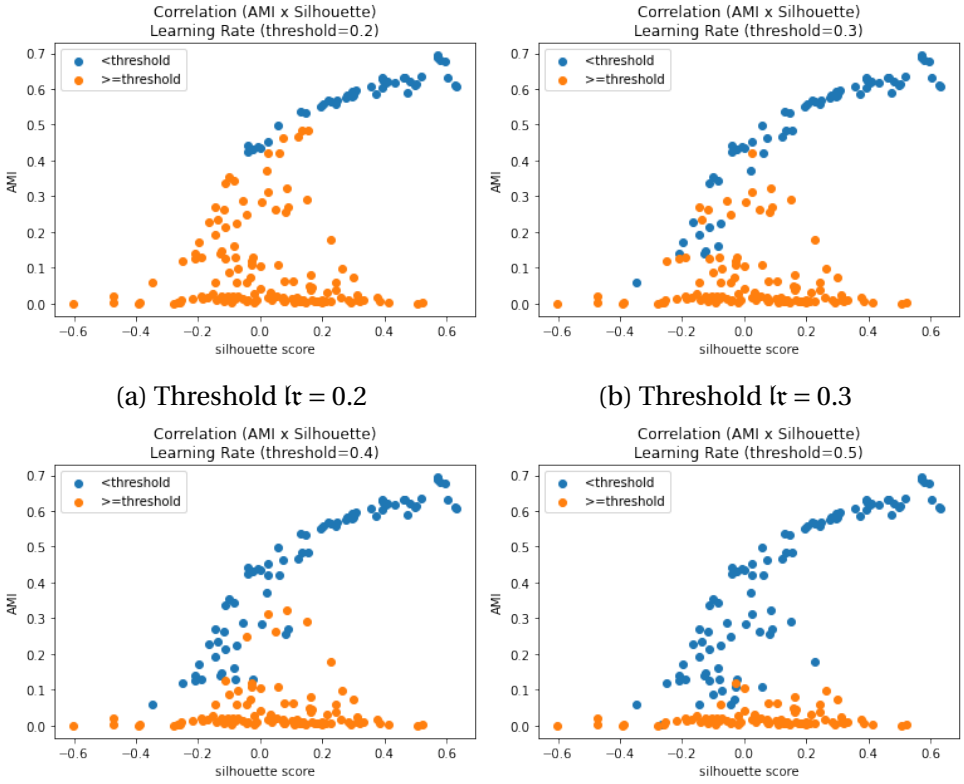


Figure 5.28 – Curves of Pearson correlation coefficients (AMI x Homogeneity score).



(a) Threshold ƚ𝔯 = 0.2

(b) Threshold ƚ𝔯 = 0.3

(c) Threshold ƚ𝔯 = 0.4

(d) Threshold ƚ𝔯 = 0.5

Figure 5.29 – Correlation plots with AMI x Silhouette Score for the ECDG data view split by learning rate (ƚ𝔯).

Mutual Information of the Data Views To understand the information shared between data views, we computed AMI for all possible pairs of clusterings generated from different data views (only for the annotated dataset). However, in contrast to the AMI computations made heretofore, here we do not use the Yara labels of annotated dataset as ground truth; instead, we use the labels provided by one of the two ensembles in the pair as ground truth. This combination generated a total of 156,520 data points.

Figure 5.30 shows the score curves for all AMI scores computed with pairs of clusterings from different data views, grouped by data view. We note that clusterings based on `strings` and `n-grams` attain in average a higher AMI, meaning that `strings` and `n-grams` carry approximately the same information. The `e-ECDGs` data view attains roughly similar AMI when evaluated against `strings` and `n-grams`, but with a slightly higher AMI for the medium case.

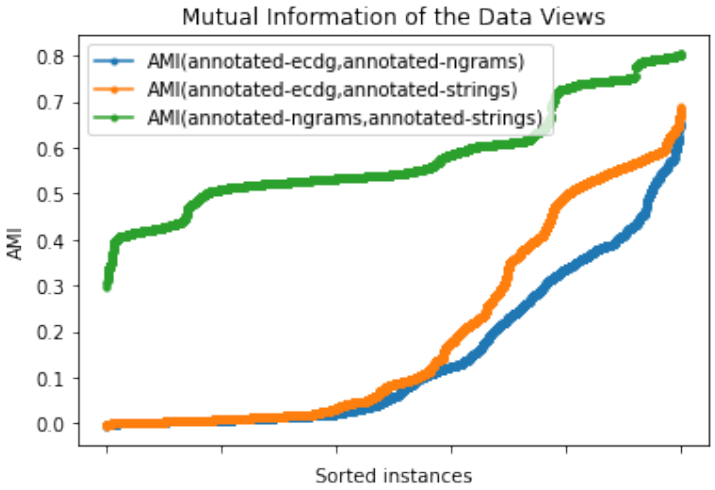


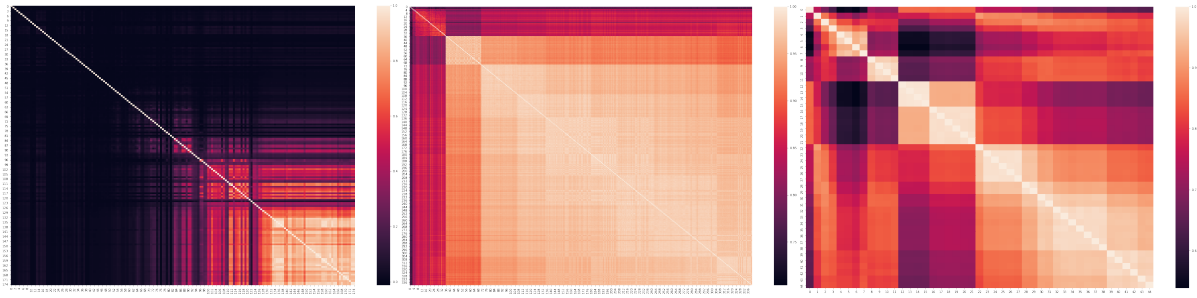
Figure 5.30 – Score curves for AMI between data views.

Figure 5.31 shows the heatmap of all the pairs of ensembles for each data view. The sorting order —which allows to visually recognize the importance order of the hyperparameters —for each data view are:

- `e-ECDGs`: learning rate
- `strings`: clustering algorithm, size of ngram, epochs, dimensions, learning rate
- `n-grams`: clustering algorithm, epochs, dimensions, learning rate

Figure 5.31a corroborates the brunt of higher values for the learning rate; they largely dominate the influence in the results, which precludes the sorting the remaining hyperparameters. Regarding the string and ngrams data views, the number of epochs proved to be an

important parameter —due to the relatively small range of the learning rate for the random search optimization of the embedding learning, we could not gauge the impact of learning rate values in the clustering.



(a) Pairwise AMI (e-ECDG) (b) Pairwise AMI (strings) (c) Pairwise AMI (ngrams)

Figure 5.31 – Heatmap of pairwise AMI for each data view.

Table 5.10 presents the average AMI for the clustering of the data views considering different learning rate thresholds (first column). For each threshold value the AMI is computed considering only vectors of embedding learning instances trained with a learning rate smaller than the threshold. In the case of `e-ECDGs` this parameter has a major impact, not only in relation of `e-ECDGs` and the other data views, but also in the AMI obtained when solely `e-ECDGs` clusterings are considered.

Table 5.10 – AMI of Data Views (constrainted by Learning Rate)

		e-ECDGs			strings		n-grams
		e-ECDGs	strings	n-grams	strings	n-grams	n-grams
Learning Rate	0.2	0.808(±0.111)	0.520(±0.080)	0.394(±0.102)	0.936(±0.051)	0.585(±0.112)	0.814(±0.107)
	0.3	0.618(±0.242)	0.431(±0.164)	0.328(±0.137)	0.934(±0.053)	0.579(±0.113)	0.810(±0.106)
	0.4	0.556(±0.243)	0.397(±0.177)	0.302(±0.145)	0.934(±0.053)	0.579(±0.113)	0.810(±0.106)
	0.5	0.449(±0.270)	0.353(±0.194)	0.266(±0.158)	0.934(±0.053)	0.579(±0.113)	0.810(±0.106)
	0.6	0.346(±0.287)	0.308(±0.210)	0.231(±0.169)	0.934(±0.053)	0.579(±0.113)	0.810(±0.106)
	0.7	0.282(±0.287)	0.278(±0.217)	0.209(±0.173)	0.934(±0.053)	0.579(±0.113)	0.810(±0.106)
	0.8	0.220(±0.277)	0.245(±0.222)	0.184(±0.175)	0.934(±0.053)	0.579(±0.113)	0.810(±0.106)
	0.9	0.209(±0.273)	0.239(±0.222)	0.179(±0.175)	0.934(±0.053)	0.579(±0.113)	0.810(±0.106)

Figure 5.32 shows the heatmap of all the pairs of ensembles selected (sorting the entries by `e-ECDGs`, `strings` and `n-grams`). We see that the interval of AMI values for the `e-ECDGs` clusterings is very wide (due to the learning rate variation), however the `e-ECDGs` clusterings of smaller learning rates attain high AMI values for pairs with other `e-ECDGs` clusterings (also computed using small learning rates) and moderate values with `strings` and `n-grams` clusterings. `strings` clusterings obtain very high AMI scores when pairing with other clusterings

and fairly high scores with `n-grams` clusterings. `n-grams` clusterings reach increasingly better AMI scores when pairing with other `n-grams` clusterings as the number of epochs grows.

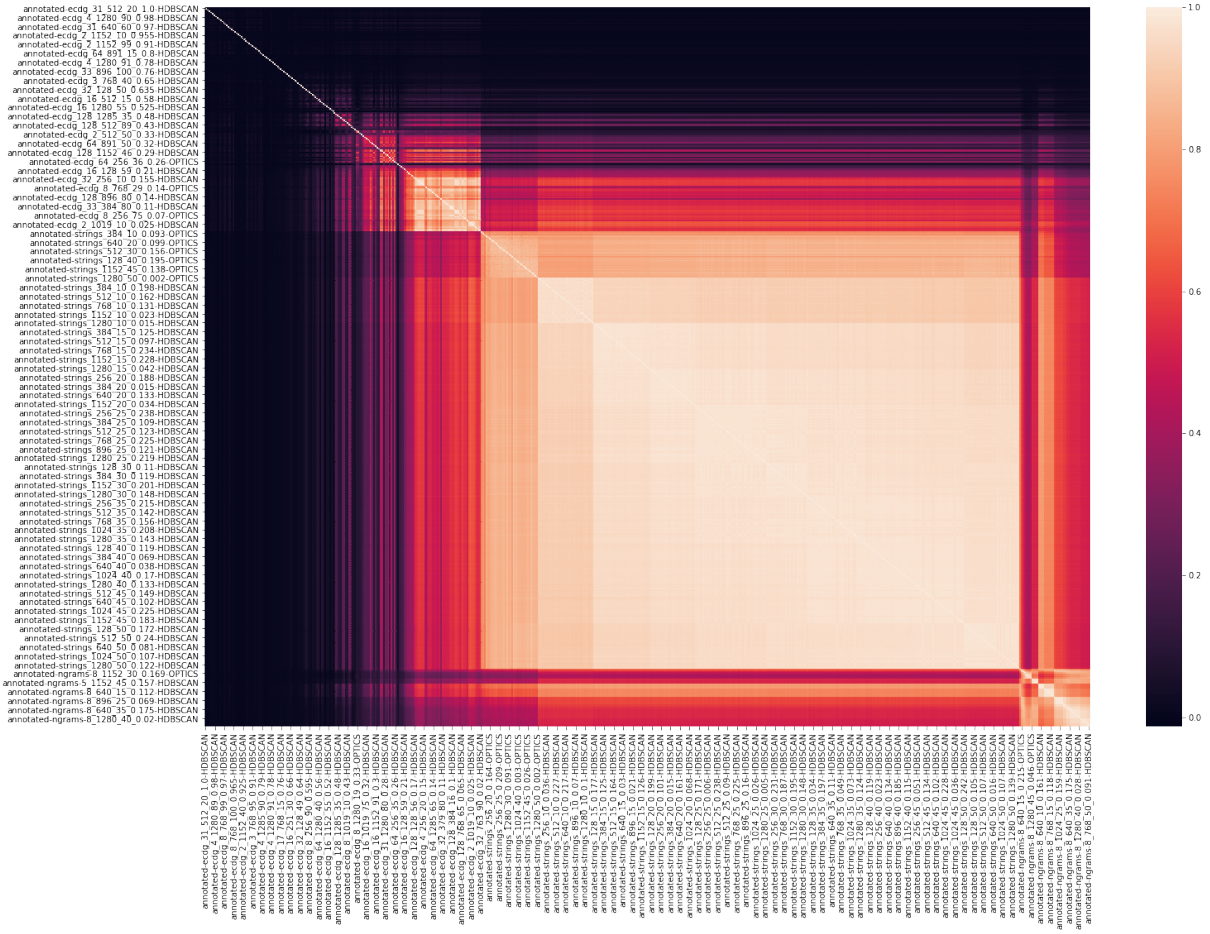


Figure 5.32 – Heatmap containing the AMI of all pairs of the selected data views.

Ensemble Clustering To evaluate ensembles clustering, we selected the 10 ensembles per data view that attained the best silhouette scores, but we discarded those whose embedding learning used a learning rate higher than 0.3. Henceforth, we call *ensemble basis* this set of 30 ensembles, to avoid confusion with the clustering obtained from them, i.e. ensemble clustering (step (iii) in figure 5.8).

By targeting the best silhouette scores and small values for the learning rate, we intended to provide a criterion that is prone to maximize the mutual AMI —and maybe even fortuitously improve the AMI with respect to the ground truth —while being based solely on an internal validation score and a controllable parameter, hence independent from any ground

truth. Table 5.11 shows the complete list of the ensemble basis, including the hyperparameters used during the embedding learning, the clustering algorithm used and the silhouette score obtained.

We tested 5 ensemble clustering methods (ECM), including:

- CSPA
- HPGA
- MCLA
- HBGF
- NMF

Our evaluation was formulated in multiple rounds with increasing number of ensembles. More precisely, the first round included one ensemble per data view (i.e. those with the leading silhouette scores), the second included two ensembles per data view (i.e. those with the leading and second best silhouette scores), and so forth. Thus, our evaluation comprised 10 rounds testing all ECMs, such that the n-th round included the n best (silhouette) scoring ensembles for each data view.

Figure 5.33 shows the execution time of each ECM as function of the number of ensembles. We note that HBGF, HPGA and CSPA finished very quickly (i.e. respectively in ~ 5s, ~ 15s and ~ 30s), while NMF takes a higher (but rather constant) time to compute and MCLA has a considerably large and linearly growing runtime. The linear regressions of the execution times for NMF and MCLA (that respectively attained $R^2 = 0.810$ and $R^2 = 0.903$) are:

$$\hat{t}_{\text{nmf}} = 1101.92 + 61.94 \cdot n_{\text{ens}} \qquad \text{(NMF)}$$

$$\hat{t}_{\text{mcla}} = -3261.09 + 423.67 \cdot n_{\text{ens}} \qquad \text{(MCLA)}$$

these equations show that the computation time for NMF increased by roughly 1 minute per additional ensemble, while MCLA increased by roughly 7 minutes.

Figure 5.34a contains a heatmap table showing the AMI of all pairs of ensembles in the ensemble basis. We can see three lighter blocks in the main diagonal, showing that ensembles of the same data views attain higher AMI. Outside of the main diagonal, we also see that the regions having pairs of strings and `n-grams` data views are lighter than those having `e-ECDGs`. For these mixed pairs, `e-ECDGs` ensembles attain lower AMI with `n-grams` or `strings` (~ 0.6) than the mixed pairs having `n-grams` and `strings`, which attain moderate

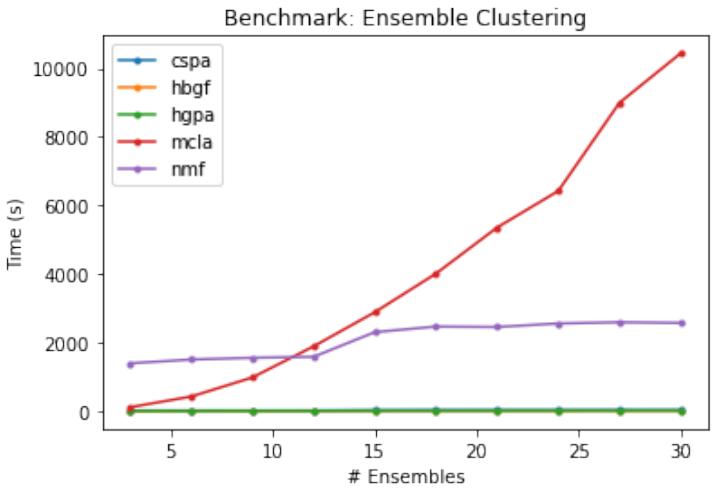


Figure 5.33 – Benchmark of the ensembles clusterings per number of ensembles.

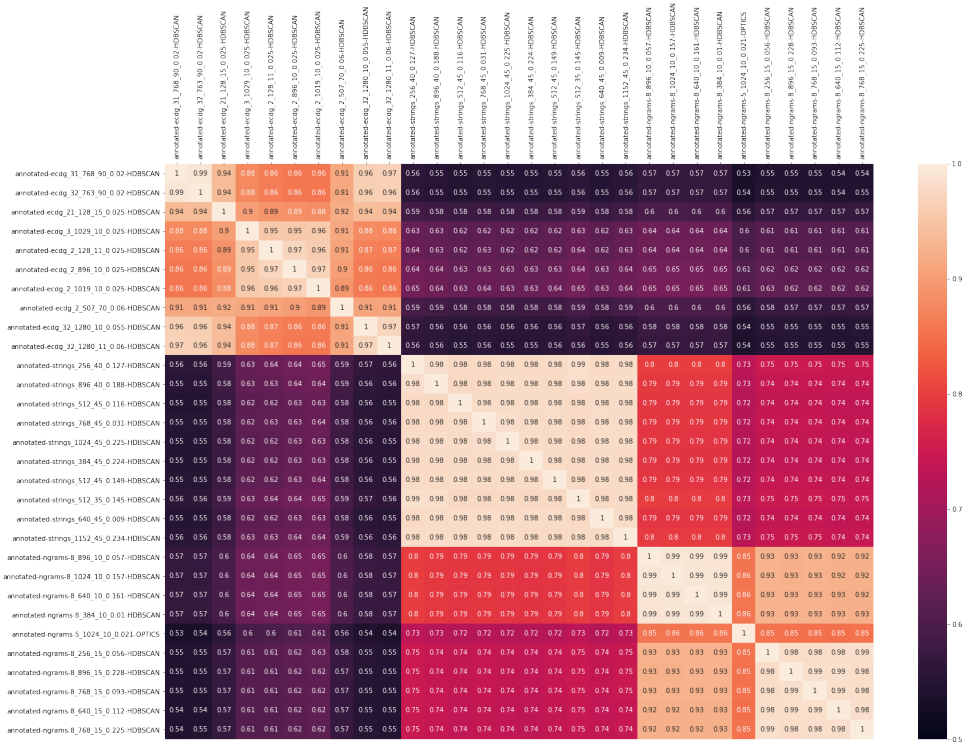
AMI scores (~ 0.75). Although the ensemble basis elements were chosen only with basis on their silhouette scores and learning rates, these scores are in consonance with the presented previously larger analysis about the mutual information of data views.

The results of all AMI scores of the ensemble clusterings with respect to the instances of the ensemble basis are presented in figure 5.34b. We note that the best overall AMI scores were obtained with NMF (~ 0.72) and MCLA (~ 0.76) ECMs.

To measure the information gain attained with the ensemble clustering approach, we computed the average AMI considering all (ensemble) clusterings obtained with each ECM (i.e. one clustering per round). We also compute the average AMI between the ensembles of each data view; however in this case, to avoid a selection bias we did not count the AMI score of ensemble pairs of the same data view. The results of this measurements are presented in table 5.12. The heatmap table 5.35 shows the average AMI per category (i.e. data view or ECM) —including the AMI values for clusterings in the same category and across those generated with different ECMs, for the sake of completeness.

Table 5.11 – List of ensembles in the *ensemble basis*

	Hyperparameters				Algo	Silhouette Score
	n or wl-iterations	dims	epochs	learning rate		
e-ECDGs	31	768	90	0.020	HDBSCAN	0.631
	32	763	90	0.020	HDBSCAN	0.630
	21	128	15	0.025	HDBSCAN	0.605
	3	1029	10	0.025	HDBSCAN	0.594
	2	128	11	0.025	HDBSCAN	0.579
	2	896	10	0.025	HDBSCAN	0.573
	2	1019	10	0.025	HDBSCAN	0.572
	2	507	70	0.06	HDBSCAN	0.520
	32	1280	10	0.055	HDBSCAN	0.503
	32	1280	11	0.06	HDBSCAN	0.499
	3	763	85	0.09	HDBSCAN	0.484
strings	-	256	40	0.127	HDBSCAN	0.590
	-	896	40	0.188	HDBSCAN	0.589
	-	512	45	0.116	HDBSCAN	0.588
	-	768	45	0.031	HDBSCAN	0.588
	-	1024	45	0.225	HDBSCAN	0.588
	-	384	45	0.224	HDBSCAN	0.587
	-	512	45	0.149	HDBSCAN	0.587
	-	512	35	0.145	HDBSCAN	0.587
	-	640	45	0.009	HDBSCAN	0.587
	-	1152	45	0.234	HDBSCAN	0.586
n-grams	8	896	10	0.057	HDBSCAN	0.300
	8	1024	10	0.157	HDBSCAN	0.300
	8	640	10	0.161	HDBSCAN	0.300
	8	384	10	0.01	HDBSCAN	0.300
	5	1024	10	0.021	OPTICS	0.198
	8	256	15	0.056	HDBSCAN	0.182
	8	896	15	0.228	HDBSCAN	0.182
	8	768	15	0.093	HDBSCAN	0.181
	8	640	15	0.112	HDBSCAN	0.181
	8	768	15	0.225	HDBSCAN	0.181



(a) Pairs of ensembles in the *ensemble basis*.

(b) Pairs of ensembles from the *ensemble basis* and the set of ensemble clusterings (mutually exclusive).)

Figure 5.34 – Heatmap table containing AMI values for clustering pairs.

Table 5.12 – Ensemble Clusterings: AMI between groups of clusterings

	Mean AMI	e-ECDGs			strings			n-grams		
		MEAN	MIN	MAX	MEAN	MIN	MAX	MEAN	MIN	MAX
e-ECDGs	0.590 (±0.0346)	-			0.591 (±0.034)	0.548	0.647	0.589 (±0.035)	0.547	0.652
strings	0.675 (±0.0897)	0.591 (±0.034)	0.548	0.647	-			0.760 (±0.026)	0.548	0.800
n-grams	0.675 (±0.091)	0.589 (±0.035)	0.547	0.652	0.760 (±0.026)	0.548	0.800	-		
mcla	0.760 (±0.0979)	0.652 (±0.034)	0.602	0.724	0.871 (±0.045)	0.753	0.915	0.758 (±0.039)	0.670	0.830
nmf	0.7189 (±0.0442)	0.690 (±0.019)	0.663	0.735	0.773 (±0.011)	0.753	0.795	0.694 (±0.030)	0.622	0.741
hgpa	0.645 (±0.0485)	0.644 (±0.023)	0.611	0.684	0.700 (±0.004)	0.692	0.709	0.591 (±0.024)	0.550	0.626
cspa	0.611 (±0.0390)	0.592 (±0.023)	0.548	0.637	0.657 (±0.008)	0.641	0.676	0.583 (±0.026)	0.531	0.617
hbgf	0.600 (±0.0513)	0.582 (±0.030)	0.509	0.636	0.661 (±0.022)	0.609	0.686	0.558 (±0.028)	0.482	0.599

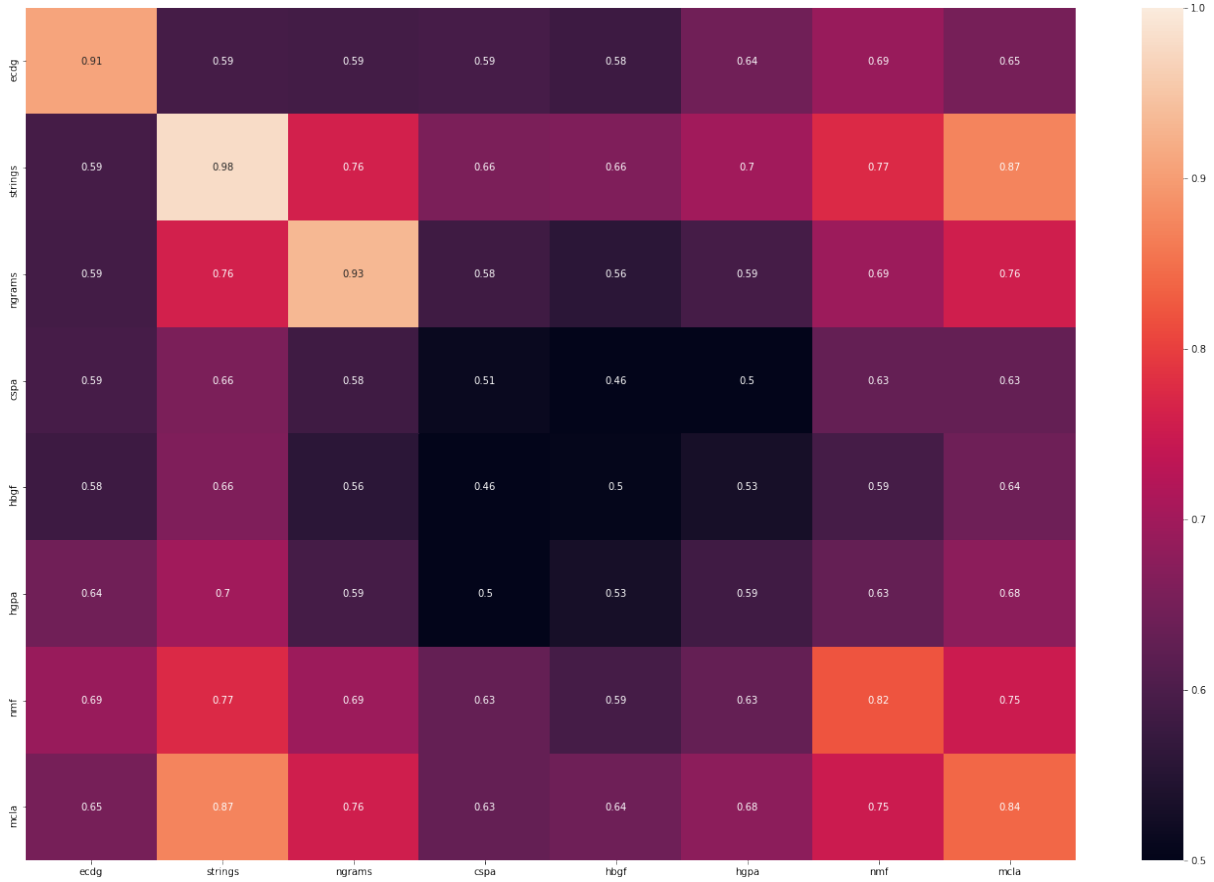


Figure 5.35 – Heatmap table containing the average AMI scores across clusterings in different categories (i.e. data views/ECM).

Ensemble Clustering versus Ground Truth To analyze the outcome of ensemble clustering in the partitioning of the annotated dataset, we compared the clustering with best mean AMI_{gt} against the Yara-based ground truth —we notate this AMI as AMI_{gt} to avoid confusion with the AMI between pairs of ensembles as done previously.

The ensemble clustering with best AMI_{gt} is *ensembles-9-mcla*, which corresponds to the instance that uses mcla as ECM and is computed using the 9 best silhouette-scoring ensembles for all three data views.

Furthermore, we also compared the scores of *ensembles-9-mcla* against the clusterings of the *ensemble basis* that have the highest AMI_{gt} for each data view, to evaluate the effects of ensemble clustering.

Tables 5.13 shows the clusterings of the *ensemble basis* per data view that have the highest AMI_{gt} scores —their ID are respectively *annotated-ecdg-2-1019-10-0.025-HDBSCAN*, *annotated-strings-256-40-0.127-HDBSCAN* and *annotated-ngrams-8-1024-10-0.157-HDBSCAN*.

Table 5.13 – List of ensembles in the *ensemble basis* with highest AMI_{gt}

Data View	Hyperparameters				Algo	AMI_{gt}
	n or wl-iterations	dims	epochs	learning rate		
`e-ECDGs`	2	1019	10	0.025	HDBSCAN	0.6941
`strings`	-	256	40	0.127	HDBSCAN	0.7634
`n-grams`	8	1024	10	0.157	HDBSCAN	0.7575

Table 5.14 shows the scores for the selected clusterings. We see that *ensembles-9-mcla* improves on the clusterings directly computed from the individual data views for all supervised metrics. Furthermore, we observe that the level of noise is remarkably lower for *ensembles-9-mcla* than for the other counterparts.

Figure 5.36 shows the heatmap drawn from the contingency matrix of *ensembles-9-mcla* with respect to the Yara-based ground truth. Each row corresponds to one individual cluster in *ensembles-9-mcla*, each column corresponds to an individual Yara rule, and the cell at (i, j) corresponds to the number of instances that belong to the i-th clusters in *ensembles-9-mcla* and the j-th Yara rule. We see that (i) only two clusters in *ensembles-9-mcla* (around indexes 39 and 249) mix a large amount of samples matched with different Yara rules; (ii) several cases of vertically aligned points (in similar colors) appear, which indicates that *ensembles-9-mcla* is able to split files matched by a single Yara rule in multiple clusters, thus working as a “sub-family” refinement.

Table 5.14 – Supervised metrics of chosen clustering instances with highest AMI_{gt}

Metric \ Clustering	ensembles-9-mcla	annotated-ecdg-2-1019-10-0.025-HDBSCAN	annotated-strings-256-40-0.127-HDBSCAN	annotated-ngrams-8-1024-10-0.157-HDBSCAN
AMI_{gt}	0.7817	0.6941	0.7634	0.7575
NMI_{gt}	0.8079	0.7310	0.7925	0.7793
homogeneity $score_{gt}$	0.7113	0.6469	0.6992	0.7686
completeness $score_{gt}$	0.9348	0.8403	0.9145	0.7903
v-measure $score_{gt}$	0.8079	0.7310	0.7925	0.7793
rand $score_{gt}$	0.5411	0.4271	0.5270	0.4287
Fowlkes-Mallows $score_{gt}$	0.6115	0.5042	0.5905	0.4775
# clusters	278	160	295	155
# noise	11	486	774	3382

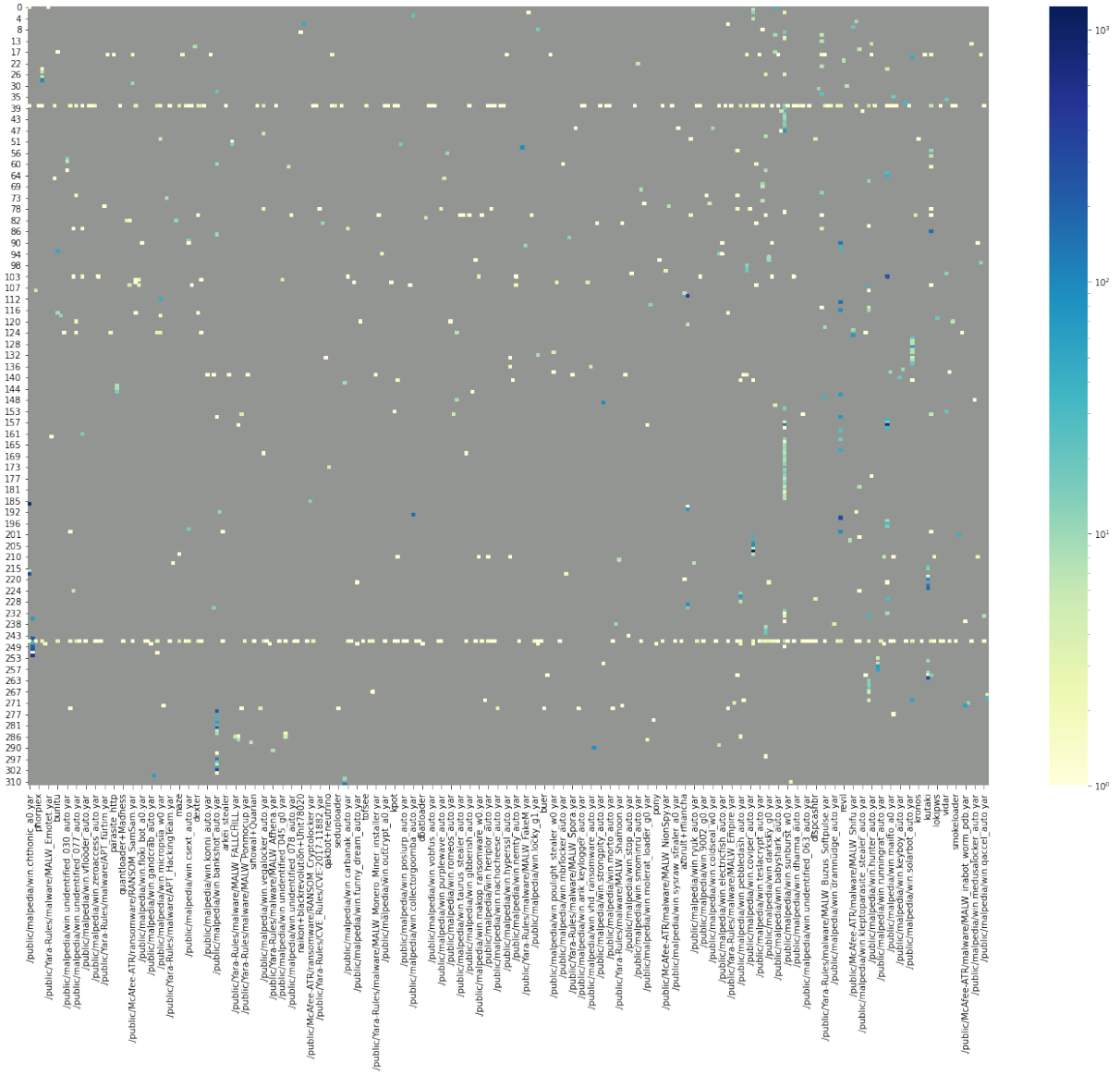


Figure 5.36 – Heatmap of the contingency matrix for *ensembles-9-mcla* (rows) versus the Yara-based ground truth (columns).

5.6 Discussion

This section discusses the results obtained in the experimental section. Our experiments were carried out in two main segments: σ^{ECDG} -Learning with SVM Regression (section 5.5.3, page 218) and EMB-DUET (section 5.5.4, page 227). Both have numerical embedding learning as core, but the former targeted the transposition of σ^{ECDG} from graphs to numerical vectors, while the latter uses numerical embedding as data view for ensemble clustering.

In the first experimental segment, as main contribution we assessed the primary influence of SVM parametrization ($\mathscr{P}_{SVM}$) on the quality of the σ^{ECDG} transposition. In particular, the choice of the SVM Kernel has shown to be a key factor in overall learning performance. The other factors (i.e. parameters for embedding learning ($\mathscr{P}_{emb}$), size of the training dataset ($\mathfrak{s}_{\text{train}}$) and size of the test dataset ($\mathfrak{s}_{\text{test}}$)) were overshadowed by $\mathscr{P}_{SVM}$ for the ranges considered in our random search.

We tested the Polynomial and RBF kernels for the SVM Regression experiments. The MAE obtained with the Polynomial kernel were relatively steady with respect to the setup of the degree parameter; furthermore, the Polynomial kernel was greatly outperformed by the RBF kernel. Presumably, the Polynomial kernel does not inherently fit σ^{ECDG} -Learning data.

The RBF Kernel attained the best scores for MAE. These scores were however still relatively high —which deterred us from using σ^{ECDG} -Learning with SVM Regression in the EMB-DUET experiments. Yet, figure 5.20 (page 225) shows that MAE score can be improved by using larger training datasets in the learning phase. Thus, a possible alternative to improve regression results is to substantially increase the training data (i.e. $\mathfrak{s}_{\text{train}}$) size.

In the second experimental segment we analyzed EMB-DUET. We considered three data views (i.e. `e-ECDGs`, `strings`, `n-grams`), which were chosen due to their complementary characteristics. `e-ECDGs` provides a structural feature that aims to provide a behavioral description of the files, whereas `strings` and `n-grams` are both syntactical-oriented features, yet very distinct from each other.

One major motivation for EMB-DUET was the speedup of the data analysis time so as to render it scalable. In our experiment we assessed a linear time growth (with respect to the number of epochs) for the embedding computation when using Doc2Vec and a bilinear growth (with respect to the number of epochs and WL-iterations) when using Graph2Vec. This is an auspicious result for the scalability of EMB-DUET, notwithstanding the impact that vocabulary size may have on the time necessary to learn the numerical embedding. Nonetheless, this time is dominated by the construction of the vocabulary, which can be

reused and updated only every so often, and it can also be limited in size, discarding terms that are less relevant for the learning —but we did not studied this possible optimization here.

Our experiments showed that computing clustering on embedded vectors is very efficient; for instance, the average time to compute clusters for 27127 embedded vectors was below 3 minutes using HDBSCAN (c.f. table 5.6, page 252). In our benchmark HDBSCAN greatly outperformed OPTICS, as it attained an overall speedup factor of 31,1 over OPTICS. Furthermore, HDBSCAN generated the clusterings with the best (internal and external) scores in our experiments. Therefore, HDBSCAN has shown to be an excellent choice for computing the ensembles of the data views.

Processing multiple data views in the analysis increases the computation cost. However, this computation is highly parallelizable, since the data views are independent from each other. Therefore, the critical path for the ensembles computation corresponds to the time taken by the slowest of them; in our experiments it was roughly the time taken to compute (without optimization) the numerical embedding for `n-grams`.

We conducted a lighter version of the AnR evaluation (c.f. section 4.4.3, page 165). The target metrics for accuracy were the AMI and the homogeneity score obtained by the ensembles, while the robustness phase also took into account Fowlkes-Mallows index, completeness score and silhouette score.

The accuracy analysis achieved fairly positive results; figure 5.25 (page 236) shows the correlation of AMI and homogeneity scores for the HDBSCAN clusterings. We note that the clusterings of the `strings` data view attained the highest AMI scores, which means that they are able to (autonomously) capture the information contained in the Yara rules adopted as ground truth. This is coherent with the fact that Yara rules in the overwhelming majority of cases target `strings` to define matching criteria. We also note that the clusterings of the `n-grams` data view attained the highest homogeneity scores, which is congruent with the vocabulary size.

We emphasize that the external evaluation takes the selected Yara rules (from Malpedia and other sources) as ground truth. Despite our effort to winnow our set of Yara rules to the ones with highest quality only, they may have their own imperfections; for instance, we had 282 matches (out of 13,065) with files that belonged to our cleanware dataset. However, since we perform an unsupervised analysis, maximizing the AMI and homogeneity scores is not a central part of our objectives.

The robustness analysis shows minor differences in the trend plots of all metrics, notwith-

standing great differences in size between the full and annotated datasets (i.e. with ratios of 1,5087×, 2,0697× and 2,0768× for the `e-ECDGs`, `strings` and `n-grams` data views, respectively). It therefore demonstrate that the use of embedding learning does not degrade the inherent robustness afforded by data views.

We analyzed the correlation of internal and external scores, so as to make propitious selection of ensembles when no ground truth is available —which is the regular use case. It demonstrated a positive correlation between AMI and silhouette score, thus evincing the relationship between both metrics. However, this analysis also revealed the importance of the learning rate; in our experiments, the optimal learning rate was found to be around 0.23, presenting a negative impact for higher values (which generated clusterings with high silhouette score and low AMI).

The analysis of the mutual information of the data views assessed two distinct scenarios: the intrinsic analysis, which considers only the pairs of clusterings from the same data view, and the extrinsic analysis, which considers only the pairs of clusterings from the two different data views.

The syntactic-based data views (i.e. `strings` and `n-grams`) attained higher intrinsic AMI scores, which means that there is less variation across different clusterings of these data views. This corroborates the general rule of thumb that considers syntactic features more precise than semantic features. Furthermore, the syntactic-based data views also attained higher extrinsic AMI scores when evaluated against them (instead of `e-ECDGs`); this means that they share more inherent similarities, which is also coherent, as both data views are syntactic-based.

In the analysis of the ensemble clustering, we tested 5 ECMs. The best results were obtained with MCLA and NMF; both were able to greatly improve the mean extrinsic AMI of the data view. In other words, the ensemble clustering was able to learn from the entire set of data views, thus retaining the preeminent properties of each data view, that are the precision of the `strings`, homogeneity of the `n-grams` and the robustness of the `e-ECDGs`.

The execution time for both ECMs grew linearly with respect to the number of ensembles in our experiments. Nonetheless, the execution time to compute the ensemble clustering remained very adequate (even for $n_{\text{ens}} = 30$), thus showing to be suitable for the practical use case. Moreover, in an actual use, the number of ensembles to cluster may well be much less than the 30 chosen for benchmark in our analysis. This is an important result for the scalability of the overall framework, because the ensemble clustering is part of the whole computation critical path.

Finally, we analyzed the instance of ensemble clustering (*ensembles-9-mcla*) that obtained the highest extrinsic AMI, thus obtaining the maximum information across all data views in our experiments. Indeed, *ensembles-9-mcla* improved all external validation scores when compared with the ensembles of the individual data views with highest silhouette score.

We note that the use of ensemble clustering precludes the internal validation of clustering, because the metrics for this intent are based on the notion of distances between data points. In our case no natural notion of distance exist. In fact, the distances are computed in the context of each data view and thus comprised in the labels produced by the clusterings. When ECM acts on the ensembles, distances are disregarded and only labels are taken into account.

Another positive outcome of the ensemble clustering was the reduced amount of noise. It was able to reduce the number of noise samples from a few hundreds of data points to dozens (e.g. only 11 singletons in the case of *ensembles-9-mcla*). This demonstrated the ability of ensemble clusterings to handle missing values in datasets.

5.6.1 Threats to Validity

This section discusses the validity [244] of our study according to: construct, internal, external validity and reliability.

Construct Validity: we have performed controlled experiments that allowed us to assess the clustering properties (i.e. accuracy, robustness and scalability) targeted by our methodology. Our evaluation dataset was painstakingly built to allow appraising the performance of the proposed framework with respect to the targeted properties and the experiment followed closely the proposed methodology; in this regard, no issue should arise from our experiential study.

Internal Validity: our analysis is derived from the results obtained from a very large parametric space, which can not be fully explored. Thus, we have access to a limited range of data points, despite our effort to cover the parametric space as much as possible using the random search optimization. This may affect the completeness of our analysis, nonetheless it does not impact soundness.

External Validity: all techniques used here for embedding learning, SVM regression and ensemble clustering are use-case agnostic, being applied without any *ad-hoc* customization to our scenario. Therefore, there should be no issue with generalization of the methodologies proposed in this study. We strove to compose a proper evaluation dataset from real

world malware and cleanware instances and a curated list Yara rules; nonetheless, as the true profile of the malware and cleanware population is unknown, misrepresentation remain possible, which could engender a selection bias. This is withal an unmanageable issue, which is a common concern in malware research.

Reliability: our study is based on a set of methods and techniques that are of public knowledge and well established in the literature. Furthermore, we provided all details for proper parametrization, whenever it was the case. We constructed our evaluation dataset with publicly available Yara rules, which allows to replicate the generation process of a new dataset that should display the same properties as ours.

5.7 Conclusion

This chapter proposes and evaluates a new malware classification framework called EMB-DUET, which is able to handle different data views obtained from the dataset under analysis. EMB-DUET builds on DUET, which leverages ensemble clustering to consolidate multiple clusterings into a single one; however, EMB-DUET adds a numerical embedding layer to normalize the input data and boost overall framework performance. Ancillary to EMB-DUET, we also proposed and tested σ^{ECDG} -Learning with SVM Regression, which targets the transposition of σ^{ECDG} to ECDGs in their vectorized form, instead of the original graph format.

Our experiments with EMB-DUET demonstrated that the use of embedding learning for feature vectorization does not encumber the original properties of the feature, as accuracy and robustness. However, this transformation is able to boost data analysis (e.g. clustering), as vectors are not only compact units of data, easy to handle in large numbers, but are also friendly to machine learning algorithms. We also demonstrated that ensemble clustering is able to coalesce the partitions from different ensembles into a single clustering, which attains a higher mutual information with each original ensemble than the mutual information between the original ensembles themselves.

The analysis revealed the importance and impact of different hyperparameters for EMB-DUET as well as σ^{ECDG} -Learning with SVM Regression. In the case of σ^{ECDG} -Learning with SVM Regression, the experiments evinced that the RBF kernel fits considerably better the regression than the Polynomial kernel. Kernel hyperparametrization was the leading performance factor for the range of values considered in our random search optimization, overshadowing hyperparametrization for the embedding learning and dataset sizes. For EMB-DUET, we ascertained the relationship between internal and external validation scores. Experiments

showed a (potential) positive correlation between silhouette score and AMI with respect to the Yara-based ground truth. However for this correlation to hold it was necessary to properly setup the learning rate of the Graph2Vec algorithm —for learning rate equal to 0.23, the Pearson coefficient measuring the correlation between AMI and silhouette score was 0.928.

The analysis of EMB-DUET also substantiated some empirical notions related to data views. Data views based on syntactical features (i.e. `strings` and `n-grams`) shared a higher AMI and attained higher precision and homogeneity with respect to the Yara-based ground truth. However, the semantic-based data view (i.e. `e-ECDGs`) obtained better silhouette scores and more malleability with respect to the hyperparametrization, which denotes its ability to handle highly diversified datasets (thus more likely to contain a higher rate of outliers). Thanks to ensemble clustering, all data views could be consolidated into a single cluster that improved the entire set of external validation scores while greatly reducing the amount of noise when compared with the original ensembles (c.f. table 5.14, page 250).

Besides accuracy and robustness (inherent to the choice of data views), *scalability* was another important characteristic sought for the proposed framework. The vocabulary size and chosen number of epochs for training were confirmed to be a burden on the execution time of the embedding learning with Doc2Vec and Graph2Vec. In case of Graph2Vec, the parametrization of the *WL-iteration* has also a substantial impact on the execution time. Yet, we were able to analyze a considerably large dataset ($\sim 27K$ samples) in few hours, considering only a single setting of embedding learning and clustering. The analysis showed a multi-linear growth rate (with manageable constant factors) to parallelize the execution time with respect to the set of hyperparameters.

We fortuitously verified the superiority of HDBSCAN over OPTICS. We used both clustering methods in our evaluation because of their autonomy in the process of cluster computation, which does not need any major parametrization to work —the only, optional, parameter that we set was the minimum number of data points to form a cluster. The results obtained with HDBSCAN were frequently better (for both for internal and external scores), and execution time was remarkably shorter, attaining an overall speedup factor of 31,1.

As main contributions of this chapter:

(i) we proposed EMB-DUET as a data-driven framework that is able to handle heterogeneous features, which leverages numerical embedding to boost overall performance (c.f. section 5.4.1, page 208);

(ii) we verified the accuracy and robustness afforded by the aggregation of the different data views (i.e. `strings`, `n-grams` and `e-ECDGs`) (c.f. section 5.5.4, page 236);

(iii) we analyzed the impact and effects of different hyperparameters involved in EMB-DUET and σ^{ECDG} -Learning with SVM Regression, so as to provide a guideline for proper parameter setting (c.f. sections 5.5.4 and 5.5.3, page 227 and 218).

Additionally, we have contributions that derive from our experimental setup, i.e.:

(i) the deployment and evaluation of the Cuckoo framework for sandboxing, in particular the development of our custom report module that outputs the ECDG of the analyzed file and the related benchmarks obtained during the use of Cuckoo for the experiments;

(ii) the procedure developed to build the dataset, in particular the proposal of the *matching graph* as a means to refine the dataset labeling process.

CONCLUSION & FUTURE WORK

In this thesis we studied the problem of malware classification, in particular we endeavor to develop methods and methodology that can respond to the modern scenario of multitudinous and highly complex malware. This situation requires building a malware analysis framework that is highly automated and autonomous —where the autonomy requirement is particularly important to guarantee a true independence from human intervention (c.f. section 1.3.2, page 50).

The bedrock of our malware classification framework are methods and methodologies of machine learning. Therefore, before tackling the technical matter of our research, we revisited and discussed the epistemological concerns related to malware research and machine learning (c.f. chapter 1, page 25).

Initially we confronted the matter of malware ontology (c.f. section 1.2, page 26) in order to identify the basis of our approach to characterize malware instances. As a result of this analysis, we formulated a realist malware ontology (c.f. section 1.2.2, page 42) that comprises a dual facet (the *realization level* and the *teleological function*) which guided our axiology (c.f. section 1.4, page 54).

Then, we examined the prevalent epistemological basis in malware research (c.f. section 1.3.2, page 50), which allowed us to identify the preponderance of *Foundationalism* in the wrought structure of knowledge in the field. We also brought to the light of malware research the JTB definition of knowledge (c.f. section 1.3.1, page 45), which spurred the analysis that pinpoints the pitfalls of external justification in the use of machine learning methods. Finally, we took the argument that security is fundamentally a social discipline, which conditions the intractability of entirely supplanting human acumen by machines, despite the effort to make systems as autonomous as possible. These epistemological considerations are also taken into account in our axiology.

We then took an accruing approach that followed the stages of our malware classification workflow: enhancement of tooling for malware analysis using symbolic execution for call tracing (chapter 3,page 111), study of ECDGs to represent the behavior of binary codes (chapter 4, page 153), and the proposal of EMB-DUET as a polyvalent framework design (chapter 5, page 189), which leverages the technique of numerical embedding to efficiently

handle big corpus of multifarious data and in addition uses methods of ensemble clustering, which are able to consolidate heterogeneous data in a single clustering.

The initial stage targeted the use of symbolic execution as an alternative method for binary analysis. Symbolic execution implements a very different technique than traditional static and dynamic methods, which are much more frequently evaded by malware. Our focus was to leverage symbolic execution to trace calls made by programs during their execution —which is essential to ECDGs, our graph-based behavioral representation of programs, studied in the chapter 4 (page 153) —therefore we investigated which are the effects of different heuristic parameterizations on SMT solvers and on the overall analysis. Our experiments (c.f. section 3.6, page 147) showed that:

(i) the number of active paths in the symbolic execution should be maximized, provided that this does not cause the analysis to exceed the available memory ;

(ii) the SMT optimization (notably the chosen setting of tactics) is effective in enhancing the overall analysis, despite not being the main optimization factor;

(iii) limiting the number of loops in the execution improves the number of unique calls obtained in the analysis, although this resulted in a minor improvement in the quality of our ECDGs;

(iv) other factors (e.g. step timeout, the minimum size of traces) do not have a relevant impact on the analysis outcome.

The next stage analyzed the effectiveness of ECDGs in providing a birthmark representation for the structure of computer programs. Based on ECDGs, we defined a new similarity function (σ^{ECDG}) (c.f. section 4.4.2, page 164) that, in our efficiency benchmark (c.f. section 4.5.3, page 173), outperformed *radiff2*, a component of the well established open-source suite *radare2*. Our σ^{ECDG} also achieved positive results in the evaluation of its accuracy and robustness. Striving for a stronger evaluation methodology, we formulated the *Accuracy and Robustness (AnR) paradigm* (c.f. section 4.4.3, page 165), which supports the use of mixtures of datasets, built according to different heuristic methodologies so as to test different aspects of the framework (i.e. its accuracy and robustness). To the best of our knowledge, no previous work in the literature adopted a similar holistic phased evaluation methodology. Our results (c.f. section 4.5.2, page 169) attained a high accuracy score (0.983), which shows that a σ^{ECDG} -based framework can autonomously describe malware families as accurately as our set of manually created signatures, and a strong resistance to outliers —hence robustness —when the test dataset intentionally contained a high rate of noise —the NMI score

between the initial and final (highly polluted) clusterings of this phase was 0.974, meaning that the information contained in the initial clustering is greatly conserved after the insertion of noise. On top of that, we showed that our framework is able to produce descriptive cluster prototypes (c.f. section 4.5.6, page 180) which can be leveraged by other machine learning methods (e.g. semi-supervised learning). Our cluster prototypes can assist the human analysis with a graphical representation of a common behavior in a set of programs and enhance classification models used in malware detection due to their resemblance to signatures.

The last stage of our work focused on the data analysis aspect capacities of a malware classification framework. For this, we proposed EMB-DUET as an improvement of the design architecture proposed in the DUET system [124]. Like the DUET system, EMB-DUET takes advantage of ensemble clustering to consolidate multiple partitions of different features in a single clustering. However, EMB-DUET features a numerical embedding layer that vectorizes the different data views, thereby speeding up the whole analysis process (c.f. chapter 5.4.1, page 208). We also studied the transposition of σ^{ECDG} on embedded vectors computed from ECDGs using Graph2Vec —experimental segment referred as σ^{ECDG} -Learning with SVM Regression (c.f. chapter 5.4.2, page 210). Our experiments showed that the embedding layer does not degrades the properties related to the accuracy and robustness of the transformed features (c.f. chapter 5.5.4, page 227). We also measured the impact and effects of different hyperparameters involved in EMB-DUET and σ^{ECDG} -Learning with SVM Regression, to provide a guideline for optimal parameter setting.

We consider that guidelines stipulated by our axiology were followed along our work, because:

— **Axiology-1** [21]: in this work we did not pursue the hypothesis that existed any *natural* upper classes for software, like malware/cleaware or any innate type of software (e.g. virus, worm, etc). We do rely on constituted malware families for analysis (in chapter 3, page 111) and evaluation (in chapters 4 and 5, pages 111 and 189), but in the perspective of *software birthmarks* as assented by our axiology.

— **Axiology-2** [22]: as we pursued a *data driven* approach to classify malware, we turned to unsupervised machine learning methods (with special regard to internal validation metrics). We still deem valuable the use of ground truth for external validation, but for this we carefully built our evaluation dataset avoiding any outsourced labeling —the

21. Axiology-1: pursuing no ontological difference between malware and software at the “realization level”.
22. Axiology-2: being more coherent driven.

only exception is in chapter 3, but the focus there is to improve of binary analysis and not unsupervised learning for malware classification.

— **Axiology-3** [23]: as a consequence of our careful preparation of our evaluation dataset for unsupervised learning, we consider that we attained a satisfactory level of doxastic justification.

By and large, we built our malware classification that managed to fulfill our set of objectives. This included the improvement of tool, methods and methodologies for malware analysis as well as the creation of a data-driven approach to construct a malware classification framework that is accurate, robust and scalable.

Furthermore, our incremental strategy allowed us to take a greedy approach to optimize each stage of the end-to-end analysis, otherwise we would be confronted with a gigantic number of parameters to handle. As a consequence, part of the early stages of our workflow could be re-evaluated as we progressed, which reinforced our analysis and provided more results to back up our prior conclusions —which was precisely the case with the call tracing with symbolic execution, and with the use of ECDG as a representation for the behavior of computer programs.

5.7.1 Future Work

Our work comprehends a diversified scope of themes related to malware classification. As each of these topics can undertake its own direction without interfering with the others, we split their presentation in separated sections.

Call Tracing with Symbolic Execution

We have several leads to improve the effectiveness of our call tracer. A patent idea is to evolve our call tracer to run a full-fledged *concolic* execution, which carries out concrete and symbolic analysis simultaneously [17]. The goal is to obtain a *deep and wide* exploration of the program state space, relying on the respective strengths of concrete and symbolic execution.

Past works comparing the different analysis methods presented a noticeable improvement in code coverage of concolic execution over (pure) symbolic [175, 271]. Toward this optimization, we tested the integration of angr with unicorn [227]; our preliminary experiments with full concrete execution obtained a minor speedup with at the expense of lower

23. Axiology-3: being more doxastically justified.

code coverage. This trial evinced that the strategy to duly interleave both execution methods is a complex topic, which deserves a full study dedicated to this task.

To improve our symbolic call tracer for the generation ECDGs, another noteworthy enhancement is to implement taint analysis targeting the symbolic variables in order to track the data flow during the execution. In our work, we rely on the def-use dependence as defined by Christodorescu *et al* [47] to build the edges of the ECDGs. This def-use strategy is based on the *data type* and the *data value* of the common arguments. However, with the taint analysis —facilitated by the symbolic execution —we should be able to improve this definition to attain a finer strategy to connect calls, based on a finer analysis of data dependancies.

Testing other granularity levels is also an axis for improvement. Our current granularity considers the whole binary code, but with symbolic execution it is possible to explore smaller chunks of the program (e.g. a single function). By breaking down the scope of analysis, it is possible to largely parallelize the analysis and to keep the burden on systems (e.g. memory consumption) under a controlled level.

As for SMT optimization, there are several re-assessments that can be explored in other contexts to polish the interpretation of our results. It would be interesting to implement the proposed optimizations into other tools exploiting similar combinations of formal and heuristic approaches for software analysis [11, 18, 31, 63, 135, 172] with the purpose of understanding the presence of domain-specific results and to evaluate the possibility of generalizing these results.

Structural-Based Binary Code Similarity

Working with graph-based similarities of binary codes is in general costly due to the graph matching step involved, as it frequently requires solving instances of NP-complete or NP-hard problems. Our choice of gSpan has proven to be rewarding, but it would be interesting to test alternative approaches for frequent subgraph mining —the survey of T. Ramraja and R. Prabhakarb lists alternative algorithms [229].

Other strategies such as Deep Graph Learning can also be considered to address the problem of frequent subgraph mining. Although partially explored in our last chapter (c.f. section 5, page 189), further exploring how to use numerical embedding and other learning techniques to learn about inner structures of graphs, thus working around the problem of graph matching, is also worth investigating [173].

Different methods to measure the graph similarity (or distance) can also be adopted,

such as the (widely used) Graph Edit Distance (GED), Markov Chain Monte Carlo sampling technique [161], adjacency matrix rewriting to represent graph as a least-square problem [322], or Graph Convolutional Networks [208].

Furthermore, exploring other graph mining techniques [232] such as graph clustering and graph classification (e.g. [144]) to analyze ECDGs can also be considered, withal this would require changing the σ^{ECDG} measure of similarity. This leads to manifold directions that can leverage ECDGs to explore the similarity of binary codes beyond σ^{ECDG}, which is a coarse-granularity similarity function (i.e. it takes into account the entire graph and produces a unique similarity measure as outcome).

Alternatively, it is possible to explore the similarity of ECDGs differently, taking into account multiple and spread matches in the graphs, which could reveal multiple behaviors subsumed in EDCG. This approach contrasts with ours, as our approach targets the largest common subgraph —roughly the "largest common behavior", which is not necessarily the most relevant one (despite making for a good software birthmark in practice) —whereas an approach that looks for (multiple and) decentralized similarities in (sub-)graphs could obtain very different results.

Enhanced versions for ECDGs is also a possible topic for further research. We partially explored such an idea with the fusion of ECDGs obtained from symbolic and (pure) dynamic analysis (c.f. section 5, page 189), but the closer investigation of the effects and optimization of this (and other) composition(s) remains open. It is also possible to study the association of calls (i.e. the nodes of the graph) to a set of categories (similarly to some previous works [85, 112]), which is an auspicious alternative to get more compact graphs and even to progress towards an agnostic description that is able to represent behaviors of different types of programs (e.g. Windows PE, ELF Linux, APKs, scripts, etc).

EMB-DUET

Our work on EMB-DUET can be improved for thorough completion. Due to the computation cost of the random search exploration and the lack of initial knowledge about the impact magnitude of the hyperparameter settings in the final results, in our study we followed a fixed design process [244]. Thus, new exploratory settings can still provide a deeper understanding about the effects of different hyperparametrizations of the learning layers (i.e. embedding learning + partition clustering) on the final ensemble clustering.

A major milestone yet to be achieved is the full integration of EMB-DUET with σ^{ECDG} -learning. However, a successful integration presumes the improvement of σ^{ECDG} -learning to

very low MAE scores in order to avoid propagating regression errors to the clustering of the data view. This requires at first place to further improve the SVM hyperparametrization and to enlarge the training dataset. Further improvements on the embedding learning hyperparametrization may also attain an important impact on performance of σ^{ECDG} -learning, but their effects were outweighed by the other components of the workflow. Alternatively, new regression approaches can be used such as the ensemble model for SVM Regression [43] and ensemble of deep learning belief networks (DBN) [226], to list but a few possibilities.

Further improvements and enhancements for EMB-DUET are also possible. We considered only three data views in our analysis, however there is no theoretical upper limit to the number of features that can be associated, therefore new analysis can explore other combinations of feature sets. There is considerable room for improvement in the clustering step that computes the ensemble basis, since we restricted the analysis only to HDBSCAN and OPTICS with a single setting (i.e. minimum number of points to form a cluster set to 5). The embedding learning step can be also optimized to work with smaller vocabulary sizes and other hyperparameter settings, which can potentially speed up computation and eventually improve the quality of the computed vectors.

EMB-DUET also opens new perspectives for future research. The vector representation of can be exploited for other usages, such as fast malware search, benefiting from this short vector encoding that also enables efficient lookup. Another perspective would be to evolve EMB-DUET in order to furnish it with the ability to perform clustering with constraints so that standard clusterings, which are computed solely with basis on the affinity of the data points, could adapt to supplementary information available. In practice, this would allow to integrate the autonomous approach of EMB-DUET with external information (e.g. labels from signature matching or from expert analysis), thus consolidating every facet of machine and expert analysis in a final clustering.

Future Perspectives

The field of malware analysis faces the challenging task of assessing the *malice* of programs. On the one hand, human expertise is needed to fulfill this complex task, as machine fail to interpret the nuances of contexts and stratagems behind a cyberattack; on the other hand, human effort alone cannot keep pace with the exponential growth of cyber threats. Therefore, the future of malware analysis inevitably goes through the synergy of human and computer work. The challenge is to find the optimal balance between both and develop means of cooperation, so that machines are used to facilitate human work as much as pos-

sible.

The leading trend in this direction is *machine learning* —as approached in this thesis. However, the scenario of malware analysis encompasses epistemological issues that are not often present in traditional uses of machine learning:

— establishing a universal malware detector is an undecidable problem, which impairs the composition of a unbiased ground-truth;

— since it depends on the use context (e.g. malware detection), the outcome of the analysis is only tentative at best, because of the possibility of software to malfunction;

— adversaries are by definition malicious, thus leading to an adaptive adversarial model against the machine learning techniques used in the analysis;

As a consequence, malware research has to deal with these peculiarities if it is to succeed in a real scenario. This calls for the development of methodologies that can take these issues into account.

In particular, the adaptive adversarial model[24] has been for a long time an important issue in malware analysis. To tackle this scenario, the focus in malware research has been greatly put on dealing with anti-reverse engineering techniques developed by malware authors and on creating signatures as countermeasure. However, in a scenario where machine learning is a central piece of the security workflow, this traditional focus overly concentrates on just part of the issue, namely the feature extraction phase of a machine learning-based analysis. So far, little has been proposed to cope with *adversarial machine learning* in the context of malware analysis (e.g. [155]).

Another analysis perspective, complementary to the one we developed in this thesis, is to consider the grouping of malware instances by cyberattack, i.e. associating instances that do not necessarily display an inherent kinship (e.g. similar behavior) —as we did here —but that were seen together in the context of a cyberattack. This approach targets malware instances that are likely to participate together in a cyberattack, thus working as indicators of the actors behind the cyberattack; this allows to gain information about the *Tactics, Techniques and Procedures* of attackers, which can be availed to infer *attribution*.

Convergence is yet another important topic for the future of malware analysis. With the emergence of new technologies that are being connected to the Internet, cyberattackers have a larger attack surface to exploit. This allows for a much more complex composition of attack

24. Commonly referred as the “cat and mouse game”.

vectors, which may well include different technologies (e.g. desktop, mobile, wearable, IoT devices, etc).

Therefore, it will become increasingly necessary to treat a diversified ensemble of technologies in the analysis. To handle that, agnostic descriptions of software birthmarks will have to be developed and analysis methodologies will have to evolve to contemplate this prospective scenario.

BIBLIOGRAPHY

[1] Adel Abusitta, Miles Q. Li, and Benjamin C.M. Fung, « Malware classification and composition analysis: A survey of recent developments », *in*: *Journal of Information Security and Applications* 59 (2021), p. 102828, ISSN: 2214-2126, DOI: https://doi.org/10.1016/j.jisa.2021.102828, URL: https://www.sciencedirect.com/science/article/pii/S2214212621000648.

[2] Hira Agrawal et al., « Detection of global, metamorphic malware variants using control and data flow analysis », *in*: *31st IEEE Military Communications Conference, MILCOM 2012, Orlando, FL, USA, October 29 - November 1, 2012*, 2012, pp. 1–6, DOI: 10.1109/MILCOM.2012.6415581, URL: https://doi.org/10.1109/MILCOM.2012.6415581.

[3] Mansour Ahmadi et al., « Novel feature extraction, selection and fusion for effective malware family classification », *in*: *Proceedings of the sixth ACM conference on data and application security and privacy*, 2016, pp. 183–194.

[4] Frances E. Allen, « Control Flow Analysis », *in*: *Proceedings of a Symposium on Compiler Optimization*, Urbana-Champaign, Illinois: Association for Computing Machinery, 1970, 1–19, ISBN: 9781450373869, DOI: 10.1145/800028.808479, URL: https://doi.org/10.1145/800028.808479.

[5] Mauricio Almeida et al., « Basic Formal Ontology 2.0 », *in*: (2015).

[6] Blake Anderson, Curtis Storlie, and Terran Lane, « Improving malware classification: bridging the static/dynamic gap », *in*: *Proceedings of the 5th ACM workshop on Security and artificial intelligence*, 2012, pp. 3–14.

[7] Nicola Angius, Giuseppe Primiero, and Raymond Turner, « The Philosophy of Computer Science », *in*: *The Stanford Encyclopedia of Philosophy*, ed. by Edward N. Zalta, Spring 2021, Metaphysics Research Lab, Stanford University, 2021.

[8] Quynh Nguyen Anh, « Capstone: next generation disassembly framework », *in*: *USA: BlackHat* (2014).

[9] Mihael Ankerst et al., « OPTICS: Ordering Points to Identify the Clustering Structure », *in*: *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1999, 49–60, ISBN: 1581130848, DOI: 10.1145/304182.304187, URL: https://doi.org/10.1145/304182.304187.

[10] Mihael Ankerst et al., « OPTICS: Ordering Points to Identify the Clustering Structure », *in*: *SIGMOD Rec.* 28.*2* (June 1999), 49–60, ISSN: 0163-5808, DOI: 10.1145/304181.304187, URL: https://doi.org/10.1145/304181.304187.

[11] Andrea Aquino, Giovanni Denaro, and Pasquale Salza, « Worst-Case Execution Time Testing via Evolutionary Symbolic Execution », *in*: *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, 2018, pp. 76–87, DOI: 10.1109/ISSRE.2018.00019.

[12] Robert Arp, Barry Smith, and Andrew D. Spear, *Building Ontologies with Basic Formal Ontology*, The MIT Press, 2015, ISBN: 0262527812.

[13] David Arthur and Sergei Vassilvitskii, *k-means++: The advantages of careful seeding*, tech. rep., Stanford, 2006.

[14] Yara Awad, Mohamed Nassar, and Haidar Safa, « Modeling malware as a language », *in*: *2018 IEEE International Conference on Communications (ICC)*, IEEE, 2018, pp. 1–6.

[15] Jinrong Bai and Junfeng Wang, « Improving malware detection using multi-view ensemble learning », *in*: *Security and Communication Networks* 9.*17* (2016), pp. 4227–4241, DOI: https://doi.org/10.1002/sec.1600, eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/sec.1600, URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/sec.1600.

[16] Michael Bailey et al., « Automated classification and analysis of internet malware », *in*: *International Workshop on Recent Advances in Intrusion Detection*, Springer, 2007, pp. 178–197.

[17] Roberto Baldoni et al., « A Survey of Symbolic Execution Techniques », *in*: *ACM Comput. Surv.* 51.*3* (2018).

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

[19] Clark Barrett, Leonardo De Moura, and Pascal Fontaine, « Proofs in satisfiability modulo theories », *in*: *All about proofs, Proofs for all* 55.*1* (2015), pp. 23–44.

[20] Clark Barrett, Pascal Fontaine, and Cesare Tinelli, *The SMT-LIB Standard - Version 2.6*, website, May 12, 2021, URL: http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-r2021-05-12.pdf.

[21] Clark Barrett, Daniel Kroening, and Thomas Melham, *Problem solving for the 21st century: Efficient solver for satisfiability modulo theories*, English (US), Knowledge Transfer Report, Technical Report 3, London Mathematical Society, Smith Institute for Industrial Mathematics, and System Engineering, June 2014.

[22] Ulrich Bayer et al., « Scalable, Behavior-Based Malware Clustering », *in*: *Proceedings of the Network and Distributed System Security Symposium, NDSS 2009, San Diego, California, USA, 8th February - 11th February 2009*, The Internet Society, 2009, URL: https://www.ndss-symposium.org/ndss2009/scalable-behavior-based-malware-clustering/.

[23] Ulrich Bayer et al., « Scalable, behavior-based malware clustering. », *in*: *NDSS*, vol. 9, Citeseer, 2009, pp. 8–11.

[24] Najah Ben Said et al., « Detection of Mirai by Syntactic and Behavioral Analysis », *in*: *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, 2018, pp. 224–235, DOI: 10.1109/ISSRE.2018.00032.

[25] James Bergstra and Yoshua Bengio, « Random search for hyper-parameter optimization. », *in*: *Journal of machine learning research* 13.*2* (2012).

[26] GNU Binutils, *objdump*, 2021, URL: https://linux.die.net/man/1/objdump.

[27] Nikolaj Bjørner et al., *Programming Z3*, website, URL: https://theory.stanford.edu/~nikolaj/programmingz3.html.

[28] Peter Boonstoppel, Cristian Cadar, and Dawson Engler, « RWset: Attacking path explosion in constraint-based test generation », *in*: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2008, pp. 351–366.

[29] Marcus Botacin et al., « Challenges and pitfalls in malware research », *in*: *Computers & Security* 106 (2021), p. 102287.

[30] Marcus Botacin et al., « Challenges and pitfalls in malware research », *in*: *Computers & Security* 106 (2021), p. 102287, ISSN: 0167-4048, DOI: https://doi.org/10.1016/j.cose.2021.102287, URL: https://www.sciencedirect.com/science/article/pii/S0167404821001115.

[31] Mark Bradley et al., « GoannaSMT–A Static Analyzer with SMT-based Refinement », *in*: *Tools for Automatic Program AnalysiS (TAPAS 2012)*, ENTCS Deauville, France, 2012.

[32] Alexandre Braga et al., « Practical Evaluation of Static Analysis Tools for Cryptography: Benchmarking Method and Case Study », *in*: *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, 2017, pp. 170–181, DOI: 10.1109/ISSRE.2017.27.

[33] Rodrigo Rubira Branco, « Ltrace internals », *in*: *Proceedings of the Linux Symposium*, vol. 1, 2007, pp. 41–52.

[34] Rory Bray, Daniel Cid, and Andrew Hay, *OSSEC host-based intrusion detection guide*, Syngress, 2008.

[35] Thomas Brewster, « An NSA Cyber Weapon Might Be Behind A Massive Global Ransomware Outbreak », *in*: *Forbes* (May 12, 2017), URL: https://www.forbes.com/sites/thomasbrewster/2017/05/12/nsa-exploit-used-by-wannacry-ransomware-in-global-explosion/?sh=9b5d58be599b (visited on 05/24/2022).

[36] « Burma hit by massive net attack ahead of election », *in*: *BBC News* (Nov. 4, 2010), URL: https://www.bbc.com/news/technology-11693214 (visited on 05/24/2022).

[37] Cristian Cadar and Koushik Sen, « Symbolic execution for software testing: three decades later », *in*: *Communications of the ACM* 56.*2* (2013), pp. 82–90.

[38] Cristiano Calcagno and Dino Distefano, « Infer: An Automatic Program Verifier for Memory Safety of C Programs », *in*: *NASA Formal Methods*, ed. by Mihaela Bobaru et al., Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 459–465, ISBN: 978-3-642-20398-5.

[39] Ricardo JGB Campello, Davoud Moulavi, and Jörg Sander, « Density-based clustering based on hierarchical density estimates », *in*: *Pacific-Asia conference on knowledge discovery and data mining*, Springer, 2013, pp. 160–172.

[40] Davide Canali et al., « A quantitative study of accuracy in system call-based malware detection », *in*: *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, 2012, pp. 122–132.

[41] John Capps, « The Pragmatic Theory of Truth », *in*: *The Stanford Encyclopedia of Philosophy*, ed. by Edward N. Zalta, Summer 2019, Metaphysics Research Lab, Stanford University, 2019.

[42] Sang Kil Cha et al., « Unleashing mayhem on binary code », *in*: *2012 IEEE Symposium on Security and Privacy*, IEEE, 2012, pp. 380–394.

[43] Snehamoy Chatterjee, Ansuman Dash, and Sukumar Bandopadhyay, « Ensemble support vector machine algorithm for reliability estimation of a mining machine », *in*: *Quality and Reliability Engineering International* 31.*8* (2015), pp. 1503–1516.

[44] Thomas Chen and Jean-Marc Robert, « The Evolution of Viruses and Worms », *in*: (Dec. 2004), DOI: 10.1201/9781420030884.ch16.

[45] Julia Yu-Chin Cheng, Tzung-Shian Tsai, and Chu-Sing Yang, « An information retrieval approach for malware classification based on Windows API calls », *in*: *2013 International conference on machine learning and cybernetics*, vol. 4, IEEE, 2013, pp. 1678–1683.

[46] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea, « S2E: A platform for in-vivo multi-path analysis of software systems », *in*: *Acm Sigplan Notices* 46.*3* (2011), pp. 265–278.

[47] Mihai Christodorescu, Somesh Jha, and Christopher Kruegel, « Mining Specifications of Malicious Behavior », *in*: *Proceedings of the 6th Joint Meeting of the ESEC and the ACM SIGSOFT SFE*, ESEC-FSE '07, Dubrovnik, Croatia: Association for Computing Machinery, 2007, 5–14, ISBN: 9781595938114, DOI: 10.1145/1287624.1287628.

[48] Mihai Christodorescu et al., « Semantics-Aware Malware Detection », *in*: *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, SP '05, USA: IEEE Computer Society, 2005, 32–46, ISBN: 0769523390, DOI: 10.1109/SP.2005.20, URL: https://doi.org/10.1109/SP.2005.20.

[49] Michael Chui et al., *IoT value set to accelerate through 2030: Where and how to capture it*, Accessed: 2022-05-24, 2021, URL: https://www.mckinsey.com/business-functions/mckinsey-digital/our-insights/iot-value-set-to-accelerate-through-2030-where-and-how-to-capture-it.

[50] Emily Chung, « PlayStation data breach deemed in 'top 5 ever' », *in*: *CBC News* (Apr. 27, 2011), URL: https://www.cbc.ca/news/science/playstation-data-breach-deemed-in-top-5-ever-1.1059548 (visited on 05/24/2022).

[51] Cisco, *Cisco Annual Internet Report*, Accessed: 2022-05-24, URL: https://www.cisco.com/c/en/us/solutions/executive-perspectives/annual-internet-report/air-highlights.html.

[52] ClamAV, *ClamAV 0.99b Meets YARA!*, ClamAV blog, Accessed: 2022-02-03, June 3, 2015, URL: https://blog.clamav.net/2015/06/clamav-099b-meets-yara.html.

[53] *CLANG*, Accessed: 2021-10-26, URL: http://clang.org/.

[54] Fred Cohen, « Computer viruses: Theory and experiments », *in*: *Computers & Security* 6.*1* (1987), pp. 22–35, ISSN: 0167-4048, DOI: https://doi.org/10.1016/0167-4048(87)90122-2, URL: https://www.sciencedirect.com/science/article/pii/0167404887901222.

[55] Julian Cohen, *Contemporary Automatic Program Analysis*, Slideshow presented at Blackhat USA 2014, 2014, URL: https://www.blackhat.com/docs/us-14/materials/us-14-Cohen-Comtemporary-Automatic-Program-Analysis.pdf.

[56] Christian Collberg et al., *The Tigress diversifying C virtualizer*, 2015.

[57] Gerald Combs, *Wireshark*, 2022, URL: https://www.wireshark.org.

[58] Juan Comesaña and Peter Klein, « Skepticism », *in*: *The Stanford Encyclopedia of Philosophy*, ed. by Edward N. Zalta, Winter 2019, Metaphysics Research Lab, Stanford University, 2019.

[59] EUROPEAN COMMISSION, *Shaping Europe's digital future*, Accessed: 2022-05-24, 2020, URL: https://eur-lex.europa.eu/legal-content/en/TXT/?uri=CELEX:52020DC0067.

[60] Federal Trade Commission, *Monitoring Software on Your PC: Spyware, Adware, and Other Software*, Mar. 2005, URL: hhttps://www.ftc.gov/sites/default/files/documents/reports/spyware-workshop-monitoring-software-your-personal-computer-spyware-adware-and-other-software-report/050307spywarerpt.pdf.

[61] Keith D. Cooper et al., « ACME: Adaptive Compilation Made Efficient », *in*: *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '05, Chicago, Illinois, USA: Association for Computing Machinery, 2005, 69–77, ISBN: 1595930183, DOI: 10.1145/1065910.1065921, URL: https://doi.org/10.1145/1065910.1065921.

[62] Target Corporation, *Target Confirms Unauthorized Access to Payment Card Data in U.S. Stores*, 2013, URL: https://corporate.target.com/press/releases/2013/12/target-confirms-unauthorized-access-to-payment-car (visited on 05/24/2022).

[63] Patrick COUSOT et al., « Varieties of Static Analyzers: A Comparison with ASTREE », *in*: *First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering (TASE '07)*, 2007, pp. 3–20, DOI: 10.1109/TASE.2007.55.

[64] CULT OF THE DEAD COW, *RUNNING A MICROSOFT OPERATING SYSTEM ON A NETWORK? OUR CONDOLENCES*, 1998, URL: https://media.defcon.org/DEF%20CON%206/DEF%20CON%206%20articles/DEF%20CON%206%20-%20Cult%20of%20the%20Dead%20Cow%20-%20Back%20Orifice%20Announcement.txt.

[65] George E. Dahl et al., « Large-scale malware classification using random projections and neural networks », *in*: *2013 IEEE International Conference on Acoustics, Speech and Signal Processing* (2013), pp. 3422–3426.

[66] Ben Dai, Xiaotong Shen, and Junhui Wang, « Embedding learning », *in*: *Journal of the American Statistical Association* (2020), pp. 1–13.

[67] Khanh Huu The Dam and Tayssir Touili, « Precise extraction of malicious behaviors », *in*: *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1, IEEE, 2018, pp. 229–234.

[68] Anusha Damodaran et al., « A comparison of static, dynamic, and hybrid analysis for malware detection », *in*: *Journal of Computer Virology and Hacking Techniques* 13.*1* (2017), pp. 1–12.

[69] Leonardo De La Rosa et al., « Efficient characterization and classification of malware using deep learning », *in*: *2018 Resilience Week (RWS)*, IEEE, 2018, pp. 77–83.

[70] Department of Defense (DoD), *Department of Defense Strategy for Operating in Cyberspace*, July 2011, URL: https://csrc.nist.gov/CSRC/media/Projects/ISPAB/documents/DOD-Strategy-for-Operating-in-Cyberspace.pdf (visited on 05/24/2022).

[71] Richard C Detmer, *Introduction to 80x86 Assembly Language and Computer Architecture*, Jones & Bartlett Learning, 2006.

[72] Marko Dimjašević et al., « Evaluation of android malware detection based on system calls », *in*: *Proceedings of the 2016 ACM on International Workshop on Security And Privacy Analytics*, 2016, pp. 1–8.

[73] Yuxin Ding et al., « A malware detection method based on family behavior graph », *in*: *Computers & Security* 73 (2018), pp. 73–86, ISSN: 0167-4048, DOI: https://doi.org/10.1016/j.cose.2017.10.007, URL: https://www.sciencedirect.com/science/article/pii/S0167404817302146.

[74] Brendan L Douglas, « The weisfeiler-lehman method and graph isomorphism testing », *in*: *arXiv preprint arXiv:1101.5211* (2011).

[75] William Duncan, *Making Ontological Sense of Hardware and Software*, Accessed: 2021-12-08, 2009, URL: https://cse.buffalo.edu/~rapaport/584/S10/duncan09-HWSWOnt.pdf.

[76] William Duncan, « Ontological distinctions between hardware and software », *in*: *Applied Ontology* 12 (Feb. 2017), pp. 1–28, DOI: 10.3233/AO-170175.

[77] Joseph C Dunn, « Well-separated clusters and optimal fuzzy partitions », *in*: *Journal of cybernetics* 4.*1* (1974), pp. 95–104.

[78] Editors, « A Pathology of Computer Viruses », *in*: *Virus Bulletin* (Mar. 1992), book review, ISSN: 0956-9979, URL: https://www.virusbulletin.com/uploads/pdf/magazine/1992/199203.pdf.

[79] Editors, « Towards a Common Language », *in*: *Virus Bulletin* (Dec. 1989), editorial, ISSN: 0956-9979, URL: https://www.virusbulletin.com/uploads/pdf/magazine/1989/198912.pdf.

[80] Ammar Elhadi et al., « Enhancing the Detection of Metamorphic Malware using Call Graphs », *in*: *Computers & Security* 46 (Oct. 2014), pp. 62–78.

[81] C.C. Elisan, *Malware, Rootkits & Botnets A Beginner's Guide*, Beginner's Guide, McGraw-Hill Education, 2012, ISBN: 978-0-07-179205-9.

[82] Adam Entous, Ellen Nakashima, and Greg Miller, « Secret CIA assessment says Russia was trying to help Trump win White House », *in*: *The Washington Post* (Dec. 9, 2016), URL: https://www.washingtonpost.com/world/national-security/obama-orders-review-of-russian-hacking-during-presidential-campaign/2016/12/09/31d6b300-be2a-11e6-94ac-3d324840106c_story.html (visited on 05/24/2022).

[83] Ikpeme Erete and Alessandro Orso, « Optimizing Constraint Solving to Better Support Symbolic Execution », *in*: *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, 2011, pp. 310–315, DOI: 10.1109/ICSTW.2011.98.

[84] Mojtaba Eskandari, Zeinab Khorshidpour, and Sattar Hashemi, « HDM-Analyser: a hybrid analysis approach based on data mining techniques for malware detection », *in*: *Journal of Computer Virology and Hacking Techniques* 9.*2* (2013), pp. 77–93.

[85] Mojtaba Eskandari and Hooman Raesi, « Frequent sub-graph mining for intelligent malware detection », *in*: *Security and Communication Networks* 7.*11* (2014), pp. 1872–1886.

[86] Yong Fang et al., « Semi-supervised malware clustering based on the weight of bytecode and API », *in*: *IEEE Access* 8 (2019), pp. 2313–2326.

[87] Jonathan Fildes, « Stuxnet virus targets and spread revealed », *in*: *BBC News* (Feb. 15, 2011), URL: https://www.bbc.com/news/technology-12465688 (visited on 05/24/2022).

[88] Jim Finkle, « U.S. firm blames Russian 'Sandworm' hackers for Ukraine outage », *in*: *Reuters* (Jan. 8, 2016), URL: https://www.reuters.com/article/us-ukraine-cybersecurity-sandworm-idUSKBN0UM00N20160108 (visited on 05/24/2022).

[89] Andrea Fioraldi, « Symbolic Execution and Debugging Synchronization », Bachelor's thesis, Sapienza University of Rome, Oct. 2018.

[90] Halvar Flake, « Structural Comparison of Executable Objects », *in*: *In Proceedings of the IEEE Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA*, 2004, pp. 161–173.

[91] Luciano Floridi, « The Method of Levels of Abstraction », *in*: *Minds and Machines* 18.*3* (Sept. 2008), 303–329, ISSN: 1572-8641, DOI: 10.1007/s11023-008-9113-7.

[92] Luciano Floridi, Nir Fresco, and Giuseppe Primiero, « On malfunctioning software », *in*: *Synthese* 192.*4* (Apr. 2015), 1199–1220, ISSN: 1573-0964, DOI: 10.1007/s11229-014-0610-3.

[93] Pascal Fontaine, « Satisfiability Modulo Theories: state-of-the-art, contributions, project », Habilitation à diriger des recherches, Université de lorraine, Oct. 2018, URL: https://tel.archives-ouvertes.fr/tel-01968404.

[94] Matt Fredrikson et al., « Synthesizing near-optimal malware specifications from suspicious behaviors », *in*: *2010 IEEE Symposium on Security and Privacy*, IEEE, 2010, pp. 45–60.

[95] Lucas Galante et al., « Machine learning for malware detection: Beyond accuracy rates », *in*: *Anais Estendidos do XIX Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais*, SBC, 2019, pp. 47–56.

[96] Tianchong Gao et al., « Android Malware Detection via Graphlet Sampling », *in*: *IEEE Transactions on Mobile Computing* 18.*12* (2019), pp. 2754–2767, DOI: 10.1109/TMC.2018.2880731.

[97] Daniel B Garrie, Alan F Blakley, and Matthew J Amstrong, « Legal Status of Spyware », *in*: *Fed. Comm. LJ* 59 (2006), p. 157.

[98] Ilir Gashi et al., « A study of the relationship between antivirus regressions and label changes », *in*: *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, 2013, pp. 441–450, DOI: 10.1109/ISSRE.2013.6698897.

[99] Xiuting Ge et al., « AMDroid: Android Malware Detection Using Function Call Graphs », *in*: *2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, 2019, pp. 71–77, DOI: 10.1109/QRS-C.2019.00027.

[100] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins, « Learning to forget: Continual prediction with LSTM », *in*: *Neural computation* 12.*10* (2000), pp. 2451–2471.

[101] Ibrahim Ghafir, Vaclav Prenosil, et al., « Advanced persistent threat attack detection: an overview », *in*: *Int J Adv Comput Netw Secur* 4.*4* (2014), p. 5054.

[102] Babu Nath Giri, Nitin Jyoti, and M Avert, « The emergence of ransomware », *in*: *AVAR, Auckland* (2006).

[103] Michael Glanzberg, « Truth », *in*: *The Stanford Encyclopedia of Philosophy*, ed. by Edward N. Zalta, Summer 2021, Metaphysics Research Lab, Stanford University, 2021.

[104] Patrice Godefroid, Nils Klarlund, and Koushik Sen, « DART: Directed Automated Random Testing », *in*: *SIGPLAN Not.* 40.*6* (June 2005), 213–223, ISSN: 0362-1340, DOI: 10.1145/1064978.1065036, URL: https://doi.org/10.1145/1064978.1065036.

[105] Allen T Goldberg, *On the complexity of the satisfiability problem*, New York University, 1979.

[106] Dan Goodin, « Hackers expose 453,000 credentials allegedly taken from Yahoo service », *in*: *Ars Technica* (July 12, 2012), URL: https://arstechnica.com/information-technology/2012/07/yahoo-service-hacked/ (visited on 05/24/2022).

[107] Ronald L Graham and Pavol Hell, « On the history of the minimum spanning tree problem », *in*: *Annals of the History of Computing* 7.*1* (1985), pp. 43–57.

[108] Kent Griffin et al., « Automatic Generation of String Signatures for Malware Detection », *in*: *Recent Advances in Intrusion Detection*, ed. by Engin Kirda, Somesh Jha, and Davide Balzarotti, Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 101–120, ISBN: 978-3-642-04342-0.

[109] Aditya Grover and Jure Leskovec, « node2vec: Scalable feature learning for networks », *in*: *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, 2016, pp. 855–864.

[110] André Grégio et al., « Ontology for malware behavior: A core model proposal », *in*: *2014 IEEE 23rd International WETICE Conference*, 2014, pp. 453–458, DOI: 10.1109/WETICE.2014.72.

[111] Deepak Gupta and Rinkle Rani, « Improving malware detection using big data and ensemble learning », *in*: *Computers & Electrical Engineering* 86 (2020), p. 106729.

[112] Sanchit Gupta, Harshit Sharma, and Sarvjeet Kaur, « Malware characterization using windows API call sequences », *in*: *International Conference on Security, Privacy, and Applied Cryptography Engineering*, Springer, 2016, pp. 271–280.

[113] Ibai Gurrutxaga et al., « Evaluation of Malware clustering based on its dynamic behaviour », *in*: *Proceedings of the Seventh Australasian Data Mining Conference (AusDM 2008)*, ed. by John F. Roddick et al., vol. 87, CRPIT, Australian Computer Society, 2008, pp. 163–170, URL: http://crpit.scem.westernsydney.edu.au/abstracts/CRPITV87Gurrutxaga.html.

[114] Nicolás Gálvez Ramírez et al., « Towards automated strategies in satisfiability modulo theory », *in*: *European Conference on Genetic Programming*, Springer, 2016, pp. 230–245.

[115] Gavin Hackeling, *Mastering Machine Learning with scikit-learn*, Packt Publishing Ltd, 2017.

[116] Maria Halkidi, Yannis Batistakis, and Michalis Vazirgiannis, « Cluster validity methods: part I », *in*: *ACM Sigmod Record* 31.*2* (2002), pp. 40–45.

[117] Greg Hamerly and Charles Elkan, « Alternatives to the k-means algorithm that find better clusterings », *in*: *Proceedings of the eleventh international conference on Information and knowledge management*, 2002, pp. 600–607.

[118] Steven Strandlund Hansen et al., « An approach for detection and family classification of malware based on behavioral analysis », *in*: *2016 International conference on computing, networking and communications (ICNC)*, IEEE, 2016, pp. 1–5.

[119] Irfan Ul Haq and Juan Caballero, *A Survey of Binary Code Similarity*, 2019, arXiv: 1909.11424 [cs.CR].

[120] Ali Hasan and Richard Fumerton, « Foundationalist Theories of Epistemic Justification », *in*: *The Stanford Encyclopedia of Philosophy*, ed. by Edward N. Zalta, Fall 2018, Metaphysics Research Lab, Stanford University, 2018.

[121] Mehadi Hassen and Philip K Chan, « Scalable function call graph-based malware classification », *in*: *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, 2017, pp. 239–248.

[122] The White House, *International Strategy for Cyberspace*, May 2011, URL: https://obamawhitehouse.archives.gov/sites/default/files/rss_viewer/international_strategy_for_cyberspace.pdf (visited on 05/24/2022).

[123] Dr Jan Hruska, *The Story of Sophos: From 0 to 1200+ employees*, YouTube, URL: https://www.youtube.com/watch?v=n5SeafKv_M0&t=208s (visited on 05/24/2022).

[124] Xin Hu and Kang G Shin, « DUET: integration of dynamic and static analyses for malware clustering with cluster ensembles », *in*: *Proceedings of the 29th annual computer security applications conference*, 2013, pp. 79–88.

[125] Xin Hu et al., « {MutantX-S}: Scalable Malware Clustering Based on Static Features », *in*: *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, 2013, pp. 187–198.

[126] Na Huang et al., « Deep Android Malware Classification with API-Based Feature Graph », *in*: *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, 2019, pp. 296–303, DOI: 10.1109/TrustCom/BigDataSE.2019.00047.

[127] Intelligence Community Assessment (ICA), *Background to "Assessing Russian Activities and Intentions in Recent US Elections": The Analytic Process and Cyber Incident Attribution*, Jan. 6, 2017, URL: https://www.dni.gov/files/documents/ICA_2017_01.pdf (visited on 05/24/2022).

[128] Jonathan Jenkins Ichikawa and Matthias Steup, « The Analysis of Knowledge », *in*: *The Stanford Encyclopedia of Philosophy*, ed. by Edward N. Zalta, Summer 2018, Metaphysics Research Lab, Stanford University, 2018.

[129] Nwokedi C. Idika and Aditya P. Mathur, « A Survey of Malware Detection Techniques », *in*: 2007.

[130] Michael Isikof, « Chinese hacked Obama, McCain campaigns, took internal documents, officials say », *in*: *NBC News* (June 7, 2013), URL: https://www.nbcnews.com/id/wbna52133016 (visited on 05/24/2022).

[131] Rafiqul Islam et al., « Classification of malware based on integrated static and dynamic features », *in*: *Journal of Network and Computer Applications* 36.*2* (2013), pp. 646–656, ISSN: 1084-8045, DOI: https://doi.org/10.1016/j.jnca.2012.10.004, URL: https://www.sciencedirect.com/science/article/pii/S1084804512002214.

[132] International Telecommunication Union (ITU), *2.9 billion people still offline*, Accessed: 2022-05-24, 2021, URL: https://www.itu.int/en/mediacentre/Pages/PR-2021-11-29-FactsFigures.aspx.

[133] Chuntao Jiang, Frans Coenen, and Michele Zito, *A Survey of Frequent Subgraph Mining Algorithms*, 2004.

[134] Haodi Jiang, Turki Turki, and Jason TL Wang, « Dlgraph: Malware detection using deep learning and graph embedding », *in*: *2018 17th IEEE international conference on machine learning and applications (ICMLA)*, IEEE, 2018, pp. 1029–1033.

[135] Andreas Johnsen et al., « Experience Report: Evaluating Fault Detection Effectiveness and Resource Efficiency of the Architecture Quality Assurance Framework and Tool », *in*: *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, 2017, pp. 271–281, DOI: 10.1109/ISSRE.2017.31.

[136] Maximilian Junker et al., « SMT-Based False Positive Elimination in Static Program Analysis », *in*: *Formal Methods and Software Engineering*, ed. by Toshiaki Aoki and Kenji Taguchi, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 316–331, ISBN: 978-3-642-34281-3.

[137] Martin Jurecek, Olha Jurecková, and Róbert Lórencz, « Improving Classification of Malware Families using Learning a Distance Metric. », *in*: *ICISSP*, 2021, pp. 643–652.

[138] Martin Jureček and Róbert Lórencz, « Malware detection using a heterogeneous distance function », *in*: *Computing and Informatics* 37.*3* (2018), pp. 759–780.

[139] Aparna Sunil Kale, Fabio Di Troia, and Mark Stamp, « Malware Classification with Word Embedding Features », *in*: *arXiv preprint arXiv:2103.02711* (2021).

[140] Alex Kantchelian et al., « Better Malware Ground Truth: Techniques for Weighting Anti-Virus Vendor Labels », *in*: *Proceedings of the 8th ACM Workshop on Artificial Intelligence and Security*, AISec '15, Denver, Colorado, USA: Association for Computing Machinery, 2015, 45–56, ISBN: 9781450338264, DOI: 10.1145/2808769.2808780, URL: https://doi.org/10.1145/2808769.2808780.

[141] Youngjoon Ki, Eunjin Kim, and Huy Kang Kim, « A novel approach to detect malware based on API call sequence analysis », *in*: *International Journal of Distributed Sensor Networks* 11.*6* (2015), p. 659101.

[142] Eric Kim, *Everything you wanted to know about the kernel trick*, 2017, URL: https://www.eric-kim.net/eric-kim-net/posts/1/kernel_trick_blog_ekim_12_20_2017.pdf (visited on 10/17/2022).

[143] Sungkwan Kim et al., « A Brief Survey on Rootkit Techniques in Malicious Codes. », *in*: *J. Internet Serv. Inf. Secur.* 2.*3/4* (2012), pp. 134–147.

[144] Joris Kinable and Orestis Kostakis, « Malware classification based on call graph clustering », *in*: *Journal in Computer Virology* 7.*4* (Nov. 1, 2011), pp. 233–245, DOI: 10.1007/s11416-011-0151-y, URL: https://doi.org/10.1007/s11416-011-0151-y.

[145] James C. King, « A New Approach to Program Testing », *in*: *SIGPLAN Not.* 10.*6* (Apr. 1975), 228–233, ISSN: 0362-1340, DOI: 10.1145/390016.808444, URL: https://doi.org/10.1145/390016.808444.

[146] Kevin Kirchner and Stefan Rosenthaler, « bin2llvm: analysis of binary programs using LLVM intermediate representation », *in*: *Proceedings of the 12th International Conference on Availability, Reliability and Security*, 2017, pp. 1–7.

[147] Thomas Kluyver et al., *Jupyter Notebooks-a publishing format for reproducible computational workflows*. Vol. 2016, 2016.

[148] Clemens Kolbitsch et al., « Effective and Efficient Malware Detection at the End Host », *in*: *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, Montreal, Canada: USENIX Association, 2009, 351–366.

[149] Deguang Kong and Guanhua Yan, « Discriminant malware distance learning on structural information for automated malware classification », *in*: *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2013, pp. 1357–1365.

[150] Keith Allen Korcz, « The Epistemic Basing Relation », *in*: *The Stanford Encyclopedia of Philosophy*, ed. by Edward N. Zalta, Spring 2021, Metaphysics Research Lab, Stanford University, 2021.

[151] Orestis Kostakis, « Classy: Fast Clustering Streams of Call-Graphs », *in*: *Data Mining and Knowledge Discovery* 28.*5–6* (Sept. 2014), 1554–1585, ISSN: 1384-5810, DOI: 10.1007/s10618-014-0367-9, URL: https://doi.org/10.1007/s10618-014-0367-9.

[152] Christopher Kruegel et al., « Polymorphic Worm Detection Using Structural Information of Executables », *in*: *Recent Advances in Intrusion Detection*, ed. by Alfonso Valdes and Diego Zamboni, Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 207–226, ISBN: 978-3-540-31779-1.

[153] Tarald O Kvalseth, « Entropy and correlation: Some comments », *in*: *IEEE Transactions on Systems, Man, and Cybernetics* 17.*3* (1987), pp. 517–519.

[154] Kaspersky Lab, *Machine Learning and Human Expertise*, tech. rep., White Paper, 2020, URL: https://media.kaspersky.com/en/business-security/machine-learning-human-expertise.pdf.

[155] Raphael Labaca-Castro et al., « Universal adversarial perturbations for malware », *in*: *arXiv preprint arXiv:2102.06747* (2021).

[156] Bell Labs, *Unix Programmer's Manual (First Edition)*, URL: https://www.bell-labs.com/usr/dmr/www/man21.pdf (visited on 05/30/2022).

[157] 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*, PPREW '13, Rome, Italy: Association for Computing Machinery, 2013, ISBN: 9781450318570, DOI: 10.1145/2430553.2430558, URL: https://doi.org/10.1145/2430553.2430558.

[158] Erik Lambrechts et al., « Round-robin tournaments generated by the circle method have maximum carry-over », *in*: *Mathematical Programming* 172.*1* (2018), pp. 277–302.

[159] Chris Lattner, « LLVM and Clang: Next generation compiler technology », *in*: *The BSD conference*, vol. 5, 2008.

[160] Quoc Le and Tomas Mikolov, « Distributed representations of sentences and documents », *in*: *International conference on machine learning*, PMLR, 2014, pp. 1188–1196.

[161] Jusuk Lee, Kyoochang Jeong, and Heejo Lee, « Detecting Metamorphic Malwares Using Code Graphs », *in*: *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, Sierre, Switzerland: Association for Computing Machinery, 2010, 1970–1977, ISBN: 9781605586397, DOI: 10.1145/1774088.1774505, URL: https://doi.org/10.1145/1774088.1774505.

[162] Wan-Jui Lee, Sergey Verzakov, and Robert PW Duin, « Kernel combination versus classifier combination », *in*: *International Workshop on Multiple Classifier Systems*, Springer, 2007, pp. 22–31.

[163] John Healy Leland McInnes and Steve Astels, *The hdbscan Clustering Library*, Accessed: 2022-03-02, 2016, URL: https://hdbscan.readthedocs.io/en/latest/index.html.

[164] Robert Lemos, « Home Depot estimates data on 56 million cards stolen by cybercriminals », *in*: *Ars Technica* (Sept. 19, 2014), URL: https://arstechnica.com/information-technology/2014/09/home-depot-estimates-data-on-56-million-cards-stolen-by-cybercrimnals/ (visited on 05/24/2022).

[165] Peng Li et al., « On Challenges in Evaluating Malware Clustering », *in*: *RAID 2010, Ottawa, Ontario, Canada, September 15-17, 2010. Proceedings*, ed. by Somesh Jha, Robin Sommer, and Christian Kreibich, vol. 6307, Lecture Notes in Computer Science, Springer, 2010, pp. 238–255, DOI: 10.1007/978-3-642-15512-3_13, URL: https://doi.org/10.1007/978-3-642-15512-3_13.

[166] Yi Li et al., « Symbolic Optimization with SMT Solvers », *in*: *SIGPLAN Not.* 49.*1* (Jan. 2014), 607–618, ISSN: 0362-1340, DOI: 10.1145/2578855.2535857, URL: https://doi.org/10.1145/2578855.2535857.

[167] You Li et al., « Steering symbolic execution to less traveled paths », *in*: *ACM SigPlan Notices* 48.*10* (2013), pp. 19–32.

[168] Andrew Liptak, « The WannaCry ransomware attack has spread to 150 countries », *in*: *The Verge* (May 14, 2017), URL: https://www.theverge.com/2017/5/14/15637888/authorities-wannacry-ransomware-attack-spread-150-countries (visited on 05/24/2022).

[169] Bo Long, Philip S Yu, and Zhongfei Zhang, « A general model for multiple view unsupervised learning », *in*: *Proceedings of the 2008 SIAM international conference on data mining*, SIAM, 2008, pp. 822–833.

[170] Renjie Lu, « Malware detection with lstm using opcode language », *in*: *arXiv preprint arXiv:1906.04593* (2019).

[171] Chi-Keung Luk et al., « Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation », *in*: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, Chicago, IL, USA: Association for Computing Machinery, 2005, 190–200, ISBN: 1595930566, DOI: 10.1145/1065010.1065034, URL: https://doi.org/10.1145/1065010.1065034.

[172] Weilin Luo and Ou Wei, « WAP: SAT-Based Computation of Minimal Cut Sets », *in*: *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, 2017, pp. 146–151, DOI: 10.1109/ISSRE.2017.13.

[173] Guixiang Ma et al., « Deep graph similarity learning: A survey », *in*: *Data Mining and Knowledge Discovery* 35.*3* (2021), pp. 688–725.

[174] Hugo Daniel Macedo and Tayssir Touili, « Mining malware specifications through static reachability analysis », *in*: *European Symposium on Research in Computer Security*, Springer, 2013, pp. 517–535.

[175] Rupak Majumdar and Koushik Sen, « Hybrid concolic testing », *in*: *29th International Conference on Software Engineering (ICSE'07)*, IEEE, 2007, pp. 416–426.

[176] *Malware Attribute Enumeration and Characterization (MAEC)*, Accessed: 2021-12-01, URL: https://maecproject.github.io.

[177] William B March, Parikshit Ram, and Alexander G Gray, « Fast euclidean minimum spanning tree: algorithm, analysis, and applications », *in*: *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2010, pp. 603–612.

[178] John Markoff, « Before the Gunfire, Cyberattacks », *in*: *The New York Times* (Aug. 18, 2008), URL: https://www.nytimes.com/2008/08/13/technology/13cyber.html (visited on 05/24/2022).

[179] Mohammad M Masud, Latifur Khan, and Bhavani Thuraisingham, « A scalable multi-level feature extraction technique to detect malicious executables », *in*: *Information Systems Frontiers* 10.*1* (2008), pp. 33–45.

[180] Mohammad M Masud, Latifur Khan, and Bhavani Thuraisingham, « Feature based techniques for auto-detection of novel email worms », *in*: *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, Springer, 2007, pp. 205–216.

[181] Deborah L McGuinness, Frank Van Harmelen, et al., « OWL web ontology language overview », *in*: *W3C recommendation* 10.*10* (2004), p. 2004.

[182] Leland McInnes and John Healy, « Accelerated Hierarchical Density Based Clustering », *in*: *Data Mining Workshops (ICDMW), 2017 IEEE International Conference on*, IEEE, 2017, pp. 33–42.

[183] Leland McInnes, John Healy, and Steve Astels, « hdbscan: Hierarchical density based clustering. », *in*: *J. Open Source Softw.* 2.*11* (2017), p. 205.

[184] Leland McInnes, John Healy, and Steve Astels, « hdbscan: Hierarchical density based clustering », *in*: *The Journal of Open Source Software* 2.*11* (Mar. 2017), DOI: 10.21105/joss.00205, URL: https://doi.org/10.21105%2Fjoss.00205.

[185] Dirk Merkel, « Docker: Lightweight Linux Containers for Consistent Development and Deployment », *in*: *Linux J.* 2014.*239* (Mar. 2014), ISSN: 1075-3583.

[186] Jason Merrill, « Generic and gimple: A new tree representation for entire functions », *in*: *Proceedings of the 2003 GCC Developers' Summit*, Citeseer, 2003, pp. 171–179.

[187] Microsoft, *Microsoft Malware Classification Challenge (BIG 2015)*, Website, URL: https://www.kaggle.com/c/malware-classification (visited on 05/24/2022).

[188] Microsoft, *Process Monitor (ProcMon)*, 2022, URL: https://docs.microsoft.com/en-us/sysinternals/downloads/procmon.

[189] Tomáš Mikolov, Wen-tau Yih, and Geoffrey Zweig, « Linguistic regularities in continuous space word representations », *in*: *Proceedings of the 2013 conference of the north american chapter of the association for computational linguistics: Human language technologies*, 2013, pp. 746–751.

[190] Tomas Mikolov et al., « Distributed Representations of Words and Phrases and their Compositionality », *in*: *Advances in Neural Information Processing Systems*, ed. by C. J. C. Burges et al., vol. 26, Curran Associates, Inc., 2013, URL: https://proceedings.neurips.cc/paper/2013/file/9aa42b31882ec039965f3c4923ce901b-Paper.pdf.

[191] Tomas Mikolov et al., « Efficient estimation of word representations in vector space », *in*: *arXiv preprint arXiv:1301.3781* (2013).

[192] Jiang Ming et al., « {TaintPipe}: Pipelined Symbolic Taint Analysis », *in*: *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 65–80.

[193] Volodymyr Mnih et al., « Playing atari with deep reinforcement learning », *in*: *arXiv preprint arXiv:1312.5602* (2013).

[194] Aziz Mohaisen, Omar Alrawi, and Manar Mohaisen, « Amal: High-fidelity, behavior-based automated malware analysis and classification », *in*: *computers & security* 52 (2015), pp. 251–266.

[195] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar, *Foundations of machine learning*, MIT press, 2018.

[196] James H. Moor, « Three Myths of Computer Science », *in*: *The British Journal for the Philosophy of Science* 29.3 (1978), pp. 213–222, ISSN: 00070882, 14643537, URL: http://www.jstor.org/stable/687009.

[197] A. Moser, C. Kruegel, and E. Kirda, « Limits of Static Analysis for Malware Detection », *in*: *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, Dec. 2007, pp. 421–430, DOI: 10.1109/ACSAC.2007.21.

[198] Leonardo de Moura and Grant Olney Passmore, « The Strategy Challenge in SMT Solving », *in*: *Automated Reasoning and Mathematics: Essays in Memory of William W. McCune*, ed. by Maria Paola Bonacina and Mark E. Stickel, Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 15–44, ISBN: 978-3-642-36675-8, DOI: 10.1007/978-3-642-36675-8_2, URL: https://doi.org/10.1007/978-3-642-36675-8_2.

[199] David A Mundie and David M Mcintire, « An ontology for malware analysis », *in*: *2013 International Conference on Availability, Reliability and Security*, IEEE, 2013, pp. 556–558.

[200] Annamalai Narayanan et al., « graph2vec: Learning distributed representations of graphs », *in*: *arXiv preprint arXiv:1707.05005* (2017).

[201] NATO, *Wales Summit Declaration*, Sept. 5, 2014, URL: https://www.nato.int/cps/en/natohq/official_texts_112964.htm (visited on 05/24/2022).

[202] NATO, *Warsaw Summit Key Decisions*, Feb. 6, 2017, URL: https://www.nato.int/nato_static_fl2014/assets/pdf/pdf_2017_02/20170206_1702-factsheet-warsaw-summit-key-en.pdf (visited on 05/24/2022).

[203] Shah Nazir, Sara Shahzad, and Neelam Mukhtar, « Software Birthmark Design and Estimation: A Systematic Literature Review », *in*: *Arabian Journal for Science and Engineering* 44.*4* (Apr. 1, 2019), pp. 3905–3927, DOI: 10.1007/s13369-019-03718-9, URL: https://doi.org/10.1007/s13369-019-03718-9.

[204] Nicholas Nethercote and Julian Seward, « Valgrind: a framework for heavyweight dynamic binary instrumentation », *in*: *ACM Sigplan notices* 42.*6* (2007), pp. 89–100.

[205] Stavros D. Nikolopoulos and Iosif Polenakis, « A graph-based model for malware detection and classification using system-call groups », *in*: *Journal of Computer Virology and Hacking Techniques* 13.*1* (Feb. 2017), 29–46, ISSN: 2263-8733, DOI: 10.1007/s11416-016-0267-1.

[206] Lamine Noureddine et al., « SE-PAC: A Self-Evolving PAcker Classifier against rapid packers evolution », *in*: *Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy*, 2021, pp. 281–292.

[207] Leo Obrst, Penny Chase, and Richard Markeloff, « Developing an Ontology of the Cyber Security Domain. », *in*: *STIDS*, Citeseer, 2012, pp. 49–56.

[208] Angelo Oliveira and R Sassi, « Behavioral malware detection using deep graph convolutional neural networks », *in*: *TechRxiv* (2019).

[209] Paul Oppenheim and Hilary Putnam, « Unity of Science as a Working Hypothesis », *in*: *Minnesota Studies in the Philosophy of Science* 2 (1958), pp. 3–36.

[210] UN Secretary-General's High level Panel, *The Age of Digital Interdependence*, Accessed: 2022-05-24, 2019, URL: https://www.un.org/en/pdfs/DigitalCooperation-report-for%20web.pdf.

[211] Younghee Park, D.s Reeves, and Mark Stamp, « Deriving common malware behavior through graph clustering », *in*: *Computers & Security* 39 (Nov. 2013), pp. 419–430, DOI: 10.1016/j.cose.2013.09.006.

[212] Younghee Park et al., « Fast Malware Classification by Automated Behavioral Graph Matching », *in*: *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*, CSIIRW '10, Oak Ridge, Tennessee, USA: Association for Computing Machinery, 2010, ISBN: 9781450300179, DOI: 10.1145/1852666.1852716, URL: https://doi.org/10.1145/1852666.1852716.

[213] Sunhera Paul and Mark Stamp, « Word embedding techniques for malware evolution detection », *in*: *Malware Analysis Using Artificial Intelligence and Deep Learning*, Springer, 2021, pp. 321–343.

[214] Fabian Pedregosa et al., « Scikit-learn: Machine learning in Python », *in*: *the Journal of machine Learning research* 12 (2011), pp. 2825–2830.

[215] Jeffrey Pennington, Richard Socher, and Christopher D Manning, « Glove: Global vectors for word representation », *in*: *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543.

[216] Harold Emanuel Petersen and Rein Turn, « System implications of information privacy », *in*: *Proceedings of the April 18-20, 1967, spring joint computer conference*, 1967, pp. 291–300.

[217] Rob Pike, *The go programming language*, 2009, URL: https://9p.io/sources/contrib/ericvh/go-plan9/doc/go_talk-20091030.pdf (visited on 10/17/2022).

[218] Daniel Plohmann et al., « Malpedia: a collaborative effort to inventorize the malware landscape », *in*: *Proceedings of the Botconf*, 2017.

[219] Pavel Polityuk and Steve Holland, « Cyberattack hits Ukraine as U.S. warns Russia could be prepping for war », *in*: *Reuters* (Jan. 14, 2022), URL: https://www.reuters.com/world/europe/expect-worst-ukraine-hit-by-cyberattack-russia-moves-more-troops-2022-01-14/ (visited on 05/24/2022).

[220] Mila Dalla Preda et al., « A Semantics-based Approach to Malware Detection », *in*: *SIGPLAN Not.* 42.*1* (Jan. 2007), pp. 377–388, ISSN: 0362-1340, DOI: 10.1145/1190215.1190270, URL: http://doi.acm.org/10.1145/1190215.1190270.

[221] Robert Clay Prim, « Shortest connection networks and some generalizations », *in*: *The Bell System Technical Journal* 36.*6* (1957), pp. 1389–1401.

[222] Giuseppe Primiero, « Information in the philosophy of computer science », *in*: *The Routledge handbook of philosophy of information* (2016), pp. 90–106.

[223] Giuseppe Primiero, Frida J. Solheim, and Jonathan M. Spring, « On Malfunction, Mechanisms and Malware Classification », *in*: *Philosophy & Technology* 32.*2* (June 2019), 339–362, ISSN: 2210-5441, DOI: 10.1007/s13347-018-0334-2.

[224] GNU Project, *GNU Debugger (GDB)*, 2022, URL: https://www.gnu.org/software/gdb.

[225] Yara Project, *Yara documentation*, Website, Accessed: 2022-02-03, URL: https://yara.readthedocs.io/.

[226] Xueheng Qiu et al., « Ensemble deep learning for regression and time series forecasting », *in*: *2014 IEEE symposium on computational intelligence in ensemble learning (CIEL)*, IEEE, 2014, pp. 1–6.

[227] NGUYEN Anh Quynh and DANG Hoang Vu, « Unicorn: Next generation cpu emulator framework », *in*: *BlackHat USA* 476 (2015).

[228] Team Radare, *Radare2 github repository*, 2020, URL: https://github.com/radareorg/radare2.

[229] T Ramraj and R Prabhakar, « Frequent subgraph mining algorithms–a survey », *in*: *Procedia Computer Science* 47 (2015), pp. 197–204.

[230] Nidhi Rastogi et al., « Malont: An ontology for malware threat intelligence », *in*: *International Workshop on Deployable Machine Learning for Security Defense*, Springer, 2020, pp. 28–44.

[231] hex rays, *IDA Free*, 2022, URL: https://hex-rays.com/ida-free/.

[232] Saif Ur Rehman, Asmat Ullah Khan, and Simon Fong, « Graph mining: A survey of graph mining techniques », *in*: *Seventh International Conference on Digital Information Management (ICDIM 2012)*, IEEE, 2012, pp. 88–92.

[233] Radim Rehurrek, Petr Sojka, et al., « Gensim—statistical semantics in python », *in*: *Retrieved from genism. org* (2011).

[234] Microsoft Research, *Z3*, URL: https://www.microsoft.com/en-us/research/project/z3-3/.

[235] Konrad Rieck et al., « Automatic analysis of malware behavior using machine learning », *in*: *Journal of Computer Security* 19.*4* (2011), pp. 639–668.

[236] Konrad Rieck et al., « Learning and Classification of Malware Behavior », *in*: *Detection of Intrusions and Malware, and Vulnerability Assessment*, ed. by Diego Zamboni, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 108–125, ISBN: 978-3-540-70542-0.

[237] Lior Rokach and Oded Maimon, « Clustering Methods », *in*: Jan. 2005, pp. 321–352, DOI: 10.1007/0-387-25465-X_15.

[238] Xin Rong, « word2vec parameter learning explained », *in*: *arXiv preprint arXiv:1411.2738* (2014).

[239] Andrew Rosenberg and Julia Hirschberg, « V-measure: A conditional entropy-based external cluster evaluation measure », *in*: *Proceedings of the 2007 joint conference on empirical methods in natural language processing and computational natural language learning (EMNLP-CoNLL)*, 2007, pp. 410–420.

[240] Florian Roth, *How to Write Simple but Sound Yara Rules*, 2015, URL: https://www.nextron-systems.com/2015/02/16/write-simple-sound-yara-rules/.

[241] Florian Roth, *yarGen*, 2013, URL: https://github.com/Neo23x0/yarGen.

[242] Peter J. Rousseeuw, « Silhouettes: A graphical aid to the interpretation and validation of cluster analysis », *in*: *Journal of Computational and Applied Mathematics* 20 (1987), pp. 53–65, ISSN: 0377-0427, DOI: https://doi.org/10.1016/0377-0427(87)90125-7, URL: https://www.sciencedirect.com/science/article/pii/0377042787901257.

[243] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams, « Learning representations by back-propagating errors », *in*: *nature* 323.*6088* (1986), pp. 533–536.

[244] Per Runeson and Martin Höst, « Guidelines for conducting and reporting case study research in software engineering », *in*: *Empirical software engineering* 14.*2* (2009), pp. 131–164.

[245] Mark E Russinovich, David A Solomon, and Alex Ionescu, *Windows internals, part 2*, Pearson Education, 2012.

[246] Barbara G. Ryder, « Constructing the Call Graph of a Program », *in*: *IEEE Trans. Softw. Eng.* 5.*3* (May 1979), 216–226, ISSN: 0098-5589, DOI: 10.1109/TSE.1979.234183, URL: https://doi.org/10.1109/TSE.1979.234183.

[247] Imtithal A Saeed, Ali Selamat, and Ali MA Abuagoub, « A survey on malware and malware detection systems », *in*: *International Journal of Computer Applications* 67.*16* (2013).

[248] Aleieldin Salem, Sebastian Banescu, and Alexander Pretschner, « Maat: Automatically Analyzing VirusTotal for Accurate Labeling and Effective Malware Detection », *in*: *arXiv preprint arXiv:2007.00510* (2020).

[249] David E. Sanger, « Obama Order Sped Up Wave of Cyberattacks Against Iran », *in*: *The New York Times* (June 1, 2012), URL: https://www.nytimes.com/2012/06/01/world/middleeast/obama-ordered-wave-of-cyberattacks-against-iran.html (visited on 05/24/2022).

[250] Florent Saudel and Jonathan Salwan, « Triton: A dynamic symbolic execution framework », *in*: *Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes*, 2015, pp. 31–54.

[251] Prateek Saxena et al., « Loop-extended symbolic execution on binary programs », *in*: *Proceedings of the eighteenth international symposium on Software testing and analysis*, 2009, pp. 225–236.

[252] Hossein Sayadi et al., « Ensemble learning for effective run-time hardware-based malware detection: A comprehensive analysis and classification », *in*: *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, IEEE, 2018, pp. 1–6.

[253] Robert Schaefer, « The Epistemology of Computer Security », *in*: *SIGSOFT Softw. Eng. Notes* 34.*6* (Dec. 2009), 8–10, ISSN: 0163-5948, DOI: 10.1145/1640162.1655274, URL: https://doi.org/10.1145/1640162.1655274.

[254] Jonathan Schaffer, « Is There a Fundamental Level? », *in*: *Noûs* 37.*3* (2003), pp. 498–517, DOI: 10.1111/1468-0068.00448.

[255] MANDIANT: Intelligent Information Security, *OpenIOC*, Accessed: 2021-12-01, URL: http://www.mandiant.com.

[256] A. A. Selçuk, F. Orhan, and B. Batur, « Undecidable problems in malware analysis », *in*: *2017 12th International Conference for Internet Technology and Secured Transactions (ICITST)*, 2017, pp. 494–497, DOI: 10.23919/ICITST.2017.8356458.

[257] Koushik Sen et al., « MultiSE: Multi-Path Symbolic Execution Using Value Summaries », *in*: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, Bergamo, Italy: Association for Computing Machinery, 2015, 842–853, ISBN: 9781450336758, DOI: 10.1145/2786805.2786830, URL: https://doi.org/10.1145/2786805.2786830.

[258] Shanhu Shang et al., « Detecting malware variants via function-call graph similarity », *in*: Nov. 2010, pp. 113 –120, DOI: 10.1109/MALWARE.2010.5665787.

[259] Geoff Shaw, « Spyware & Adware: the risks facing businesses », *in*: *Network security* 2003.*9* (2003), pp. 12–14.

[260] John F Shoch and Jon A Hupp, « The "worm" programs—early experience with a distributed computation », *in*: *Communications of the ACM* 25.*3* (1982), pp. 172–180.

[261] Yan Shoshitaishvili et al., « Sok:(state of) the art of war: Offensive techniques in binary analysis », *in*: *2016 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2016, pp. 138–157.

[262] Michael Sikorski and Andrew Honig, *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*, 1st, San Francisco, CA, USA: No Starch Press, 2012, ISBN: 1593272901, 9781593272906.

[263] Jagsir Singh and Jaswinder Singh, « A survey on machine learning-based malware detection in executable files », *in*: *Journal of Systems Architecture* 112 (2021), p. 101861, ISSN: 1383-7621, DOI: https://doi.org/10.1016/j.sysarc.2020.101861, URL: https://www.sciencedirect.com/science/article/pii/S1383762120301442.

[264] Steven Skiena, *Implementing discrete mathematics - combinatorics and graph theory with Mathematica*. Addison-Wesley, 1990, pp. I–VIII, ISBN: 978-0-201-50943-4.

[265] Alex J Smola and Bernhard Schölkopf, « A tutorial on support vector regression », *in*: *Statistics and computing* 14.*3* (2004), pp. 199–222.

[266] Fu Song and Tayssir Touili, « Pushdown model checking for malware detection », *in*: *International Journal on Software Tools for Technology Transfer* 16.*2* (2014), pp. 147–173.

[267] Open Source, *Cuckoo Sandbox*, 2010, URL: https://www.cuckoosandbox.org.

[268] Lars St, Svante Wold, et al., « Analysis of variance (ANOVA) », *in*: *Chemometrics and intelligent laboratory systems* 6.*4* (1989), pp. 259–272.

[269] Richard M Stallman, « GNU compiler collection internals », *in*: *Free Software Foundation* (2002).

[270] Luke Stark and Jevan Hutson, « Physiognomic artificial intelligence », *in*: *Available at SSRN 3927300* (2021).

[271] Nick Stephens et al., « Driller: Augmenting fuzzing through selective symbolic execution. », *in*: *NDSS*, vol. 16, 2016, 2016, pp. 1–16.

[272] Matthias Steup and Ram Neta, « Epistemology », *in*: *The Stanford Encyclopedia of Philosophy*, ed. by Edward N. Zalta, Fall 2020, Metaphysics Research Lab, Stanford University, 2020.

[273] Alexander Strehl and Joydeep Ghosh, « Cluster ensembles - A Knowledge Reuse Framework for Combining Multiple Partitions », *in*: *Journal of machine learning research* 3.*Dec* (2002), pp. 583–617.

[274] Peter Suber, « What is software? », *in*: *The Journal of Speculative Philosophy* (1988), pp. 89–119.

[275] Bowen Sun et al., « Malware family classification method based on static feature extraction », *in*: *2017 3rd IEEE International Conference on Computer and Communications (ICCC)*, IEEE, 2017, pp. 507–513.

[276] Morton Swimmer, *Towards An Ontology of Malware Classes*, Accessed: 2021-12-01, 2009, URL: http://www.scribd.com/doc/24058261/Towards-an-Ontology-of-Malware-Classes.

[277] Peter Szor, *The Art of Computer Virus Research and Defense*, Addison-Wesley Professional, 2005, ISBN: 0321304543.

[278] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar, *Introduction to data mining*, Pearson Education India, 2016.

[279] Andrew S. Tanenbaum, *Structured computer organization, 5th Edition*, Pearson Education, 2006, ISBN: 978-0-13-148521-1.

[280] AV TEST, *Malware Statistics*, Website, URL: https://www.av-test.org/en/statistics/malware/ (visited on 05/24/2022).

[281] Ken Thompson, « Reflections on Trusting Trust », *in*: *Commun. ACM* 27.*8* (Aug. 1984), 761–763, ISSN: 0001-0782, DOI: 10.1145/358198.358210, URL: https://doi.org/10.1145/358198.358210.

[282] Ronghua Tian et al., « Differentiating malware from cleanware using behavioural analysis », *in*: *2010 5th international conference on malicious and unwanted software*, Ieee, 2010, pp. 23–30.

[283] Flavio Toffalini et al., « Detection of Masqueraders Based on Graph Partitioning of File System Access Events », *in*: *2018 IEEE Security and Privacy Workshops (SPW)*, 2018, pp. 217–227, DOI: 10.1109/SPW.2018.00037.

[284] Philipp Trinius et al., « A malware instruction set for behavior-based analysis », *in*: (2009).

[285] Raymond Turner, « Specification », *in*: *Minds and Machines* 21.*2* (May 2011), 135–152, ISSN: 1572-8641, DOI: 10.1007/s11023-011-9239-x.

[286] Grigorios Tzortzis and Aristidis Likas, « Convex mixture models for multi-view clustering », *in*: *International Conference on Artificial Neural Networks*, Springer, 2009, pp. 205–214.

[287] Daniele Ucci, Leonardo Aniello, and Roberto Baldoni, « Survey of machine learning techniques for malware analysis », *in*: *Computers & Security* 81 (2019), pp. 123–147, ISSN: 0167-4048, DOI: https://doi.org/10.1016/j.cose.2018.11.001, URL: https://www.sciencedirect.com/science/article/pii/S0167404818303808.

[288] Jamal Uddin, Rozaida Ghazali, and Mustafa Mat Deris, « Does number of clusters effect the purity and entropy of clustering? », *in*: *International Conference on Soft Computing and Data Mining*, Springer, 2016, pp. 355–365.

[289] Xabier Ugarte-Pedrero, Mariano Graziano, and Davide Balzarotti, « A close look at a daily dataset of malware samples », *in*: *ACM Transactions on Privacy and Security (TOPS)* 22.*1* (2019), pp. 1–30.

[290] Xabier Ugarte-Pedrero et al., « SoK: Deep packer inspection: A longitudinal study of the complexity of run-time packers », *in*: *2015 IEEE Symposium on Security and Privacy*, IEEE, 2015, pp. 659–673.

[291] Sandro Vega-Pons and José Ruiz-Shulcloper, « A survey of clustering ensemble algorithms », *in*: *International Journal of Pattern Recognition and Artificial Intelligence* 25.*03* (2011), pp. 337–372.

[292] Swapna Vemparala et al., « Malware detection using dynamic birthmarks », *in*: *Proceedings of the 2016 ACM on international workshop on security and privacy analytics*, 2016, pp. 41–46.

[293] Nguyen Xuan Vinh, Julien Epps, and James Bailey, « Information theoretic measures for clusterings comparison: Variants, properties, normalization and correction for chance », *in*: *The Journal of Machine Learning Research* 11 (2010), pp. 2837–2854.

[294] VirusTotal, *Malware Hunting*, Website, Accessed: 2022-02-03, URL: https://www.virustotal.com/#/hunting-overview.

[295] John Von Neumann, « Theory and organization of complicated automata », *in*: *Burks (1966)* (1949), pp. 29–87.

[296] Roland Vossen, *Win 95 Source code in c!! (Archive)*, URL: https://groups.google.com/g/rec.games.programmer/c/hqQmsBR0ltg (visited on 05/30/2022).

[297] Jane Wakefield, « Tax software blamed for cyber-attack spread », *in*: *BBC News* (June 28, 2017), URL: https://www.bbc.com/news/technology-40428967 (visited on 05/24/2022).

[298] Xinran Wang et al., « Behavior Based Software Theft Detection », *in*: *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, Chicago, Illinois, USA: Association for Computing Machinery, 2009, 280–290, ISBN: 9781605588940, DOI: 10.1145/1653662.1653696, URL: https://doi.org/10.1145/1653662.1653696.

[299] World Economic Forum (WEF), *The Global Risks Report 2010*, Accessed: 2022-05-24, 2010, URL: https://www3.weforum.org/docs/WEF_Global_Risks_Report_2010.pdf.

[300] World Economic Forum (WEF), *The Global Risks Report 2010*, Accessed: 2022-05-24, 2017, URL: https://www3.weforum.org/docs/GRR17_Report_web.pdf.

[301] World Economic Forum (WEF), *The Global Risks Report 2012*, Accessed: 2022-05-24, 2012, URL: https://www3.weforum.org/docs/WEF_GlobalRisks_Report_2012.pdf.

[302] World Economic Forum (WEF), *The Global Risks Report 2014*, Accessed: 2022-05-24, 2014, URL: https://www3.weforum.org/docs/WEF_GlobalRisks_Report_2014.pdf.

[303] World Economic Forum (WEF), *The Global Risks Report 2018*, Accessed: 2022-05-24, 2018, URL: https://www3.weforum.org/docs/WEF_GRR18_Report.pdf.

[304] World Economic Forum (WEF), *The Global Risks Report 2019*, Accessed: 2022-05-24, 2019, URL: https://www3.weforum.org/docs/WEF_Global_Risks_Report_2019.pdf.

[305] World Economic Forum (WEF), *The Global Risks Report 2020*, Accessed: 2022-05-24, 2020, URL: https://www3.weforum.org/docs/WEF_Global_Risk_Report_2020.pdf.

[306] World Economic Forum (WEF), *The Global Risks Report 2021*, Accessed: 2022-05-24, 2021, URL: https://www3.weforum.org/docs/WEF_The_Global_Risks_Report_2021.pdf.

[307] World Economic Forum (WEF), *The Global Risks Report 2022*, Accessed: 2022-05-24, 2022, URL: https://www3.weforum.org/docs/WEF_The_Global_Risks_Report_2022.pdf.

[308] Kilian Weinberger et al., « Feature hashing for large scale multitask learning », *in*: *Proceedings of the 26th annual international conference on machine learning*, 2009, pp. 1113–1120.

[309] Carl J. Wenning, « Scientific epistemology: How scientists know what they know », *in*: 2009.

[310] Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo de Moura, « Efficiently solving quantified bit-vector formulas », *in*: *Formal Methods in System Design* 42.*1* (Feb. 1, 2013), pp. 3–23, DOI: 10.1007/s10703-012-0156-2, URL: https://doi.org/10.1007/s10703-012-0156-2.

[311] WithSecure, *Brain: Searching for the first PC virus in Pakistan*, YouTube, URL: https://www.youtube.com/watch?v=lnedOWfPKT0 (visited on 05/30/2022).

[312] Lingfei Wu et al., « A novel malware variants detection method based On function-call graph », *in*: Jan. 2013, pp. 1–5, ISBN: 978-1-4799-1660-3, DOI: 10.1109/ANTHOLOGY.2013.6784887.

[313] Xiao-Ling Xia et al., « Malware detection based on ontology », *in*: *2017 International Conference on Machine Learning and Cybernetics (ICMLC)*, vol. 1, IEEE, 2017, pp. 21–26.

[314] Dongpeng Xu, Jiang Ming, and Dinghao Wu, « Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping », *in*: *2017 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2017, pp. 921–937.

[315] Lifan Xu et al., « Dynamic Android Malware Classification Using Graph-Based Representations », *in*: *2016 IEEE 3rd International Conference on Cyber Security and Cloud Computing (CSCloud)*, 2016, pp. 220–231, DOI: 10.1109/CSCloud.2016.27.

[316] Babak Yadegari and Saumya Debray, « Symbolic execution of obfuscated code », *in*: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2015, pp. 732–744.

[317] Jinpei Yan, Yong Qi, and Qifan Rao, « Detecting malware with an ensemble method based on deep neural network », *in*: *Security and Communication Networks* 2018 (2018).

[318] Xifeng Yan and Jiawei Han, « gSpan: Graph-Based Substructure Pattern Mining. », *in*: *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002)*, Maebashi City, Japan: IEEE Computer Society, Dec. 2002, pp. 721–724, ISBN: 0-7695-1754-4, URL: http://dblp.uni-trier.de/db/conf/icdm/icdm2002.html#YanH02.

[319] *Yara - VirusTotal*, website, URL: https://virustotal.github.io/yara/.

[320] Yanfang Ye et al., « Automatic malware categorization using cluster ensemble », *in*: *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2010, pp. 95–104.

[321] Adam Young and Moti Yung, « Cryptovirology: Extortion-based security threats and countermeasures », *in*: *Proceedings 1996 IEEE Symposium on Security and Privacy*, IEEE, 1996, pp. 129–140.

[322] Mikhail Zaslavskiy, Francis Bach, and Jean-Philippe Vert, « A Path Following Algorithm for the Graph Matching Problem », *in*: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 31.*12* (2009), pp. 2227–2242, DOI: 10.1109/TPAMI.2008.245.

[323] Kim Zetter, « A Mysterious Hacker Group Is On a Supply Chain Hijacking Spree », *in*: *WIRED* (May 3, 2019), URL: https://www.wired.com/story/barium-supply-chain-hackers/ (visited on 05/30/2022).

[324] Kim Zetter, « That Insane, $81M Bangladesh Bank Heist? Here's What We Know », *in*: *WIRED* (May 17, 2016), URL: https://www.wired.com/2016/05/insane-81m-bangladesh-bank-heist-heres-know/ (visited on 05/24/2022).

[325] Jie Zhang et al., « Malware Detection using CNN via Word Embedding », *in*: *2021 8th International Conference on Dependable Systems and Their Applications (DSA)*, IEEE, 2021, pp. 600–607.

[326] Y. Zhang et al., « Based on Multi-features and Clustering Ensemble Method for Automatic Malware Categorization », *in*: *2017 IEEE Trustcom/BigDataSE/ICESS*, 2017, pp. 73–82, DOI: 10.1109/Trustcom/BigDataSE/ICESS.2017.222.

[327] Ying Zhao and George Karypis, « Criterion functions for document clustering: Experiments and analysis », *in*: (2001).

[328] Qinghai Zheng et al., « Feature concatenation multi-view subspace clustering », *in*: *Neurocomputing* 379 (2020), pp. 89–102.

[329] Hao Zhou et al., « Analysis of Android Malware Family Characteristic Based on Isomorphism of Sensitive API Call Graph », *in*: *2017 IEEE Second International Conference on Data Science in Cyberspace (DSC)*, 2017, pp. 319–327, DOI: 10.1109/DSC.2017.77.

[330] zynamics, *BinDiff*, https://www.zynamics.com/bindiff.html.

CHAPTER 6

APPENDIX

6.1 Basic Formal Ontology (BFO)

This section intends to provide a short background on some of the main concepts defined in BFO, depicted in figure 6.1. For greater details on BFO, refer to the official specification and user guide [5].

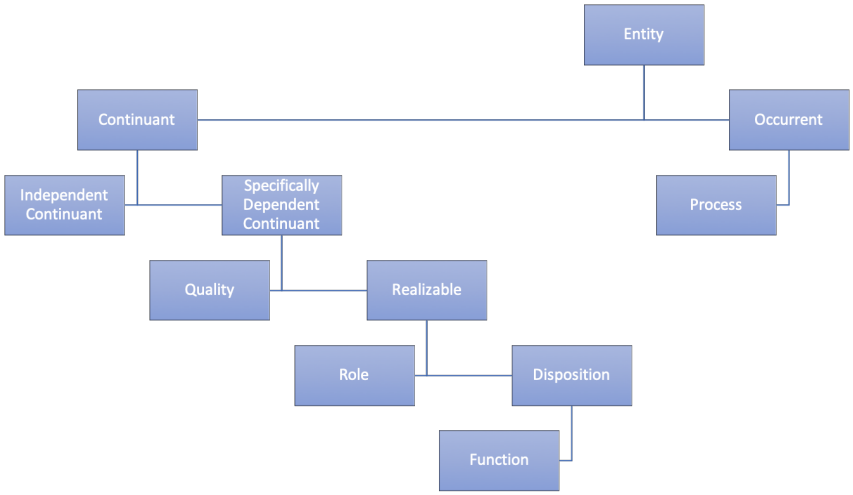


Figure 6.1 – Synoptic outline of BFO hierarchy.

In BFO, *Entity* is the most basic concept representing anything that exists. These entities are divided into *Instances* and *Universals*, where the former refers to a particular case of the latter. Entities are also divided into *Continuant* and *Occurent*, where the former "exists in full at any time in which it exists at all, persists through time while maintaining its identity and has no temporal parts" (e.g. a thesis) and the latter "has temporal parts and that happens, unfolds or develops through time" (e.g. the process of writing a thesis).

An *Independent Continuant* is "a bearer of *quality* and *realizable entities*, in which other

entities inhere and which itself cannot inhere in anything" (e.g. a thesis), whereas a *Dependent Continuant* is "dependent on one or other *independent continuant* bearers or inheres in or is borne by other entities" (e.g. the number of pages of this thesis). In other words, *Dependent Continuants* are *qualities*, *functions*, *roles*, and *dispositions* —generically, *Dependent Continuants* are "attributes" —that can exist only insofar as they are borne or carried by *Independent Continuants*.

A *Generically Dependent Continuant* is "dependent on one or other *independent continuant* bearers" (e.g. the content of this thesis), whereas a *specifically dependent continuant* "inheres in or is borne by other entities" (e.g. a printed version of this thesis).

BFO defines two types of *specifically dependent continuant*: *quality* and *realizable*. *Quality* is a *specifically dependent continuant* that "is exhibited if it inheres in an *entity* or *entities* at all (a categorical property)" (e.g. the number of words in this thesis). In contrast, a *Realizable Entity* is a *specifically dependent continuant* that "inheres in *continuant entities* and is exhibited in full at every time in which it inheres in an *entity* or group of *entities*. The exhibition or actualization of a *realizable entity* is a particular manifestation, functioning or process that occurs under certain circumstances" (e.g. the use of this thesis as reference by another work).

BFO defines two types of *Realizable Entity*: *role* and *disposition*. *Role* is a "manifestation of which brings about some result or end that is not essential to a *continuant* in virtue of the kind of thing that it is but that can be served or participated in by that kind of *continuant* in some kinds of natural, social or institutional contexts" (e.g. the use of this thesis as reference by another work). *Disposition* is a "*realizable entity* that essentially causes a specific process or transformation in the *continuant* in which it inheres, under specific circumstances and in conjunction with the laws of nature." (e.g. the number of pages of this thesis, given the font size and policy). Essentially, *role* is an *extrinsic* (or *externally-grounded*) *realizable entity*, whereas *disposition* is an *intrinsic* (or *internally-grounded*) *realizable entity*.

BFO also defines *Function* as a special kind of *realizable entity*, which consists in "the manifestation of which is an **essentially end-directed activity** of a *continuant* entity in virtue of that *continuant* entity being a specific kind of *entity* in the kind or kinds of contexts that it is made for" (e.g. the function of a thesis to present the research about some topic). To further clarify the BFO notion of *function*, we can think about the following example: "the function of a hammer is to drive in nails"; although it is possible to play drums with a hammer, its *essential end-directed activity* is driving nails.

6.2 Logistic Equation and the Sigmoid Function

The logistic function was introduced by Pierre François Verhulst as a model of population growth. It is modeled as a first-order non-linear ordinary differential equation, known as the *logistic equation*:

$$\frac{d}{dt}f(t) = f(t)(a - bf(t)) \tag{6.1}$$

The term a represents the positive growth rate that takes place when the birth rate is higher than the death rate, whereas the term b represents the slow down in the growth rate due to competition which increases as the population grows. Implicitly, both a and b are assumed to be positive values.

Defining $z(t)$ as $z(t) = 1/f(t)$, we have:

$$\frac{d}{dt}z(t) = -\frac{1}{f(t)^2}\frac{d}{dt}f(t) = -\frac{1}{f(t)^2}f(t)(a - bf(t)) = -az(t) + b \tag{6.2}$$

The solution to equation 6.2 is $z(t) = \frac{b}{a} + c_0\, e^{-at}$, therefore the solution to equation 6.1 is:

$$f(t) = \frac{a}{c_0\, e^{-at} + b} \tag{6.3}$$

For $c_0 > 0$, $f(t)$ is strictly increasing monotonic because $f'(t) = \frac{a^2 e^{-at}}{c_0\, e^{-at} + b} > 0$. We notice that $f(t \to -\infty) = 0$, whereas $f(t \to \infty) = a/b$.

Furthermore, $f'(t) = f(t)(a - bf(t))$ is zero for $f(t) = 0$ or $f(t) = a/b$, which happens for $t \to -\infty$ and $t \to \infty$. Those points where the derivative are zero are the *steady states* of the logistic equation.

By computing a second derivative of $f(t)$ we obtain:

$$\begin{aligned} f''(t) &= \frac{d}{dt}(af(t) - bf(t)) = af'(t) - 2bf(t)f'(t) \\ &= f(t)(a - bf(t))(a - 2bf(t)) \end{aligned}$$

The values of t for which $f''(t) = 0$ are those for which $f(t) = 0$, $f(t) = a/b$ or $f(t) = a/2b$. As seen before, the first two cases correspond to $t \to -\infty$ and $t \to \infty$. For the third case, we refer to this point as $t = \tau$:

$$
\begin{aligned}
f(\tau) = \frac{a}{2b} &\Rightarrow \frac{a}{c_0\ e^{-a\tau} + b} = \frac{a}{2b} \\
&\Rightarrow c_0\ e^{-a\tau} + b = 2b \\
&\Rightarrow \tau = \frac{ln(c_0) - ln(b)}{a}
\end{aligned}
$$

We now show that $f(t)$ is rotationally symmetrical about $(\tau,\ f(\tau))$, i.e. $f(\tau+\alpha) - f(\tau) = f(\tau) - f(t-\alpha)$.

For this we first expand $f(t+\alpha)$:

$$
\begin{aligned}
f(\tau+\alpha) &= \frac{a}{c_0\ e^{-a(\tau+\alpha)} + b} \\
&= \frac{a}{c_0\ e^{-a\tau - a\alpha} + b} \\
&= \frac{a}{\frac{c_0\ e^{-a\tau}}{e^{a\alpha}} + b} = \frac{ae^{a\alpha}}{c_0\ e^{-a\tau} + be^{a\alpha}} \\
&= \frac{ae^{a\alpha}}{c_0\ e^{-a(\frac{ln(c_0)-ln(b)}{a}))} + be^{a\alpha}} = \frac{ae^{a\alpha}}{c_0\ \frac{e^{ln(b)}}{e^{ln(c_0)}} + be^{a\alpha}} \\
&= \frac{ae^{a\alpha}}{b + be^{a\alpha}} = \frac{a/b}{1 + e^{-a\alpha}}
\end{aligned}
$$

Now we expand $f(t-\alpha)$:

$$
\begin{aligned}
f(\tau-\alpha) &= \frac{a}{c_0\ e^{-a(\tau-\alpha)} + b} \\
&= \frac{a}{c_0\ e^{-a\tau + a\alpha} + b} \\
&= \frac{a}{e^{a\alpha}\ c_0\ e^{-a\tau} + b} = \frac{ae^{-a\alpha}}{c_0\ e^{-a\tau} + be^{-a\alpha}} \\
&= \frac{ae^{-a\alpha}}{c_0\ e^{-a(\frac{ln(c_0)-ln(b)}{a}))} + be^{-a\alpha}} = \frac{ae^{-a\alpha}}{c_0\ \frac{e^{ln(b)}}{e^{ln(c_0)}} + be^{-a\alpha}} \\
&= \frac{ae^{-a\alpha}}{b + be^{-a\alpha}} = \frac{a/b}{1 + e^{a\alpha}}
\end{aligned}
$$

And we compute $f(\tau+\alpha)-f(\tau)$ and $f(\tau)-f(t-\alpha)$ as follows:

$$f(\tau+\alpha)-f(\tau)=\frac{a/b}{1+e^{-a\alpha}}-\frac{a}{2b}=\frac{a}{b}\left[\frac{1}{1+e^{-a\alpha}}-\frac{1}{2}\right]=\frac{a(1-e^{-a\alpha})}{2b(1+e^{-a\alpha})}=\frac{a(e^{a\alpha}-1)}{2b(e^{a\alpha}+1)}$$
$$f(\tau)-f(\tau-\alpha)=\frac{a}{2b}-\frac{a/b}{1+e^{a\alpha}}=\frac{a}{b}\left[\frac{1}{2}-\frac{1}{1+e^{a\alpha}}\right]=\frac{a(e^{a\alpha}-1)}{2b(e^{a\alpha}+1)}$$
$$\Rightarrow f(\tau+\alpha)-f(\tau)=f(\tau)-f(\tau-\alpha)$$

□

The value of $f'(t)=f(t)(a-bf(t))$ at τ is $f'(\tau)=(a/2b)(a-b(a/2b))=a^2/2b$, which is always positive. As $f'(t)$ and $f''(t)$ are two continuous functions, $f''(t)=0$ for $t\to-\infty$, $t\to\infty$ and $t=\tau$, and $f'(t)=0$ for $t\to-\infty$, $t\to\infty$, it means that the peak of $f'(t)$ occurs for $t=\tau$.

It means that the logistic function (1) is rotationally symmetrical about $(\tau,\ f(\tau))$, (2) increases slowly for large absolute values of t, (3) increases faster for values t near to τ. All these factors combined give a "S-shape" form to the curve —reason why the logistic curve is also referred as the "S-curve".

We note that if $c_0=b$ it implies that $\tau=0$, which results in:

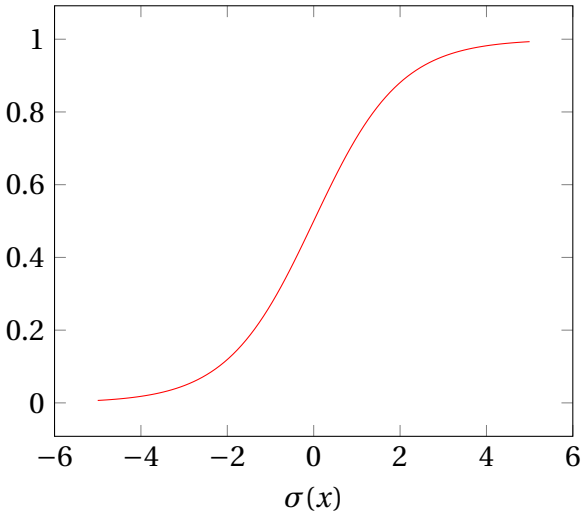
$$f(t)=\frac{a}{b(1+e^{-at})}$$

For $a=b$ we have $f(t)\in[0,1]$, which can be used to represent a likelihood function. The special case in which $a=b=c_0=1$ is known as the *sigmoid function* $(\sigma(x))$, i.e.:

$$\sigma(x)=\frac{1}{(1+e^{-x})}$$

The plot of the sigmoid function is displayed below:

sigmoid function



6.3 Density-Based Clustering Methods

Algorithm 1: Pseudocode of DBSCAN algorithm

```
1  DBSCAN ({p_i}, ε, MinPts);
   Input  : {p_i}, ε and MinPts
   Output: 𝒞
2  𝒞 ← {};
3  cnt ← 0 ;                                          /* Cluster counter */
4  ℒ(p_i) ← ∅, ∀{p_i};
5  foreach p in {p_i} do
6      if ℒ(p) ≠ ∅ then continue ; ;                  /* Already visited */
7
8      𝒩 ← FindNeighbors(p, {p_i}, ε);
9      if |𝒩| < MinPts −1 then
10         ℒ(p) ← η ;                                  /* η:  noise */
11         continue
12     end
13     cnt ← cnt +1;
14     𝒞(cnt) ← {p};
15     ℒ(p) ← cnt ;                                   /* Cluster label */
16     foreach p_n in 𝒩 do
17         if ℒ(p_n) = η then ℒ(p_n) ← cnt;
18         if ℒ(p_n) ≠ ∅ then continue ; ;            /* Already clustered */
19
20         𝒞(cnt) ← 𝒞(cnt) ∪ {p_n};
21         ℒ(p_n) ← cnt;
22         𝒩_n ← FindNeighbors(p_n, {p_i}, ε);
23         if |𝒩| ≥ MinPts −1 then
24             𝒩 ← 𝒩 ∪ 𝒩_n ;                          /* Add reachable points */
25         end
26     end
27 end
```

6.4 Mutual Information

Here we demonstrate that the mutual information of two random variables X and Y is indeed:

$$I(X,Y) = \sum_{x \in X, y \in Y} p(x,y)\ log_2\left(\frac{p(x,y)}{p(x)p(y)}\right)$$

Starting from the definition of entropy:

$$H(X) = \sum_{x \in X} p(x)\ log_2\left(\frac{1}{p(x)}\right)$$

Since $\sum_{y \in Y} p(x,y) = p(x)$, we have:

$$H(X) = \sum_{x \in X} (\sum_{y \in Y} p(x,y)) log_2\left(\frac{1}{p(x)}\right) = \sum_{x \in X, y \in Y} p(x,y)\ log_2\left(\frac{1}{p(x)}\right)$$

The same goes for $H(Y)$, i.e.:

$$H(Y) = \sum_{y \in Y} (\sum_{x \in X} p(x,y)) log_2\left(\frac{1}{p(y)}\right) = \sum_{x \in X, y \in Y} p(x,y)\ log_2\left(\frac{1}{p(y)}\right)$$

Since the definition of the mutual information $I(X,Y)$ is:

$$I(X,Y) = H(X) + H(Y) - H(X,Y)$$

we can replace $H(X)$ and $H(Y)$ by the previous equations, i.e.:

$$\begin{aligned} I(X,Y) &= \sum_{x \in X, y \in Y} p(x,y)\ log_2\left(\frac{1}{p(x)}\right) + \sum_{x \in X, y \in Y} p(x,y)\ log_2\left(\frac{1}{p(y)}\right) - \sum_{x \in X, y \in Y} p(x,y)\ log_2\left(\frac{1}{p(x,y)}\right) \\ &= \sum_{x \in X, y \in Y} p(x,y)\left(\ log_2\left(\frac{1}{p(x)}\right) + \ log_2\left(\frac{1}{p(y)}\right) - \ log_2\left(\frac{1}{p(x)p(y)}\right)\right) \\ &= \sum_{x \in X, y \in Y} p(x,y)\ log_2\left(\frac{p(x,y)}{p(x)p(y)}\right) \end{aligned}$$

□

6.5 PE Headers

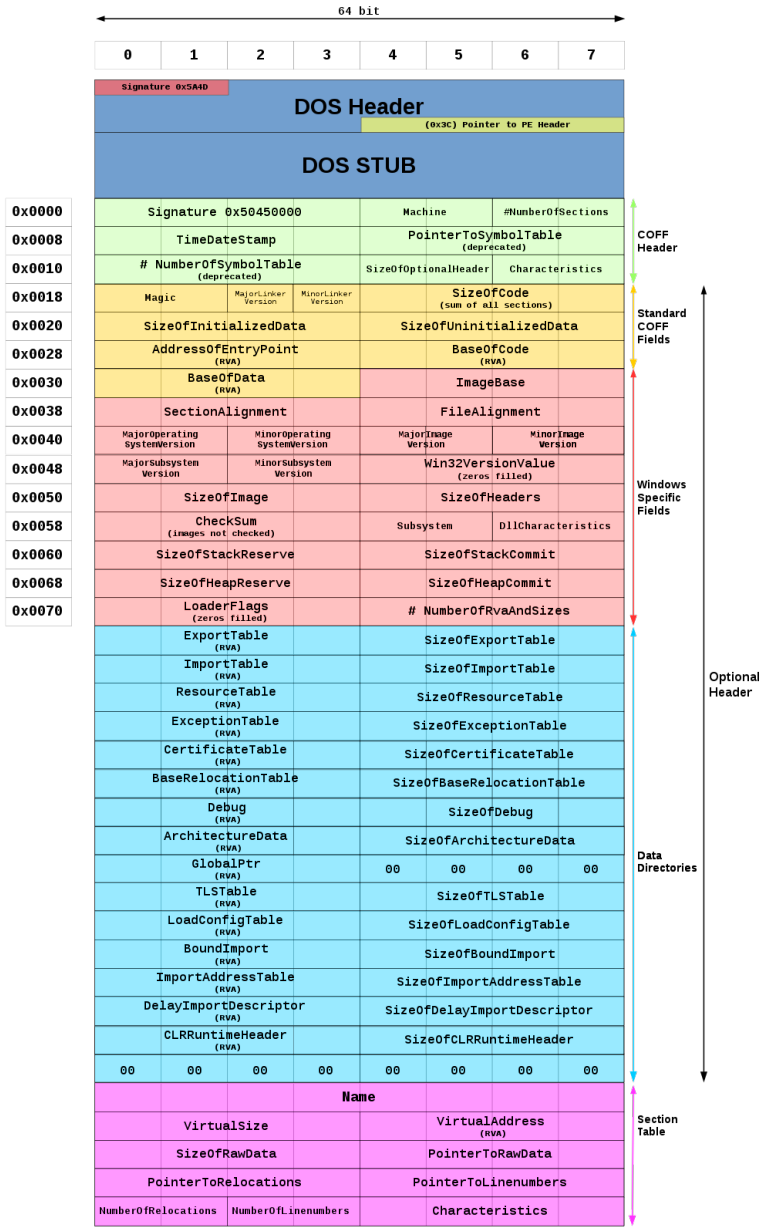


Figure 6.2 – Layout of headers in a PE file [1].

6.6 Call Tracing with Symbolic Execution

6.6.1 Supplementary analysis

Main effect of graph and execution heuristics on the F-score

Figure 6.3 shows the impact on the F-score of the seven studied factors (i.e. the graph building and execution heuristics). The plots aim to represent how classifier and the levels to which a given factor are related.

In each plot, points on a column represent the micro average F-score achieved by all the configurations having a specific setting (on the x-axis) for the heuristic reported in the caption. Each plot represents the levels for a given factor on the x-axis. On the y-axis the quality of the classifier, F_1-score by family in our case.

Interactions between factors

The interactions between each possible combination of factors are represented by means of the Pareto plot in figure 6.4. Basically, such a plot allows to observe factors and interaction effects having the highest impact on the performance (the F-score in our case). The impact, positive or negative (respectively in black and gray in figure 6.4), of each combination of factors on the F-score is on the x-axis.

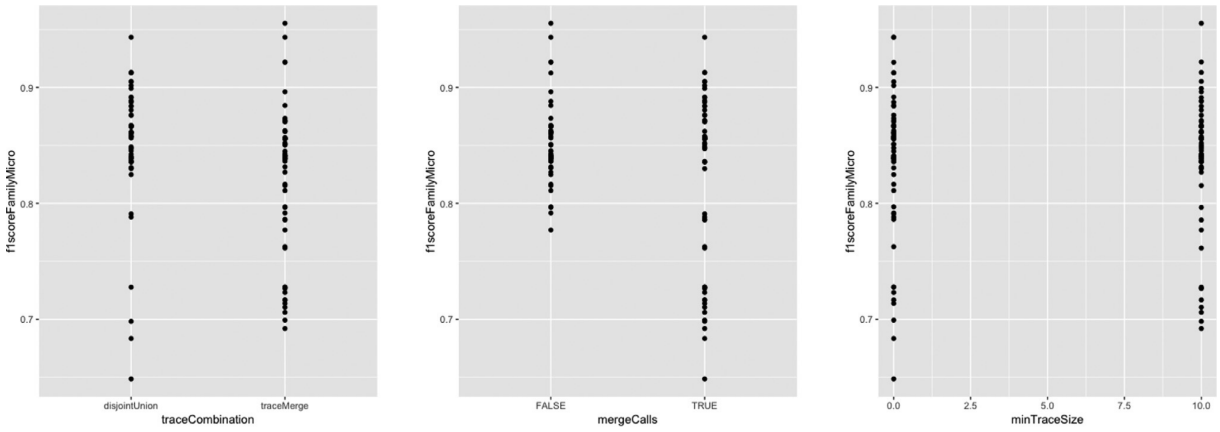
Interaction plots in figures 6.5, 6.6 and 6.7 provide a more detailed view by showing the relations between each pair of factors (by means of a linear model).

Relationship between connected components and F-score

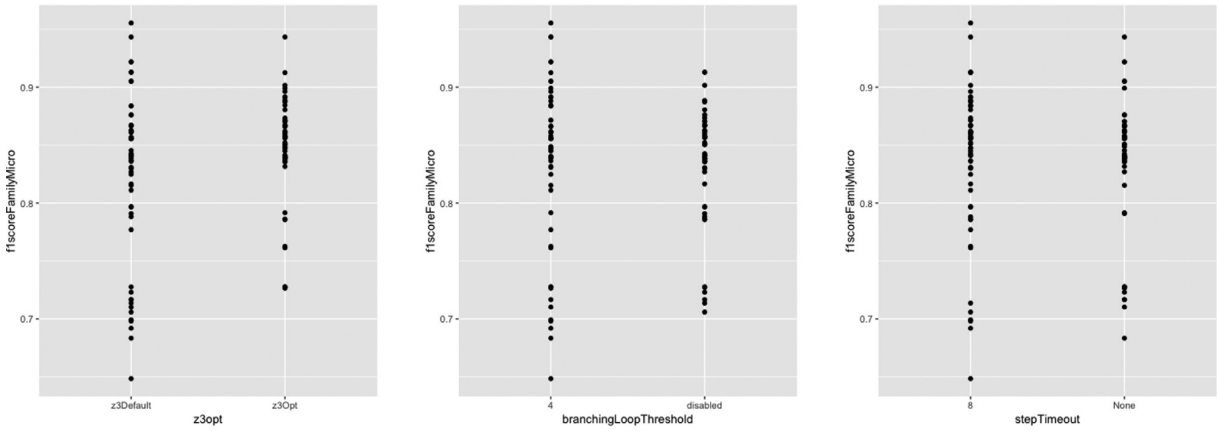
During our analysis we pointed out that the litmus test for the quality of an ECDG-based classifier is the presence and the size of connected components (c.f. section 3.5.3). Here we report, by means of box plots the relationship between the classifier performance (with respect to the binary and multi-class classifiers) and number (c.f. figures 6.8, 6.9, and 6.10) and size (c.f. figures 6.11, 6.12 and 6.13) of the largest connected components in the set of ECDG-based signatures used by the classifier, for all the 128 experimental units analyzed in this work.

In the plots, on the secondary y-axis the red stars refer to the F-score of the multi-class classification (malware classifier), whereas the triangles represent the F-score of the binary

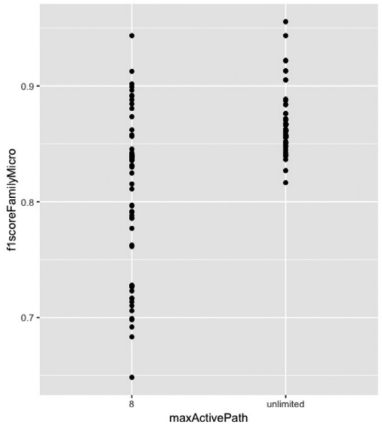
1. Image published at https://fi.wikipedia.org/wiki/Portable_Executable#/media/Tiedosto:Portable_Executable_32_bit_Structure_in_SVG_fixed.svg (accessed October 2022)



(a) Trace combination (b) `Merge calls` (c) `Min trace size`



(d) `z3 config` (e) `branching loop threshold` (f) `step timeout`



(g) `max active path`

Figure 6.3 – Main effect of the factors.

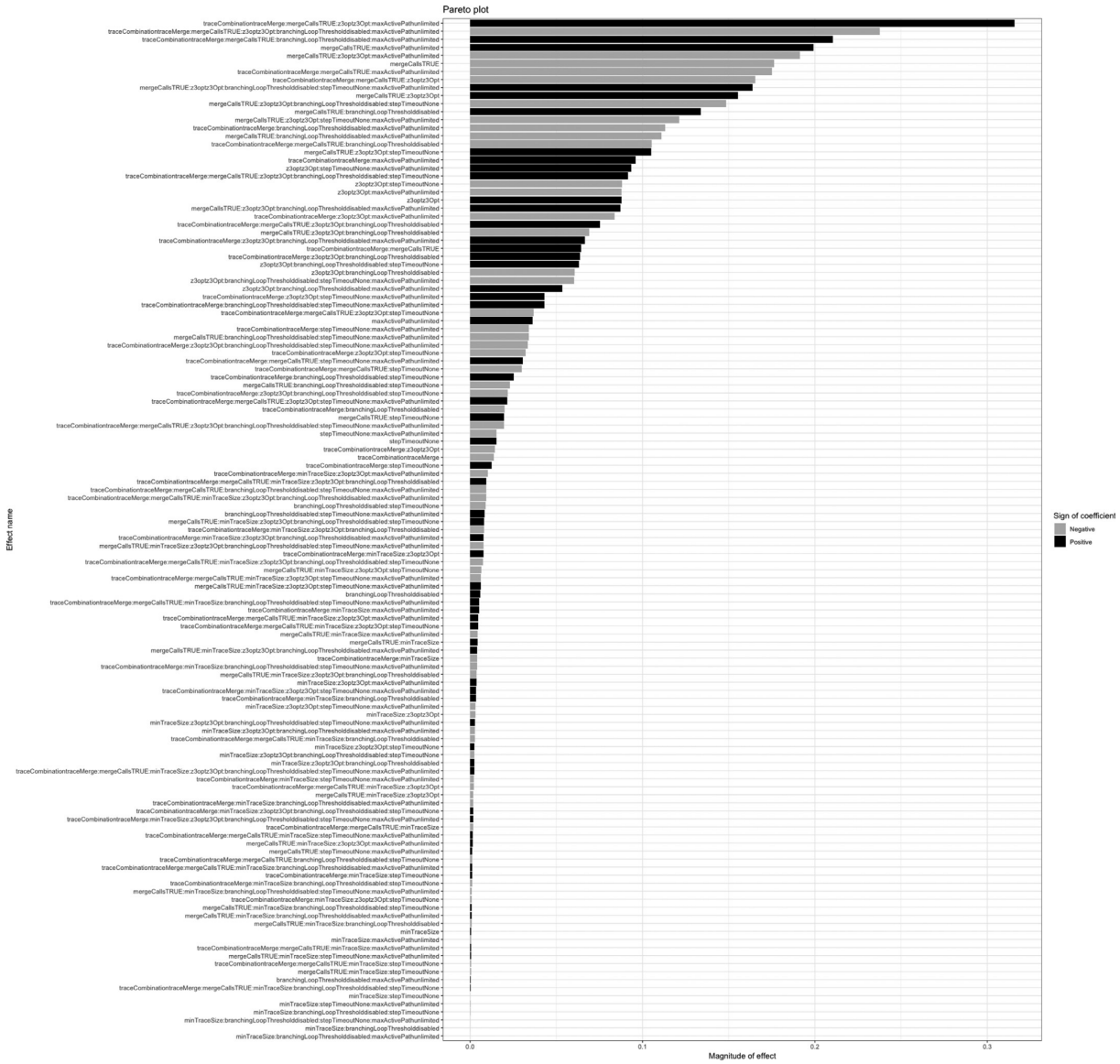
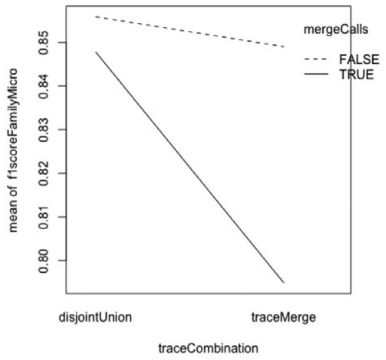
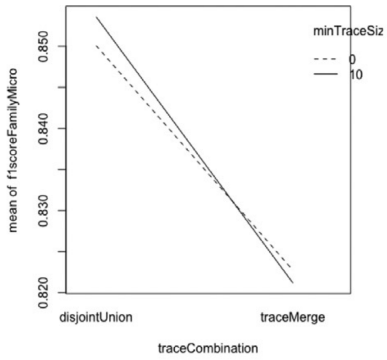


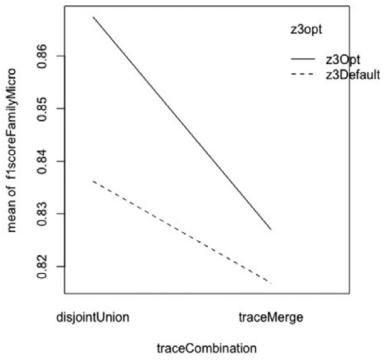
Figure 6.4 – Pareto plot for the factorial experiments. In black positive coefficients, in gray the negative ones.



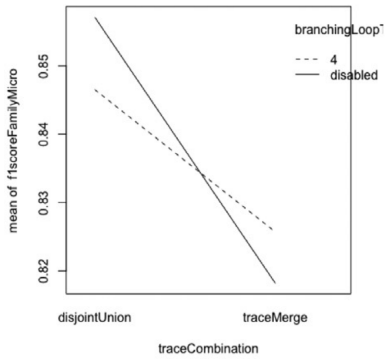
(a) Trace combination and merge calls



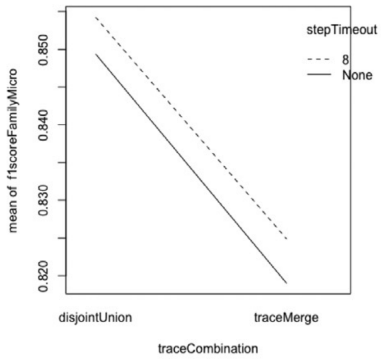
(b) Trace combination and min-trace-size



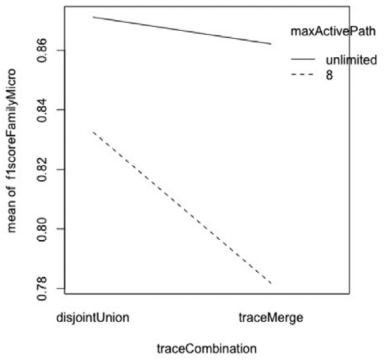
(c) Trace combination and z3 configuration



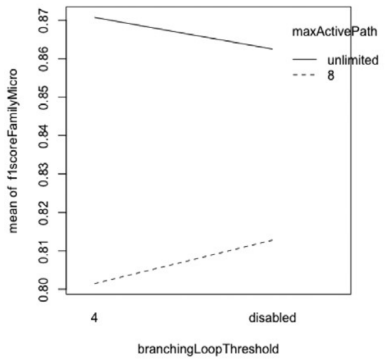
(d) Trace combination and branching loop



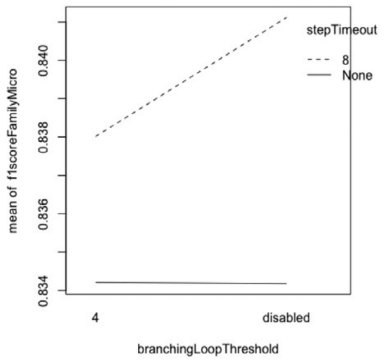
(e) Trace combination and step timeout



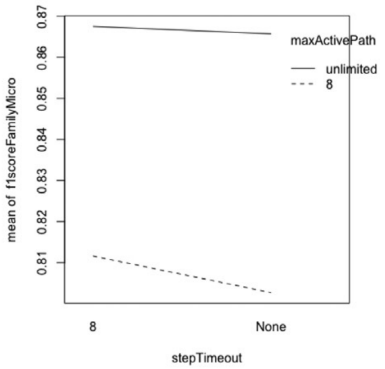
(f) Trace combination and max active paths



(g) Branching loop and max active paths

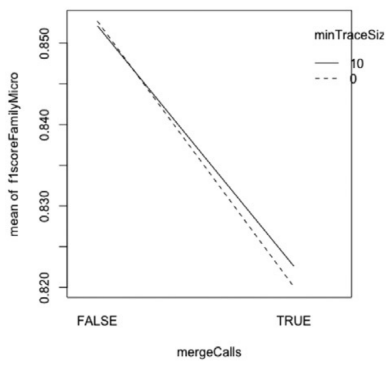


(h) Branching loop and step timeout

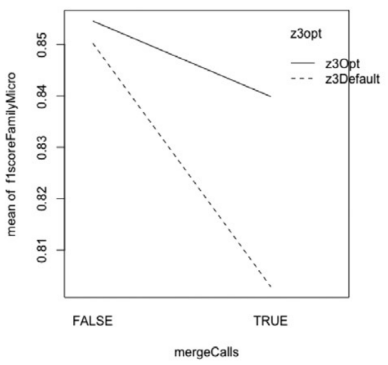


(i) Step timeout and max active path

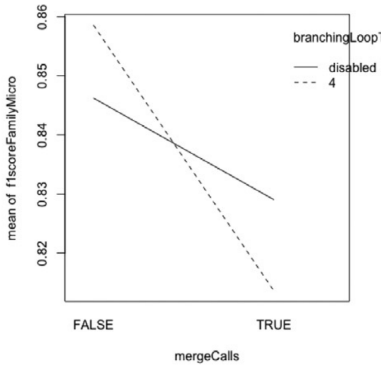
Figure 6.5 – Interaction plots for the ANOVA analysis (1/3): interaction effects on the mean F_1-score by malware family for each possible pair of factors.



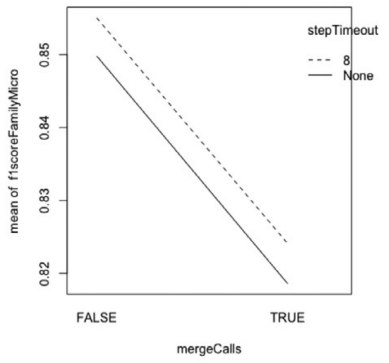
(a) Merge calls and min trace size



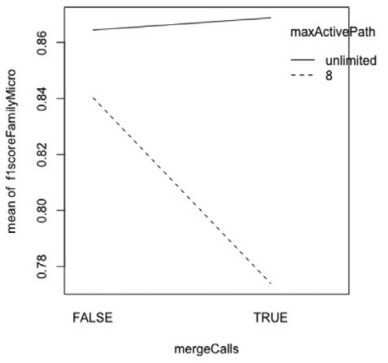
(b) Merge calls and z3 configuration



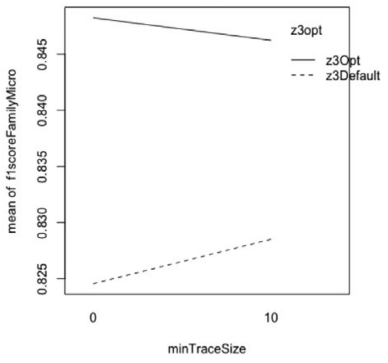
(c) Merge calls and branching loop



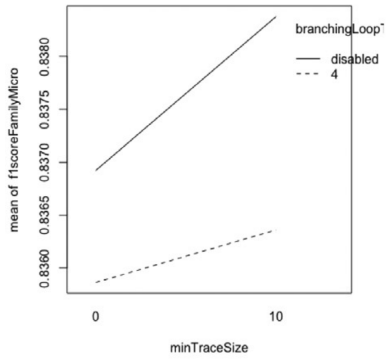
(d) Merge calls and step timeout



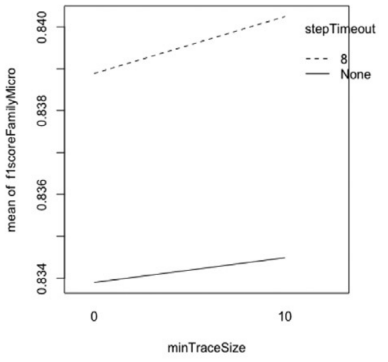
(e) Merge calls and max active paths



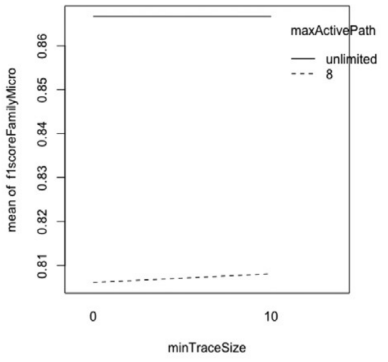
(f) Min trace size and z3 configuration



(g) Min trace size and branching loop

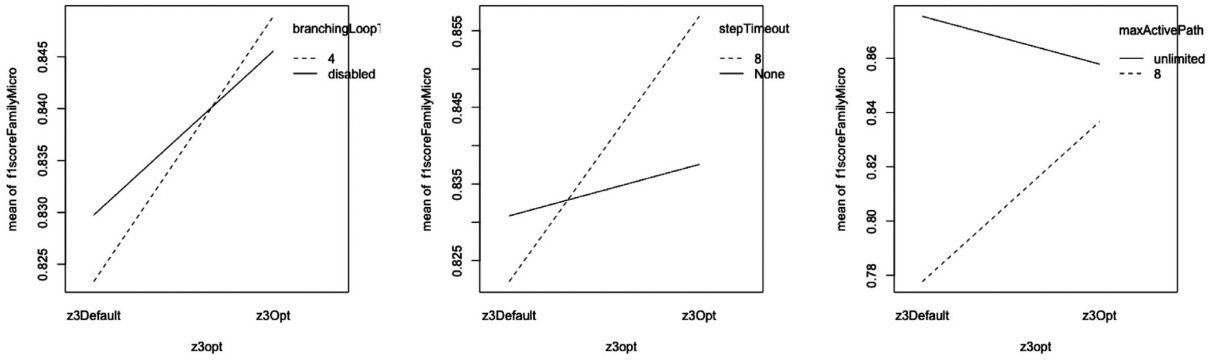


(h) Min trace size and step timeout



(i) Min trace size and max active path

Figure 6.6 – Interaction plots for the ANOVA analysis (2/3): interaction effects on the mean F_1-score by malware family for each possible pair of factors.



(a) z3 configuration and branching loop

(b) z3 configuration and step timeout

(c) z3 configuration and max active paths

Figure 6.7 – Interaction plots for the ANOVA analysis (3/3): interaction effects on the mean F_1-score by malware family for each possible pair of factors.

classification (malware detection). For the sake of readability, the 128 configurations have been grouped in sets of 32 for each plot.

Impact of the graph similarity threshold on the F-score

In this work, according to our previous experience and after the preliminary analysis, we have chosen to set the similarity threshold for the graph matching algorithm gSpan to 0.7. In the following (c.f. figures 6.14, 6.15, 6.16, 6.17, 6.18, 6.19, 6.20, and 6.21), we report the F-score for the binary (malware detection) and multi-class (malware classification) classifiers, for all the 128 studied configurations, by varying the similarity threshold with steps of 0.1 in the range from 0 to 1. Here the blue circles indicate binary classification, and the red stars multi-class classification.

Dataset

Due to a NDA (Non-Disclosure Agreement) and security concerns we are not allowed to share our dataset. Notwithstanding, for the sake of reproducibility, we provide below the SHA1 and the corresponding family for each binary analyzed in our work (c.f. tables 6.1, 6.2, 6.3, 6.4 and 6.5). As files are (overwhelmingly likely to be) mapped to a single SHA1 value, interested researchers can lookup for the samples in other available databases (e.g. VirusTotal).

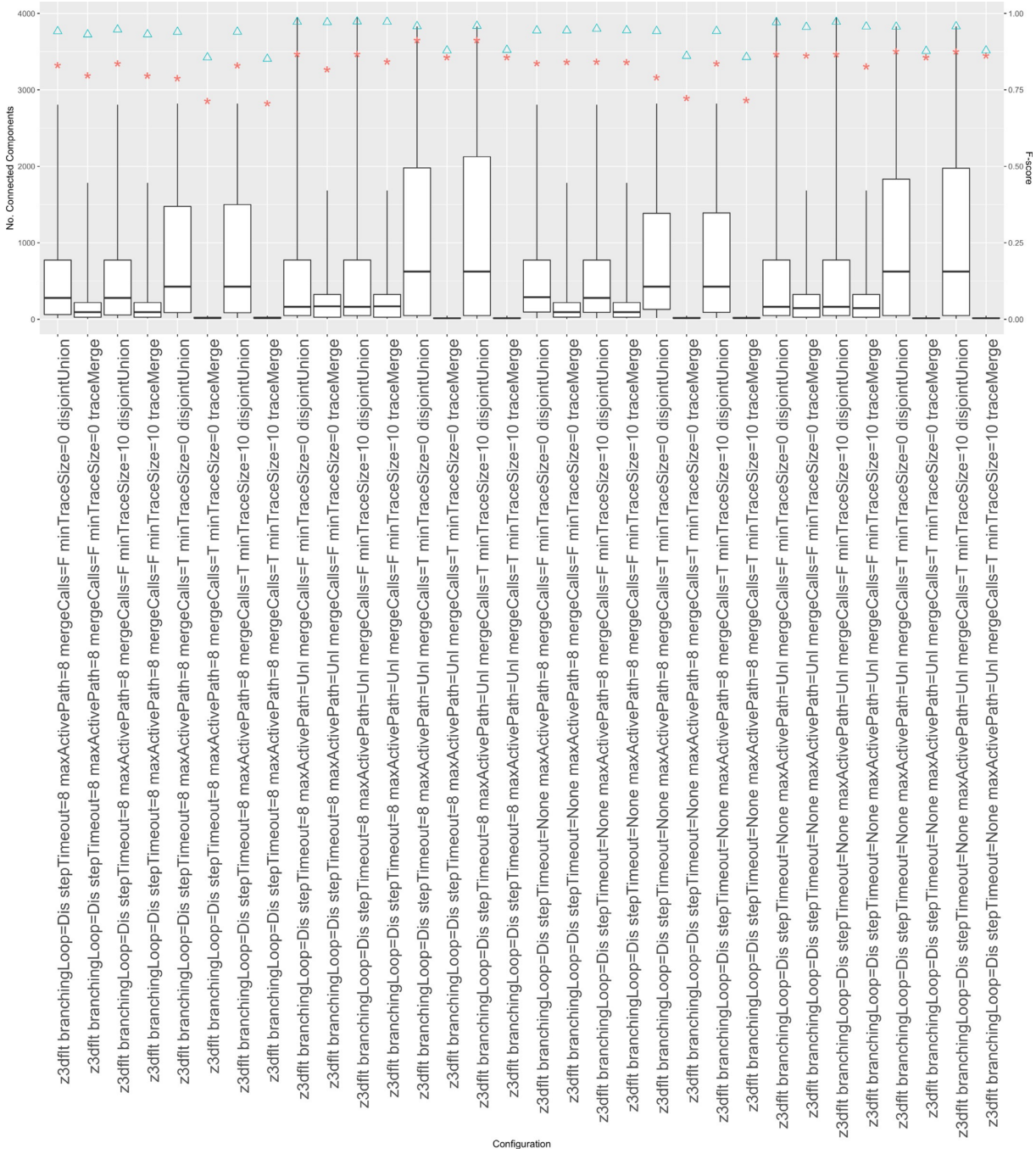


Figure 6.8 – Number of connected components for the ECDGs in each configuration and F-scores (triangles represent the binary classifier, stars the multi-class classifier) - (1/3).

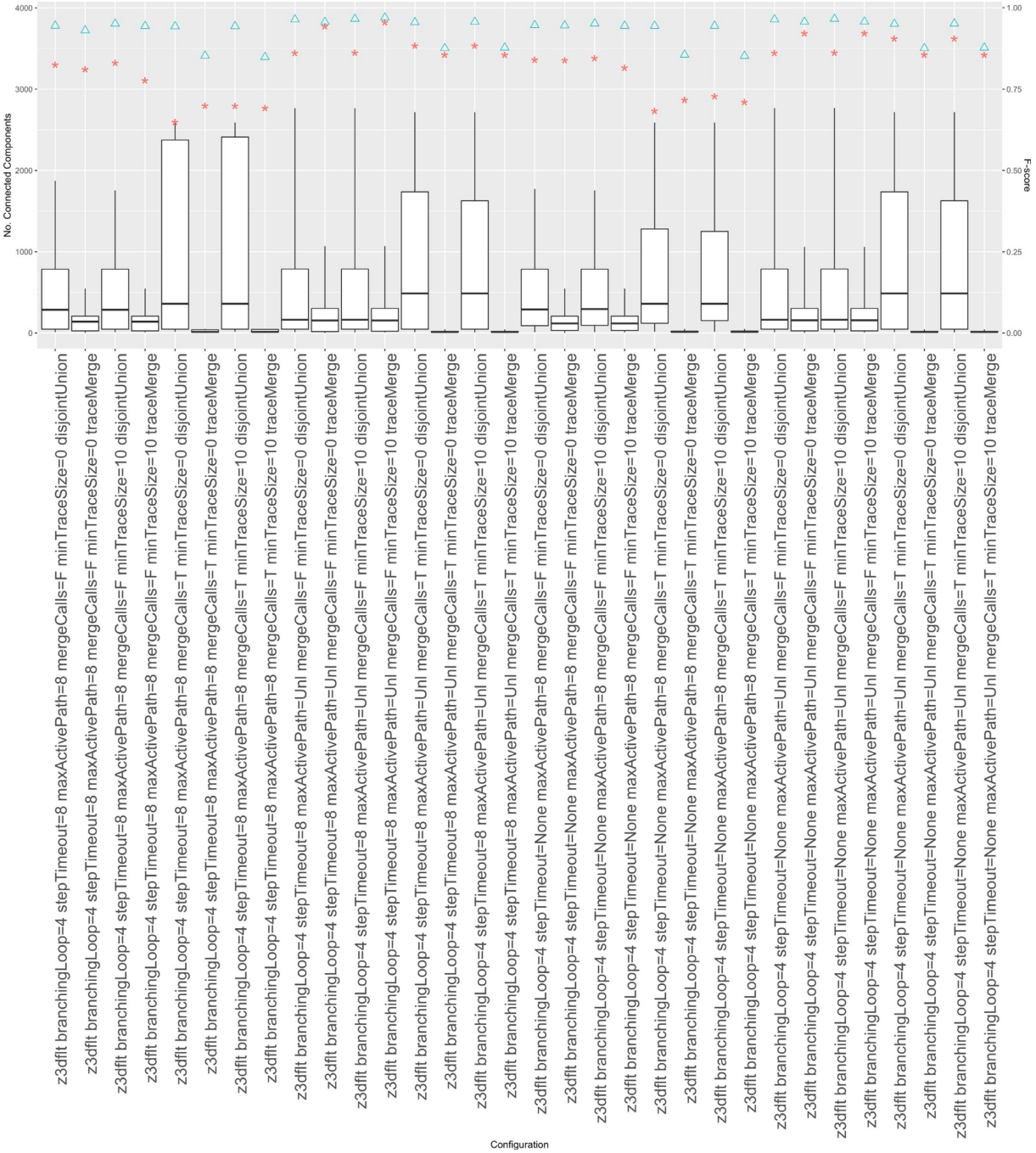


Figure 6.9 – Number of connected components for the ECDGs in each configuration and F-scores (triangles represent the binary classifier, stars the multi-class classifier) - (2/3).

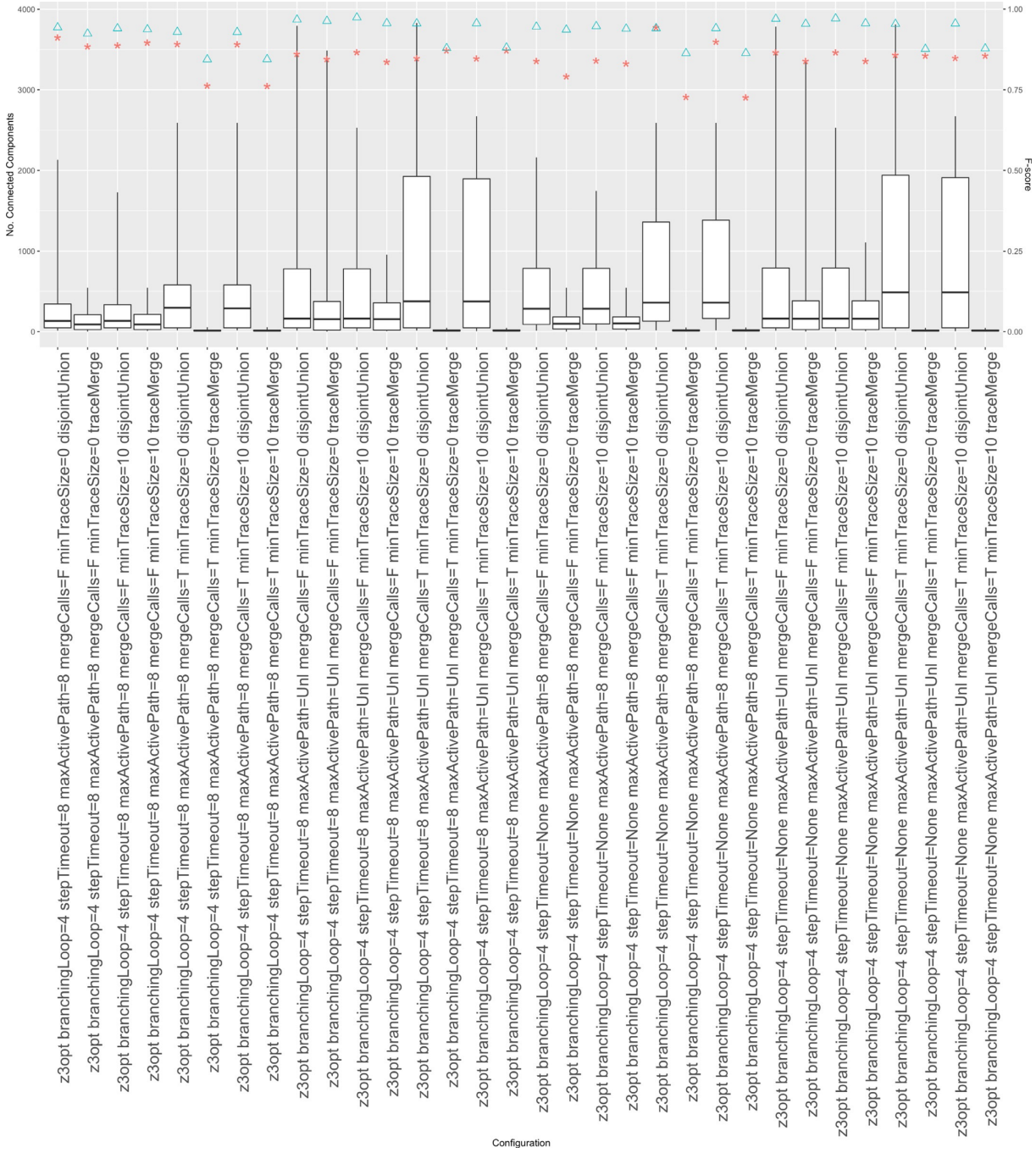


Figure 6.10 – Number of connected components for the ECDGs in each configuration and F-scores (triangles represent the binary classifier, stars the multi-class classifier) - (3/3).

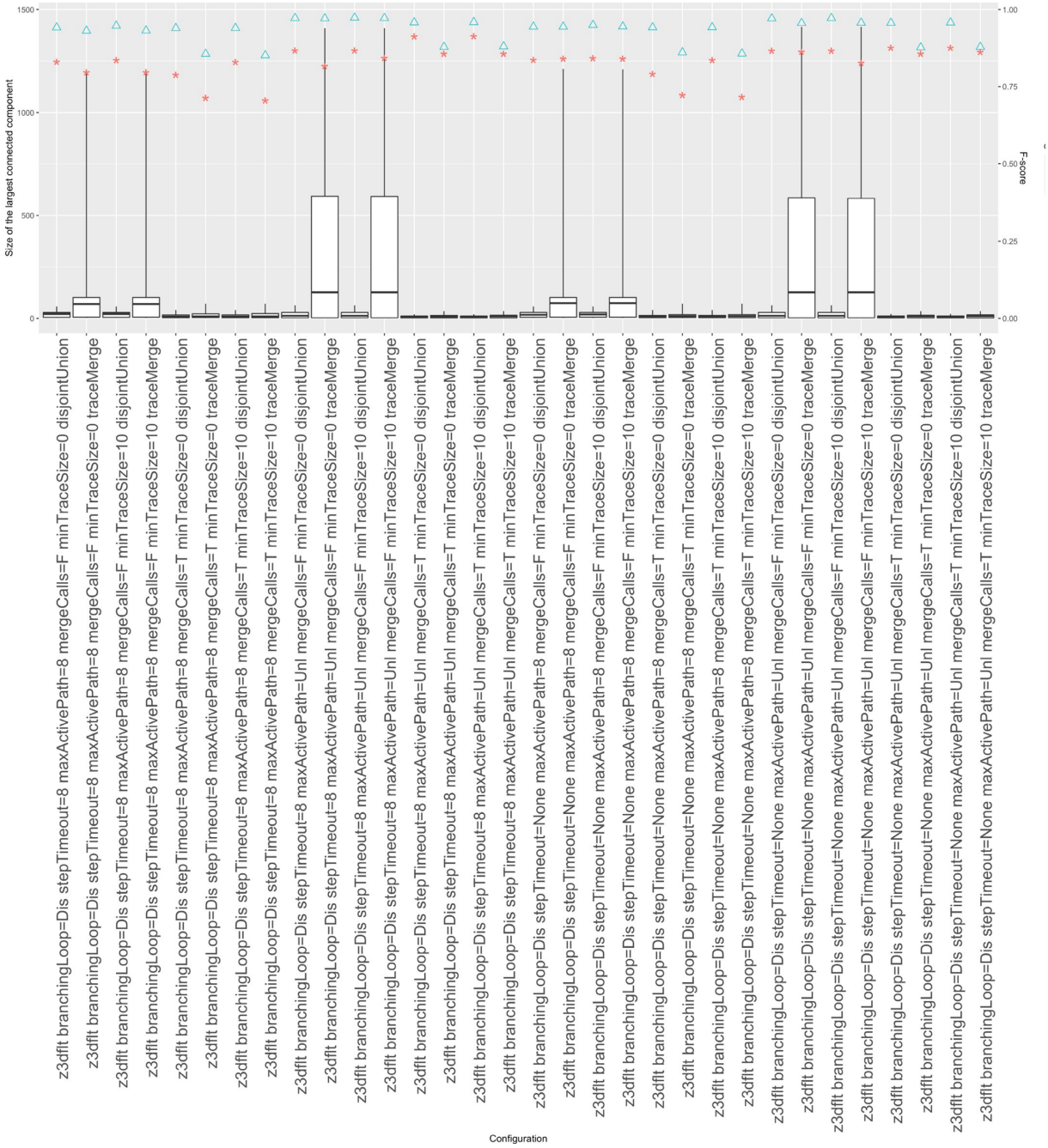


Figure 6.11 – Size of the largest connected component for the ECDGs in each configuration and F-scores (triangles represent the binary classifier, stars the multi-class classifier) - (1/3).

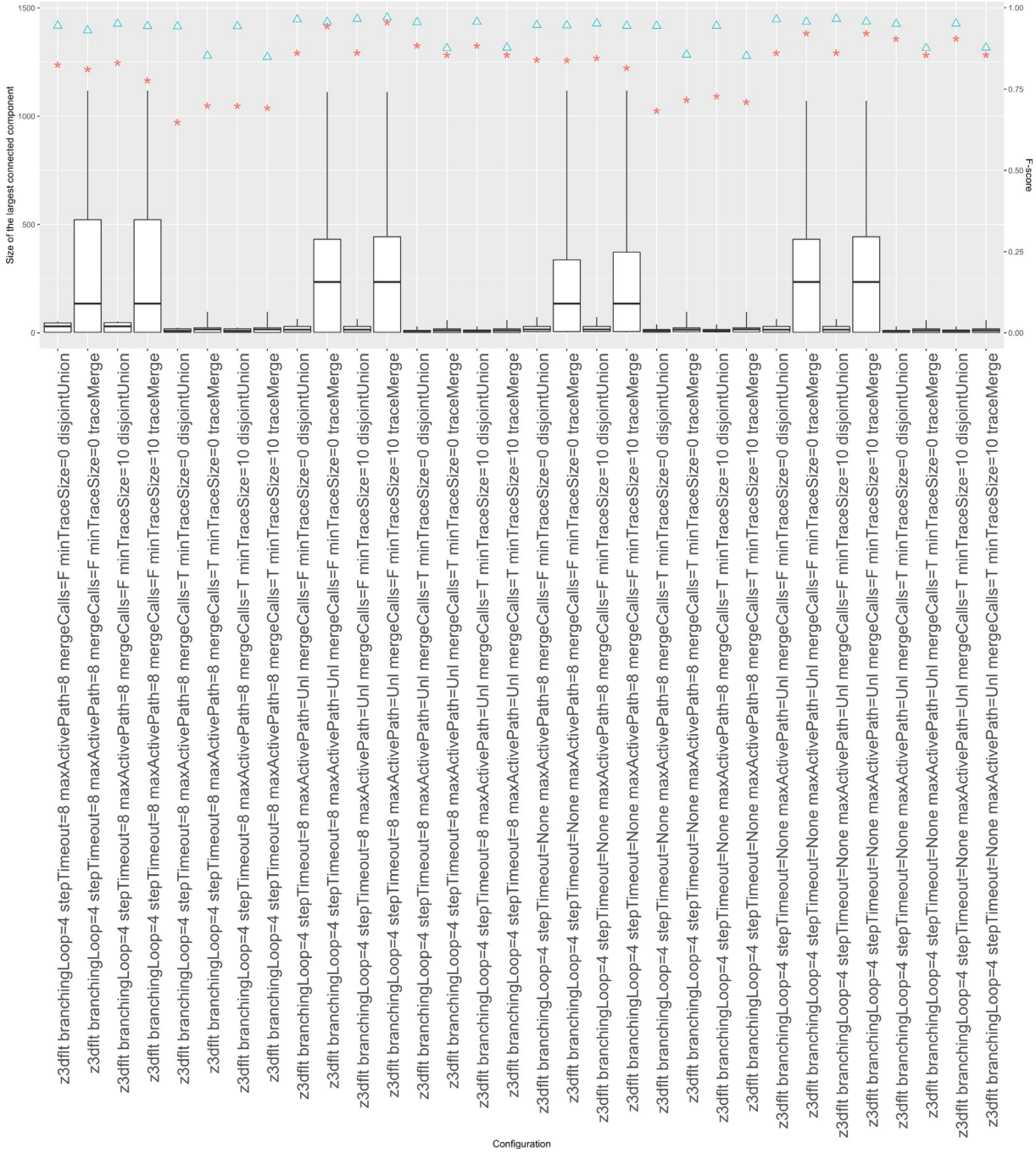


Figure 6.12 – Size of the largest connected component for the ECDGs in each configuration and F-scores (triangles represent the binary classifier, stars the multi-class classifier) - (2/3).

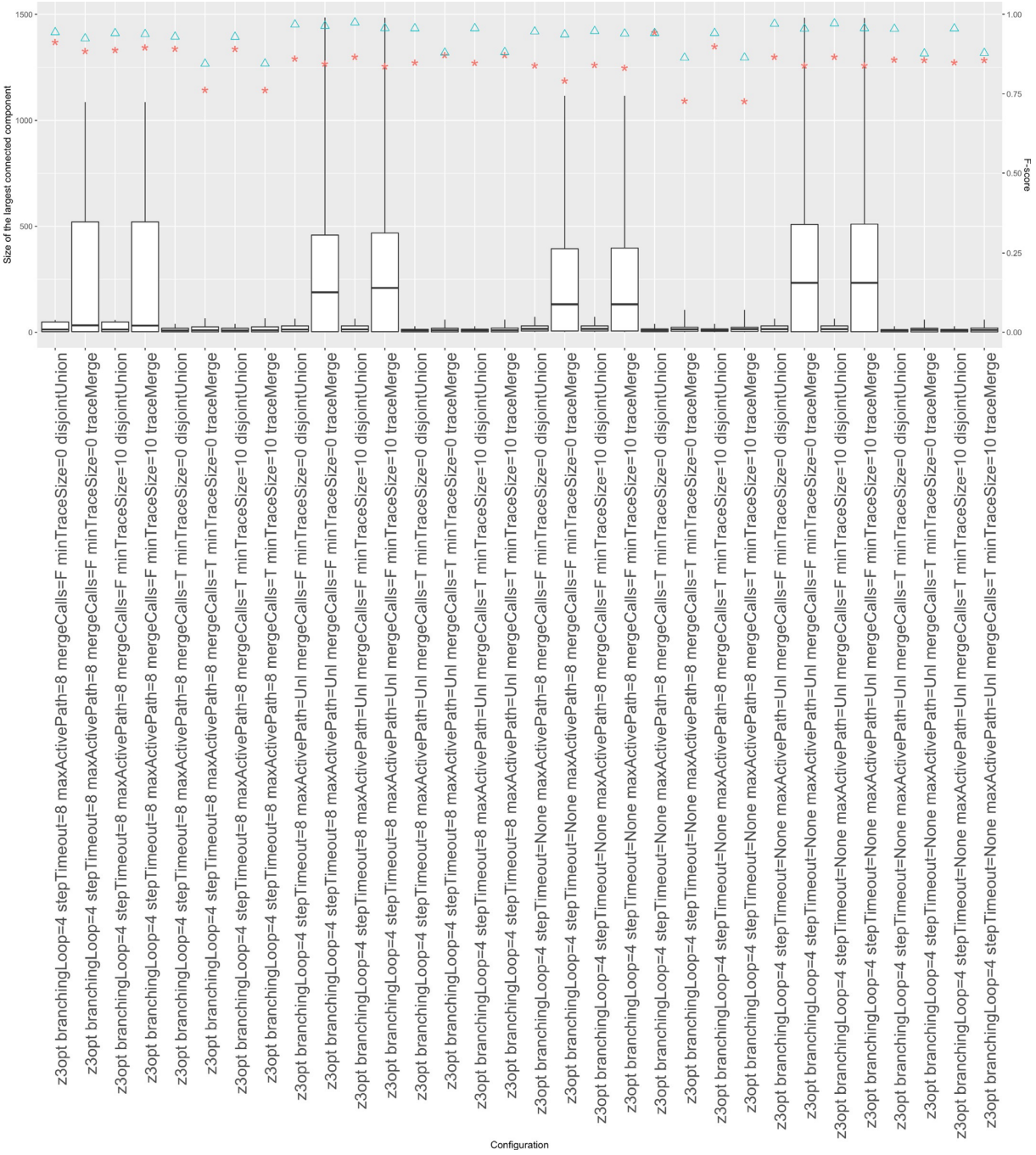
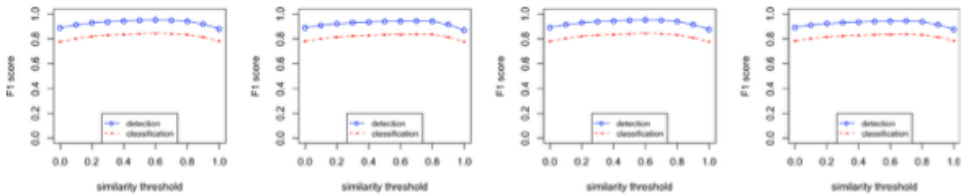


Figure 6.13 – Size of the largest connected component for the ECDGs in each configuration and F-scores (triangles represent the binary classifier, stars the multi-class classifier) - (3/3).

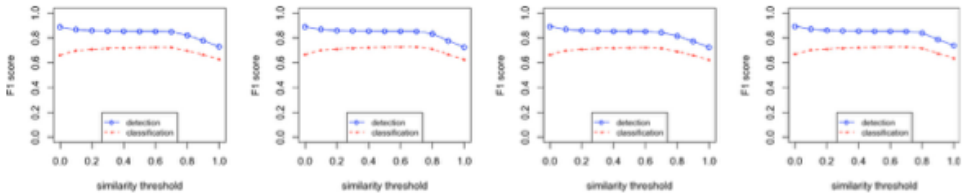


(a) z3dflt, branchingLoop=Dis, stepTimeout=None, maxActivePath=8, mergeCall=F, minTraceSize=0, traceMerge

(b) z3opt, branchingLoop=Dis, stepTimeout=None, maxActivePath=8, mergeCall=F, minTraceSize=0, traceMerge

(c) z3dflt, branchingLoop=Dis, stepTimeout=None, maxActivePath=8, mergeCall=F, minTraceSize=10, traceMerge

(d) z3opt, branchingLoop=Dis, stepTimeout=None, maxActivePath=8, mergeCall=F, minTraceSize=10, traceMerge

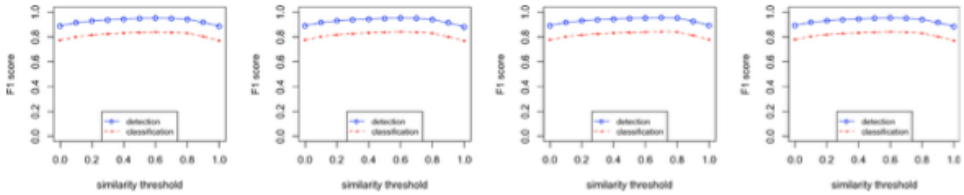


(e) z3dflt, branchingLoop=Dis, stepTimeout=None, maxActivePath=8, mergeCall=T, minTraceSize=0, traceMerge

(f) z3opt, branchingLoop=Dis, stepTimeout=None, maxActivePath=8, mergeCall=T, minTraceSize=0, traceMerge

(g) z3dflt, branchingLoop=Dis, stepTimeout=None, maxActivePath=8, mergeCall=T, minTraceSize=10, traceMerge

(h) z3opt, branchingLoop=Dis, stepTimeout=None, maxActivePath=8, mergeCall=T, minTraceSize=10, traceMerge

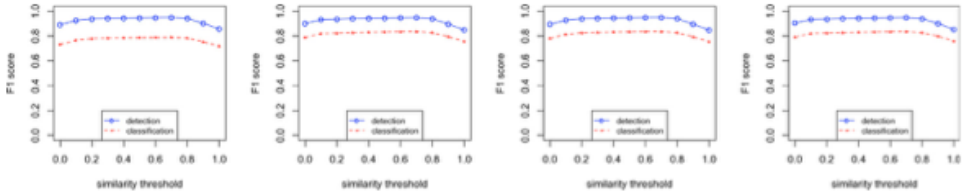


(i) z3dflt, branchingLoop=Dis, stepTimeout=None, maxActivePath=8, mergeCall=F, minTraceSize=0, disjointUnion

(j) z3opt, branchingLoop=Dis, stepTimeout=None, maxActivePath=8, mergeCall=F, minTraceSize=0, disjointUnion

(k) z3dflt, branchingLoop=Dis, stepTimeout=None, maxActivePath=8, mergeCall=F, minTraceSize=10, disjointUnion

(l) z3opt, branchingLoop=Dis, stepTimeout=None, maxActivePath=8, mergeCall=F, minTraceSize=10, disjointUnion



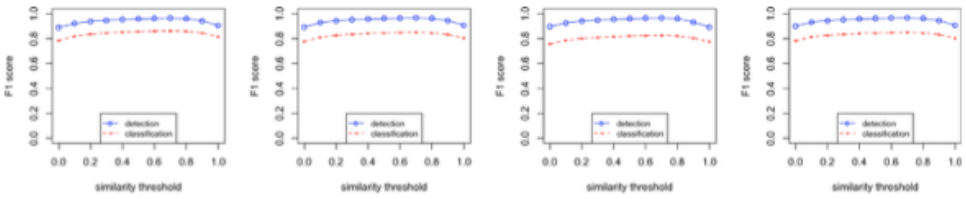
(m) z3dflt, branchingLoop=Dis, stepTimeout=None, maxActivePath=8, mergeCall=T, minTraceSize=0, disjointUnion

(n) z3opt, branchingLoop=Dis, stepTimeout=None, maxActivePath=8, mergeCall=T, minTraceSize=0, disjointUnion

(o) z3dflt, branchingLoop=Dis, stepTimeout=None, maxActivePath=8, mergeCall=T, minTraceSize=10, disjointUnion

(p) z3opt, branchingLoop=Dis, stepTimeout=None, maxActivePath=8, mergeCall=T, minTraceSize=10, disjointUnion

Figure 6.14 – Graph similarity vs. F-scores (1/8)

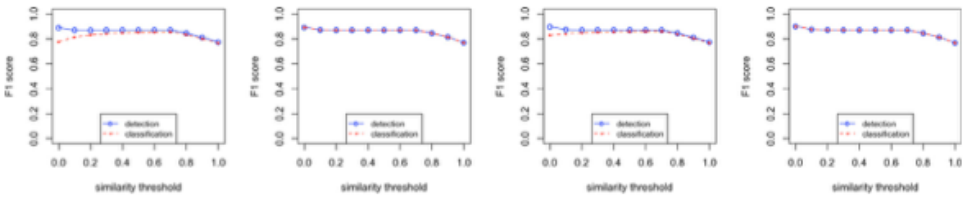


(a) z3dflt, branchingLoop=Dis, stepTimeout=None, maxActivePath=Unl, mergeCall=F, minTraceSize=0, traceMerge

(b) z3opt, branchingLoop=Dis, stepTimeout=None, maxActivePath=Unl, mergeCall=F, minTraceSize=0, traceMerge

(c) z3dflt, branchingLoop=Dis, stepTimeout=None, maxActivePath=Unl, mergeCall=F, minTraceSize=10, traceMerge

(d) z3opt, branchingLoop=Dis, stepTimeout=None, maxActivePath=Unl, mergeCall=F, minTraceSize=10, traceMerge

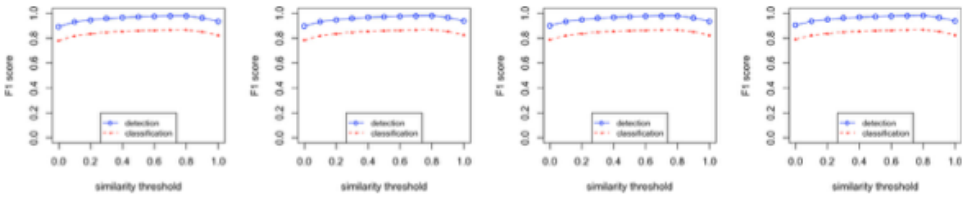


(e) z3dflt, branchingLoop=Dis, stepTimeout=None, maxActivePath=Unl, mergeCall=T, minTraceSize=0, traceMerge

(f) z3opt, branchingLoop=Dis, stepTimeout=None, maxActivePath=Unl, mergeCall=T, minTraceSize=0, traceMerge

(g) z3dflt, branchingLoop=Dis, stepTimeout=None, maxActivePath=Unl, mergeCall=T, minTraceSize=10, traceMerge

(h) z3opt, branchingLoop=Dis, stepTimeout=None, maxActivePath=Unl, mergeCall=T, minTraceSize=10, traceMerge

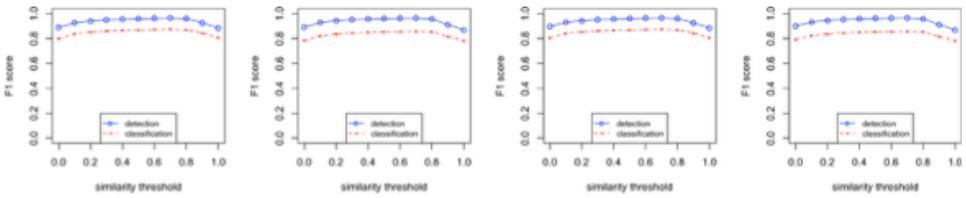


(i) z3dflt, branchingLoop=Dis, stepTimeout=None, maxActivePath=Unl, mergeCall=F, minTraceSize=0, disjointUnion

(j) z3opt, branchingLoop=Dis, stepTimeout=None, maxActivePath=Unl, mergeCall=F, minTraceSize=0, disjointUnion

(k) z3dflt, branchingLoop=Dis, stepTimeout=None, maxActivePath=Unl, mergeCall=F, minTraceSize=10, disjointUnion

(l) z3opt, branchingLoop=Dis, stepTimeout=None, maxActivePath=Unl, mergeCall=F, minTraceSize=10, disjointUnion



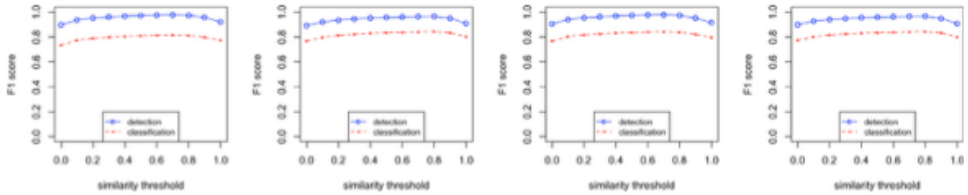
(m) z3dflt, branchingLoop=Dis, stepTimeout=None, maxActivePath=Unl, mergeCall=T, minTraceSize=0, disjointUnion

(n) z3opt, branchingLoop=Dis, stepTimeout=None, maxActivePath=Unl, mergeCall=T, minTraceSize=0, disjointUnion

(o) z3dflt, branchingLoop=Dis, stepTimeout=None, maxActivePath=Unl, mergeCall=T, minTraceSize=10, disjointUnion

(p) z3opt, branchingLoop=Dis, stepTimeout=None, maxActivePath=Unl, mergeCall=T, minTraceSize=10, disjointUnion

Figure 6.15 – Graph similarity vs. F-scores (2/8)

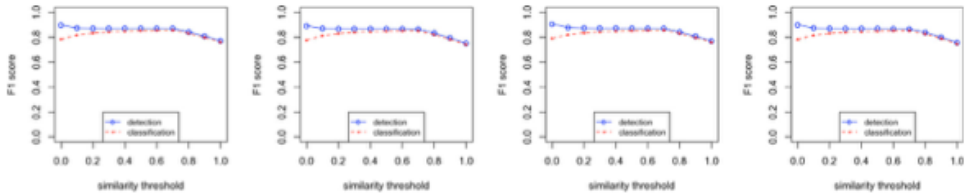


(a) z3dflt, branchingLoop=Dis, stepTimeout=8, maxActivePath=Unl, mergeCall=F, minTraceSize=0, traceMerge

(b) z3opt, branchingLoop=Dis, stepTimeout=8, maxActivePath=Unl, mergeCall=F, minTraceSize=0, traceMerge

(c) z3dflt, branchingLoop=Dis, stepTimeout=8, maxActivePath=Unl, mergeCall=F, minTraceSize=10, traceMerge

(d) z3opt, branchingLoop=Dis, stepTimeout=8, maxActivePath=Unl, mergeCall=F, minTraceSize=10, traceMerge

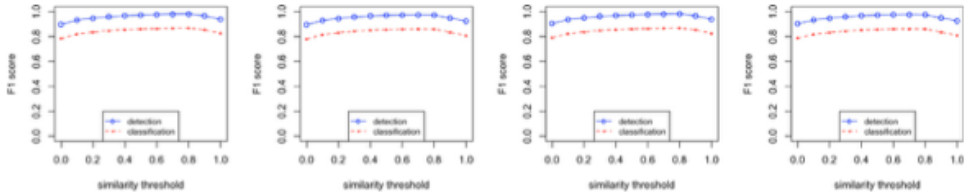


(e) z3dflt, branchingLoop=Dis, stepTimeout=8, maxActivePath=Unl, mergeCall=T, minTraceSize=0, traceMerge

(f) z3opt, branchingLoop=Dis, stepTimeout=8, maxActivePath=Unl, mergeCall=T, minTraceSize=0, traceMerge

(g) z3dflt, branchingLoop=Dis, stepTimeout=8, maxActivePath=Unl, mergeCall=T, minTraceSize=10, traceMerge

(h) z3opt, branchingLoop=Dis, stepTimeout=8, maxActivePath=Unl, mergeCall=T, minTraceSize=10, traceMerge

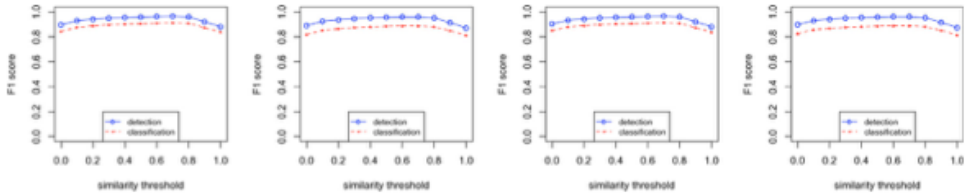


(i) z3dflt, branchingLoop=Dis, stepTimeout=8, maxActivePath=Unl, mergeCall=F, minTraceSize=0, disjointUnion

(j) z3opt, branchingLoop=Dis, stepTimeout=8, maxActivePath=Unl, mergeCall=F, minTraceSize=0, disjointUnion

(k) z3dflt, branchingLoop=Dis, stepTimeout=8, maxActivePath=Unl, mergeCall=F, minTraceSize=10, disjointUnion

(l) z3opt, branchingLoop=Dis, stepTimeout=8, maxActivePath=Unl, mergeCall=F, minTraceSize=10, disjointUnion



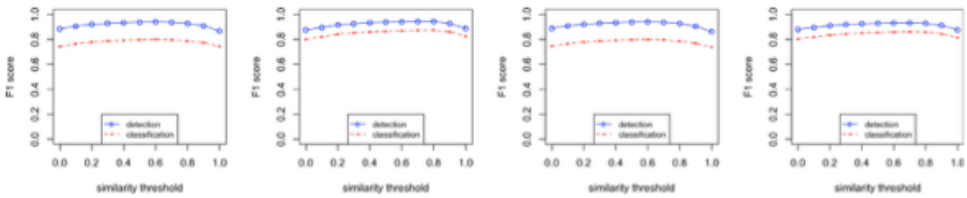
(m) z3dflt, branchingLoop=Dis, stepTimeout=8, maxActivePath=Unl, mergeCall=T, minTraceSize=0, disjointUnion

(n) z3opt, branchingLoop=Dis, stepTimeout=8, maxActivePath=Unl, mergeCall=T, minTraceSize=0, disjointUnion

(o) z3dflt, branchingLoop=Dis, stepTimeout=8, maxActivePath=Unl, mergeCall=T, minTraceSize=10, disjointUnion

(p) z3opt, branchingLoop=Dis, stepTimeout=8, maxActivePath=Unl, mergeCall=T, minTraceSize=10, disjointUnion

Figure 6.16 – Graph similarity vs. F-scores (3/8)

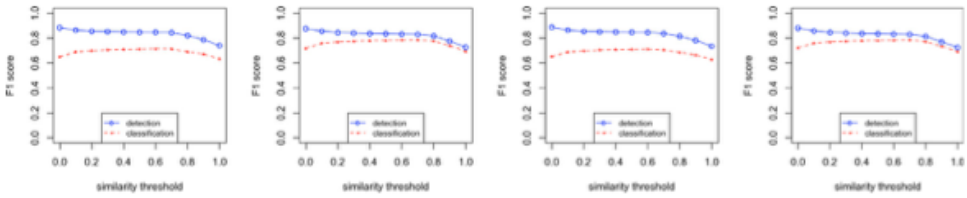


(a) z3dflt, branchingLoop=Dis, stepTimeout=8, maxActivePath=8, mergeCall=F, minTraceSize=0, traceMerge

(b) z3opt, branchingLoop=Dis, stepTimeout=8, maxActivePath=8, mergeCall=F, minTraceSize=0, traceMerge

(c) z3dflt, branchingLoop=Dis, stepTimeout=8, maxActivePath=8, mergeCall=F, minTraceSize=10, traceMerge

(d) z3opt, branchingLoop=Dis, stepTimeout=8, maxActivePath=8, mergeCall=F, minTraceSize=10, traceMerge

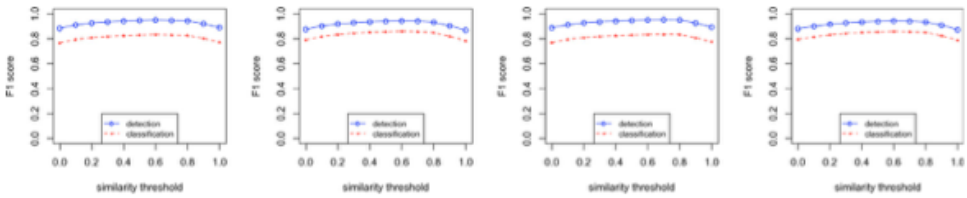


(e) z3dflt, branchingLoop=Dis, stepTimeout=8, maxActivePath=8, mergeCall=T, minTraceSize=0, traceMerge

(f) z3opt, branchingLoop=Dis, stepTimeout=8, maxActivePath=8, mergeCall=T, minTraceSize=0, traceMerge

(g) z3dflt, branchingLoop=Dis, stepTimeout=8, maxActivePath=8, mergeCall=T, minTraceSize=10, traceMerge

(h) z3opt, branchingLoop=Dis, stepTimeout=8, maxActivePath=8, mergeCall=T, minTraceSize=10, traceMerge

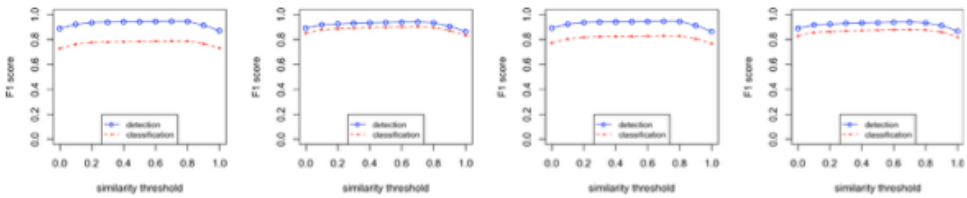


(i) z3dflt, branchingLoop=Dis, stepTimeout=8, maxActivePath=8, mergeCall=F, minTraceSize=0, disjointUnion

(j) z3opt, branchingLoop=Dis, stepTimeout=8, maxActivePath=8, mergeCall=F, minTraceSize=0, disjointUnion

(k) z3dflt, branchingLoop=Dis, stepTimeout=8, maxActivePath=8, mergeCall=F, minTraceSize=10, disjointUnion

(l) z3opt, branchingLoop=Dis, stepTimeout=8, maxActivePath=8, mergeCall=F, minTraceSize=10, disjointUnion



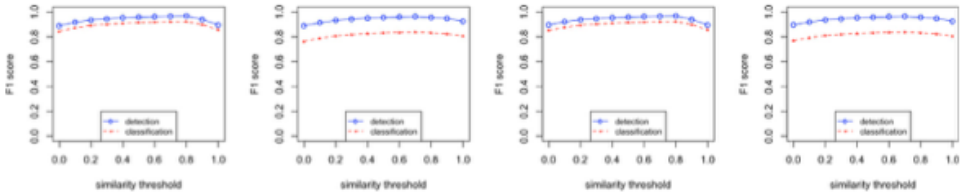
(m) z3dflt, branchingLoop=Dis, stepTimeout=8, maxActivePath=8, mergeCall=T, minTraceSize=0, disjointUnion

(n) z3opt, branchingLoop=Dis, stepTimeout=8, maxActivePath=8, mergeCall=T, minTraceSize=0, disjointUnion

(o) z3dflt, branchingLoop=Dis, stepTimeout=8, maxActivePath=8, mergeCall=T, minTraceSize=10, disjointUnion

(p) z3opt, branchingLoop=Dis, stepTimeout=8, maxActivePath=8, mergeCall=T, minTraceSize=10, disjointUnion

Figure 6.17 – Graph similarity vs. F-scores (4/8)

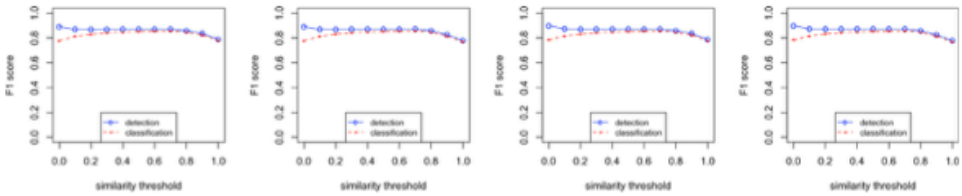


(a) z3dflt, branchingLoop=4, stepTimeout=None, maxActivePath=Unl, mergeCall=F, minTraceSize=0, traceMerge

(b) z3opt, branchingLoop=4, stepTimeout=None, maxActivePath=Unl, mergeCall=F, minTraceSize=0, traceMerge

(c) z3dflt, branchingLoop=4, stepTimeout=None, maxActivePath=Unl, mergeCall=F, minTraceSize=10, traceMerge

(d) z3opt, branchingLoop=4, stepTimeout=None, maxActivePath=Unl, mergeCall=F, minTraceSize=10, traceMerge

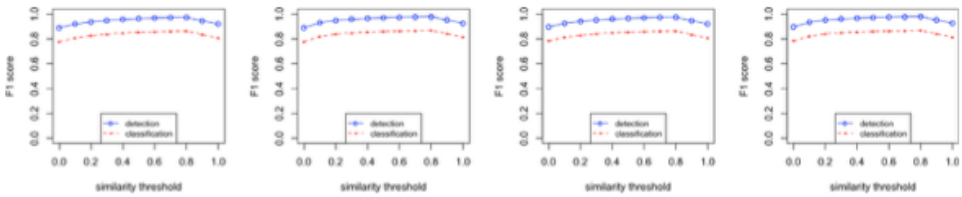


(e) z3dflt, branchingLoop=4, stepTimeout=None, maxActivePath=Unl, mergeCall=T, minTraceSize=0, traceMerge

(f) z3opt, branchingLoop=4, stepTimeout=None, maxActivePath=Unl, mergeCall=T, minTraceSize=0, traceMerge

(g) z3dflt, branchingLoop=4, stepTimeout=None, maxActivePath=Unl, mergeCall=T, minTraceSize=10, traceMerge

(h) z3opt, branchingLoop=4, stepTimeout=None, maxActivePath=Unl, mergeCall=T, minTraceSize=10, traceMerge

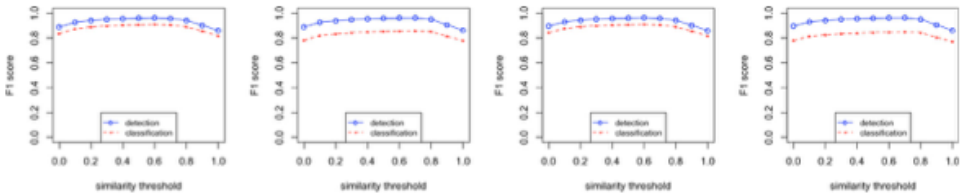


(i) z3dflt, branchingLoop=4, stepTimeout=None, maxActivePath=Unl, mergeCall=F, minTraceSize=0, disjointUnion

(j) z3opt, branchingLoop=4, stepTimeout=None, maxActivePath=Unl, mergeCall=F, minTraceSize=0, disjointUnion

(k) z3dflt, branchingLoop=4, stepTimeout=None, maxActivePath=Unl, mergeCall=F, minTraceSize=10, disjointUnion

(l) z3opt, branchingLoop=4, stepTimeout=None, maxActivePath=Unl, mergeCall=F, minTraceSize=10, disjointUnion



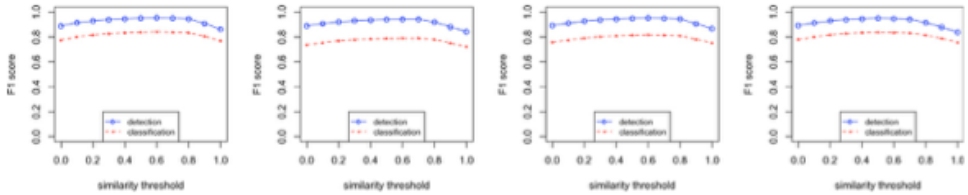
(m) z3dflt, branchingLoop=4, stepTimeout=None, maxActivePath=Unl, mergeCall=T, minTraceSize=0, disjointUnion

(n) z3opt, branchingLoop=4, stepTimeout=None, maxActivePath=Unl, mergeCall=T, minTraceSize=0, disjointUnion

(o) z3dflt, branchingLoop=4, stepTimeout=None, maxActivePath=Unl, mergeCall=T, minTraceSize=10, disjointUnion

(p) z3opt, branchingLoop=4, stepTimeout=None, maxActivePath=Unl, mergeCall=T, minTraceSize=10, disjointUnion

Figure 6.18 – Graph similarity vs. F-scores (5/8)

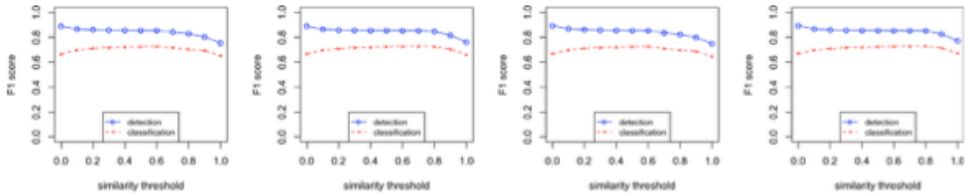


(a) z3dflt, branchingLoop=4, stepTimeout=None, maxActivePath=8, mergeCall=F, minTraceSize=0, traceMerge

(b) z3opt, branchingLoop=4, stepTimeout=None, maxActivePath=8, mergeCall=F, minTraceSize=0, traceMerge

(c) z3dflt, branchingLoop=4, stepTimeout=None, maxActivePath=8, mergeCall=F, minTraceSize=10, traceMerge

(d) z3opt, branchingLoop=4, stepTimeout=None, maxActivePath=8, mergeCall=F, minTraceSize=10, traceMerge

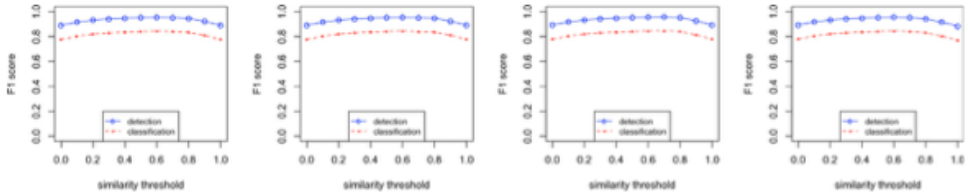


(e) z3dflt, branchingLoop=4, stepTimeout=None, maxActivePath=8, mergeCall=T, minTraceSize=0, traceMerge

(f) z3opt, branchingLoop=4, stepTimeout=None, maxActivePath=8, mergeCall=T, minTraceSize=0, traceMerge

(g) z3dflt, branchingLoop=4, stepTimeout=None, maxActivePath=8, mergeCall=T, minTraceSize=10, traceMerge

(h) z3opt, branchingLoop=4, stepTimeout=None, maxActivePath=8, mergeCall=T, minTraceSize=10, traceMerge

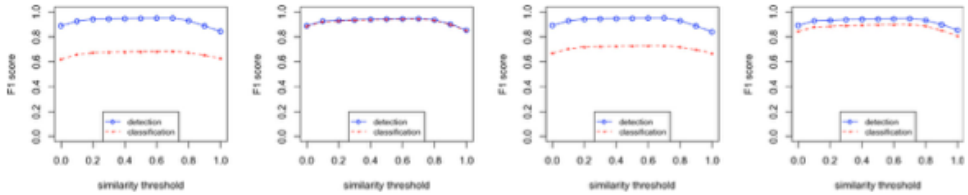


(i) z3dflt, branchingLoop=4, stepTimeout=None, maxActivePath=8, mergeCall=F, minTraceSize=0, disjointUnion

(j) z3opt, branchingLoop=4, stepTimeout=None, maxActivePath=8, mergeCall=F, minTraceSize=0, disjointUnion

(k) z3dflt, branchingLoop=4, stepTimeout=None, maxActivePath=8, mergeCall=F, minTraceSize=10, disjointUnion

(l) z3opt, branchingLoop=4, stepTimeout=None, maxActivePath=8, mergeCall=F, minTraceSize=10, disjointUnion



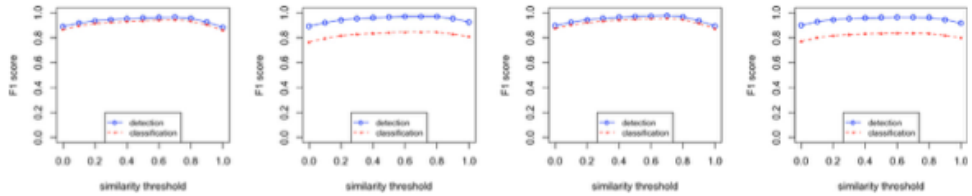
(m) z3dflt, branchingLoop=4, stepTimeout=None, maxActivePath=8, mergeCall=T, minTraceSize=0, disjointUnion

(n) z3opt, branchingLoop=4, stepTimeout=None, maxActivePath=8, mergeCall=T, minTraceSize=0, disjointUnion

(o) z3dflt, branchingLoop=4, stepTimeout=None, maxActivePath=8, mergeCall=T, minTraceSize=10, disjointUnion

(p) z3opt, branchingLoop=4, stepTimeout=None, maxActivePath=8, mergeCall=T, minTraceSize=10, disjointUnion

Figure 6.19 – Graph similarity vs. F-scores (6/8)

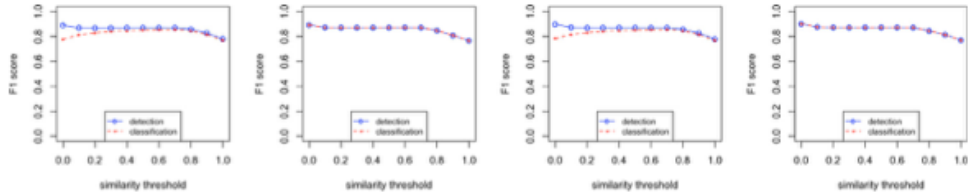


(a) z3dflt, branchingLoop=4, stepTimeout=8, maxActivePath=Unl, mergeCall=F, minTraceSize=0, traceMerge

(b) z3opt, branchingLoop=4, stepTimeout=8, maxActivePath=Unl, mergeCall=F, minTraceSize=0, traceMerge

(c) z3dflt, branchingLoop=4, stepTimeout=8, maxActivePath=Unl, mergeCall=F, minTraceSize=10, traceMerge

(d) z3opt, branchingLoop=4, stepTimeout=8, maxActivePath=Unl, mergeCall=F, minTraceSize=10, traceMerge

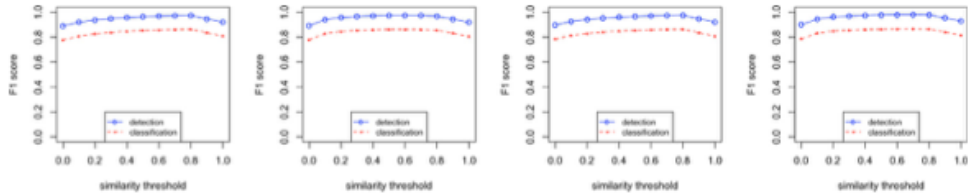


(e) z3dflt, branchingLoop=4, stepTimeout=8, maxActivePath=Unl, mergeCall=T, minTraceSize=0, traceMerge

(f) z3opt, branchingLoop=4, stepTimeout=8, maxActivePath=Unl, mergeCall=T, minTraceSize=0, traceMerge

(g) z3dflt, branchingLoop=4, stepTimeout=8, maxActivePath=Unl, mergeCall=T, minTraceSize=10, traceMerge

(h) z3opt, branchingLoop=4, stepTimeout=8, maxActivePath=Unl, mergeCall=T, minTraceSize=10, traceMerge

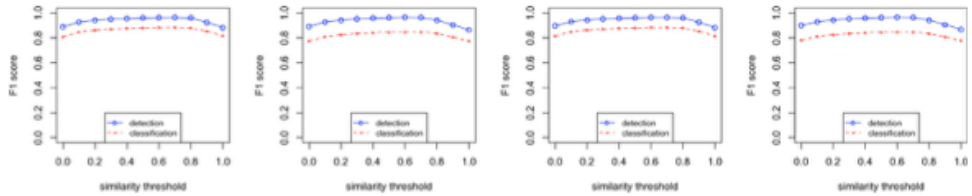


(i) z3dflt, branchingLoop=4, stepTimeout=8, maxActivePath=Unl, mergeCall=F, minTraceSize=0, disjointUnion

(j) z3opt, branchingLoop=4, stepTimeout=8, maxActivePath=Unl, mergeCall=F, minTraceSize=0, disjointUnion

(k) z3dflt, branchingLoop=4, stepTimeout=8, maxActivePath=Unl, mergeCall=F, minTraceSize=10, disjointUnion

(l) z3opt, branchingLoop=4, stepTimeout=8, maxActivePath=Unl, mergeCall=F, minTraceSize=10, disjointUnion



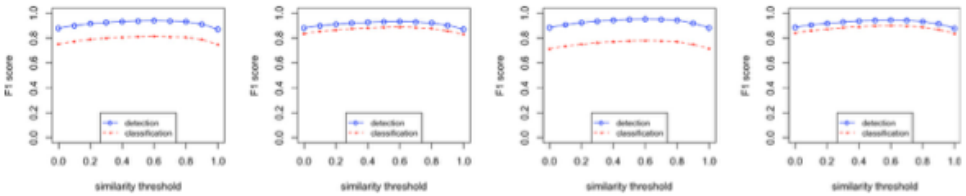
(m) z3dflt, branchingLoop=4, stepTimeout=8, maxActivePath=Unl, mergeCall=T, minTraceSize=0, disjointUnion

(n) z3opt, branchingLoop=4, stepTimeout=8, maxActivePath=Unl, mergeCall=T, minTraceSize=0, disjointUnion

(o) z3dflt, branchingLoop=4, stepTimeout=8, maxActivePath=Unl, mergeCall=T, minTraceSize=10, disjointUnion

(p) z3opt, branchingLoop=4, stepTimeout=8, maxActivePath=Unl, mergeCall=T, minTraceSize=10, disjointUnion

Figure 6.20 – Graph similarity vs. F-scores (7/8)

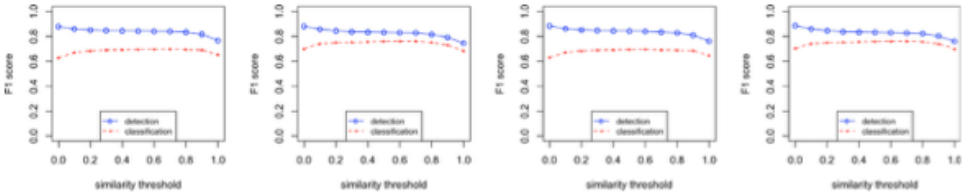


(a) z3dflt, branchingLoop=4, stepTimeout=8, maxActivePath=8, mergeCall=F, minTraceSize=0, traceMerge

(b) z3opt, branchingLoop=4, stepTimeout=8, maxActivePath=8, mergeCall=F, minTraceSize=0, traceMerge

(c) z3dflt, branchingLoop=4, stepTimeout=8, maxActivePath=8, mergeCall=F, minTraceSize=10, traceMerge

(d) z3opt, branchingLoop=4, stepTimeout=8, maxActivePath=8, mergeCall=F, minTraceSize=10, traceMerge

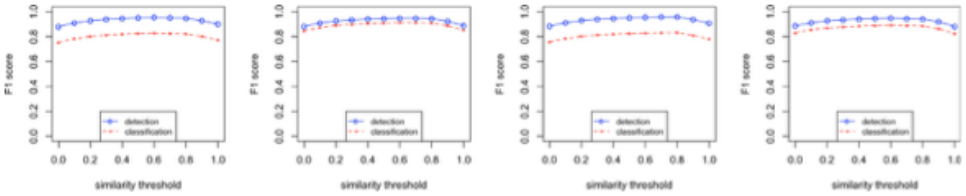


(e) z3dflt, branchingLoop=4, stepTimeout=8, maxActivePath=8, mergeCall=T, minTraceSize=0, traceMerge

(f) z3opt, branchingLoop=4, stepTimeout=8, maxActivePath=8, mergeCall=T, minTraceSize=0, traceMerge

(g) z3dflt, branchingLoop=4, stepTimeout=8, maxActivePath=8, mergeCall=T, minTraceSize=10, traceMerge

(h) z3opt, branchingLoop=4, stepTimeout=8, maxActivePath=8, mergeCall=T, minTraceSize=10, traceMerge

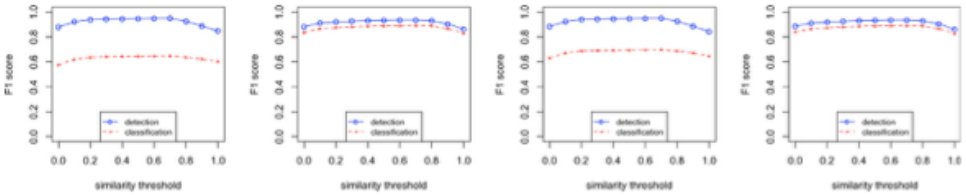


(i) z3dflt, branchingLoop=4, stepTimeout=8, maxActivePath=8, mergeCall=F, minTraceSize=0, disjointUnion

(j) z3opt, branchingLoop=4, stepTimeout=8, maxActivePath=8, mergeCall=F, minTraceSize=0, disjointUnion

(k) z3dflt, branchingLoop=4, stepTimeout=8, maxActivePath=8, mergeCall=F, minTraceSize=10, disjointUnion

(l) z3opt, branchingLoop=4, stepTimeout=8, maxActivePath=8, mergeCall=F, minTraceSize=10, disjointUnion



(m) z3dflt, branchingLoop=4, stepTimeout=8, maxActivePath=8, mergeCall=T, minTraceSize=0, disjointUnion

(n) z3opt, branchingLoop=4, stepTimeout=8, maxActivePath=8, mergeCall=T, minTraceSize=0, disjointUnion

(o) z3dflt, branchingLoop=4, stepTimeout=8, maxActivePath=8, mergeCall=T, minTraceSize=10, disjointUnion

(p) z3opt, branchingLoop=4, stepTimeout=8, maxActivePath=8, mergeCall=T, minTraceSize=10, disjointUnion

Figure 6.21 – Graph similarity vs. F-scores (8/8)

6.6.2 Evaluation Dataset

Table 6.1 – Evaluation dataset (1/5)

Family	SHA1
jeefo	f1f06713bfce019a1d33185bf65dbd3ff4854b34
	12c35183ab2c960234e8b3224fea8653d14d882e
	f59b3f97f596daba094e974b90c37fd111add32a
	7324e63211332c945c0db095317409dba5b8aa89
	4971eae31449da08f120d6d4cab2ddf57c1a5115
	8eb11682498a8291aba5c0836c7992a6dcbe54ed
	667194c6c6f672a0a77d89937073072d3b707862
	31448f16a4845ebfa4864dabc000b54871e03acf
	41e97ad145fb417cd9a214b8e169ea1bb43d5850
	8c5903f36018d0613f70704d0b12adeb9daa4ce2
	6b998c847c35691fdf71adcfed5cc759ecfdd11b
	02c0362de786d276eb5dd67065eef7e49ad6d4cc
	f5b4782760fe2c064ca9d97f0f39b7fcb03957a5
	ae51bc9ed3f9d90ef2bda01fd76d22c97e46cdb6
	bf2d30db511ee0d469de55386bf6e7849439a45c
	ef827d0aefff1572e1816c907cea4b4612142375
	9a04faa46701d9ef53852dd896ec9312b580d9d6
	dce188f3809291ac046a72bb78265b182b2e7c75
	fb030359343a43d4d6ff142589dd58b1db9a8fc2
	1a60a66557f0c4e82629b88ddb18886f6bbeccba
	150b079cdefa6d664a63fe64107b31af1a258350
	c1af341a10eac11cafa0c51a3c7165fb6020eacd
	37f0a683ebfad0af8c590345372c0bc95ec63de5
	0fb629bfdfccaa12405e141a418edc2f13c771ac
	17b547234616c69682cc7c11fa6f8b75626ebfdf
gamemodding	e60663028faa214e2ab493c44019365f6ac7a349
	5bd39ababe69d4023089e53f3ca23a2e55ea1c06
	6f3a5a4f7884363947efe33f66d9ebcfc361e472
	83849d556715a99ccd98b78ec9aa1db783a60251
	641fe0e576267a1a582409283776645104a9270e
	45b1c294e3674883257d4ebdcd75a19f8bb2d751
	6fb04001558d7b9c7e27204c6a338d343f05cc15
	9fac4eed08c9d16e05605116a22c58b47fa6a67f
	44547232398772218535157ff88eea48fed5d7d1
	a003ffe778c404eed9b5f3c8e721fe1afc81a40f
	d3f6956727f02b5b832aecf4afc6792c873aa118
	73f0b47703243aa61d3ed3282e62a90bb68df102
	66a422dffe441a3a3f3347044078cc5c70d05353
	5f9f64952b42f1b95fb47fbe99a4dd3e511712b8
	46e9671f16a363f2033d1756ce171032107b5a48
	614dcd5d165edae6d9cdba83303f1984c74addb3
	7e528bb6a4c608bff1175061aaeae84061008054
	f6a41df6b8442e9c5b2a9b5bb3821872b6e3a25e
	2bb11518070e7d7e4ba1b02bb56cd4140eae3c5a
	fa7824cf823126910eb1c8e06276bbb31d117c12
	fb4533c5aeb3df9a4d0278a7d3fb588da4cb71c3
	41b6b8e2f24202503328c417e502a02105419684
	6342e8acbafe064d22753400f2696140e91e8773
	0c33ba4fc177e2c2c017ea224b22b60ff48357d4
	875c71ce5f363e6fd90d20d8d8eea838a2c3926e

Table 6.2 – Evaluation dataset (2/5)

Family	SHA1
mira	09cebe61bd8a85fd1b8e15a87bc528e50e9146d1
	8b171470662a7f77c711d1f8219f742b67bf6636
	8be30a111087292144c4a105b1e59336c646c7b2
	9f93c3603f91455ffa10611ed505682ad3750222
	f2678a18b5760e50e1496cc10793b8dd0d43babb
	8735fb97613720390ff44ec4088e19a6bf20fc82
	7301f70dab808be9cb9d61eb829c0ac84e22fed6
	22531cac47269c6b542700e11686ba310b474425
	4c2716baa548f10a97cb581ac3b3ff22970dbad3
	d1dac653c287606c58c1344a21bc98ff1cab33ae
	dbb489de8a845ce8cec003f9a73c7b848dc4c04d
	69dff1bdad1773ec9aa187b33cd0b1c19e8e3cc9
	cab35a276d1631ef5b62a91c0b1fe51e95dd4e7b
	b74cf183dfdadf124ef29954d855eb0d4493a603
	8777e4a3528e399e94223f4f8b0e9d0c29fe6c0a
	ec3fa64c6a0778e610b59c9639abed71e408984c
	59ac1c3502cce4128fe56e953aa59614275eaf1d
	03c60cb6dfe950ab8abfc79db64a4485304ed4b4
	f19390b920a515b752f5c5827d7e909e32912caf
	4c68e704f180f51f60c5864760837802ae42ed4b
	7fb49935dc90a6ca38f3e287f6862f335dfe1ca3
	a69aa13f110a258e6be1edfa14c9cf6453440cd7
	5ef1d37d3ac6e376719af055956a3f7ccb961b05
	b993fca2a7e5475de0503ac6fe53ec502ffdda71
	042e5606a5469abefcff6f63f76933459baa3cf5
installbrain	42314c2e1ddc0366aef7e6227f23e170fd34e517
	312596700c2326168e08d2c339400e8b740317f1
	d5e9fc66f70d40bff172470f3e8c76affaeabba8
	2ae5536f48452423d7f32db71460cabd16107df6
	a082516a4389503d15f4069922e1b18e2537f750
	87a116c78c4cca00024cbbc672ab3a2bb3dec27c
	2ce3f5baa7d7f4b2e3a64e8b9e8549c3dfc945a6
	dfdb89211d2f15e92cab0cb0f42da063e1d3b840
	00364bf2ad09a975efacdd420876d29199a1e388
	1e2aeaf7e811fdb550302a0603b0b30fb370f503
	5f3272cd501f346e6075a58ba94ff977e241b225
	2839fda2772f066150dca6b7d95efa33e85bfc79
	667e6f22bf45c38fc33059e9f7399295b2d897f8
	8d8e373ed0845ed4ed2d983ee2b1c5b00cad9076
	ead0d438833687ba989318c9e27290da5581191b
	e7be4cb61f06a72cc1b300508608d6689edd68d2
	03c437977840383bb01b9581cb3eb343cfa852a4
	21029fa0e28350101d038ad6d27ca251666d6b2e
	c443922da881d8ad5fd3ea0e40ae5ffd4938df6c
	9cb055b6f6df6e3e42372b2dc60771f33cf80fb4
	26a55c1e24e3429bae6de47e50d7c46cdc4115ce
	78339b24ffd2d0e8801b889aa00b8d9255830772
	8bf70b22639c9bd822f90778327a521d19e37ca2
	7ca87cd015492dea5eeff97d023add97efe277ba
	fc9d8bd601f53343bf0f5c4116d2f840423eec70

Table 6.3 – Evaluation dataset (3/5)

Family	SHA1
addrop	ed268e89978f59dbd6a34613a251322b1f7e7f70
	9ee17a6fe4da450ab751712264af58bdfe8cdab8
	a5216221219a3fa2f0e03f742f62f3c670a68b34
	f058a34fa7a0ad2dfcae1edcad9b42bae31e15dd
	13dce4594d01c430e3098d5da26bf7cec629ad0c
	5c632ad6f5029e05becd4b47924a29c2ba6033af
	2ae7f1ef08b8657de3060068bbdcf375e2f20cd3
	f8a5215a4bffc2fc90f115e914fb26b5817d50e3
	adeda6a02c5312bb12fceaf71a055627f50ba180
	9c240d316d0b8766a5cf6ec07e5aa4fac6c819cc
	d97825ab75f08a1efbc924e2969487a0a6408e7f
	65f46c8a82782640e502396a107f4403fbbacc72
	8fcd4ffcd87c1283a470aeb5184ae5dafa42fd54
	37c443875ebbf6056ccaf22ed6f184e724520336
	cfb14ce62e981a610f88be8811e1dc343db12b92
	7188044570e204369e3bab83039ee56753a381cb
	010b38298a4a4d1e256a27e87cc0d1f34f47b64a
	14606a9aada603389684160b59794a766efe9f0b
	181433c9aa6780710b44dd1c48ad8796dca18cb8
	e0fdd45fdf81ecde6823a4bf4ffccbc35cabd1cb
	c8ed6e3a1353616d0232b5ce0ec74fdfac61b4b1
	3e746871396e56491bcdf24c4d0120e15f98ef37
	5d62957020594d5e66d55d4d72921154475ff8a9
	c060d2975a5089b6501bbb8839d856ee5dc931a6
	e51ecb575bbd772b1e7e7391655e777514d8b8d5
couponmarvel	9e25305bc47f2046e133b2f09b3b828715da811e
	07fc7ccac20b4f412876e7a992c340a0a65e2f63
	17bb3448fae7c6b3409feaa6002a82348d193949
	a5f1500704238efae5084368028111066b89d05d
	1ecb611f1c29e9776baf619675fbc44403d534a3
	b3ea56b1995e14544202880c0cb211b197a9e95a
	96c3b39b61552fbe511ee7ada13a29334b4aa286
	c5c19840d6887f61e0e10df92ee95adb3af2ea6d
	eaa9a1a1383cac81e36189d9324aad0028dc68a1
	d2d5fe3d8ed8a4f0572c36e99e86226b0d827d24
	d94f7fecac0c27903ded491b233dfbb2a17e647b
	6828ec0826075baab6186545748db93677689b41
	2aeef97c7984303e3c02a0b70701ed2702dac98c
	6e04ed657682a9ba46bb8e1c19e26bd750722655
	aa55f46ed48b9be0989a3cbb58191bbd99533cc2
	9486e83fd4ca8f75e8c57135fb4e31d4849b65e3
	467cd38943f87ffabe4a848efa7da3f2c9bf95e8
	3e99db71a9b35285c920d576c989c184d5387e5b
	3b08d213f1938280cb49a215875a724eb40a7983
	6723086317f9721401ee2d874af7d9c32a831364
	8a95de1e8776d2ae66ab3d2d83148e59d3d27b0c
	363d98aab25bd882fb7732af3ec21d92cbeb483e
	afbc4811e111cf1df0d6740aef4f0ce57a0707e3
	4840f4f717359e13ff5d600021d037b4b85ee5b6
	5d165d2374bf547f299180ff8abcb8e2372b5906

Table 6.4 – Evaluation dataset (4/5)

Family	SHA1
multiplug	2cac3884cca13c4a24caf24235d5f0006913ea26
	63e138c4b9af4a951c753de8866328eec8cbe3d4
	a6f9db9d1717013e2dfe8e47e24e25f831ed5278
	7e91a2248abed93eafebfb2bf05810bb0b32802e
	d9a8f8e3d97e6d47045c4209371a424bdd977026
	01531ada6cc172dfa4e86d826222b09090cd956a
	afb72959f2b0b7d5f131130bc5fe21f9bde9c9ec
	0479e8499c1a3e94da5231e7aa47eff0d84cc5eb
	235fca383cb5ab9d892faf3608489cd56655ed87
	50771e305929168d44091e11e7491da35559a9b4
	be4c1c202a2583d9d49a8e9881f9f57b37ae16b2
	1e2575e266514625ca7591abd6807877842b91c1
	1d9d53f16af5ea60fef032102f710f0a2f0a098a
	3534182525c52ccec2f508d8664dc90899c54c6d
	3b6805317c906a98d34c3d483e720bccfb138922
	b47e7221fcee6b995728eec97b093e7a201990c9
	69a09fbb98d0f8679f0717f3bdbf32ae24bf69cc
	53304574aa944b6696c44f76f8ce45f75f3cf641
	a32a0faeeec65891f93157bc368dea54383f6d89
	381b398188662098c306121a92c7164b753eb1ee
	7a9f8f0e7a443ac2ed5d7dab4e5f9568a85c7479
	3c4b3dd2375681a3d59d1ea2af7deb8cdf4c6990
	e3df83d9b798be3284194bb4390579ac7d196873
	393580b1980ac00490dc8e14c68091e8ac5694ad
	afa991896b637884e7d60aaf6024a6457c96c0b0
detroie	c9d4d366439d144fdb92c60562cda4c9be0e0b04
	61e7feb76f046262e7b8ebd08450e50c4cef6ff1
	801c0bd52c63d8aa17491e4e7f017435314ae360
	62c42f510f93dd0ce86d5666449ac9d733b15e9a
	576a946cdfa6feadf06142ae06f55b7fed577115
	09031d53c01585fc2a1ff8358cb0541d2006e799
	3ab89751d14866299b2cb78eb41f931adc816717
	1aed3dd03d42b96a376598c7a53b5edab14e69f9
	e6fb750288954e94917a0f7a2ee7b5b271672885
	6abb9084345b9e2990e14fe907e6fd213cc47cc4
	e9abd5e7bf0d77723991df905615d14dd4e31692
	2cad3027ce9674f00cdfd7487e83956033dee1d8
	2739c2937b4de40924a0400a2200caa9f271f51d
	59037dc39c4a18d6b98280a9e598c83a33c3e64b
	9917463fd83f55c5afe4cd5fa5dec1ef8e2128e2
	13029bb8fd12b4ec54dc49ea49a90abdc2fc75ee
	04226396c0de4dce851f7529f88fdb96dd112d57
	6fc2c149f385fb2c35ade0b0dcea54c7ec4b9422
	97d69adc224831a186c5036e0985ce991177dae9
	bf63e4ae6b88b1fa3b7426adaf6e1ecbf6c9af4a
	7f6c111ac542f49c83a3a90514497f0ede8b7392
	2834829c99ef5cafcbcdfe0f5c194f3f0c096ad1
	bd287cfe53be4f13404d9c081db5cc223ab380ad
	6a63611845806950c9f3f03f9dc8adb660aae4a0
	144837eb6d598644ae5664eadaa3efe308d4a2cd

Table 6.5 – Evaluation dataset (5/5)

Family	SHA1
cleanware	168518b3b4aa0ae5c9ade193177ff1a0125ec665
	c00a10899c17f66ad4b47d9b2e57a9cef4004b6d
	934ce1475dcaff413a60d792143a582064a62f5b
	a1bf405584a3b93912c8ae0fb1aa137b4a1f387c
	0b1d02a4d87c2780dd3d8958cfe9b9b6757bc71d
	43b18970cb4571b8b516ec4eab5dcba6517b4dd8
	1ab2d93096bcc18c6956118860c3e9aaa4c2f8cf
	cdf15ab96e7d9e50bd98d20564b67bfa4113c8fb
	976d42ff400556a15946c1bad7d687212d0c96b3
	cfba36ab1f6a8842cfa405e922e6c261ac2b45a2
	60e870128b89da49334f107e289d7a74eef7ccad
	62ec427462f477ff810198aefd122af73a68c73c
	b8abcc65de8e8f8edc4b27569b34d10bb8c13e4d
	ee9847051a898d88060ca52bab0c229851acfbf0
	eed8781c92830087cc2cc41b507b7b6899ae8f17
	9adf02ba06b5297d95e8288cc50ae44ec9f8c28f
	a51959cb4e11467f0fbdf490be2078ff5d9d7391
	b63b81aa89fd8db3d9ff2386cc816d5849d95869
	39a6e114bc4f4bcaf9c915abc5158c98fa2ab4da
	1898cd90ad7095121511383170d5b3ee85325448
	139926599045c7de0b37c0906b51c0322c9ab82a
	f36ce7091903b73a6905460069877ddc209ad2e7
	e0a85bee630d831e3f1aa99bf87b2d7b379f1b40
	50260b3c03863a375f226c6e1e124b93c325c12e
	3b5ce23f1e64cb41f639b68fddc44a51f871f70a

6.7 Structural-Based Binary Code Similarity

6.7.1 NEF Selection: Detailed Analysis

This section details the analysis conducted to select the *nef* value for the AnR experiment. For this, we computed the evaluation metrics for different *nef* values using four different clustering algorithms (i.e. DBSCAN, HDBSCAN, OPTICS and Agglomerative). Furthermore, as DBSCAN requires the definition of input parameters ϵ and Agglomerative the values for linkage (λ) and distance threshold (τ), they also undergo a parameter exploration to pick the best value according to a given *tuned metric*.

The list of considered metrics is:

— Silhouette score

— (Mean) similarity score

— Similarity score

— Completeness score

— V-measure score

While silhouette score and mean similarity do not depend on any given labeling, the similarity, completeness and v-measure are ground truth-dependent. In the latter case, the labels are taken from the Yara rules defined in experiment of the *accuracy phase* (section 4.5.5, page 177).

We split the analysis into two different categories: tuning and no-tuning clustering algorithms. The former includes algorithms that require the definition of input parameters and therefore undergo a parameter exploration against a given *tuning metric*. The latter includes algorithms that do not require input parameters and therefore respond autonomously as the *nef* value varies.

No-tuning Clustering Algorithms

This category includes the following clustering algorithms:

— HDBSCAN

— OPTICS

This category is simpler to analyze since it does not involve any underlying parameter tuning.

HDBSCAN In figure 6.22 we see the variation of each metric as function of *nef*.

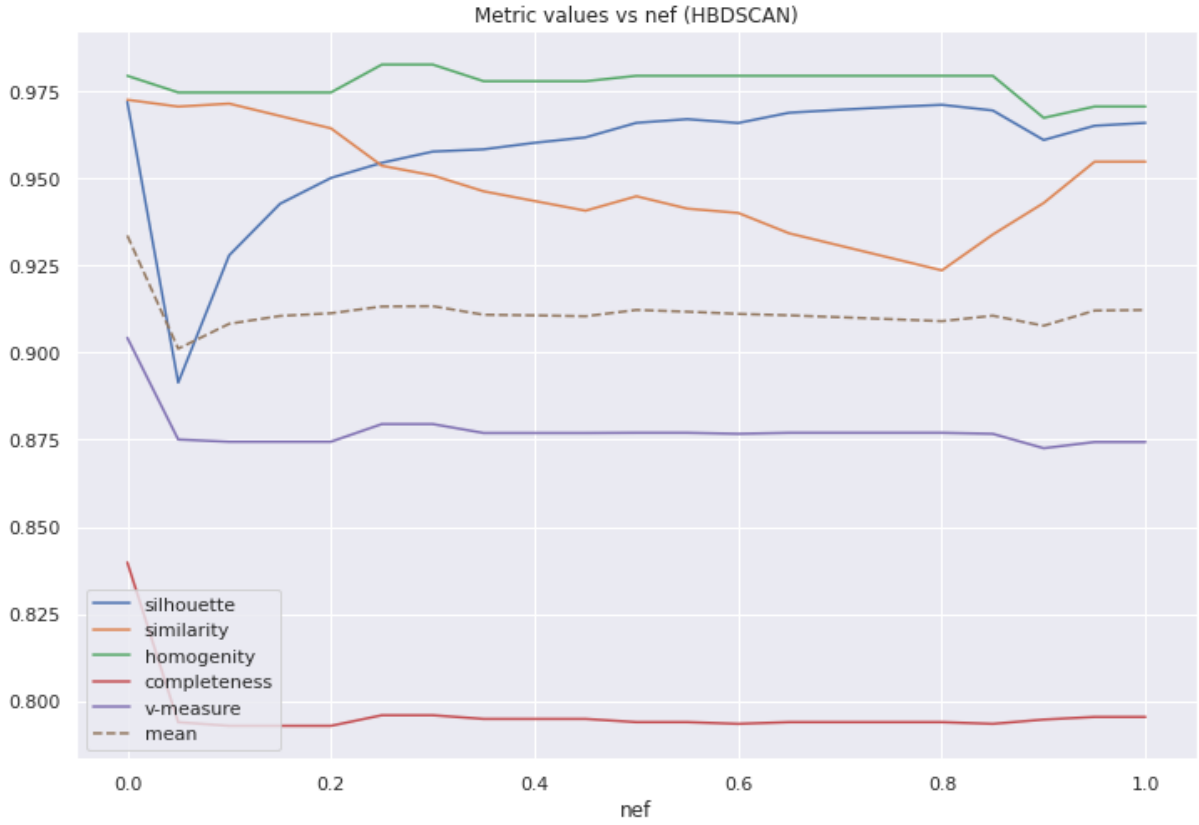


Figure 6.22 – Metric values for HBSCAN as function of *nef*

we observe that:

— the scores for completeness, homogeneity and v-score almost do not change;

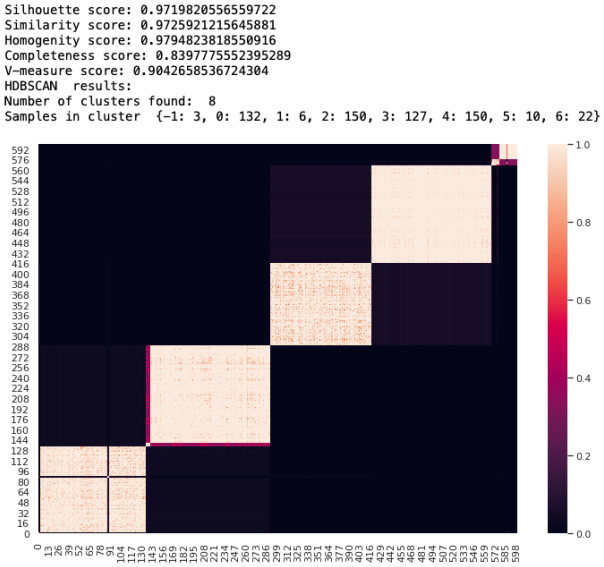
— for *nef* between 0 and 0.8, there is a somewhat stable regime where the silhouette grows (asymptotically) and the similarity reduces linearly;

— the lines for silhouette and similarity scores cross for *nef* between 0.20 and 0.205 (figure 6.23b);

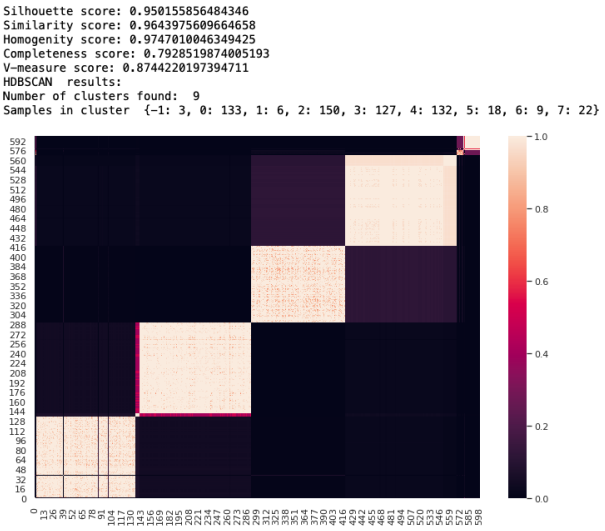
— the worst value for the similarity score is produced for *nef* equal 0.8 (figure 6.23c);

— the worst value for the silhouette score is produced for *nef* equal 0 (figure 6.23a);

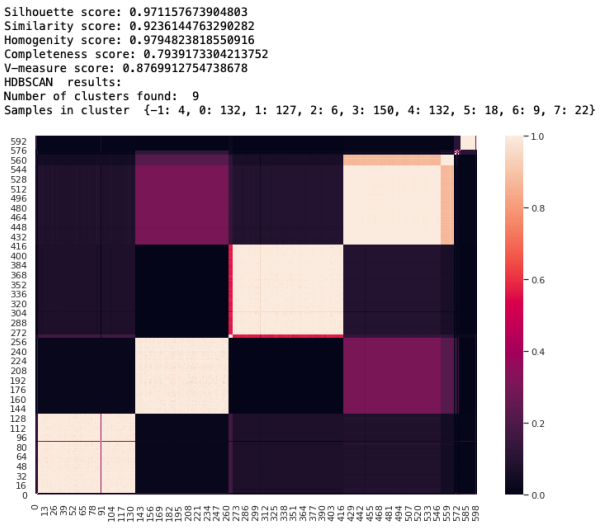
OPTICS In figure 6.24 we see the variation of each metric as function of *nef*.
we observe that:



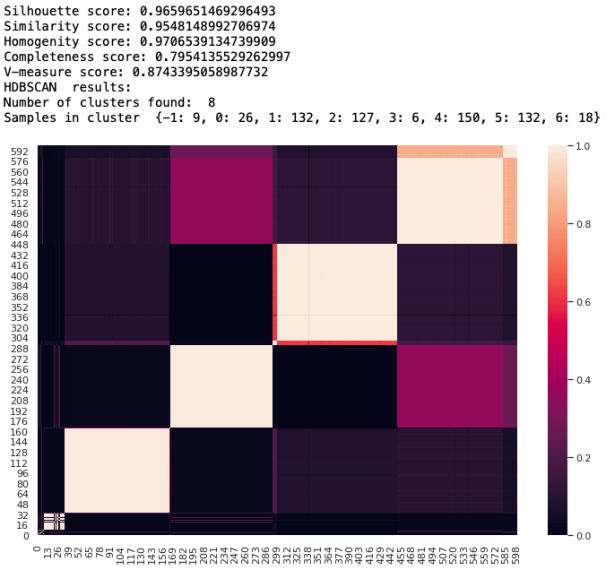
(a) *nef* =0.0



(b) *nef* =0.20



(c) *nef* =0.80



(d) *nef* =1.0

Figure 6.23 – Results for HBSCAN clustering with different *nef* values

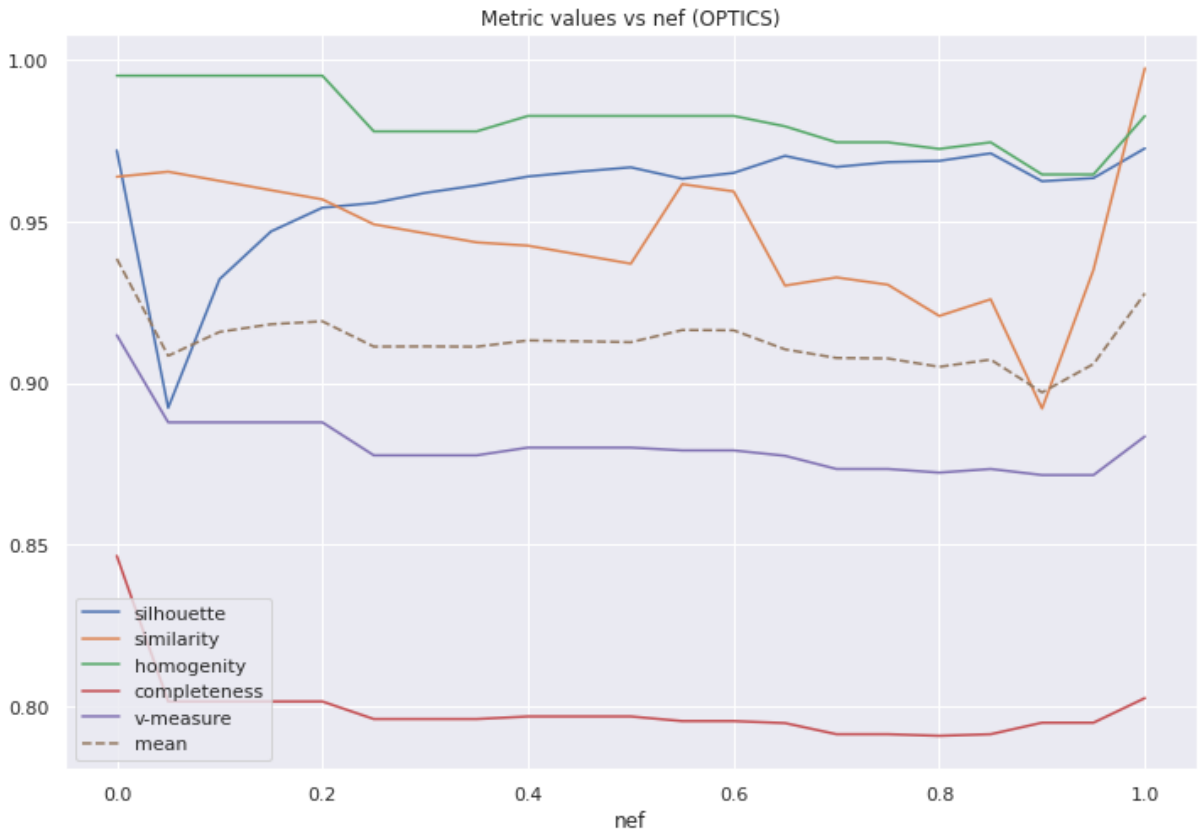


Figure 6.24 – Metric values for OPTICS as function of *nef*

— the score for completeness almost does not change;

— the score for homogeneity follows a step down pattern for *nef* equal 0.2, which impacts the v-measure score;

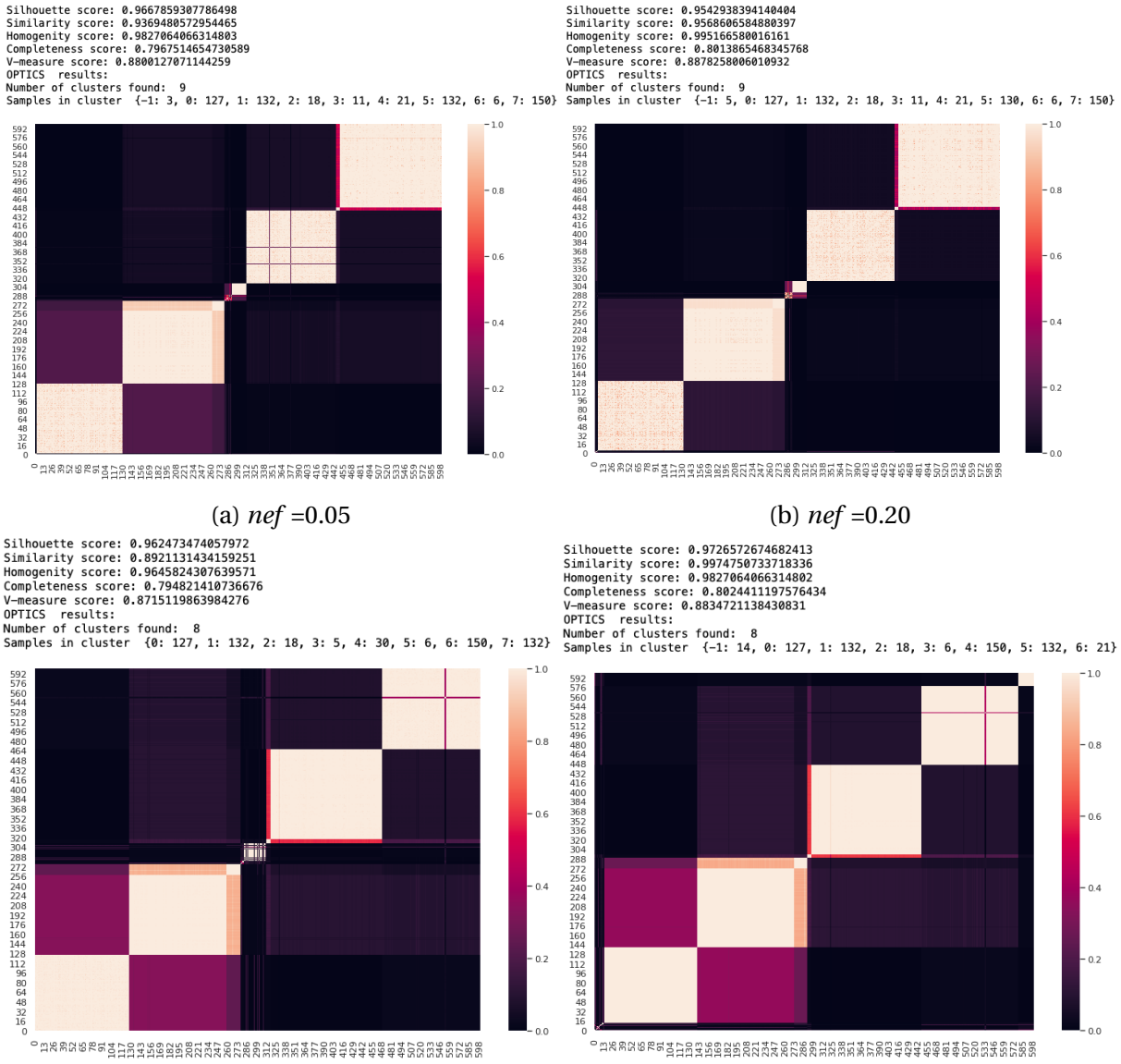
— the silhouette score follows a somewhat stable asymptotically growth trajectory;

— the similarity score decreases almost linearly for *nef* between 0 and 0.8, with exception of an interval between 0.5 and 0.65, where it grows rapidly. A fast growth is also found for $nef > 0.9$;

— the lines for silhouette and similarity scores cross for *nef* between 0.20 and 0.205, furthermore a step downwards is observed for the homogeneity score at this point (figure 6.25b);

— the worst value for the similarity score is produced for *nef* equal 0.8 (figure 6.25c);

— the worst value for the silhouette score is produced for *nef* equal 0.05 (figure 6.25a);



(a) *nef* =0.05

(b) *nef* =0.20

(c) *nef* =0.90

(d) *nef* =1.0

Figure 6.25 – Results for OPTICS clustering with different *nef* values

Tuning Clustering Algorithms

This category includes the following clustering algorithms:

— DBSCAN

— Agglomerative

DBSCAN In the figures 6.26a,6.26b, 6.26c, 6.26d, 6.26e we see the variation of each metric as function of *nef* by tuning DBSCAN by silhouette, similarity, homogeneity, completeness and v-measure respectively.
we observe that:

— tuning by *completeness* gives by far the worst average results for all metrics;

— tuning by *v-measure* or *silhouette* results in the best average scores;

— when tuned by *silhouette*, the *similarity* grows steps upwards for *nef* equal 0.5, whereas the *completeness* steps downwards. It suggests that more clusters are found.

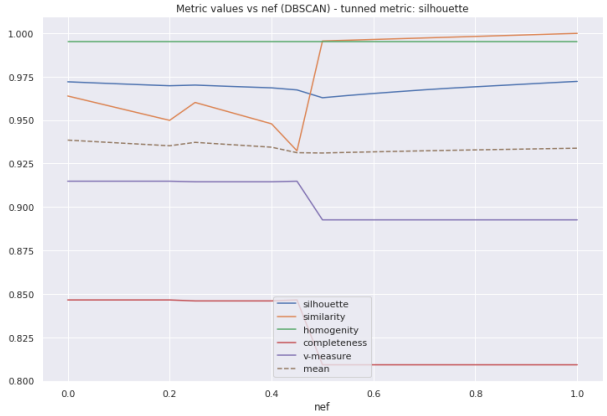
— when tuned by *silhouette*, the best value for *nef* < 0.5 is 0.25 (c.f. figure 6.27)

Agglomerative In the figures 6.28a,6.28b, 6.28c, 6.28d, 6.28e we see the variation of each metric as function of *nef* by tunning the Agglomerative clustering by silhouette, similarity, homogeneity, completeness and v-measure respectively.
We observe that:

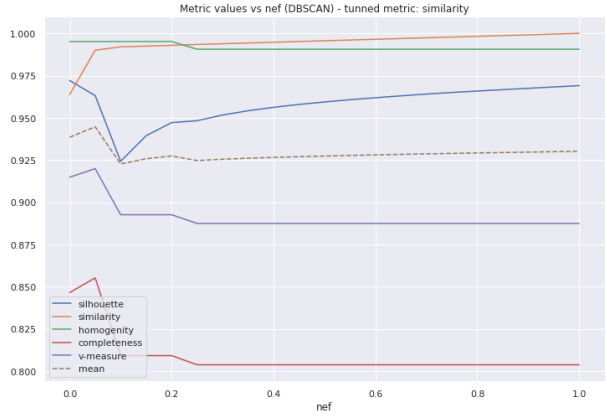
— tuning by *completeness* gives by far the worst average results for all metrics;

— tuning by *v-measure* or *silhouette* results in the best average scores;

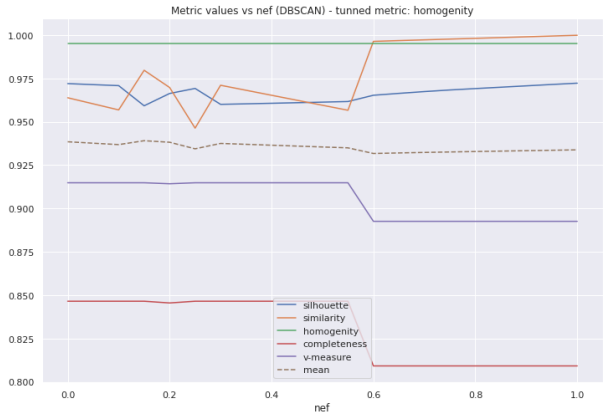
— when tuned by *silhouette*, the best result is obtained for *nef* equal 0.45 (c.f. figure 6.29)



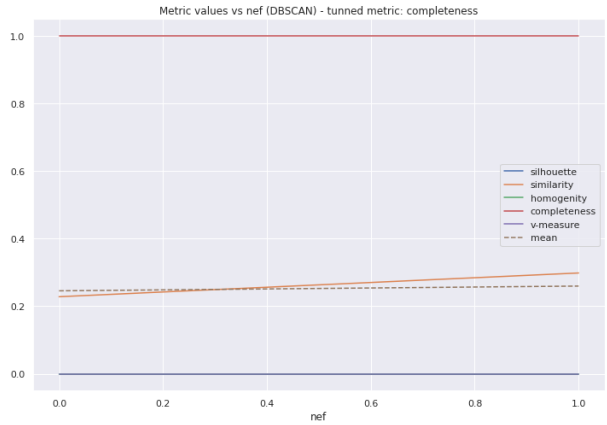
(a) Tuned by silhouette



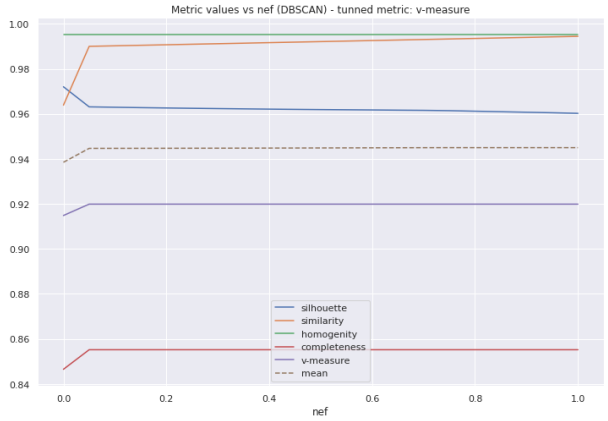
(b) Tuned by similarity



(c) Tuned by homogeneity



(d) Tuned by completeness



(e) Tuned by v-measure

Figure 6.26 – Results for DBSCAN as function of different tuning metrics

```
Silhouette score: 0.970149077531073
Similarity score: 0.9601410699196623
Homogenity score: 0.9951665800161608
Completeness score: 0.8459319591320458
V-measure score: 0.9145009860094633
DBSCAN  results:
Number of clusters found:  8
Samples in cluster  {-1: 6, 0: 127, 1: 150, 2: 10, 3: 21, 4: 6, 5: 150, 6: 130}
```

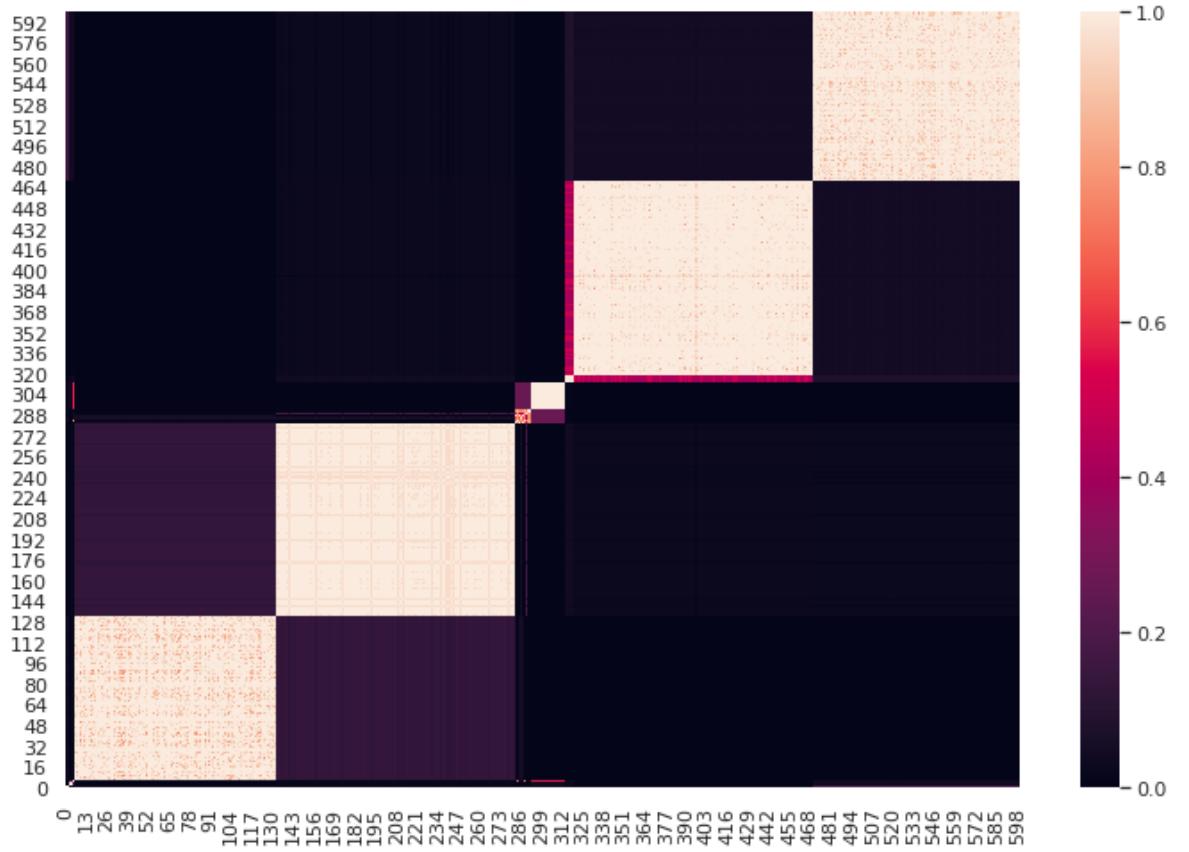
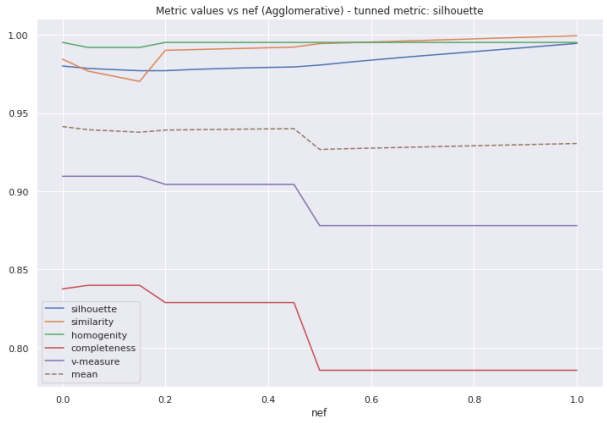
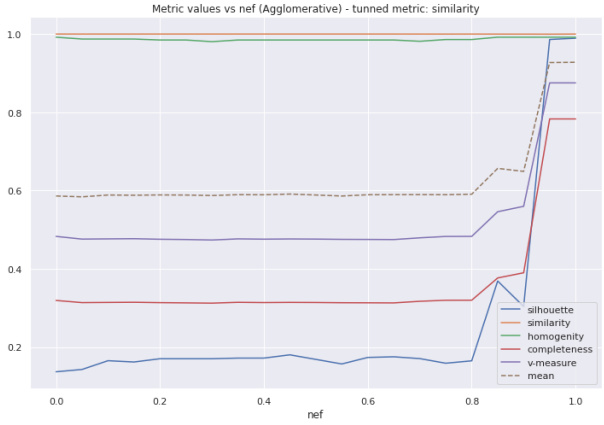


Figure 6.27 – DBSCAN clustering (*nef* =0.25, tunned by silhouette)

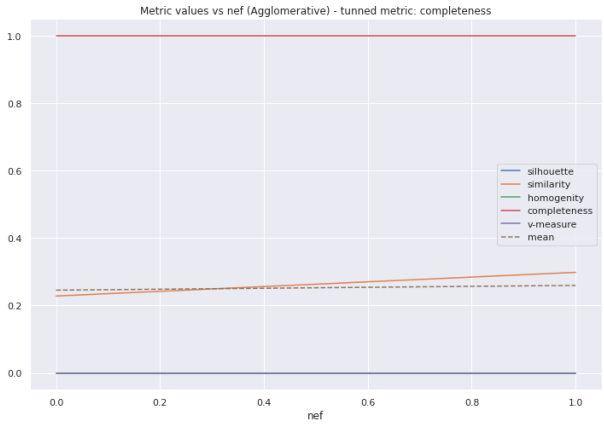


(a) Tuned by silhouette

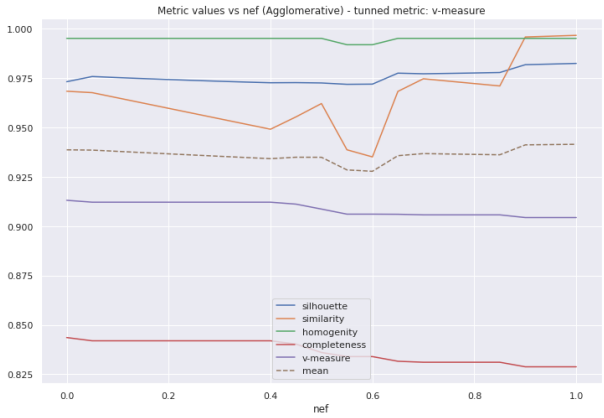


(b) Tuned by similarity

(c) Tnned by homogeneity



(d) Tuned by completeness



(e) Tuned by v-measure

Figure 6.28 – Results for Agglomerative as function of different tuning metrics

```
Silhouette score: 0.9794611960329693
Similarity score: 0.9921724503557144
Homogenity score: 0.995166580016161
Completeness score: 0.8287779314607285
V-measure score: 0.9043828849560831
Agglomerative  results:
Number of clusters found:  15
Samples in cluster  {-1: 3, 0: 2, 1: 150, 2: 127, 3: 150, 4: 130, 5: 3, 6: 2, 8: 2, 9: 6, 11: 21, 12: 2, 14: 2}
```

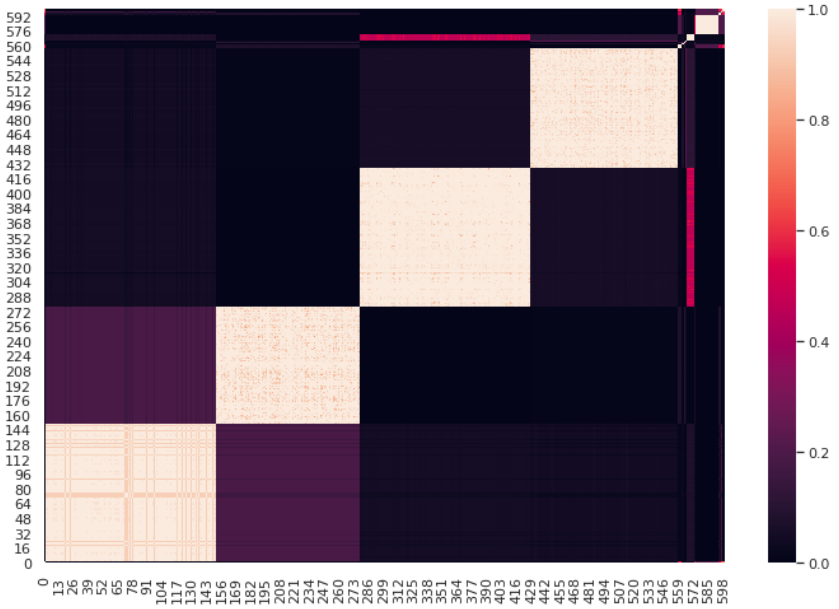


Figure 6.29 – Agglomerative clustering (*nef* =0.45, tuned by silhouette)

6.8 EMB-DUET: Multi-Feature Clustering Based on Numerical Embedding

6.8.1 SVM

Distance Between Hyperplanes

Here we show that the distance between the margins in SVM is equal to $2/||w||$. For this, let x_0 be a point in the hyperplane $\langle w, x\rangle + b = -1$ (i.e. class -1) and d denote the perpendicular distance from x_0 to the hyperplane $\langle w, x\rangle + b = 1$ (i.e. class 1) —the distance that we want to find is precisely d.

The point on the hyperplane of class 1 corresponds to:

$$\langle w, x_0 + d\frac{w}{||w||}\rangle + b = 1$$

because $w/||w||$ is a unit normal vector to the hyperplanes.

Expanding this equation, we have:

$$\begin{aligned}
\langle w, x_0\rangle + \langle w, d\frac{w}{||w||}\rangle + b &= 1 \\
\langle w, x_0\rangle + d\frac{||w||^2}{||w||} + b &= 1 \\
\langle w, x_0\rangle + b &= 1 - d||w|| \\
-1 &= 1 - d||w|| \\
d &= \frac{2}{||w||}
\end{aligned}$$

□

6.8.2 EMB-DUET

Scatter plot having the data points of the different OPTICS clusterings with respect to their AMI and homogeneity score:

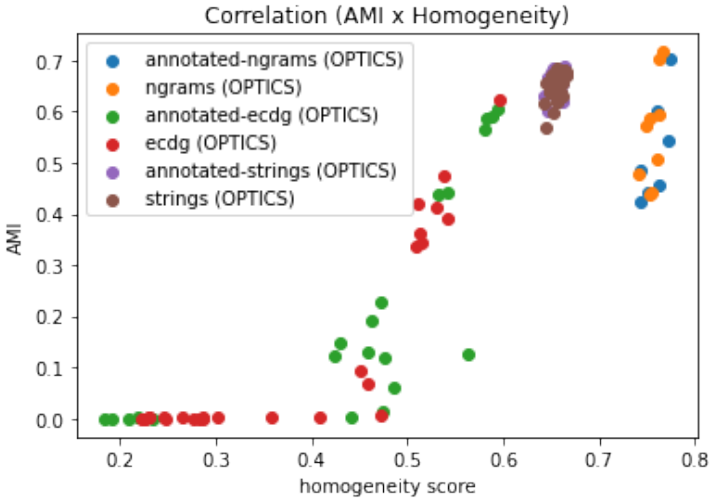


Figure 6.30 – Correlation plot (AMI x Homogeneity score)

UNIVERSITÉ DE RENNES 1

Titre : Classification de Logiciels Malveillants Dirigée par les Données et Assistée par des Méthodes d'Apprentissage Automatique

Mot clés : classification des logiciels malveillants, analyse des logiciels malveillants, exécution symbolique, partitionnement de données

Résumé : Historiquement, l'analyse des logiciels malveillants (ou *malware*, MW) a fortement fait appel au savoir-faire humain pour la création manuelle de signatures permettant de détecter et de classer les MW. Cette procédure est très coûteuse et prend beaucoup de temps, ce qui ne permet pas de faire face aux scénario modernes de cybermenaces. La solution consiste à automatiser largement l'analyse des MW.

Dans ce but, la classification des MW permet d'optimiser le traitement de grands corpus de MW en identifiant les ressemblances entre des instances similaires. La classification des MW est donc une activité clé liée à l'analyse des MW.

Cette thèse aborde le problème de la classification des MW en adoptant une approche pour laquelle l'intervention humaine est évitée autant que possible. De plus, nous contournons la subjectivité inhérente à l'analyse humaine en concevant la classification uniquement à partir de données directement issues de l'analyse des MW, adoptant ainsi une approche dirigée par les données. Notre objectif est d'améliorer l'automatisation de l'analyse des MW et de la combiner avec des méthodes d'apprentissage automatique capables de repérer et de révéler de manière autonome des points communs imprévisibles au sein des données.

Nous avons échelonné notre travail en trois étapes. Dans un premier temps, nous nous sommes concentrés sur l'amélioration de l'analyse des MW et sur son automatisation, étudiant de nouvelles façons d'exploiter l'exécution symbolique dans l'analyse des MW et développant un cadre d'exécution distribué pour augmenter notre puissance de calcul. Nous nous sommes ensuite concentrés sur la représentation du comportement des MW, en accordant une attention particulière à sa précision et à sa robustesse. Enfin, nous nous sommes focalisés sur le partitionnement des MW, en concevant une méthodologie qui qui ne restreint pas la combinaison des caractéristiques syntaxiques et comportementales, et qui monte bien en charge en pratique.

Quant à nos principales contributions, nous revisitions l'usage de l'exécution symbolique pour l'analyse des MW en accordant une attention particulière à l'utilisation optimale des tactiques des solveurs SMT et aux réglages des hyperparamètres ; nous concevons un nouveau paradigme d'évaluation pour les systèmes d'analyse des MW ; nous formulons une représentation compacte du comportement sous la forme de graphe, ainsi qu'une fonction associée pour le calcul de la similarité par paire, qui est précise et robuste ; et nous élaborons une nouvelle stratégie de partitionnement des MW basée sur un partitionnement d'ensemble flexible en ce qui concerne la combinaison des caractéristiques syntaxiques et comportementales.

Title: Data-Driven Malware Classification Assisted by Machine Learning Methods

Keywords: malware classification, malware analysis, symbolic execution, clustering

Abstract:

Historically, malware (MW) analysis has heavily resorted to human savvy for manual signature creation to detect and classify MW. This procedure is very costly and time consuming, thus unable to cope with modern cyber threat scenario. The solution is to widely automate MW analysis.

Toward this goal, MW classification allows optimizing the handling of large MW corpora by identifying resemblances across similar instances. Consequently, MW classification figures as a key activity related to MW analysis, which is paramount in the operation of computer security as a whole.

This thesis addresses the problem of MW classification taking an approach in which human intervention is spared as much as possible. Furthermore, we steer clear of subjectivity inherent to human analysis by designing MW classification solely on data directly extracted from MW analysis, thus taking a data-driven approach. Our objective is to improve the automation of malware analysis and to combine it with machine learning methods that are able to autonomously spot and reveal unwitting commonalities within data.

We phased our work in three stages. Initially we focused on improving MW analysis and its automation, studying new ways of leveraging symbolic execution in MW analysis and developing a distributed framework to scale up our computational power. Then we concentrated on the representation of MW behavior, with painstaking attention to its accuracy and robustness. Finally, we fixed attention on MW clustering, devising a methodology that has no restriction in the combination of syntactical and behavioral features and remains scalable in practice.

As for our main contributions, we revamp the use of symbolic execution for MW analysis with special attention to the optimal use of SMT solver tactics and hyperparameter settings; we conceive a new evaluation paradigm for MW analysis systems; we formulate a compact graph representation of behavior, along with a corresponding function for pairwise similarity computation, which is accurate and robust; and we elaborate a new MW clustering strategy based on ensemble clustering that is flexible with respect to the combination of syntactical and behavioral features.