On Malware Diversity and Similarity : Understanding Variability Across Malware Families

Antonino Vitale

THESE DE DOCTORAT DE
SORBONNE UNIVERSITE
préparée à EURECOM

École doctorale EDITE de Paris n° ED130
Spécialité: «Informatique, Télécommunications et Électronique»

Sujet de la thèse:

On Malware Diversity and Similarity: Understanding Variability Across Malware Families

Thèse présentée et soutenue à Biot, le 29/09/2025, par

Antonino Vitale

devant le jury composé de:

Examinateur & President	Prof. Melek Önen	EURECOM
Rapporteur	Prof. Valerie Viet Triem Tong	CentraleSupelec/CNRS/INRIA
Rapporteur	Prof. Juan Tapiador	Universidad Carlos III de Madrid
Examinateur	Prof. Matteo Dell'Amico	Università degli studi di Genova
Co-Directeur de thèse	Prof. Simone Aonzo	EURECOM
Directeur de thèse	Prof. Davide Balzarotti	EURECOM

Abstract

The growing volume of malware circulating daily, combined with its increasing structural diversity, presents substantial challenges for automated malware analysis. Machine-learning classifiers, widely adopted for malware detection and family classification, often struggle to generalize when confronted with new malware families or datasets affected by labeling inconsistencies, class imbalance, and incomplete feature sets.

Beyond classification, the structural variability observed within malware families complicates similarity measurement. Malware authors actively use techniques such as packing and deliberate binary modifications to generate diverse variants from the same code base. In addition to these challenges, many malware datasets collected from live feeds contain truncated samples, files that are incomplete due to errors during collection or transmission. While truncation is not a source of meaningful diversity, it introduces noise that pollutes datasets and wastes analysis resources when processed by tools or sandboxes.

At the same time, unrelated malware samples often display misleading structural similarities due to common build environments, shared packers, and recurring compiler toolchains. These inter-family artifacts undermine the precision of static similarity features, leading to clustering errors and incorrect associations between distinct malware families.

This thesis addresses these challenges through a measurement-driven investigation of malware diversity across three perspectives: the impact of dataset composition and feature selection on machine-learning classifiers, the extent and nature of intra-family polymorphism in malware binaries, and the structural factors driving false similarities between unrelated families. Together, these studies provide a comprehensive empirical foundation for improving the design, evaluation, and reliability of malware classification and similarity analysis techniques.

Résumé

Le volume croissant de programmes malveillants circulant quotidiennement, combiné à leur diversité structurelle toujours plus grande, représente un défi majeur pour l'analyse automatisée des malwares. Les classificateurs basés sur l'apprentissage automatique, largement adoptés pour la détection et la classification des familles de malwares, peinent souvent à généraliser lorsqu'ils sont confrontés à de nouvelles familles ou à des jeux de données affectés par des incohérences de labellisation, des déséquilibres de classes ou des ensembles de caractéristiques incomplètes.

Au-delà de la classification, la variabilité structurelle observée au sein des familles de malwares complique considérablement la mesure de similarité. Les auteurs de malwares recourent activement à des techniques telles que le packing et les modifications binaires intentionnelles pour générer des variantes diverses issues d'une même base de code. À ces défis s'ajoute la présence fréquente d'échantillons tronqués dans les jeux de données collectés à partir de flux en temps réel. Bien que ces fichiers incomplets ne constituent pas une source de diversité significative, ils introduisent du bruit et gaspillent des ressources d'analyse lorsqu'ils sont traités par des outils ou des sandboxes.

Parallèlement, des échantillons provenant de familles de malwares totalement distinctes présentent souvent des similarités structurelles trompeuses, résultant d'environnements de compilation communs, de packers partagés ou de chaînes d'outils de compilation récurrentes. Ces artefacts inter-familles compromettent la précision des caractéristiques de similarité statique, entraînant des erreurs de regroupement et des associations incorrectes entre familles distinctes.

Cette thèse aborde ces défis à travers une étude empirique de la diversité des malwares selon trois perspectives : l'impact de la composition des jeux de données et du choix des caractéristiques sur les performances des classificateurs, l'ampleur et la nature du polymorphisme intra-famille dans les binaires malveillants, et les facteurs structurels à l'origine des sim-

ilarités trompeuses entre familles non apparentées. Ensemble, ces travaux offrent une base empirique solide pour améliorer la conception, l'évaluation et la fiabilité des techniques de classification et d'analyse de similarité des malwares.

Contents

List of Figures

Chapter 1

Introduction

1.1 Context and Motivation

The volume of malware circulating in the wild continues to grow at an unprecedented rate. Security vendors, malware repositories, and threat-intelligence platforms collectively collect and process millions of potentially malicious samples every month. As of 2025, both the AV-TEST Institute and Kaspersky report more than 450,000 new malicious programs appearing daily [58, 20], with the extreme case of 11 millions of new samples in April 2025 [48]. This relentless influx of binaries has made automation essential at every stage of malware analysis, from detection and triage to clustering and behavioral characterization [73, 43, 116].

However, the scale of the problem is not merely a matter of quantity. The diversity and variability observed across malware samples present profound challenges for automated analysis systems. This diversity manifests at multiple levels: between families, within families, and even across the feature representations extracted for similarity measurement. Understanding how this diversity affects analysis pipelines is essential for building reliable and scalable malware-research tools.

At the structural level, malware authors have become adept at generating large numbers of polymorphic variants of the same malware strain [35, 78, 75]. Techniques like packing, code mutation, binary reordering, and intentional structural manipulations ensure that even functionally identical samples appear structurally distinct. This intra-family diversity severely impacts the reliability of many static similarity features, which often assume structural consistency within malware families. Features such as fuzzy hashes [61, 86] or import-table signatures can vary dramatically between two samples of the same family, making static feature-based classification and

clustering error-prone [134, 14].

Conversely, unrelated malware samples frequently exhibit misleading structural similarities. The widespread adoption of common build environments, off-the-shelf packers, and shared compiler toolchains introduces recurring artifacts that transcend family boundaries [63, 59, 78]. Two binaries from entirely different families may share identical resources, section layouts, or other structural characteristics, not because of shared behavior, but as a by-product of being processed through the same build environments. This inter-family similarity can lead to false positives in clustering and distort feature-based analyses, especially when using coarse-grained or syntactic similarity metrics.

Adding to the complexity, the way malware datasets are constructed, labeled, and curated has a direct impact on the evaluation and development of analysis techniques [66]. Public and private datasets differ in family coverage, sample balance, collection time-frames, and labeling consistency [16, 116, 90]. Antivirus-based labeling, still the de-facto standard for most research datasets, suffers from inconsistencies and vendor-specific taxonomies [112, 43]. Even tools designed to reconcile these inconsistencies, such as AVClass [112], operate on top of inherently noisy inputs. These dataset-related issues become particularly important when evaluating classification techniques, where performance can be heavily influenced by dataset composition and splitting strategies.

Feature design further amplifies the effect of malware diversity on analysis outcomes. Widely used static features, ranging from whole-binary fuzzy hashes to PE-structure descriptors like PEHash [135], Imphash [1], and RichPE [134], are highly sensitive both to intra-family polymorphism and inter-family artifact sharing. Minor binary modifications can dramatically change feature outputs, while unrelated samples processed through the same build environments may produce artificially similar feature representations. This fragility undermines the reliability of similarity measurements, clustering outcomes, and even machine-learning classifiers trained on such features.

The combination of these factors creates a landscape where measuring malware similarity, clustering samples, or training generalizable machine-learning models becomes a non-trivial task. Without a systematic understanding of how diversity influences each step of the analysis pipeline, from dataset construction to feature extraction to evaluation, the risk is high that research conclusions will be biased, non-reproducible, or misrepresentative of real-world conditions.

Addressing this challenge requires looking at malware diversity holisti-

cally, considering how structural variability, feature fragility, dataset biases, and evaluation inconsistencies collectively shape the strengths and limitations of current malware analysis techniques.

1.2 Challenges

The structural diversity and variability found in modern malware introduce a series of interrelated challenges that affect classification and clustering tasks across the malware analysis pipeline. These challenges emerge at multiple levels, spanning dataset composition, intra-family structural variability, and inter-family similarity. Understanding and characterizing these challenges is essential for building a reliable foundation for malware analysis research.

- **Dataset Composition and Its Impact on Classification.** The first set of challenges concerns the role of dataset characteristics in shaping classification outcomes. Existing malware datasets often differ significantly in terms of family coverage, number of samples per family, and labeling consistency. Such discrepancies raise important questions about how these factors influence the performance and generalization of machine-learning-based malware classifiers. Further uncertainty comes from the presence of off-the-shelf packers and protectors, missing or incomplete feature values, especially from dynamic analyses, and the inherent inconsistencies in antivirus family labeling. Different antivirus vendors often apply distinct naming conventions and classification criteria, introducing ambiguity and noise into the ground truth used for training and evaluating classifiers. A related issue is the inclusion of truncated samples, executables that are partially downloaded or corrupted during collection. Although not intentionally created by malware authors, these truncated files pollute malware feeds and waste storage and analysis resources when mistakenly processed as valid samples. An additional open question relates to classifier behavior when evaluated on families unseen during training, reflecting real-world deployment scenarios.

- **Structural Variability Within Malware Families.** A second challenge stems from the high degree of structural variability that can exist among samples belonging to the same malware family. Techniques such as packing and intentional binary modifications are commonly used by malware authors to produce polymorphic variants [35]. These transformations introduce variability across multiple binary

components, including PE headers, sections, and resources. Understanding how to systematically quantify these differences and determine their distribution across large malware collections remains an open problem. Additionally, there is a need to identify which specific causes, whether related to packing, binary manipulation, or other structural changes, are responsible for the observed polymorphism in different families.

- **Structural Similarities Across Different Families.** Finally, a third challenge arises from the unintended structural similarities that can appear between samples from entirely different malware families. Common build environments, widespread use of shared packers, and recurring compiler toolchains introduce artifacts that make unrelated samples appear structurally alike. This poses significant difficulties for clustering approaches that rely on static similarity features, as these artifacts can cause unrelated samples to be grouped together. The extent to which different feature types, such as fuzzy hashes, section-based metrics, or PE metadata descriptors, are vulnerable to such confounding factors remains largely uncertain. Additionally, it is unclear which structural artifacts are most responsible for creating these misleading inter-family similarities, making it difficult to anticipate or mitigate their impact when analyzing large malware collections.

1.3 Contributions

This work offers a comprehensive, data-driven exploration of malware diversity and similarity, examining its impact on key components of automated malware analysis. The contributions span from dataset construction to feature evaluation, with a strong emphasis on empirical measurement and methodological rigor.

In this first chapter, we presented the context and motivations behind this thesis, the key challenges in the field, and our contributions to address them. Chapter 2 introduces the necessary background for this dissertation. Chapter 3 provides a detailed description of the datasets used across the thesis. The following three chapters each correspond to one of the individual studies conducted. Finally, Chapter 7 summarizes the main findings and discusses possible future research directions.

Contribution I (Chapter 4)

The first study focuses on the design, construction, and analysis of a large-scale malware dataset explicitly engineered to minimize common biases found in existing public datasets. We collect and label 67,000 Windows PE malware samples, evenly distributed across 670 families, ensuring balanced class representation. This careful dataset construction allows for fair and reproducible evaluation of machine-learning-based malware classifiers. Using this dataset, we conduct an extensive evaluation of a wide range of classification models, covering static, dynamic, and hybrid feature-based approaches. The study investigates how different types of features contribute to classification accuracy, how the presence of off-the-shelf packers and protectors affects performance, and how missing values in dynamic features influence the outcomes. It also explores how dataset splitting strategies and family coverage affect classifier generalization, especially when encountering families not seen during training. Particular attention is paid to understanding how label inconsistencies and feature selection impact binary and family classification tasks under realistic deployment conditions.
This study and its findings are published in:

- Savino Dambra, Yufei Han, Simone Aonzo, Platon Kotzias, Antonino Vitale, Juan Caballero, Davide Balzarotti, and Leyla Bilge
  "Decoding the Secrets of Machine Learning in Malware Classification: A Deep Dive into Datasets, Feature Extraction, and Model Performance."
  In ACM Conference on Computer and Communications Security. ACM, November 2023.[34]

Contribution II (Chapter 5)

Building on the curated dataset, the second study investigates the extent and causes of structural polymorphism within malware families. To do this, we develop a component-level structural analysis framework that decomposes each malware sample into its distinct binary components (e.g., headers, sections, resources) and systematically analyzes their presence and variability across samples of the same family. This is enabled by *PEdiff* [10], a dedicated PE structural comparison tool developed as part of this study.

The study focuses on quantifying the structural differences observed within families at a fine granularity and assessing the distribution of different sources of polymorphism, such as packing, truncation, and targeted binary modifications. It also examines how these structural changes manifest across various PE components and how multiple factors may contribute to the

observed variability within the same family.
This study and its findings are published in:

- Antonino Vitale, Simone Aonzo, Savino Dambra, Nanda Rani, Lorenzo Ippolito, Platon Kotzias, Juan Caballero, and Davide Balzarotti
  "The Polymorphism Maze: Understanding Diversities and Similarities in Malware Families."
  In European Symposium on Research in Computer Security (ESORICS) 2025.[130]

Contribution III (Chapter 6)

The third study turns attention to inter-family diversity and the risk of false similarities across unrelated malware samples. Focusing on eleven widely adopted static similarity features, including fuzzy hashes, section-based features, and PE metadata descriptors, we conduct a large-scale precision-oriented analysis to assess how these features behave in the presence of inter-family structural similarities and shared build artifacts.

The study evaluates the precision of each feature in separating malware families and examines the structural artifacts that most frequently lead to clustering errors. It analyzes how common build environments, packers, and compiler toolchains contribute to structural similarity between unrelated samples and investigates the susceptibility of different feature types to such confounding factors. The work provides detailed insights into the structural characteristics most responsible for violating family boundaries during static feature-based clustering.
This study and its findings are currently under submission for peer-review.

Chapter 2

Background and Related Works

2.1 Context and Motivation

This chapter provides the necessary background for understanding the contributions of this thesis, which focuses on exploring malware similarity and diversity from different perspectives. Specifically, this dissertation examines how malware diversity affects machine-learning-based classification, how structural variability manifests within malware families, and how static features capture or fail to capture similarities across different families. To contextualize this research, we first review prior studies on ML-driven malware classification and dataset design. We then cover research on malware polymorphism and structural variability, followed by works on code and binary similarity. Finally, we discuss existing approaches that use static features for malware clustering, highlighting their strengths and limitations in handling malware diversity.

2.2 Malware Classification and Machine Learning Challenges

Table 2.1 presents a categorization of previous works on Windows malware classification, according to their goal (binary detection or family classification), features (static or dynamic), and dataset size (both in terms of malware executables and malware families). Among the approaches in Table 2.1, the choice of classification models varies widely, including Support Vector Machines, GradientBoost, Random Forest, and neural networks.

		Goal		Features		Dataset	
Work	**Year**	**D**	**C**	**S**	**D**	**#**	**Fam.**
Rieck et al. [102]	2008	-	✓	-	✓	10K	14
McBoost [91]	2008	✓	-	✓	✓*	5.5K	-
PE-Miner [114]	2009	✓	-	✓	-	16K	-
Nataraj et al. [84]	2011	-	✓	✓	✓	67K	561*
OPEM [109]	2012	✓	-	✓	✓	1K	-
Santos et al. [108]	2013	✓	-	✓	-	1K	-
Dahl et al. [33]	2013	✓	-	-	✓	1.8M	-
Kancherla et al. [56]	2013	✓	-	✓	-	25K	-
Saxxe et al. [110]	2015	✓	-	✓	-	350K	-
Miller et al. [77]	2016	✓	-	✓	✓	1.1M	-
MtNet [47]	2016	✓	✓	-	✓	2.8M	98
MAAR [107]	2017	✓	-	-	✓	3K	-
MalConv [99]	2018	✓	-	✓	-	284K	-
EMBER [16]	2018	✓	-	✓	-	400K	-
Rhode et al. [101]	2018	✓	-	-	✓	5.1K	-
MalDy [57]	2019	✓	✓	-	✓	20K	15
NeurLux [51]	2019	✓	-	-	✓	34K	-
MalInsight [43]	2019	✓	✓	✓	✓	3.5K	5
MalDAE [42]	2019	✓	-	✓	✓	5.5K	-
MALDC [142]	2020	✓	-	-	✓	54K	-
IMCFN [127]	2020	✓	-	✓	-	9.4K	-
Zhang et al. [143]	2020	✓	-	-	✓	27.7K	-
Rabadi et al. [98]	2020	✓	-	-	✓	7.1K	-
Joyce et al. [53]	2022	-	✓	✓	-	3K	454
This work	2023	✓	✓	✓	✓	67K	670

Table 2.1: Related work on ML-based Detection and family Classification of Windows malware (S=Static, D=Dynamic)

Most approaches rely on feature extraction, such as n-grams of bytes, op-codes, or system calls, though some operate directly on raw bytes and API sequences [99, 51].

MalInsight [43] is the only study to provide a comprehensive coverage of both feature choice and classification tasks, though it includes only 5 families. On the other end of the spectrum, Nataraj et al. [84] studied family classification on an unbalanced dataset with over 500 classes, but their use of raw AV labels means that the number does not correspond to real malware families. In contrast, this thesis investigates the factors impacting ML classifier performance using a large-scale, balanced dataset with 670 families.

Two main challenges have been identified in ML-driven cybersecurity research. First, the problem of missing observations, which impacts predic-

tion accuracy in fields like network intrusion detection [89, 119]. Second, the I.I.D. assumption, where training and test data are expected to be drawn from the same distribution, rarely holds in real-world malware classification due to rapidly evolving malware behaviors [24]. The breakdown of this assumption introduces significant performance degradation when classifiers encounter out-of-distribution (OOD) samples.

Arp et al. [19] provide a broad review of ML-based cybersecurity research, highlighting biases introduced by sampling errors, labeling noise, and inappropriate evaluation metrics. Their work underscores the need for evaluation methodologies that account for data imbalance and distributional shifts. Building on these concerns, this thesis focuses on the bottlenecks encountered when deploying ML-based malware classifiers in practice, including the impact of malware family coverage during training and classifier behavior under OOD conditions.

2.3 Dataset Construction for Malware Classification

In 2015, the Microsoft Malware Classification Challenge [116] was launched as a Kaggle competition to promote research on malware family classification. This dataset consists of disassembly and byte data for 20K Windows malware samples from 9 families and has been widely used in subsequent studies.

Other works have addressed dataset scale and diversity in Android malware. For example, [90] evaluates spatial and temporal bias over 129,728 Android apps, while [66] examines the variance-bias trade-off in clustering 134,698 apps. In the Windows malware space, MOTIF [53] provides a manually labeled dataset of 3,095 samples across 454 families, representing the most diversified public dataset in terms of family coverage. However, the dataset suffers from extreme imbalance, with more than half the families containing fewer than five samples, creating a few-shot learning challenge for ML-based classifiers.

This dissertation draws inspiration from these prior datasets while focusing on large-scale Windows malware collections with balanced family distributions, enabling a controlled evaluation of how dataset characteristics impact classification performance.

2.4 Polymorphism and Malware Diversity

Malware achieves polymorphism through a range of obfuscation techniques, including dead-code insertion, register reassignment, and instruction substitution [141], making static detection increasingly ineffective [18]. As a result, prior works have largely focused on behavioral analysis to detect polymorphic malware. Techniques include behavior-aware hidden Markov models [118], mixed static and dynamic analysis approaches [87], and flow-graph matching using emulators [29].

This thesis departs from traditional detection efforts and instead focuses on quantitatively characterizing the structural causes of polymorphism across malware families using a component-level analysis.

2.5 Code and Binary Similarity

Several approaches measure similarity between executables at the code level by examining disassembled output [41, 45, 46, 71]. These techniques have been used for binary diffing [41, 71], similarity search [45], and malware clustering [46]. This study leverages BinDiff [41] as a representative tool for identifying code similarity within malware families.

At the binary level, fuzzy hashing techniques such as SSDEEP [62], TLSH [86], SDHASH [105], and MRSH-v2 [27] are widely used for forensic analysis [106], malware detection [113, 79, 81], and clustering [21, 125, 115, 23]. Several studies have evaluated their effectiveness [88, 26]. This research introduces a fine-grained structural comparison across 12 PE file components to localize byte-level differences. As part of this, it uses TLSH for pairwise comparison of component values across samples, observing that aggregation over small components (such as PE headers) often produces high volatility in family-wise similarity scores.

2.6 Static Features for Malware Clustering

Extensive research has addressed the problem of clustering malware samples into families [25, 92, 50, 46, 100, 69, 85]. Table 2.2 summarizes prior works that have employed at least one of the static similarity features analyzed in this thesis.

Some studies have explored the use of certificates for grouping samples [63, 59], while others have evaluated Authentihash grouping independently of signing status [139]. Feature-specific evaluations include work on

Work	Year	Authentihash	Cert Subject	Cert Thumbprint	Icon DHash	Icon Hash	Imphash	PEHash	RichPE	ssdeep	TLSH	vhash
PEHash [135]	2009	-	-	-	-	-	-	✓	-	-	-	-
French et al. [37]	2012	-	-	-	-	-	-	-	-	✓	-	-
TLSH [86]	2013	-	-	-	-	-	-	-	-	-	✓	-
Certified PUP [63]	2015	✓	✓	-	-	-	-	✓	-	-	-	-
Li et al. [69]	2015	-	-	-	-	-	-	-	-	✓	-	-
Webster et al. [134]	2017	-	-	-	-	-	-	-	✓	-	-	-
Certified Malware [59]	2017	-	✓	✓	-	-	-	-	✓	-	-	-
Chikapa et al. [30]	2018	-	-	-	-	-	✓	✓	-	-	-	-
Joyce et al. [54]	2019	-	-	-	-	-	✓	✓	✓	-	-	-
Posluvsny et al. [95]	2019	-	-	-	-	-	-	-	✓	-	-	-
Kim et al. [60]	2020	-	-	-	✓	-	-	-	-	-	-	-
Fuzzy-Import [81]	2020	-	-	-	-	-	✓	-	-	✓	-	-
Namanya et al. [82]	2020	-	-	-	-	-	✓	✓	-	✓	-	-
Ali et al. [15]	2020	-	-	-	-	-	-	-	-	-	✓	-
HAC-T [85]	2020	-	-	-	-	-	-	-	-	-	✓	-
Botacin et al. [26]	2021	-	-	-	-	-	-	-	-	✓	-	-
RecMaL [139]	2023	✓	-	-	-	-	✓	✓	-	-	-	✓
VirusTotal [129]	2024	✓	✓	✓	✓	✓	✓	-	✓	✓	✓	✓

Table 2.2: Related work on clustering using static features.

icon similarity using image comparison algorithms [60], as well as clustering features such as Imphash [1], PEHash [135], RichPE [134], and vhash [131].

Existing literature can be broadly divided into two groups: studies evaluating the precision of individual features [30, 54, 95] and those combining multiple features to improve clustering quality [139, 63, 59].

In the area of fuzzy hashes, most prior work has focused on evaluating their precision across datasets [37, 86, 69, 15, 85, 26]. Some studies explored combining fuzzy hashes with other features, as in Naik et al. [81] and Namanya et al. [82], who combined SSDeep, Import Hash, and PEHash for improved detection and clustering.

To summarize, while some features such as PEHash [135, 63, 30, 54, 82, 139], Import Hash [30, 54, 81, 82, 139], and SSDeep [37, 69, 81, 82, 26] have been widely evaluated, other features such as certificate-based grouping [63, 59, 139], Icon DHash [60], TLSH [86, 15, 85], and vhash [139] remain underexplored. Notably, Icon Hash has not been studied at all.

This dissertation builds on these prior efforts by conducting a systematic, cross-feature evaluation of static similarity features under conditions of high

malware diversity, with a particular focus on inter-family false similarities caused by shared structural artifacts.

Chapter 3

Datasets

This chapter introduces the malware datasets referenced across the three chapters of this thesis. Not all datasets are used in every chapter, and the experiments in each chapter rely on different subsets or filtered versions of these datasets. The goal of this chapter is to provide readers with a clear overview of each dataset's characteristics and origins, facilitating a better understanding of their role in the subsequent studies.

3.1 Balanced Dataset

The Balanced Dataset [34] contains 67,000 hashes of 32-bit PE malware samples collected from the VirusTotal (VT) feed between August 2021 and March 2022. Samples are evenly distributed across 670 malware families, with exactly 100 samples per family.

Family labels were assigned using the AVClass [112] tool, and sample selection prioritized diversity and balance for large-scale classification tasks. The full details of the dataset creation process are provided in Section 4.1.

3.2 MOTIF Dataset

The Malware Open-source Threat Intelligence Family (MOTIF) dataset [53] contains 3,095 Windows PE malware samples drawn from 454 distinct families. Samples and family labels were derived from open-source threat intelligence reports published by 14 major cybersecurity organizations over a five-year period (January 2016 to December 2020).

MOTIF emphasizes coverage diversity, focusing on long-tail, real-world malware families from operational threat intelligence sources. Sample hashes

and family names were collected from these reports and mapped into family aliases when applicable. Samples may be assigned a group of labels considered aliases. For each alias group, we select the name used by AVClass, or the most popular one in case AVClass does not know any.

The dataset is imbalanced: 131 families (29%) contain only one sample, 232 families (51%) contain 2–9 samples, and 91 families (20%) have at least 10 samples. Only one family (*icedid*) exceeds 100 samples. Samples were retrieved from VirusTotal, using the public hash lists provided by the MOTIF repository [9].

3.3 Malicia Dataset

The Malicia dataset [83] contains 9,908 Windows PE malware samples collected between March 2012 and February 2013 from drive-by download campaigns. Collection was conducted using honeypots designed to attract drive-by download traffic.

Family labeling was performed using clustering techniques that leveraged network behavior, execution screenshots, and embedded icon analysis, followed by manual expert validation. The final dataset includes 53 family-level clusters: 19 labeled with well-known malware family names and 34 labeled with generic cluster identifiers.

Similar to MOTIF, Malicia is heavily imbalanced: 23 families (43%) contain only one sample, 17 families (32%) contain 2–9 samples, and 13 families (25%) contain at least 10 samples. Only 4 families include at least 100 samples. The largest family, *winwebsec*, accounts for 5,820 samples (58.7% of the dataset).

Samples are distributed across common malware classes such as adware, backdoors, downloaders, ransomware, rogueware, viruses, and worms, providing class diversity for malware analysis tasks.

The dataset was originally created by the authors for academic research but has since been discontinued. However, we contacted the dataset authors, who kindly agreed to share it with us for our studies.

3.4 Dataset Usage

The three datasets, *Balanced Dataset*, *MOTIF*, and *Malicia*, are used in different ways across the next three chapters in this thesis. Table 3.1 summarizes which datasets are used in each chapter, along with any filtering or sampling applied.

Chapter	Balanced Dataset	MOTIF	Malicia	Other Samples/Families
Chapter 4	✓ (created here)	~✓ (Motif-like)	-	✓
Chapter 5	✓†	✓†	✓†	-
Chapter 6	✓	✓	✓	-

Table 3.1: Summary of the datasets used in each chapter.
~✓ indicates that a MOTIF-like, unbalanced dataset (not the actual MOTIF dataset).
† indicates that only families with at least 10 non-truncated samples were used.

Chapter 4. This chapter focuses on building and analyzing the *Balanced Dataset*, whose collection process is described in detail in Section 4.1. All experiments rely on this dataset and on other malware samples and families collected during the same period that do not belong to any of the three datasets described here.

Chapter 5. Experiments in this chapter leverage all three datasets introduced in this chapter. Section 5.1 provides details about the selection and filtering criteria applied to the datasets for this study.

Chapter 6. All three datasets (*Balanced Dataset*, *MOTIF*, and *Malicia*) are used. Section 6.1 details the procedures followed to integrate and deduplicate samples and families for clustering evaluation.

Chapter 4

An Empirical Study of Malware Binary and Family Classification

In the previous chapters, we introduced the challenges posed by malware diversity and highlighted the crucial role that datasets play in shaping the outcomes of malware analysis research. We also described the datasets that will be used across this thesis. Building on this context, this chapter focuses on the first step of our investigation: understanding how dataset composition, feature choice, and labeling inconsistencies affect the performance of Machine Learning (ML) models for malware classification.

Malware classification remains one of the most widely adopted applications of Machine Learning in cybersecurity, driven by the need to process vast volumes of incoming samples with limited human resources. As discussed in Chapter 1, the number of new malicious programs discovered daily remains exceptionally high, exceeding 450,000 samples per day in 2025 according to both the AV-TEST Institute and Kaspersky [58, 20].

To address this scale, both academia and industry have increasingly turned to ML-driven malware classification models. These models offer flexible detection and labeling mechanisms that go beyond traditional signature-based systems, but they also introduce new challenges. ML classifiers fundamentally rely on learning statistical correlations from historical training data. As a result, their ability to generalize to novel, previously unseen malware variants is inherently limited, particularly when adversaries deliberately craft evasive samples that differ from known patterns.

Another important aspect is the structure of the classification pipeline itself. Most ML-based malware analysis workflows adopt a multi-stage pro-

cess [73, 137, 43], typically starting with a *binary classification* step (malicious vs. benign) followed by a *family classification* phase (identifying the malware family). Although high classification accuracy has been reported in the literature [73, 137, 43], many existing studies have relied on overly optimistic experimental assumptions, often due to issues in dataset design and construction.

In addition, a ground-truth of malware families is hard to obtain. Antivirus companies will not likely use the same name for the same family. Although the CARO (Computer Antivirus Research Organization) naming convention has been proposed to mitigate this issue, it still faces usage obstacles. Scientific research tackled this problem and produced AVClass [112]: given a list of AV labels (e.g., from a VirusTotal JSON report), the tool returns the *single most likely* family name. However, even if AVClass returns a single family name according to a consensus algorithm by default, it can also output a ranking of all alternative family names. Thus, the problem is that AVClass is often used to carry out studies using its default output as ground truth, even though it is probabilistic in nature.

Moreover, while it is straightforward to collect a high number of samples for popular families, collecting a large diverse malware dataset remains difficult and time-consuming [16, 116, 90, 66]. In this chapter, we collect PE malware executables from the VirusTotal (VT) feed [132], a real-time stream of JSON-encoded reports of samples submitted to VirusTotal. Despite the appearance of more than 44M VT reports over a period of nearly three months and the collection of 227k samples from 13.8k families, only 780 malware families of those contain at least 100 samples.

To further complicate the matter, malware authors often use off-the-shelf packers and protectors [78, 75]. Both modify a program to hinder its analysis while still preserving its original behavior. Based on their design, different malware that undergo the packing or protection procedures may generate executables that share a highly similar structure. This easily makes a ML classifier trained over these malware samples overfit the packed or protected file structure, rather than capturing its true malicious component.

Therefore, we put considerable effort to create four heterogeneous datasets for a total of 118,111 samples to perform a large-scale measurement study. Three of them are composed of malicious samples with varying numbers of families, while the fourth contains benign samples. We devoted particular attention during the construction of the datasets, trying both to reproduce the datasets usually used in research, but also considering real-world scenarios typical of malware analysis. Such datasets allowed us to create well-controlled experiments for studying how the effectiveness of ML-based

binary and family classification change under different testing scenarios.

Finally, there is also another crucial aspect that influences ML algorithms that we further explored: *feature* extraction. The methods by which one can analyze executable files fall into two main categories, depending on what facets one wants to study, namely *static* properties and *dynamic* behavior; nonetheless, the previous two can also be *combined*. Since we wanted to study existing ML state-of-the-art solutions and **not** design new ones, we build our static and dynamic feature extraction approaches on what was described in recent papers [17, 14]. Therefore, this means that we have statically analyzed and dynamically executed in a sandbox more than a hundred thousand samples were used in this study.

The work in this chapter contributes by answering the following research questions for both binary and family classification tasks:

⟨R1⟩ How do static, dynamic, and combined models perform on different malware families/classes in binary and family classification?

⟨R2⟩ On which families and classes of malware does each model fail to produce accurate classification?

⟨R3⟩ What is the contribution of static and dynamic feature classes to the classification performance and does their contribution change when joining the two sets?

⟨R4⟩ Does the presence of off-the-shelf packers and protectors bring harm to classification accuracy?

⟨R5⟩ Do missing feature values in the runtime behaviors negatively impact the classification results?

⟨R6⟩ Is the AVClass2 confidence score correlated with ML-based decisions?

⟨R7⟩ How does the training dataset construction strategy affect the model performance?

⟨R8⟩ How does the ML-driven malware classifier perform over the families unseen in the training data?

4.1 Dataset Collection

To conduct our experiments we collected 118,111 Windows PE32 executables, divided in four datasets, as summarized in Table 4.1. This section describes the process for building those datasets.

Dataset	Samples	Families
Balanced Dataset (M_B)	67,000	670
Benign (B)	16,611	-
Malware Unbalanced (M_U)	18,000	1,500
Malware Generic (M_G)	16,500	-
All	118,111	-

Table 4.1: Dataset summary

4.1.1 Malware Samples

We collect PE malware executables from the VirusTotal (VT) feed [132]. The VT feed is a real-time stream of JSON-encoded reports. Each report contains the analysis results of a sample submitted to VirusTotal – including file hashes, filetype, size, and the detection labels assigned by a large number of antivirus (AV) engines. These reports are generated both by new samples submitted by VT users, as well as by user-requested re-analysis of files already known to VT. Samples in the feed can be of various file types (e.g., PE, APK, PDF), but our collection focuses on Windows PE executables. Samples that appear in the feed can be downloaded within 7 days from the moment they appear in the feed.

We want our dataset to be as diverse as possible in terms of the number of families, but also to be balanced, so that no malware family is over-represented or under-represented. Our initial target was to collect 1,000 malware families with a hundred samples each. The threshold of 100 samples per family was chosen to have enough samples per family to performing multi-class classification experiments, taking into account that samples are split into 60% training, 20% validation, and 20% testing. However, due to the collection, filtering, and reclassification process described below, we ended up with 670 families satisfying that threshold, as shown in Table 4.1.

To the best of our knowledge, this is the most diverse labeled malware dataset in terms of families at the time of writing. The most recent dataset was MOTIF [53] with 454 families. While the number of families in MOTIF is also large, it is 21 times smaller than our balanced dataset with 3,095 samples, and is unbalanced with a median of three samples per family. Only one family in MOTIF reaches 100 samples and 29% of the families have only one sample. Such a small number of samples for most families does not allow building an accurate multi-class classifier, as we will show in our evaluation.

Initial collection from VT feed. We collected reports and samples from the VT feed for 82 non-consecutive days between August 2021 and March 2022. We only retained reports of samples detected by at least one AV engine, and with a trID [94] filetype identification field (available in the report) equal to '32-bit non-installer PE executable'. We excluded 64-bit PE executables, dynamic-link libraries (DLLs), and executables generated by popular installer software (e.g., NSIS, InnoSetup). These restrictions are placed by our dynamic analysis sandbox, described in Section 4.2.2, which currently does not support running 64-bit PE executables or DLLs, and does not interact with GUIs in order to complete the installation of other programs. However, an analysis of the whole VT feed during the 82 collection days shows that from all malicious PE samples in the feed, 87.6% are 32-bit executables, 8.2% are DLLs (32-bit or 64-bit), 3.9% are 64-bit executables, and the remaining 0.3% are other PE types (e.g., OCX, CPL, SCR).

The retained reports are fed to the AVClass2 malware labeling tool [112], which outputs the most likely family name for the sample as well as a confidence factor that captures the number of AV engines assign that family to the sample (after removing duplicates due to AV engines that copy each other). For each family reported by AVClass2, our system downloaded 100 distinct samples. Each downloaded sample was then checked again to exclude any remaining non-32-bit PE executables and installers that were missed by trID. In particular, samples are removed if their PE header does not indicate they are 32-bit executables, or if they are detected as installers using public Yara rules by Avast [13]. As stated, our initial target was to collect 1,000 malware families with 100 samples each. However, when this target was reached, many other families had been collected with less than 100 samples, resulting in an initial dataset of 239,417 PE32 malware samples from 23,555 families.

Reclassification and family filtering. The AV labels of a sample may change over time as AV vendors refine their detection rules. These label changes may in turn change the family that AVClass2 outputs for a sample. To account for such changes, we re-collect the updated VT report for our samples 54 days after the end of our collection process, and feed the new reports to AVClass2 to obtain the (possibly) updated family. From the 239,417 samples, 9.7% (23,171) were at this point re-classified as a different family. AVClass2 uses a taxonomy to identify a wide range of non-family tokens that may appear in the AV labels. These include file properties (e.g., *FILE:packed:asprotect*, *FILE:exploit: gingerbreak*), malware classes (e.g.,

CLASS:virus, *CLASS:worm*), behaviors (e.g., *BEH:ddos*, *BEH:filedelete*), and generic tokens (e.g., *GEN:malicious*, *GEN:behaveslike*). However, the AVClass2 taxonomy is assumed to be incomplete by design [112]. Thus, it may output a label for a sample that does not correspond to a real family, but rather to a previously unknown instance of the above categories. To address this issue, we manually inspected the collected family labels and conservatively filtered out any labels that may not correspond to real family names. This step identified 86 likely non-family tokens not in the AVClass2 taxonomy, such as *gametool*, *testsample*, *nsismod*, *dllinject*, and *processhijack*. We also removed random-looking labels (e.g., *005376ae*) that AVClass2 failed to filter. As a byproduct of our effort, we will contribute our extended AVClass2 taxonomy to the open-source AVClass2 project.

After reclassification and family filtering, the dataset contained 227,296 samples from 13,894 families, out of which 780 families had at least 100 samples. Thus, despite examining more than 44M VT reports over a period of nearly 3 months, we were unable to reach our goal of 1,000 families with 100 samples. This illustrates the difficulty of building a diverse malware dataset.

Feature filtering. We performed static and dynamic feature extraction (as detailed in Section 4.2) for all samples of the 780 families with at least 100 samples. This required executing each sample in a sandbox to obtain a behavioral report. We discarded 122 samples for which the static feature extraction pipeline failed. The failure reasons were corrupted headers (26 binaries), empty output from the disassembler probably due to obfuscation techniques (95 samples), and the absence of the entry point in one binary. We also discarded samples that did not exhibit any runtime behavior, and sub-sampled families to keep only 100 samples each. The result is a Balanced Dataset (hereinafter M_B in this chapter) that contains 67,000 samples from 670 families. According to AVClass2, those families belong to 13 malware classes: 36% (282) of the families are classified as grayware (including its adware subclass), 15% (120) as downloaders, 11% (87) as worms, 10% (78) as backdoors, 5% (41) as viruses, and the remaining 23% (62) includes ransomware, rogueware, spyware, miners, hacking tools, clickers, and dialers.

Dataset statistics. Over 93% of the samples in the M_B dataset are detected by at least 20 AV engines, while only 0.3% have a VT score less or equal to 3. It is worth noting that the minimum number of detections for samples in the dataset is two since AVClass2 requires at least two AV

engines to assign a label to a sample.

Samples on the VT feed can be new (i.e., collected and scanned for the very first time by VT) or resubmitted (i.e., first submitted in the past but re-scanned on the day they were collected). We compute the freshness of samples in the M_B dataset as the number of days between a sample's collection date and its VT first seen date. We observe that 53.4% of the samples were collected within a day of being first observed by VT, 7.6% within a year, and 37.8% are old samples first seen over one year before our study.

Packer and protector detection. To hamper analysis, malware authors may use packers that compress a sample and de-compress it at runtime, as well as more sophisticated protectors that may combine different obfuscations such as packing, encryption, and code virtualization. To evaluate the impact of packers and other protectors on malware classification, we determine whether a sample uses an off-the-shelf packer or protector by using the signature-based Detect It Easy (DIE) [5] tool, as well as the well-maintained Yara rules of Avast RetDec [13]. Overall, 22% of the samples in M_B use a packer or protector. The most popular packer is `upx` detected on 14.0% the samples, followed by `aspack` (3.2%) and `pecompact` (1.0%). The most popular protectors are `vmprotect` (1.9%) and `asprotect` (0.4%).

4.1.2 Testing Datasets

We create two other disjoint malware datasets, which we use in Section 4.3 to test the ability of ML classifiers to generalize beyond the M_B dataset they were built upon. The first dataset, referred as Malware Unbalanced (or M_U) in Table 4.1, contains 18K samples from 1.5K families. These samples were part of the initial VT feed collection, passed the filtering and re-classification steps, but their families never reached the threshold of 100 samples and thus were excluded from M_B. All samples are detected by at least 20 AV engines and none of the samples nor their families are part of M_B.

The second dataset, Malware Generic (M_G), contains 16.5K samples for which AVClass2 was unable to output a family, due to AV engines using only generic labels. These samples were separately collected from the VT feed between June 23rd and July 6th 2022 and underwent the filtering steps to keep only 32-bit non-installer PE executables. All samples are detected by at least 20 AV engines and none of the samples are part of M_B.

4.1.3 Benign Samples

Building a benign dataset by just relying on the number of AV detections in the VT report is prone to errors due to the presence of malicious files that are still unknown to AV engines. Therefore, we took a more conservative strategy and decided to build a benign dataset by using a fresh installation of all the community-maintained packages of Chocolatey [4] (which undergo a rigorous moderation review process to avoid pollution) in a clean machine running Windows 10. After each package was installed, we extracted all the executable files present on the hard disk, which may correspond to Windows system files or third-party publishers.

We exclude files that are not 32-bit PE executables and those with more than three detections on VT. This allowed us to discard borderline cases, i.e., benign files with characteristics very similar to malware, like hacking and scanning tools. Using this procedure we collected a dataset B of 16,611 benign samples. The code signatures of those samples indicate a large diversity of publishers with over 1.4K different signers – including both small companies and large software publishers such as Microsoft, Oracle, and Google.

4.2 Methodology

We aim to answer the 8 research questions raised in the introduction of Chapter 4. Notably, we aim to explore the performances of ML-driven malware classifiers that use features extracted statically, dynamically, or a combination of both with varied coverage of malware families and changed volumes of training samples. Developing novel ML-based malware classification models is beyond the scope of our study. Instead, we focus on discussing and evaluating the analysed issues using state-of-the-art ML models for malware classification. As explained next, we use features presented in previous works [14, 47, 43, 17]. This imposes a limitation as other features could provide better results.

4.2.1 Static Features

Hojjat et al. [14] performed a literature review to identify the static features that carry the most useful information for binary classification. We implement their feature extraction methodology to extract the same classes of static features. Similar to Hojjat et al. [14], we do not attempt to unpack the executables and perform the same feature extraction regardless of whether the files are packed or not.

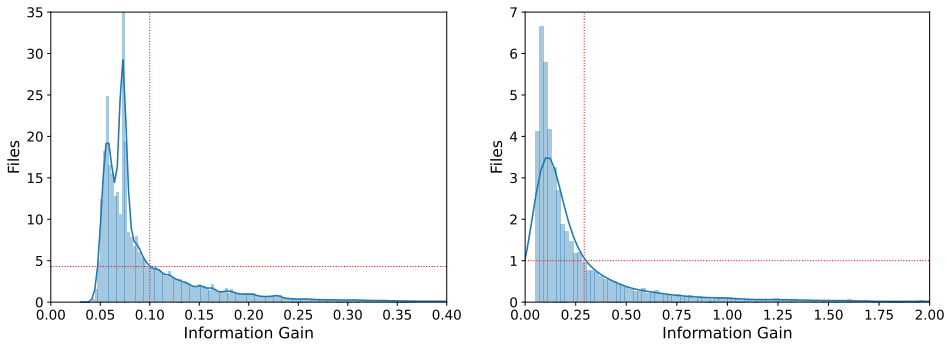
ID	Class	Extraction	Features
s-headers	PE headers	static	29
s-sections	PE sections	static	590
s-file	File Generic	static	2
s-dll	DLL imports	static	131
s-imports	API imports	static	3,732
s-strings	Strings	static	10,402
s-bytegrams	Byte n-grams	static	13,000
s-opcodegrams	Opcodode n-grams	static	2,500
d-network	Network activity	dynamic	438
d-file	File activity	dynamic	60,555
d-mutex	Mutexes used	dynamic	7
d-registry	Registry operations	dynamic	60
d-service	Services activity	dynamic	736
d-process	Process activity	dynamic	28,198
d-thread	Thread actitivy	dynamic	7

Table 4.2: Feature classes used in the classifiers.

The upper half of Table 4.2 summarizes the static feature classes (prefixed by s-). The *s-headers* class captures 29 integer features (Table 1 in the Appendix) from the *Optional* and *COFF* headers of the executable [31]. The *s-sections* class captures 590 Boolean features from each section in the executable (Table 2 in the Appendix). The *s-file* features capture the file size in bytes and the whole file Shannon entropy [74].

For the remaining 5 feature classes the exact number of features may differ from those reported by [14] because they undergo a dataset-dependent feature selection step that retains only the features that show variability or that provide higher information gain (IG) [96]. For instance, in *s-bytegrams* and *s-opcodegrams*, the selection process enumerates all values observed in the validation set (20% of samples in M_B), excludes rare values appearing in less than 1% of the samples, computes IG, uses the elbow method to identify a threshold value for IG, and only retains features with at least that threshold IG. As in [14], for *s-dll*, *s-imports*, and *s-strings*, the selection process only excludes rare values, but does not select an IG threshold.

The *s-dll* and *s-imports* class contain Boolean features extracted from the import table (imported libraries in case of *s-dll* and imported functions for *s-imports*). We extracted 637 unique libraries and 28,667 functions and retained only those that appear in at least 1% of the files in the validation set,



(a) s-bytegrams (b) s-opcodegrams

Figure 4.1: Distribution of number of *s-bytegrams* and *s-opcodegrams* based on their Information Gain value.

reducing the number to 131 DLLs and 3,732 library functions. Similarly, for the *s-strings* class, we extracted 106,352,885 strings of at least 4 characters, filter those that appear in over 1% of the files, and kept 10,402 Boolean features capturing whether the string appears or not in the binary. The *s-bytegrams* class captures the presence of selected 4-grams, 5-grams, and 6-grams. As proposed in [14], to keep memory usage manageable, the selection process for this feature class is performed on 1,000 randomly chosen files from M_B, instead of the full validation dataset. From the 1,363,150,788 *s-bytegrams* extracted, the selection retained the 13,000 features with the highest IG (Figure 4.1a). The *s-opcodegrams* class captures 1-gram, 2-grams and 3-grams from the sequence of opcodes disassembled using Capstone [28]. Given an initial set of 255,812 opcode n-grams, we computed the TF-IDF and used the elbow method on the IG distribution to retain the top 2,500 float features (Figure 4.1b).

4.2.2 Sandbox

We have built a sandbox for executing malware using the best practices proposed by previous works [104, 78, 75, 140]. We configured a Windows 10 Pro 32-bit virtual machine (VM) with 2 CPUs (Intel Xeon Platinum 8160 @ 2.10GHz) and 2 GiB of RAM. We installed popular apps and populated the file system with common file types to resemble a legitimate desktop workstation as suggested by Miramirkhani et al. [78]. Malware runs on clones of this VM orchestrated using Proxmox VE [12]. To improve performance, we stored all virtual disk images and VM snapshots in a RAM disk. As

recommended by Rossow et al. [104], each machine runs on its isolated local network with full Internet access through an ADSL line of our institution dedicated to this purpose. Recent works have measured that 40%–80% of modern malware use at least one evasive technique [75, 38]. To limit the impact of such evasions, we base our analysis on the Intel PIN-based *JuanLesPIN* tool [7, 75], which handles common evasive techniques, thereby increasing the likelihood that malware detonates. Unfortunately, it does not support 64-bit Windows executables, so we focus on 32-bit malware. We modified *JuanLesPIN* to monitor Windows APIs responsible for network, processes, services, registry, mutexes, file system, and DLL loading. Finally, we tested our analysis environment with the Al-Khaser [8] tool to confirm that our sandbox could not be identified. To measure the overhead introduced by our analysis system we executed 1,000 malware samples randomly chosen among those that: (i) terminate the execution, (ii) use at least one evasive technique, and (iii) **detonates** according to the threshold proposed in [65], i.e., the sample calls at least 50 Windows APIs. We measured their execution time with and without instrumentation by observing a percentage increase of $\mu = 125$, $\sigma = 31$, $min = 26$, $med = 106$, $max = 206$. This overhead is in line with that in [75]. Kuechler et al. [65] recently showed that the amount of code executed by malware samples plateaus after two minutes, and little additional information can be obtained thereafter. Thus considering the overhead mentioned above, we took a conservative approach and ran each sample for up to five minutes.

4.2.3 Dynamic Features

We extract 7 classes of dynamic features from the API calls (including their arguments) invoked by the malware during execution in the sandbox. The features were chosen to cover those used in previous works that built classifiers from malware executions (e.g., [47, 43, 17]).

The lower half of Table 4.2 summarizes the 7 dynamic feature classes (prefixed by `d-`). Categorical features such as filenames and domains are one-hot encoded to Boolean features. To encode each feature, we count all its possible values and exclude those appearing less than five times in the training set. The *d-network* class (438 features) captures the HTTP, TCP, and UDP traffic. Of those, 430 features capture unique domains contacted by the malware and HTTP User-Agent strings used; three count the number of HTTP requests, TCP connections, and UDP pseudo-sessions; and 5 randomness-related features capture the mean/median/min/max/std likelihood of domain names and URLs contacted according to a recently proposed Markov Chain model [17]. The *d-file* class features (60,555) capture

the name and extension of 60,547 files created or accessed by the malware, the number of files read, written, and deleted; and 5 capture the randomness of the filenames. The *d-mutex* class features (7) capture the number of mutex objects created and the randomness of the mutex names. The *d-registry* class features (60) capture 55 unique registry keys written, and the count of registry keys created, opened, read, written, and deleted. The *d-service* class features (736) capture the count, randomness, and names of services and service managers created, started, and halted. The *d-process* class features (28,198) capture the count of processes created, processes terminated, and shell commands invoked, as well as 28,195 unique process names. The *d-thread* class features (7) capture the number of the threads opened, created, resumed, terminated, and suspended, as well as the number of the interactions with the context of a given thread and the number of asynchronous procedure calls (APC) queued to a thread. The last two features help capture suspicious behaviors.

Missing features. When a dynamic feature cannot be computed (e.g.,due to lack of activity), we assign them default place-holder values that do not belong to the domain of the features. We refer to such features as *missing* features. For example, if a sample has no file system activity, we cannot compute the *d-file* filename randomness features. As a result, the 5 statistical features related to the randomness of the file names are thus not available. We perform dynamic feature extraction only over detonated malware samples (i.e., those that called at least 50 APIs as defined in 4.2.2), but even for detonated samples, there are still missing observations of feature values. To facilitate the analysis of the impact of the missing features, we define the *feature missing rate* (FMR) of a malware family as the fraction of family samples that have missing values in the file, registry, service, and process features (which, among the seven dynamic features classes we consider, are the most relevant for classification according to Table 4.10). Missing values over all these four feature classes considerably degrades both the amount and quality of useful information available to the classifier. According to our analysis, over 54% of the malware families studied in chapter contain on average 77% of the malware samples per family with missing feature values in these four dynamic feature classes. Missing observations can negatively impact ML classifiers by overfitting the data and reducing the model's accuracy. Recently, Aonzo et al. [17] showed that classifier models tend to focus on static features, rather than dynamic ones, precisely because static features are rarely missing. In Section 4.3.2 we analyze the impact of missing features in the classification results.

4.2.4 Models

We train multiple models to capture different axis: classification task (i.e., binary or family classification), features (i.e., static, dynamic, combined), classifiers (i.e., Random Forest, XGBoost), dataset construction (i.e., distribution of families in training dataset), and a different number of families and samples.

Classification task. We build models for binary and family classification tasks. The binary classification models detect whether a given sample is malicious (positive class) or benign (negative class). The family classification models identify the family of a given malicious sample, that is, there is one class per malware family and no goodware class. We prefix the name of a model with *binary-* or *family-* to indicate the classification task.

Features. We build models that use all static features, all dynamic features, and all combined features (i.e., all static and all dynamic). The name of a model includes *-static-*, *-dynamic-*, or *-combined-* to indicate the features used.

Classifiers. Given a large number of ML classifiers, it is not possible for us to systematically evaluate all of them. In our experiments we selected *Random Forest* and *XGBoost* because they are consistently among the best-performing classifiers evaluated in previous works (summarized in Table 2.1 and Section 2.2). Moreover, being tree-based, they are easier to interpret, they allow direct analysis of feature importance, and they are also intrinsically capable of handling both categorical features (e.g., unique filenames accessed during execution) and continuous features (e.g., filename mean randomness). We also considered neural networks, but discarded them because to achieve good performance they require larger training datasets (e.g., $\geq 400k$ samples in [99]). It was not clear whether we could build a balanced family dataset of the required size. In addition, there exist many potential neural architectures to evaluate and their training times are longer, which is critical given the large number of models we evaluate.

Dataset construction. For the binary classification task, we experiment with two ways of building our dataset, namely *uniform* and not *nonUniform*. The *uniform* approach builds datasets that balance the number of goodware and malware, using a sampling-with-replacement approach, as follows. We uniformly select from each family in M_B a number of samples

so that the total number of malicious samples matches the size of the benign dataset (i.e., each family in M_B provides 24–25 samples for a total of 16,611 malware samples). We repeat the process five times avoiding repetitions (i.e., each time selecting a different set of malware samples from each family in M_B), to completely cover all the malicious samples in each family. These steps produce 5 balanced datasets. Each dataset is split into 60% of samples for training, 20% for validation (i.e., selecting the classifier hyper-parameters), and 20% for testing. To evaluate a model, for each of the five datasets, we perform a 10-fold cross validation to ensure that all the samples equally contribute to the training and testing datasets. We report average results across the five rounds and their respective folds. Thus, obtaining the accuracy results from one model requires us to train and test 50 times.

The *nonUniform* approach replicates the unbalanced distribution of samples per family in the MOTIF dataset [53]. The motivation for this dataset is to study whether the family distribution in the training set of a binary classification task (where family labels are not used) affects the detection accuracy. In MOTIF, 29% of families have only one sample, 41% have 2-5 samples, 12% 6-10, 10% 11-20, 4% 21-30, 2% 31-40, 1% 41-50, and 1% has over 142 samples. We replicate this distribution on the 670 families in M_B. For example, we select one sample from 29% (randomly-chosen without replacement) of the 670 M_B families and 142 samples from one randomly-chosen family. The resulting dataset comprises all 16,611 benign samples and 4,821 samples from 670 families that follow the per-family sample distribution in MOTIF.

Number of families and samples. To measure the impact that the number of families to classify and the available samples for each family have on the results, we build multiple ML-based classifiers for the family classification task by uniformly sampling 70, 170, 270, 370, 470 and 570 families from the total 670 families. For each of them, we also experiment with a version trained and tested on 50, 60, 70, and 80 malware samples for each family. As indicated above, we have 20% samples used as the validation data. Therefore, at maximum, there are 80 malware samples for training and testing use.

4.3 Experimental study

This section presents the results of the experiments we conducted to answer the research questions presented in the introduction. We have adopted the

Task	Features	Precision	Recall	F1-score	Families with 100% accuracy
Binary	Static	0.956	0.957	0.957	242 (36.12%)
Binary	Dynamic	0.945	0.892	0.926	465 (69.40%)
Binary	Combined	0.963	0.934	0.948	450 (67.16%)
Family	Static	0.856	0.850	0.848	68 (10.15%)
Family	Dynamic	0.734	0.708	0.704	114 (17.17%)
Family	Combined	0.874	0.867	0.865	138 (20.60%)

Table 4.3: Overall classification results using Random Forest.

following structure for ease of reading: the reader will find the discussion to ⟨**Rx**⟩ in Section 4.3.x and a summary with the answer ⟨**Ax**⟩ at the end of each subsection.

4.3.1 Overall Classification Results

In this section, we examine how static, dynamic, and combined features impact binary and family classification. We first discuss the results using Random Forest and then discuss the XGBoost results. Table 4.3 summarizes the accuracy results using Random Forest. The results correspond to the uniform dataset construction approach. Each line in the table reports the averaged precision, recall, and F1 score of 10-fold cross validation. It also reports the fraction of malware families with 100% family-wise accuracy. In binary classification, 100% family-wise accuracy for a family denotes that the family can be perfectly differentiated from goodware. In family classification, 100% family-wise accuracy instead means that samples from a malware family are not misclassified as another malware family.

The static features achieve a higher F1 score than the dynamic features in both binary and family classification. However, the fraction of perfectly classified malware families is higher for dynamic features, suggesting that dynamic features work very well for some malware families, but poorly on others. The combination of static and dynamic features brings marginal improvements in F1 score over static-only features. It improves it by 1% for family classification, but decreases it by 2% for binary classification. On the other hand, adding dynamic features increases the percentage of perfectly classified families over the static case, although for binary classification the fraction reduces compared to dynamic-only features. The accuracy reduction with more features might seem counter-intuitive, but it can happen

Task	Features	Precision	Recall	F1-score
Binary	Static	0.907	0.902	0.904
Binary	Dynamic	0.978	0.820	0.892
Family	Static	0.705	0.690	0.697
Family	Dynamic	0.720	0.689	0.691

Table 4.4: Overall classification results for XGBoost. We failed to run XGBoost on the combined features due to its higher memory consumption.

when the two feature sets are not independent and bring different strengths and weaknesses that lead to mistakes on different parts of the input space. It is well known as the curse-of-dimensionality in machine learning [120]. Adding more features does not necessarily improve the overall accuracy, more features may bring unexpected variance and noise into the classification module [70].

Our results may raise concerns about the value of dynamic analysis. On the one hand, dynamic features outperform static features for a fraction of families, significantly raising the number of perfectly classified families (e.g., nearly doubling it for binary classification). This confirms the value of dynamic analysis, for example when researchers are interested to build behavioral signatures for specific malware families. On the other hand, the overall impact of adding dynamic features to static features is unclear. This might be the consequence of malware families for which dynamic features do not work well, because of intrinsic properties of the malware family (or malware class), but also because the sandbox might fail to stimulate samples adequately (e.g., due to evasion techniques or to the lack of a live command-and-control server). Adding dynamic features to the models may still provide other benefits. For example, recent work has shown that dynamic features are preferred by humans for interpretability [17]. Furthermore, dynamic features can increase the robustness of the model, making it more resilient to obfuscations designed to hamper static analysis.

XGBoost. Table 4.4 shows the classification results using XGBoost for static and dynamic features. We failed to run XGBoost on the combined features due to XGBoost's higher memory consumption, which becomes a bottleneck given the large number of features (roughly 100k) in the combined model. The results correspond to the uniform dataset construction approach and 10-fold cross validation. Similar to Random Forest, the static features achieve higher F1 score than the dynamic features in both binary

and family classification, although the advantage of static over dynamic is smaller in this case. Compared to Random Forest classifiers, XGBoost classifiers have lower F1-score for both binary classification (4.4% lower) and family classification (8.2% lower). Since Random Forest classifiers have higher accuracy, and they also run faster than XGBoost classifiers while consuming less memory, we use Random Forest classifiers in the rest of our evaluation.

Time-aware experiments. To avoid the temporal bias that cross-validation may introduce, Pendlebury et al. [90] suggested to split training samples into temporal bins. However, since our dataset only contains 100 samples per family, the individual bins would be too small and thus we decided to not perform temporal binning. Instead, in Section 4.3.8 we perform a separate out-of-distribution (OOD) evaluation with unseen families and singletons not present in the training dataset, which addresses the main bias that cross-validation introduces.

⟨**A1**⟩ For both binary and family classification tasks, models trained on static features alone provide higher accuracy than the models trained only on dynamic features. The latter is able to perfectly classify more families, but perform poorly on others, producing an overall lower classification accuracy.
Adding dynamic features on top of the static features brings marginal accuracy improvement for family classification and even negatively affects binary classification. On the other hand, dynamic features may offer benefits for model robustness and interpretability.

4.3.2 Hard-to-Detect Malware

This section analyzes which malware classes and families pose a greater challenge for classifiers based on static and dynamic features. Note that our multi-class classification models are for families. We only use here the coarser malware class (e.g., virus, worm) to draw conclusions on similar families.

Table 4.5 shows Recall and F1-scores for each malware class in binary and family classification respectively. In binary classification, the recall value is defined as the number of correctly classified samples in the class over the total number of samples in the class. The numbers differ from those in Table 4.3 because Table 4.5 only considers the classification results of malware samples, while Table 4.3 covers the classification of both goodware and malware samples (thus taking also false positives into account).

Class	Binary class. Recall			Family class. F1 score		
	Static	Dyn.	Comb.	Static	Dyn.	Com.
Adware	0.905	0.915	0.981	0.926	0.761	0.925
Backdoor	0.966	0.943	0.996	0.830	0.730	0.838
Clicker	0.971	0.929	1.000	0.817	0.692	0.821
Dialer	0.994	0.875	1.000	0.988	0.888	0.984
Downloader	0.974	0.899	0.996	0.864	0.695	0.874
Grayware	0.932	0.895	0.986	0.832	0.675	0.852
Miner	0.989	0.972	0.999	0.927	0.807	0.962
Ransomware	0.967	0.945	0.997	0.839	0.580	0.853
Rogueware	0.984	1.000	0.992	0.616	0.401	0.663
Spyware	0.972	0.829	0.998	0.869	0.704	0.879
Tool	0.992	0.929	1.000	0.864	0.778	0.830
Virus	0.885	0.939	0.971	0.819	0.719	0.809
Worm	0.978	0.899	0.996	0.922	0.721	0.921
Average	**0.967**	**0.920**	**0.9907**	**0.848**	**0.704**	**0.865**

Table 4.5: Classification accuracy for malware classes.

As we can see, the recall and F1 score are not uniform across all classes and can widely vary depending on the task and the features used. Static features are considerably better at detecting downloaders, dialers, and worms. In contrast, dynamic features perform better on rogueware, miner, and ransomware.

These results are confirmed also if we look at individual families. We provide Table 4.6, Table 4.7, Table.4.8 and Table.4.9 to show the 10 families with the lowest accuracy in both classification tasks using static and dynamic features. For instance, among the 10 malware families for which the static classifier makes more mistakes, we count four *viruses* (i.e., file infectors) and six *grayware*. This is even more remarkable if we consider the fact that there are only 40 families of Viruses in our entire dataset. The fact that *viruses* typically append their code to benign files results in a wide variation in terms of static features among samples of the same family, and this can explain why it is hard for a static classifier to differentiate them from *goodware* and from other families. Similarly, *grayware* is defined as undesirable code, which is not outright malicious per se, therefore making it difficult to find a clear boundary to isolate these families. In the worst 10 families using dynamic features, we can observe a similar pattern: grayware and viruses dominate the list. Besides, adware and spyware are also among

Static binary classification			
Family	**Class**	**Avg Recall**	**% packed**
pioneer	virus	0.401	6%
asparnet	grayware	0.410	5%
systweak	grayware	0.458	19%
shopper	grayware	0.500	1%
sality	virus	0.516	4%
vitro	virus	0.553	3%
installcore	grayware	0.596	10%
slugin	virus	0.598	4%
elex	adware	0.603	9%
passview	grayware	0.617	35%

Table 4.6: Top-10 malware families with the lowest binary classification accuracy using the static features (i.e., highest mispredictions as goodware).

the worst families. Malware samples in each of the classes have similar behaviors.

⟨**A2**⟩ Models employing static features find it more difficult to classify *grayware* and *viruses*. Dynamic features can identify ransomware, spyware, and adware as malware, but they have great difficulty in properly identifying their families, probably due to very similar runtime behaviors of different families in these classes.

4.3.3 Feature Class Importance

This section examines the importance of the static and dynamic features for binary and family classification using a Random Forest classifier. We measure feature importance using the average Mean Decrease Impurity (MDI) score. In a tree-based classifier, the MDI score of a feature captures how often the feature was used in the tree. The more a feature is used, the more important it is to distinguish different classes. For feature classes, we average the MDI Score across all the features belonging to the same feature class and over all the trees in the Random Forest model.

Feature classes. Table 4.10 summarizes the feature class importance. Overall, static features are ranked higher than dynamic features, especially for family classification. This matches results in Section 4.3.1 where dy-

Static family classification			
Family	**Class**	**Avg F1**	**% packed**
zpevdo	grayware	0.150	15%
vitro	virus	0.240	3%
uwamson	grayware	0.252	15%
gendal	grayware	0.280	62%
dumpex	grayware	0.290	40%
alman	virus	0.293	11%
sality	virus	0.328	4%
pasta	grayware	0.346	28%
cobra	grayware	0.381	60%
copidmbe	virus	0.387	9%

Table 4.7: Top-10 families with the lowest family classification accuracy using static features (i.e., highest mispredictions to other families)

namic features provide marginal improvements over static features. This observation is in line with recent findings that although humans prefer dynamic features, ML algorithms rely more on the *always present* static features [17].

The most contributing static feature classes for both classification tasks are *s-bytegrams*, *s-opcodegrams*, and *s-strings*. This confirms what was previously observed in the literature, with raw and opcode ngrams dominating over other static features [14]. On the other hand, the most contributing dynamic feature classes for both classification tasks are *d-file* and *d-process*. It is interesting to note that even expert human analysts used widely file and process operations to identify malicious behaviours [17].

In our dataset, over 50% of the malware samples contain missing features values in the *d-network* and *d-service* feature classes, thus missing feature values is likely the reason for their low importance. We evaluate this in Section 4.3.5. It is interesting that *d-registry* ranks second for binary classification, but only 10th for family classification. This means that registry operations are useful to differentiate malware from goodware, but they do not provide enough diversity to separate different malware families. This likely happens because multiple malware families operate on the same registry keys such as those related to achieving persistence (e.g., auto-start) and those that disable OS security features. In contrast, goodware does not need to operate on those keys.

Dynamic binary classification				
Family	**Class**	**Avg Recall**	**Packed**	**FMR**
tasker	grayware	0.0	11%	0.77
malex	downloader	0.0	1%	0.77
rostpay	grayware	0.0	96%	0.76
constructor	grayware	0.0	13%	0.78
atcpa	virus	0.0	0%	0.78
mocrt	spyware	0.0	73%	0.80
mokes	backdoor	0.0	1%	0.65
bingoml	grayware	0.0	22%	0.72
safebytes	grayware	0.0	99%	0.81
trymedia	adware	0.0	73%	0.70

Table 4.8: Top-10 malware families with the lowest binary classification accuracy using dynamic features (i.e., highest mispredictions as goodware).

Individual features. The most contributing static feature classes are *s-bytegrams* and *s-opcodegrams*, but their individual features are hard to interpret. For binary classification, the top 10 *s-strings* features capture 5 API names (*exit*, *CreateThread*, *cexit*, *CopyFileA*, *WinExec*), one section name (*.idata*), one module name (*MSVCRT.dll*), a string possibly related to the .NET runtime (*<assemblyIdentity*), and two short strings with unclear meaning (*:0806*, *L$ H*). The top *s-sections* features capture section entropy and bit 31 in the section characteristics field, which states if the section can be written to. These features are likely related to packing. We further examine which static features allow to detect packed malware in Section 4.3.4. The top *s-imports* features have some overlap with the top strings (e.g., *exit*, *cexit*), but also contain APIs possibly used for evasion (e.g., *queryperformancecounter*, *getsystemtimeasfiletime*) and popular C runtime functions (e.g., *free*, *calloc*, *malloc*, *fprintf*). For family classification, the top static individual features differ from those for binary classification with no intersection between the top 10 *s-strings* and *s-imports* for binary and family classification. For example, the top strings contain 6 API names (*WNetOpenEnumA*, *WNetEnumResourceA*, *WNetCloseEnum*, *RegisterServiceProcess*, *PathFileExistsA*, *UpdateResourceA*), a third-party library name (*StringX*), and some short strings (*QQQQS3*, *llllll*, *3.91*). These strings are not highly ranked for binary classification and are possibly associated with specific families.

Dynamic family classification				
Family	**Class**	**Avg F1**	**% packed**	**FMR**
bancos	spyware	0.0	44%	0.76
kovter	grayware	0.0	0%	0.78
safebytes	grayware	0.0	99%	0.80
winner	grayware	0.0	0%	0.80
umbra	downloader	0.0	0%	0.80
ulise	grayware	0.0	2%	0.80
contenedor	virus	0.0	0%	0.80
cobra	grayware	0.0	60%	0.79
kuaizip	adware	0.0	1%	0.80
zpevdo	grayware	0.0	15%	0.77

Table 4.9: Top-10 families with the lowest family classification accuracy using dynamic features (i.e., highest mispredictions to other families)

Among the dynamic features, the most contributing classes are *d-file* and *d-process*. In contrast to the static features, the top contributing dynamic features largely overlap between binary and family classification. The top process features are the number of processes invoking shell commands, and the number of terminated, opened, and created processes. The top file features capture the entropy of the files accessed, as well as the name of some specific files, such as *appdata\local\temp\ 7zipsfx.000*, which likely indicates the executable is an SFX installer. One difference between binary and family classification is that for family classification the number of mutexes created is a top contributor. Mutexes are often used by malware creators to avoid re-infecting the same host and their number and values are intuitively family-specific.

Overall, the interpretability of individual features can be hard, especially for n-grams. In fact, we argue that one benefit of ML classifiers is that they can select the features they consider most useful, which a human may not be able to identify based on domain knowledge. Our data release [2] includes the top individual features for the different models.

Feature Class	Binary classification			Family classification		
	Comb.	Static	Dyn.	Comb.	Static	Dyn.
s-bytegrams	40.88	51.38	-	38.60	41.67	-
d-registry	17.19	-	25.00	0.51	-	0.60
s-opcodegrams	13.44	21.08	-	23.48	20.87	-
s-strings	9.09	15.27	-	17.62	19.27	-
d-file	7.74	-	29.70	3.16	-	56.20
s-sections	3.05	6.73	-	5.62	6.48	-
s-imports	2.48	4.17	-	7.87	9.30	-
d-thread	2.06	-	7.34	0.16	-	5.26
d-network	1.51	-	3.50	0.35	-	3.70
d-process	1.47	-	32.90	0.87	-	30.70
s-headers	0.34	0.72	-	0.73	0.96	-
d-mutex	0.25	-	0.16	0.03	-	1.19
d-service	0.19	-	1.40	0.07	-	2.39
s-dll	0.17	0.28	-	0.52	0.57	-
s-file	0.13	0.35	-	0.39	0.87	-

Table 4.10: Feature class importance using MDI score.

⟨**A3**⟩ Static features are more important than dynamic features for both classification tasks, but especially for family classification. Raw and opcode n-grams are the most important feature classes in both classification tasks. The importance of a feature class may depend on the classification task. For example, *d-registry* is important to distinguish malware from goodware, but is not relevant for family classification.

4.3.4 Impact of Packers and Protectors

This section evaluates whether the presence of off-the-shelf packers and protectors harms the classification accuracy when considering static features. Our dataset comprises real malware collected from a commercial feed, so we expect the fraction of packed samples to approximate that in the wild. Overall, we identified 119 unique known packers, including highly sophisticated ones like VMProtect and Themida, covering 22% of the samples in our dataset. However, this ratio is certainly a lower bound as packer detection tools may not identify custom packers. Tables 4.6–4.9 show that the packing rate largely varies per family: some have 99% of their samples packed while others have none. As explained in Section 4.2.1, we did not attempt to unpack samples, but follow prior work in extracting static fea-

tures regardless of whether a file is packed or not. The packer information is only used for the analysis of the results.

We first investigate whether the models overfit the packers or instead can capture data that allows them to classify samples correctly. To answer this question, we first compute the family-wise classification accuracy for both binary and family classification using static features. We then compute the Pearson correlation scores between the family-wise accuracy scores and the rate of packed samples in each family. If packing negatively affects the ability to classify a sample, we would expect lower accuracy for families where packing is more prevalent. However, the correlation scores are 0.015 and 0.0001 respectively for binary and family classification. To statistically support these results, we run a T-test with the null hypothesis being that there is not a significant correlation between classification accuracy and packing presence. We respectively obtain 0.51 and 0.98 as p-values that do not allow us to reject the null hypothesis. Thus, we conclude that there is not a statistically significant correlation between the two variables. This might seem surprising, as one might expect a high correlation between packing and misclassification rate at least for models that rely only on static features. After all, packing was one of the main reasons that led researchers to introduce malware analysis sandboxes and dynamic analysis. However, this is a common misconception. In fact, while packing is very effective at impeding static *analysis* (i.e., the ability to examine a sample and statically derive its behavior), other works [14] have shown that common packers leave certain areas of the binary untouched, thus having a limited effect on the ability of a ML *classifier* to identify a sample. While our static models seem capable to detect samples protected with off-the-shelf packers, newer protectors can be designed to specifically target static models. Also, it is possible that some of the hard-to-detect families use (undetected) custom packers that indeed hamper the detection.

To understand which static features are more effective at identifying packed malware, we compute the importance of the feature classes separately for two sets: packed samples on one side and unprotected (i.e., not packed) samples on the other. Table 4.11 summarizes the results for both binary and family classification. The *All* column captures the feature importance for all samples (regardless of packing) and thus matches the values already reported in Table 4.10. The results show that for both binary and family classification of packed samples, the relative importance of *s-bytegrams* increases significantly (compared to all samples) and there are also relevant increases in the importance of *s-sections*, *s-headers*, and *s-dll*. On the other hand, the relative importance of *s-opcodegrams* and *s-imports*

Feature Class	Binary classification			Family classification		
	All	Packed	Not-Packed	All	Packed	Not-Packed
s-bytegrams	51.38	62.22	49.30	41.67	53.66	38.59
s-opcodegrams	21.08	8.30	22.69	20.87	9.95	25.02
s-strings	15.27	16.80	16.16	19.27	18.17	17.80
s-sections	6.73	7.50	6.29	6.48	9.39	10.17
s-imports	4.17	2.29	4.35	9.30	5.32	6.09
s-headers	0.72	1.42	0.63	0.96	1.30	1.17
s-dll	0.28	1.06	0.21	0.57	0.91	0.78
s-file	0.35	0.40	0.36	0.87	1.29	0.36

Table 4.11: Feature class importance using MDI score when considering all samples, packed samples only, and not-packed samples only.

is greatly reduced.

This is likely due to the fact that much of the code in packed samples is compressed or encrypted, reducing the amount of useful opcodes that can be extracted statically to those in the unpacking routine. On the other hand, raw bytegrams are still able to capture distinctive sequences of bytes, which may act like signatures for the packed samples. Those sequences can be extracted from parts of the executable that are not code (e.g., PE header and data sections). The classifier focusing on those parts for packed samples would also explain the increased importance of *s-sections*, *s-headers*, and *s-strings*. In addition, some packers use weak encryption schemes based on XOR operations with a fixed key, which may make distinctive byte sequences in the unpacked code to still be distinctive (in their encrypted form) in the packed executable. The decrease in importance for *s-imports* is likely linked to packers obfuscating the import table. Finally, most packers leave a very reduced import table that tends to use the same Windows libraries, which could explain the slight increase for *s-dll*.

⟨**A4**⟩ Packed or protected samples (with off-the-shelf tools) do not significantly correlate with their classification accuracy using static features. This means that although these technologies function well to deter static analysis (in particular reverse engineering), they do not significantly affect ML classifiers, which are still able to successfully identify byte-level signatures.

4.3.5 Impact of Missing Dynamic Feature Values

Some possible explanations for the worse results of dynamic features compared to static features are that a sandbox may fail to stimulate samples adequately to cause them to 'detonate', or that samples may not work properly due to missing local or remote components. As a result, the classifier might need to take a decision based on a partial view of the malware runtime behavior.

We computed the Pearson correlation coefficient between the family-wise recall of binary classification and the FMR to study the link between the two. Interestingly, the correlation is not statistically significant for the binary classification task (pearson -0.1 and p-value 0.11). However, there is a clear negative correlation (-0.43, p-value of $7.61 * 10^{-16}$) for the family classification task. In this case, as the fraction of samples with missing feature values for a family increases, its classification accuracy decreases. This is also confirmed by looking at the malware families that are the most difficult to classify with dynamic features, i.e., those for which the classifier has the lower accuracy (see Tables 4.8 and 4.9 in Section.4.3.2). Among the top-10 all have an FMR $> 65\%$.

This outcome demonstrates that the ML classifier might still be able to identify signs of malicious behavior in incomplete dynamic analysis reports, but more feature values are needed to precisely distinguish among different families (in particular for those, like downloaders, that might have similar behavioral profiles). In addition, binary classification is also affected by the quality of the behaviors collected from benign samples, while family classification accuracy is solely associated with the feature completeness of malware samples in each family.

⟨**A5**⟩ Globally, a statistically significant inverse correlation in the family classification task between the family-wise classification accuracy using dynamic features and the amount of missing dynamic feature values exist. The correlation is instead not significant for the binary classification task.

4.3.6 Impact of Ground Truth Confidence

To assign a family to a sample AVClass2 computes a list of (tag, confidence) pairs, e.g., (FAM:sality, 5), (CLASS:virus, 4), (FAM:zpevdo, 1). Then, it selects as family the highest confidence tag that is either a family in its taxonomy or an unknown tag not in its taxonomy. The confidence score roughly represents the number of AV engines that assign a tag to the sample, after accounting for aliases and discounting groups of AV engines that copy

their labels. This section examines whether the AVClass2 confidence score for the selected family impacts the classification accuracy.

To examine this issue, we first compute the confidence score for each family. For each sample, we obtain a normalized confidence in the [0,1] range by dividing the confidence score of the assigned family over the sum of the confidence scores for all family and unknown tags for the sample. In the case above, this step returns 0.83 as the *FAM:sality* confidence was 5, but *FAM:zpevdo* also appeared in the output. Then, we average the normalized confidence factor across all samples in the family to produce a family confidence score.

Next, we compute the correlation between the family-wise classification accuracy and the family confidence score. The hypothesis is that higher family confidence scores correlate with higher family classification accuracy, i.e., the more agreement AV engines have when tagging the sample, the easier it should be to classify the sample. The Pearson correlation coefficient is 0.083 for static features (p-value 0.03) and 0.062 for dynamic features (p-value 0.01). The correlation is positive but extremely small. Thus, we can conclude that poor family classification is not influenced by a low AVClass2 confidence score and the result is statistically significant. This is further confirmed by examining the 10 families with the lowest classification accuracy using either static-only or dynamic-only features (Table 4.6 and Table.4.8). Of those 20 families, all have a confidence score above 0.5 and 15 have a confidence score above 0.8. This suggests that even when the AV engines do not fully agree on the name of a sample, the majority vote likely selects the correct family, which provides further confidence on our AVClass2-based ground truth generation approach.

⟨**A6**⟩ The accuracy of family classification is not correlated with the AVClass2 confidence score, which captures the agreement between different AV vendors on the family name of a sample. This observation supports that AVclass2 is a valid tool for getting ground truth when it is necessary to obtain the family name of malware.

4.3.7 Impact of Training Dataset Construction

This section evaluates the effect of the construction of the training dataset on classification accuracy. We specifically investigate the impact of the size of the training dataset, the variety of malware families represented, and the uniformity of the sample-family selection. To the best of our knowledge, the question of how diversity in terms of families impact binary classification has not been studied before.

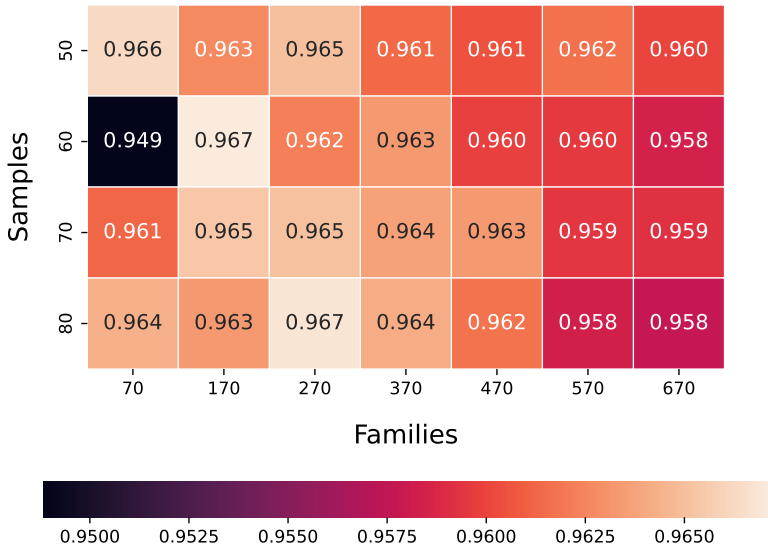


Figure 4.2: F1 score heatmap for **binary classification** using static model.

To study this aspect we plot a number of heatmaps. In each experiment, as described in Section.4.2, we reserved randomly 20 samples in each family for validation (e.g., hyper-parameter tuning) and we choose p samples from the remaining 80 samples and use them for training and testing. To study the impact of number of available samples, we vary p from 50 to 80. To study instead the impact of the number of different families in the dataset, we progressively vary the number of families involved in both binary and family classification from 70 to 670. For each combination of number of families and number of samples per family, we conduct a 10-fold cross validation test and report the averaged F1 score in the corresponding cell of each heatmap.

Figure 4.2 and Figure 4.3 present heatmaps of the F1 score for binary classification, using static features and dynamic features respectively. Figure 4.4 shows the heatmap for the combined model, for brevity only showing the variation with the number of families. Figure 4.5, Figure 4.6, and Figure 4.7 are similar but for the family classification task.

Overall, the results indicate that as the number of samples per family increases, the classification accuracy also increases. The exception is for the binary classification using static features, where increasing the samples per family may cause a decrease in overall accuracy. For example, when using 50 samples for each of the 670 families the F1 score is 0.960, but when using 80 samples it slightly decreases to 0.958. However, the trend is

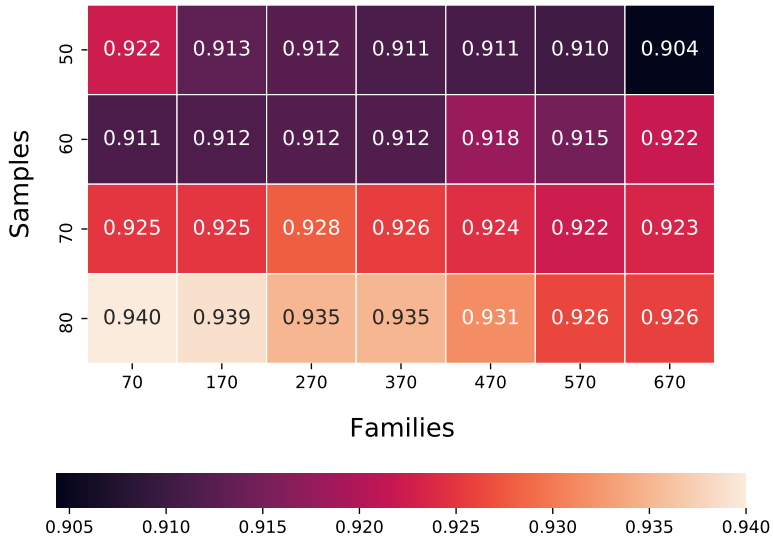


Figure 4.3: F1 score heatmap for **binary classification** using dynamic model.

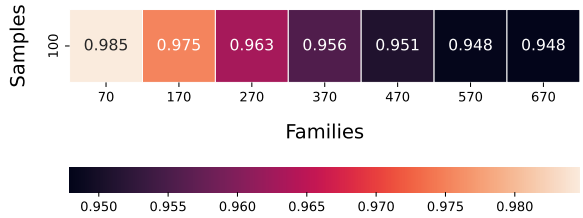


Figure 4.4: F1-score heatmap for **binary classification** with combined model.

different if we consider more families. We consider these very small changes as fluctuations due to the randomness of the sample selection process.

With respect to family diversity, the results confirm that the more families in the training dataset the more difficult their classification is. As expected, the decrease in classification accuracy is more marked for the family classification task, where intuitively the higher the number of classes the more difficult the classification becomes. The decrease is also more marked for the dynamic features than for the static ones, likely due to their lower discriminatory power as discussed in Section 4.3.1.

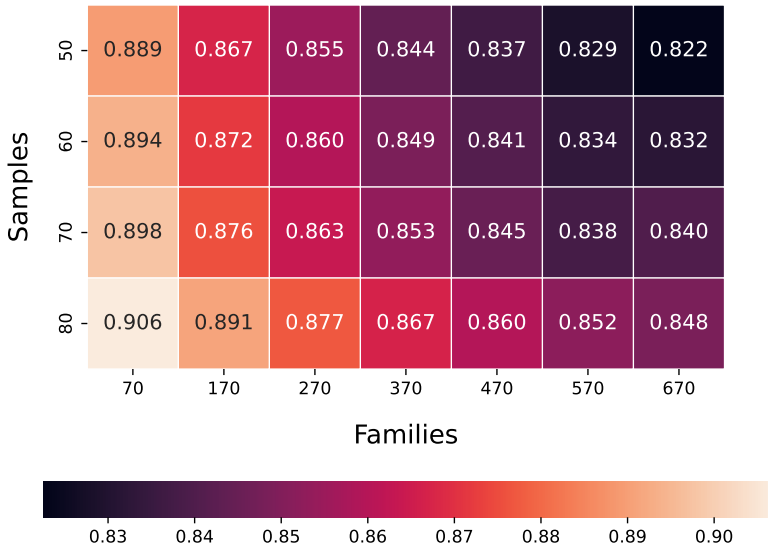


Figure 4.5: F1 score heatmap for **family classification** using Random Forest on static analysis features.

Non-uniform sampling. We also evaluate the impact of a non-uniform downsampling strategy for binary classification. For this purpose, we mimic the distribution of the recently-proposed MOTIF dataset [53], which contain 3,095 PE malware samples from 454 families with an unbalanced distribution (e.g., the median is 3 samples per family and 29% of families have a single sample). We create a new dataset by applying the MOTIF distribution to M_B. This new MOTIF-like dataset comprises 4,821 samples from all 670 families with the following distribution: 29% of the families are singletons, 41% have 2-5 samples, 12% 6-10, 10% 11-20, 4% 21-30, 2% 31-40, 1% 41-50, and 1% has over 50 samples (up to 100).

We use this to compare two sampling approaches: the *uniform* approach (which is the one we adopted so far) where we keep a balanced number of samples for each family, versus a *nonUniform* approach, where we consider a real-world case in which the number of available samples varies from one family to another, as captured by the MOTIF-like dataset. Table 4.12 shows the results for both approaches and different feature sets. We could not identify any significant difference between the two approaches, thus suggesting that training a classifier with a non-uniform amount of samples does not significantly impact its performance, under the important assumption that the testing dataset also follows the same distribution.

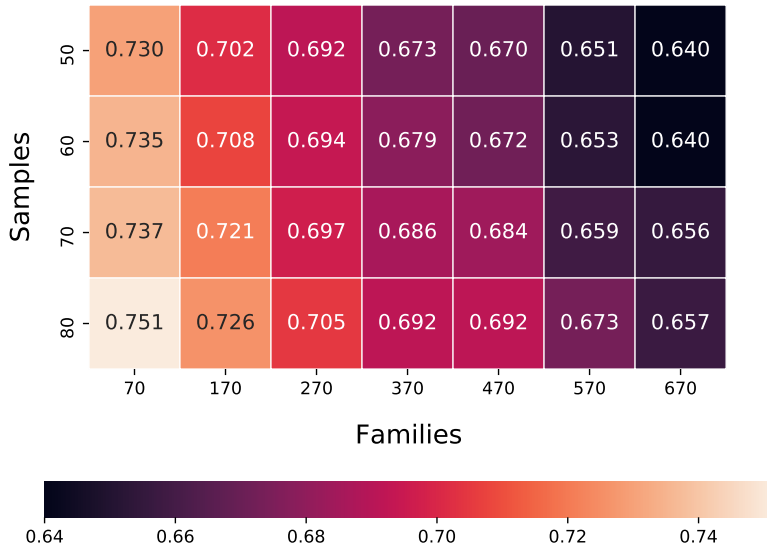


Figure 4.6: F1 score heatmap for **family classification** using Random Forest on dynamic analysis features

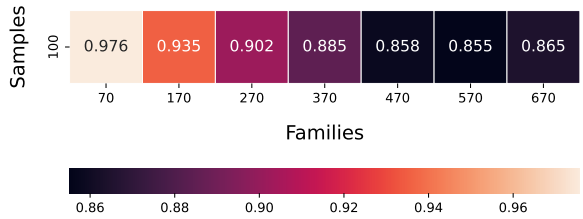


Figure 4.7: F1-score heatmap for family classification when combining features derived from static and dynamic analysis

⟨**A7**⟩ Increasing the number of malware families in the training set makes the classification more complex and generally results in lower accuracy. While not surprising, this is very important because previous studies were often performed on only a few dozens of families, with the risk of reporting inflated results that do not generalize to larger and more realistic datasets.
Increasing the number of samples per family can help to increase the classification accuracy, in particular for models based on dynamic analysis. Finally, the choice between a non-uniform and a uniform downsampling strategy does not significantly affect the binary classification accuracy.

Model	Prec.	Recall	F1	Acc.
binary-static-uniform	0.956	0.957	0.957	0.957
binary-dynamic-uniform	0.962	0.892	0.926	0.929
binary-combined-uniform	0.963	0.934	0.948	0.948
binary-static-nonUniform	0.961	0.960	0.961	0.960
binary-dynamic-nonUniform	0.959	0.886	0.921	0.924
binary-combined-nonUniform	0.955	0.927	0.940	0.927

Table 4.12: Impact of uniform and non-uniform sample selection in training dataset.

Model	Singletons	Unseen
binary-static-uniform	0.943	0.815
binary-dynamic-uniform	0.805	0.898
binary-combined-uniform	0.985	0.908
binary-static-nonuniform	0.810	0.653
binary-dynamic-nonuniform	0.328	0.855
binary-combined-nonuniform	0.758	0.637

Table 4.13: Binary classification accuracy on singletons and unseen families datasets.

4.3.8 Model Generalization

In this section, we test how well our models for binary and family classification generalize on unseen data. To this extent, we validate the performance of the previously-trained models on the singleton and unseen datasets introduced in Section 4.1.2, which include new families and have different distributions from the training data. This scenario is known as the "out-of-distribution" (OOD) test [72], where training and testing data have different distributions in the feature space. The distribution gap between the training and testing data has been frequently witnessed in malware analysis [52], as malware families evolve rapidly over time. Theoretically, one should expect the performance of a ML model to drop drastically in this more realistic scenario, as OOD samples directly violate the IID assumption of ML techniques.

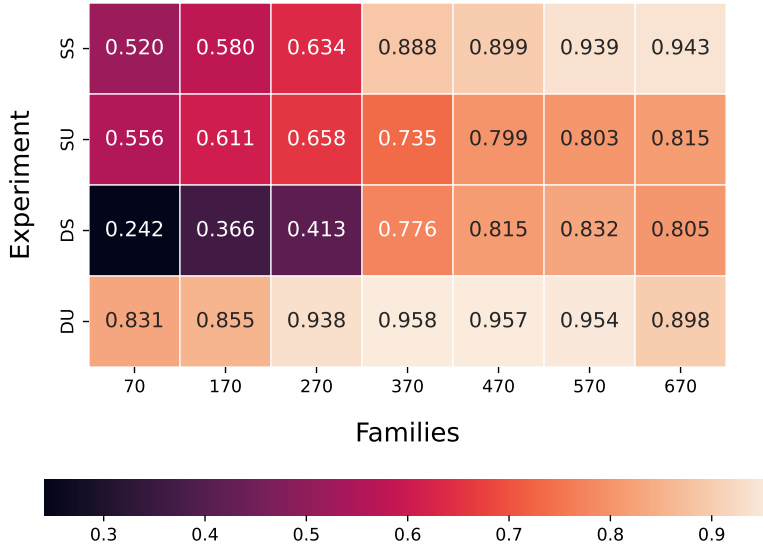


Figure 4.8: Binary classification accuracy on singletons and unseen families of the uniform dynamic and static models. (SS: Static Singleton. SU: Static Unseen. D is for Dynamic)

Binary Classification. Table 4.13 summarizes the binary classification results over the singletons and unseen families using the static, dynamic, and joint feature pool. "Uniform" and "non-uniform" in the table denote training with the 670 families with uniformly and non-uniform dataset construction methods (Section 4.2.4) The empirical measurements shown in Table 4.13 can be summarized around three main observations.

First, the accuracy of binary classification using only static or dynamic features deteriorates significantly over singleton and unseen family files. Using the combined feature set, the binary classification accuracy with the uniform setting augments over the singleton samples, whereas it deteriorates over the unseen families. In the non-uniform setting, we can observe the same tendency of accuracy drop over the OOD samples. The observations echo closely to the out-of-distribution challenge of machine learning raised in [72].

Second, the accuracy deterioration over the out-of-distribution samples is more significant in the non-uniform setting of training than that in the uniform setting, regardless of the used features.

This is different from the results of the in-distribution evaluation in Table 4.12, where we observe no major difference in accuracy between the uniform and non-uniform settings. These results show an important point:

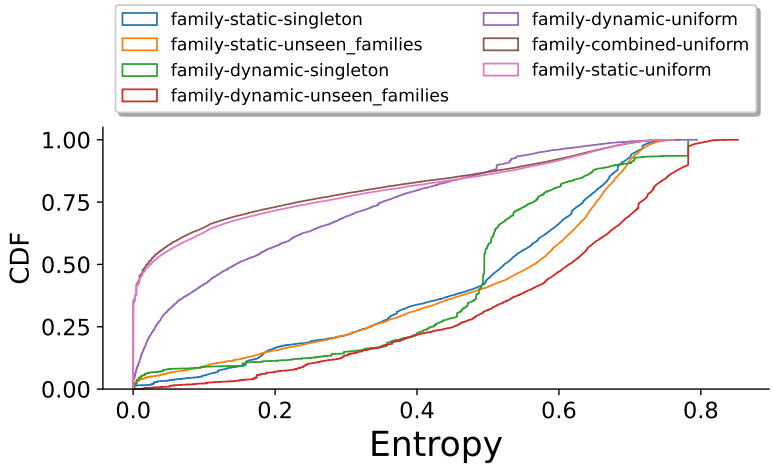


Figure 4.9: Entropy distribution comparison

classifiers built on very unbalanced datasets may perform equally well when tested on samples with the same unbalanced distribution, but generalize more poorly to other testing datasets, likely because many families were underrepresented in the training and thus the model failed to properly capture them.

Third, we can notice that static features generalize poorly to unseen families, while dynamic features perform better in this scenario. This is due to the nature of the features themselves: static information can precisely pinpoint only known samples, while dynamic behavior can better generalize also to unknown ones. Thus, compared to static features, dynamic features may provide more rich information to capture new types of malicious behaviors that never appear in the training phase.

We investigate this aspect in more detail by varying the number of families we used for training. In Figure 4.8, we can see that dynamic features perform poorly when the number of malware families for training is low (as there was not enough example of behaviors to learn from) but, with a sufficient number of families, they offer better classification results than static features. Dynamic features usually have a high dimensional and highly sparse feature representation. For example, some files or processes only appear a few times in the training set for specific malware families. A smaller number of families may aggravate the curse of dimensionality, which results in an overfitting of the classifier. Furthermore, we can observe the classification accuracy over unseen samples improves as the number of families increases, regardless of the features used in the test.

Family Classification. So far, we only tested the generalization of our models in a binary classification scenario. We now apply our family classifier trained using the 670 families over the singleton and unseen families as another out-of-distribution test scenario. Achieving high or low classification accuracy over these out-of-distribution samples is not interesting, as most of these samples share no common families as the training data and we don't have the ground truth family labels for these samples. Thus, the purpose of organizing this test is only to study how the uncertainty level of the family classifier changes over the out-of-distribution malware samples.

To measure the uncertainty difference, we define the Relative Entropy Score (*RES*) of the classifier's output as $\frac{\sum_{k=1}^{C} p_k \log p_k}{T}$, where $T = \sum_{k=1}^{C} 1/C \log 1/C$ and C is the number of the families covered by the training data building the classifier. In this experiment, C is therefore set to 670. For an input sample, the output of the family classifier is a 670-dimensional probability-valued vector $\{p_k\}$ (k=1,2,3,...,C=670). Each p_k gives the probabilistic confidence that the sample belongs to the corresponding family. By definition, the numerator $\sum_{k=1}^{C} p_k \log p_k$ provides the entropy of the classifier's output. The denominator $\sum_{k=1}^{C} 1/C \log 1/C$ denotes the maximum entropy that the classifier's classification output may have. As a result, the magnitude of *RES* is strictly normalized between 0 and 1. Higher/Lower *RES* denotes that the classifier shows higher/lower uncertainty level over the classification output.

In Figure 4.9, we demonstrate the empirical cumulative distribution function (CDF) of RES-based uncertainty distribution of the family classifier's output on the testing malware samples of the 670 families (in-distribution samples) and those belonging to singleton / unseen families. Consistently with theoretical studies [72], we can find that the uncertainty level of the family classification output over the singleton and malware samples of previously unseen families increases significantly, compared to those derived with the testing samples sharing the same families of the training data.

⟨A8⟩ Our experiments confirm a significant performance drop in binary classification over out-of-distribution samples, both in the case of singleton and unseen families. At the same time, the confidence of the ML-based classifier decreases significantly over these out-of-distribution samples. This implies that ML-based models tend to be less certain over malware samples drifted from the training samples. Our results also show that models trained on a very unbalanced dataset generalize more poorly, and that dynamic features generalize better than static over new families. Overall, as the distribution gap between training and testing malware samples is common in practice, these results raise concern over the utility of ML-based malware classification for real-world scenarios.

4.4 Final Recommendations

The goal of this Chapter was to understand the key factors that influence the performance of ML models for malware detection and family classification. Based on our experimental results, we can draw some general recommendations on the use of ML for malware classification:

1. Ideally, experiments on malware classification (both binary and family) should be performed on hundreds of different families, each containing a sufficient and balanced number of samples. However, this is often difficult to achieve in the malware field. Thus, we believe the contribution of our Chapter is not to simply re-state this obvious finding, but to provide for the first time a quantitative assessment of the impact of the lack of these characteristics on the classification results. For instance, we show that classifiers trained on a few families (like the ones using the popular Microsoft dataset) can provide misleadingly high accuracy scores while experiments conducted on unbalanced datasets tend to generalize poorly when tested over different distributions.
   Our findings can also be used to better understand and compare results reported in previous studies. For example, our results show that a family classifier with a F1 score of 0.89 over 600 families is likely better than a classifier with a score of 0.93 on 30 families.

2. Static features dominate detection and classification of samples from *known* families, by relying on signature-like information extracted from sequences of bytes and opcodes. Packing, in its current widespread implementation, does not seem to have a considerable negative effect on this. The addition of dynamic features, which are much more time-consuming

and error-prone to extract, has only a marginal impact on the classification accuracy and therefore its use should be carefully considered if the goal is to detect known families. However, static features are unable to capture samples from *unknown* families, where instead models based on dynamic behavior show a better ability to generalize. Therefore, our findings suggest that *today* static features alone are sufficient for family classification, but a combination of static and dynamic features is probably preferable for binary classification.

3. The performance of *all* ML models drop drastically when tested on OOD samples. While the feature completeness and the regular update of the training data to cover new malware families are key to obtaining good classification accuracy, both of them are difficult to achieve in the real world. It is due to the data-driven nature of ML-based classification mechanisms. The quality and coverage of training data play a core role in determining the classification performance. Beyond improving the quality of training data, our experiments suggest that the inclusion of dynamic features into the classification task can be used to alleviate the impact of the OOD issue. More specifically, we show that using dynamic features still allows us to successfully flag suspicious previously-unseen malware samples, even if with less accuracy and higher false positive rates in binary and family classification tasks.

This chapter opens several directions for future work. For example, we would like to explore how to mitigate the impact of missing features in dynamic analysis, e.g., through feature selection. We also plan to analyze the reasons behind hard-to-detect families, which could be due to custom packers, benign functionality in the malware, generic families that cover different malware, or other reasons.

Chapter 5

Understanding Intra-Family Diversity and Polymorphism

After focusing on inter-family classification challenges in the previous chapter, we now turn to an equally critical but often overlooked problem: understanding the structural variability that exists within malware families themselves. This chapter systematically investigates its root causes, quantifies its extent, and analyzes how it manifests across different parts of the PE file format.

A fundamental starting point for this discussion is clarifying what constitutes a *malware family*. Despite decades of research and thousands of publications, the scientific community still lacks a universally agreed-upon definition. One common interpretation is that a malware family consists of distinct malicious samples (i.e., with different file hashes) that all originate from the same code base, similar to how different builds or versions represent the same benign program. Samples within a family are generally expected to share behavioral traits, structural patterns, and attribution to the same threat actor group [6].

However, this expectation is challenged by the widespread use of *polymorphism*. Malware authors routinely introduce structural variation, even within the same malware version, using techniques like re-packing, binary obfuscation, and code injection. As a result, samples from the same family can differ substantially in binary representation, even if their underlying functionality remains the same. For example, file infectors spread by embedding malicious code into unrelated host executables, creating further diversity among samples. This polymorphism is one of the main drivers behind the constant growth in the number of distinct malware binaries observed by the security industry [35].

Such intra-family variability has important implications for analysis and detection. AntiVirus (AV) signatures are generally designed to match families or significant portions of them, not just individual samples. Likewise, ML-based family classifiers are expected to generalize from training samples to unseen variants from the same family. Yet, this generalization depends heavily on the nature and extent of differences between samples. Minor discrepancies in PE headers have a very different impact than deep structural changes introduced by packers or protectors.

This chapter aims to provide the first comprehensive exploration of the reasons behind the polymorphism in the samples belonging to the same malware family. For this, we leverage three malware family datasets (the Balanced Dataset proposed in Chapter 4 [34], the MOTIF dataset [53], and the Malicia dataset [83]), building a superset composed of 66,160 samples split into 743 families. Our analysis involves a static examination of malware samples to understand the syntactic variations within samples belonging to the same malware family. Static analysis is preferred for examining such syntactic characteristics, whereas dynamic analysis is necessary for assessing behaviors. However, in our dataset, the behavioral similarities among the samples are already captured by the family labels assigned to the samples. More specifically, this chapter is organized to answer the following three Research Questions:

RQ1: *How can we measure the structural differences among multiple samples from the same family?* At first, in Section 5.2, we break down the PE file format in a number of disjoint *components*, that fully cover the whole PE file format structure and content. Then, given two executables, we design a structural comparison approach to precisely locate their differences and similarities at the component level. We implement our approach in an open-source tool we named *PEdiff* [10].

RQ2: *What are the polymorphic techniques that affect multiple components, and what is their prevalence?* In Section 5.3, we examine two main reasons for cross-component differences: file truncation and packing. *Truncation* occurs when the expected size of a sample is larger than the real size of the file on disk. Truncation occurs due to errors during sample collection (e.g. samples extracted from network traffic where packets were missing). *Packing* is a technique that compresses or encrypts code on disk and then recovers it at runtime. We measure packing in two ways: by using state-of-the-art signature-based tools to reliably detect known off-the-shelf packers and by implementing a machine learning (ML) classifier proposed by Aghakhani et al. [14] to also identify custom packers.

RQ3: *What are the many reasons of polymorphism at the component*

granularity? In Section 5.4, we examine polymorphism in one or multiple components. Our results show that two-thirds of the families have no common components among their samples, meaning that all the PE components are at least slightly different. On the other hand, for 12.8% of the families, we were able to pinpoint the single reason behind the polymorphic variants.

In summary, we first developed a novel methodology for the structural comparison of PE files. Then, we highlighted the importance of two common elements (packing, truncation) which are crucial for the construction of malware datasets. We advocate for the community to conscientiously consider the elements they wish to exclude or include in their studies, given the potential bias these decisions may introduce. Lastly, we conducted a comprehensive measurement of polymorphism across 743 malware families. This analysis provides valuable insights for future research, enabling a targeted focus on the most prevalent trends and the timely development of appropriate solutions.

Finally, the scientific significance of this work is particularly relevant in the context of the design and evaluation of robust ML classifiers: we believe that their (in)ability to generalize to different samples needs to be always corroborated by an analysis of the variability of samples within the families in the dataset.

5.1 Datasets

This study uses three malware datasets: the Balanced Dataset [34], MOTIF [53], and Malicia [83]. A detailed description of each dataset, including their size, collection period, labeling methodology, and family distribution, is provided in Chapter 3.

For our experiments, we apply specific filtering to ensure reliability and comparability across analyses. We first remove truncated samples and families with fewer than 10 non-truncated samples, as detailed in Section 5.3.1. This step ensures that all samples used have valid and analyzable binary content.

For families with more than 100 non-truncated samples, we randomly select 100 samples to maintain consistency across family sizes. In the end, the final dataset used in this study comprises **66,160** samples distributed across **743** families.

Type	Components
Metadata	DOS Header, DOS Stub, Rich Header †, COFF Header, Optional Header, Data Directories †, Section Table
Sections	Entry Point Section, Resource Section †, Other Sections †
Extra	Certificate Table †, Overlay †

Table 5.1: Components of a PE executable and their type. The dagger † indicates an optional component.

5.2 Structural Comparison

While samples in a family differ in file hash, they may exhibit similarities, while their differences may be concentrated on specific parts. To examine similarities and differences within a malware family, we have designed a methodology for structural comparison of executables. It first divides each executable into 12 disjoint *components*, described in Section 5.2.1, that fully represent the PE executable format [31]. Next, it performs pairwise comparisons of all executables in a family at the component level, categorizing components as unchanged, similar, or different, as detailed in Section 5.2.2 We have implemented our methodology into *PEdiff* [10], an open-source tool comprising 1K lines of Python code.

5.2.1 PE Components

We split each executable into 12 disjoint (i.e., non-overlapping) components that capture its structure, depicted in Table 5.1. We grouped them into three parts: the *Metadata* contains the first seven components, which do not carry the actual content of the executable but define its structure and properties, the *Sections* which contain the code and data of the executable, and the *Extra*, which consists of components that are appended at the end of the file. Of the 12 components, six are optional and may not exist.

Within the Metadata, the *DOS Header* and *DOS Stub* correspond to the legacy MS-DOS information that is still present for compatibility. The *COFF header* and *Optional Header* capture the homonymous PE headers. The *Rich Header* is an undocumented component containing information about the tool versions used to build the different object files in the executable [134]. The *Data Directories* is an array of 16 entries, where each entry contains the start offset and size of a data directory, including the export, import, resource, and certificate tables. The *Section Table* is an

array that defines the name, start offset and the size of the sections that form the main body of the executable.

We identify three Sections components: The *Entry Point Section* is the section that contains the *AddressOfEntryPoint* field of the Optional Header. The *Resources Section* is a special section that contains a tree structure holding data items such as strings, images, and icons. Finally, the *Other Sections* component captures all other sections in the executable that do not contain the entry point or the resources. This is the only component that does not necessarily correspond to a contiguous sequence of bytes, since the order of the sections is defined in the Section Table and the entry point and resources sections may not be the first or last sections.

Executables may contain two optional Extra components. For signed executables, the *Certificate Table* contains a digital signature and a list of X.509 certificates for validating the file's integrity and the identity of the publisher. The *Overlay* component captures data appended at the end of an executable. This data is not described in the PE header, thus it is ignored by the loader. However, it is accessible by reading it directly from the file on disk. The presence of an overlay can be identified because the file's expected size (i.e., the sum of the start offset and size of the last section) is smaller than the real size of the file on disk. Some tools consider the certificate table to be an overlay. However, we consider it a separate component because its start offset and size are defined in the Data Directories and thus its existence is known to the loader. For signed samples, we consider that an overlay exists if and only if there is additional data after the end of the certificate table.

5.2.2 Family Component Analysis

Given the samples in a family, our goal is to identify which components are similar and different in the family. For this, we compare the contents of a component across all pairs of samples in the family. For each component in each pair of samples, we apply a pairwise similarity function to determine whether the contents of the component across the two samples are similar or not, and accumulate results across all pairs of samples.

Pairwise similarity. We experiment with three Boolean similarity functions that given the content of a component in two samples determine whether the component is similar. The first function computes the SHA256 hash of the sequence of raw bytes of the component[1] and checks if both

[1]For the Other Sections component, we sort the sections according to their offset, concatenate their raw bytes, and compute the SHA256 of the resulting buffer.

Algorithm 1 Determine Component Status: Similar, Different, or Missing

Require: Array of samples S belonging to family F, Component c, Threshold t

Initialize $C_p \leftarrow 0$
Initialize $C_d \leftarrow 0$
$NS \leftarrow |S|$, number of samples in S
$P \leftarrow$ all combinations of S
$NP \leftarrow \frac{NS(NS-1)}{2}$, number of samples combinations and cardinality of P

for all $(S_x, S_y) \in P$ **do**
    if $c \in S_x$ **and** $c \in S_y$ **and** $S_x[c] = S_y[c]$ **then**
        $C_p \leftarrow C_p + 1$
        $C_d \leftarrow C_d + 1$
    else if $c \notin S_x$ **and** $c \notin S_y$ **then**
        $C_d \leftarrow C_d + 1$
    end if
end for
$C_p \leftarrow \frac{C_p}{NP}$
$C_d \leftarrow \frac{C_d}{NP}$

if $C_p \geq t$ **and** $C_d \geq t$ **then**
    return Similar
else if $C_p < t$ **and** $C_d < t$ **then**
    return Different
else
    return Missing
end if

hashes are the same. This is the strictest similarity function requiring both samples to have identical content in the component. The second function computes instead the TLSH [86] fuzzy hash over the components' raw bytes. Fuzzy hashes output similar digests when the inputs are similar. Among all the fuzzy hashes available in the wild, we chose TLSH because it is the one that can produce a hash for the smallest stream of bytes, given that the minimum size is 50 bytes. Thus, it can handle most of the smallest components that are usually headers. Other fuzzy hashes could require very large minimum sizes (e.g. SSDEEP requires at least 4KB to compute the hash). TLSH returns a distance in the $x \in [0, \infty)$ range, which we

normalize ($y = max\{\frac{300-x}{3}, 0\}$) to a similarity in the $y \in [0, 100]$ range as suggested by other works [124, 88]. The component values are considered similar if the TLSH similarity was ≥ 90, as proposed by Oliver et al. [86]. This function is more lax because it considers the component values to be similar even if they are not identical, as long as the raw byte differences are small. Pagani et al. [88] showed that TLSH can remain robust when small modifications are introduced in the code; however, they also observed that compiling the exact same source with seemingly minor tweaks (such as slightly different compiler flags) can result in anything from negligible differences to extensive ripple effects in the final executable. Therefore, we compute the code similarity using the popular BinDiff [41] tool, which disassembles both executables (using IDA Pro 8.1 in our setup) and uses graph isomorphism and heuristics to match their functions. It returns a similarity value in $[0, 1]$. The advantage of BinDiff is that it disassembles the code and thus can ignore differences in the data between code blocks and handle some code reordering. But, it only measures code similarity, so we only apply it to the Entry Point Section component. We determine that the Entry Point Section of two samples is similar if their BinDiff similarity is > 0.85, as suggested by Egele et al. [36].

Family components. To determine if a component is similar, different, or missing across a family we use Algorithm 1. It takes as input the $10 \leq n \leq 100$ samples that belong to a family, a similarity function, and a threshold t. For each component c, it initializes to zero two counters: C_d^c and C_p^c. The first captures the number of pairs a component differs and the other the number of pairs where the component is present. For each of the $n(n-1)/2$ pairs of samples in the family, it compares each of the 12 components. If a component c is present in both samples and is similar, it increments both counters for the component; if present in only one sample, the counters are not modified; and if the component is absent in both, it increments C_d^c. Once all pairs of samples have been analyzed, counters are normalized by dividing them by the number of pairs. For each component, if $C_p \geq t$ and $C_d \geq t$, the component is deemed *similar* (present and consistent in most samples), if $C_p < t$ and $C_d < t$, the component is present with varying values and is classified as *different*, and if $C_p < t$ and $C_d \geq t$, the component is often *missing* and thus should be ignored as there is not enough information.

Threshold selection. Requiring a component to be similar across all pairs of samples (i.e., t=1.0) is too strict, e.g., some samples could have wrong family labels. Instead, we evaluate lower threshold values that allow a fraction of pairs to differ. Figure 5.1 presents the number of families with none, few (1-3), or many (>3) common components while varying the threshold t

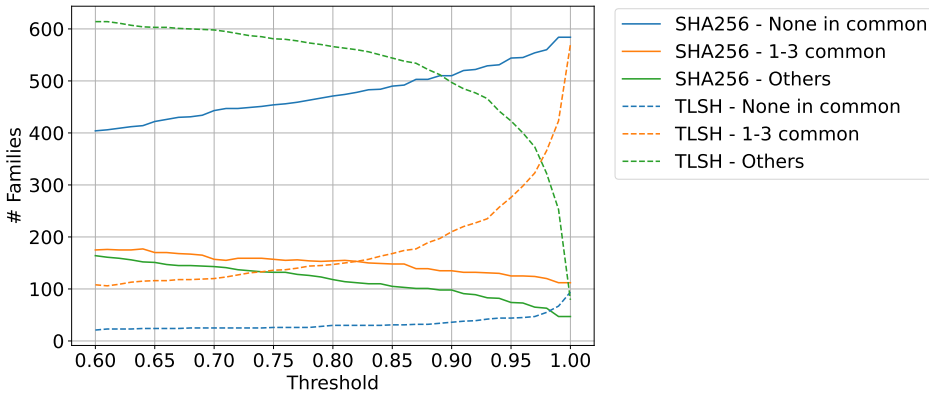


Figure 5.1: Number of families with none, 1-3 or more than 3 common components by varying the threshold, and using SHA256 or TLSH.

for both SHA256 (solid lines) and TLSH (dashed lines). The results show clear differences between both pairwise similarity functions. Using SHA256, raising the threshold increases the number of families with no similar components gradually from 405 at t=0.6 to 510 at 0.9, while the number of common components gradually decreases in opposite fashion. In contrast, using TLSH the number of families with no common components remains fairly constant around 20 and only starts to increase at high thresholds. Our manual analysis shows that using TLSH is problematic with small data components (i.e., PE headers in Metadata) where component similarity is determined even when important fields differ. Thus, we conclude that using the stricter crypto hash is more appropriate to analyze polymorphism. While changing the threshold of the crypto hash affects a relatively small subset of families, too low a threshold reduces confidence that components are truly shared, while too high a threshold increases the risk of discarding slightly different but essentially similar data. For the experiments in Section 5.4 we use the SHA256 pairwise similarity with a threshold of 0.75 which we have observed to strike a good balance by tolerating small differences without overlooking meaningful commonalities.

5.3 Cross-Component Analysis

Before proceeding to the analysis of individual components, we analyze different sources of variability that are not specific to a component but may instead affect the entire PE file structure: file truncation and packing.

5.3.1 Truncation

Millions of malware samples are collected and shared daily by various companies and services. Some are retrieved directly from filesystems by endpoint protection systems, while others come from emails, compressed archives, network traffic, or memory. Additionally, files may be pre-processed by tools like unpackers. During this collection and transformation process, a sample might become truncated, either intentionally or unintentionally. Truncated samples can be identified because their expected size (i.e., the sum of the start offset and size of the last section) is larger than the real size of the file on disk. One caveat is that if a sample has an overlay and the truncation only affects the overlay, then truncation cannot be detected as the overlay is not described in the PE headers. Among the 68,683 samples in the three datasets, 2,504 (3.6%) are truncated. Of those, the vast majority (2,486) belong to the Balanced Dataset and 18 come from MOTIF, with Malicia having none. Truncated samples are distributed among 42.9% (320/746) of the families, showing that truncation is indeed a widespread phenomenon. We found 1.9% (14/746) families where more than half of the samples are truncated. The families with the majority of truncated samples are: *stone* (97%), *kuaizip* (92%), *ranumbot* (88%), *duote* (87%), and *spybot* (82%).

To measure the missing part, we define the *truncation ratio* as the quotient of the missing bytes (expected size minus real size) over the expected size. The mean truncation ratio is 50.2% (median 52.9%), indicating that, on average, more than half of the expected file size is missing in truncated files. Then, by analyzing which components are impacted by truncation, we found that the offset from which bytes begin to be missing always starts at least after the Section Table; thus, truncation affects sections, resources, certificates, and overlay.

Another aspect of the truncation is whether the truncated samples came from the same original sample (i.e., the only difference is the truncation size). Thus, for each family, we compared all the truncated samples by checking whether the smaller sample was once again a truncated version of the largest. This analysis revealed 77 (10.3%) families having at least 2 samples coming from the same original executable and truncated in different points. The extreme case is one family, *spybot*, where 80 truncated samples come from the same original file that has been truncated at different offsets.

While truncated files are rarely, if ever, mentioned in malware studies, their consequences are very important. For instance, truncated samples cannot be executed, and thus fail dynamic analysis. But also static signatures may not match, and even popular static analysis tools produce confusing results when run on truncated files. For example, the popular *pefile* [11]

(a multi-platform Python module to parse PE files) erroneously reports an inexistent overlay on all truncated samples. Given that also VirusTotal uses pefile to parse PE executables, VT often reports truncated samples having a non-existent overlay.

We remove the truncated samples, as well as any families with less than 10 non-truncated samples, from the dataset. The dataset used in the remainder of this chapter has **66,160** samples distributed in **743** families.

5.3.2 Packing

	Samples	**Families**	
Factor	Present	Present(>0%)	Present(100%)
Packed (DiE)	15,704 (23.7%)	527 (70.9%)	20 (3.0%)
Packed (PackG)	24,134 (36.5%)	580 (78.1%)	45 (6.1%)
Packed (ML)	55,773 (84.3%)	726 (97.7%)	283 (38.1%)
Packed (All)	59,265 (89.6%)	731 (98.4%)	354 (47.6%)

Table 5.2: Packing prevalence in terms of packed samples and number of families with some/all packed samples.

We adopt a broad definition of packing, encompassing all transformations applied to a PE file, such as encryption, compression, or encoding, that require a runtime component to recover the original data. While these transformations primarily aim to thwart static analysis, they can also generate vastly different executables by re-packing the same sample with different algorithms or encryption keys. Consequently, packing is often regarded as a key driver of polymorphism.

Accurately detecting packed samples is a challenging task and an active research area. To address this, we employ two state-of-the-art approaches. Signature-based tools are effective at detecting known off-the-shelf packers, offering low false positives (FPs). However, they suffer from false negatives (FNs) due to their inability to identify custom packers or all instances of off-the-shelf packers. To overcome these limitations, we incorporate an ML classifier capable of detecting both less common off-the-shelf packers and custom packing routines. However, the classifier may overestimate the number of packed samples. In summary, we use signature-based tools to establish a lower bound and the ML classifier to set an upper bound on packing presence in our dataset.

Signature-based tools. We use two publicly available signature-based packer detection tools: PackGenome [68] and Detect-it-Easy [5]. PackGenome generates YARA rules from dynamic traces of the unpacking routine collected during program execution. The authors released the code for generating YARA rules for 20 off-the-shelf "accessible" packers. These rules identify 24,134 (36.5%) samples as being packed. The most commonly reported packers are UPX (48.6% of detected samples), WinLicense (31.7%) and PECompact (9.8%). The popular open-source Detect It Easy (DiE) tool, whose output is included in the VT reports at the time of writing, identified 15,704 samples (23.7%) being packed. The top-3 packers identified are UPX (58.8% of detected samples), ASPack (12.3%) and VMProtect (9.5%).

ML classifier. We implement a packer classifier by leveraging the datasets and the feature classes proposed by Aghakhani et al. [14]. On their datasets, we used the same nine static feature classes (56,485 individual features) they extracted to train a Random Forest classifier that predicts whether a sample is packed or not. We used 10-fold cross-validation to train and test the classifier and obtained an overall F1-score of 0.96. When applied to our dataset, the classifier predicted 55,773/66,160 (84.3%) samples as packed.

In summary, signature detection tools identify 23.7%–36.5% samples as packed, while the ML classifier identifies instead a much higher 84%, as summarized in Table 5.2. It is likely that the real fraction lies somewhere in between, so we use these numbers as lower and upper bounds, respectively. We illustrate the differences between both packer detection approaches using *privateexeprotector*, which has been mislabeled as a malware family but is rather a commercial protector. When the Balanced Malware [34] was created, AVClass taxonomy did not yet classify it as a packer, leading to its mislabeling. Executables labeled *privateexeprotector* may belong to different malware families using the same protector. Of 100 samples, 13 are tagged as "Private EXE Protector" by signature-based tools, 2 as UPX, and 87 remain undetected, showing high FNs. In contrast, the ML classifier detects 99 samples as packed, demonstrating minimal FNs, although potentially introducing FPs.

Finally, we noticed that many families use different off-the-shelf packers, a quick shortcut to achieve polymorphism. The most extreme cases are 8 families that use at least 15 different packers. In fact, we computed the Pearson correlation between the number of different components in each family and the total number of unique off-the-shelf packers in that family.

Component	Samples	Families				
	Present	Present(0%)	Present(≥50%)	Present(100%)	Differs	OnlyDifference
DOS Header	66,160 (100.0%)	-	743 (100.0%)	743 (100.0%)	524 (70.5%)	-
DOS Stub	65,059 (98.3%)	3 (0.4%)	734 (98.8%)	655 (88.2%)	484 (65.1%)	-
Rich Header †	42,577 (64.4%)	94 (12.7%)	497 (66.9%)	168 (22.6%)	494 (66.5%)	-
COFF Header	66,160 (100.0%)	-	743 (100.0%)	743 (100.0%)	622 (83.7%)	-
Optional Header	66,160 (100.0%)	-	743 (100.0%)	743 (100.0%)	652 (87.8%)	1 (0.1%)
Data Directories †	66,148 (99.9%)	-	743 (100.0%)	741 (99.7%)	638 (85.9%)	-
Section Table	66,160 (100.0%)	-	743 (100.0%)	743 (100.0%)	642 (86.4%)	-
Entry Point Section	66,155 (99.9%)	-	743 (100.0%)	739 (99.5%)	637 (85.7%)	2 (0.3%)
Resource Section †	57,230 (86.5%)	27 (3.6%)	661 (89.0%)	385 (51.8%)	625 (84.1%)	1 (0.1%)
Other Sections †	64,734 (97.8%)	3 (0.4%)	731 (98.4%)	583 (78.5%)	636 (85.6%)	7 (0.9%)
Certificate Table †	11,257 (17.0%)	344 (46.3%)	108 (14.5%)	19 (2.6%)	188 (25.3%)	1 (0.1%)
Overlay †	31,096 (47.0%)	87 (11.7%)	319 (42.9%)	87 (11.7%)	493 (66.4%)	37 (5.0%)

Table 5.3: Component statistics. For each component, number of samples where the component is present, number of families where the component is present in none/half/all samples, number of families where the component differs, and number of families where the component is the only one with differences. A dash means zero in all columns. A dagger † indicates an optional component.

We obtained a moderate relationship (0.42, p-value = 6.0e−34), namely, more packers, greater difference.

5.4 Component Analysis

We now use *PEdiff* to understand in which parts of a PE file the differences among samples of the same family are located. Section 5.4.1 first examines how frequently each component exists. Then, Section 5.4.2 identifies similar and different components in families. Section 5.4.3 examines where their polymorphism is being introduced in individual components.

5.4.1 Component Presence

Table 5.3 summarizes the presence of individual components in the dataset. Most required components are present in all samples (DOS Header, COFF Header, Optional Header, Section Table). Optional components vary significantly in presence. Three are common: Data Directories (present in 99.9% of samples), Other Sections (97.8%), and Resource Section (86.5%). Less common optional components include the Rich Header (64.4%), Overlay (47.0%), and Certificate Table (17.0%). The presence of an optional component is not consistent across all samples in a family. For example, while 399 (53.7%) families have some signed sample and 656 (88.3%) families

have a sample with an overlay, only 19 (2.6%) families have all samples signed and only 87 (11.7%) have an overlay in all samples. Similarly, there are 94 families (12.7%) where no sample has the Rich Header, 168 (22.6%) where all samples have it, but in the majority of families some samples, but not all, have it (87.3%).

5.4.2 Similar and Different Components

This section examines how many similar and different components there are in each family. Roughly two-thirds of the dataset, specifically 454 families, contain samples with *no* components in common, i.e., where all parts of the PE files are, at least partially, different. Among the remaining 289 families, 95 families contain largely similar files, with only one, two, or three different components and 157 families contains instead files that are all different *except* for 1–3 components.

We examine these three family groups separately. In most families, differences among samples span all PE file components, not just localized parts. This suggests malware authors did not achieve polymorphism by altering a few bytes or simply re-compiling (which would preserve sections like data, resources, and the Rich Header). For the 42,498 samples in this group, 40.1% are packed according to the signature-based tools, 84.7% according to the ML classifier, and 90.4% when combining both methods. Of the samples in the remaining 289 families (23,662 samples), 38.4% are packed according to the signature-based tools, 83.6% according to the ML classifier, and 90.4% using both methods. Despite slightly higher percentages in the first group, the difference is too small to attribute polymorphism to packing alone.

We focus on families with files sharing few common components. The most consistent components are the DOS Header and Stub, in 95 and 131 families respectively, and the Rich Header, in 21 families. In highly similar families, the most variable component, is the overlay (78 families), appearing over three times more frequently than others like other sections (22), resource (17) and entry point (14) section. Rare differences include headers and, in three families, only the Certificate Table.

5.4.3 Individual Component Polymorphism

So far, our analysis has been binary, components were either identical or different. In this section, we explore differences and similarities in greater detail. For example, over 5% of the families in our dataset contain samples that differ only in their overlay. What do these overlays look like? Are they

entirely different, or do they contain only a few unique bytes? To address these questions, we examine each component individually.

Rich Header. The Rich Header is an optional and undocumented component, which can be useful to detect if two executables may come from the same project. Among the 494 (66.5%) families where the Rich Header is different, in 12 families the entry IDs are similar, and in 4 family also its counts meaning that most objects used to build the samples are common.

Entry Point Section. This component contains the very first instruction of the program and thus can be considered a code section. The SHA256 pairwise similarity identifies 41 (5.5%) families where this component does not change at all and another 65 (8.7%) families where this component is similar. For the remaining 637 (85.7%) families, we use the BinDiff similarity to compare their code, identifying an additional 7 families with similar code. For 37/637 (5.81%) families BinDiff failed to properly disassemble all the samples, mainly because the entry point address points to invalid code. Such behavior is a common evasion technique in malware, where a custom loader, often implemented via TLS callbacks, dynamically reconstructs the correct code during runtime.

Resource Section. Among the 625 (84.1%) families where the resource section differs, there is one family (*dostre*) where this component is the only difference. The difference is in the resource values, in particular, there is one specific Bitmap resource whose value keeps changing. We manually reversed the code and discovered that the malicious code's payload (extracted and executed at runtime) was encoded within that image. We also identify four families (*moarider*, *winloadsda*, *axespec*, *sohana*) where their only difference lies in the order of the resource names, three families (*blackshades*, *umbra*, *virfire*) for which the resources are the same, but the padding of the section differs, ten families where the difference is in the Resource Table but not in the resources identifiers, and another (*lolbot*) has the only difference in a single string that changes for all the samples. The observed variations in the samples are likely introduced intentionally to achieve hash-bursting and, subsequently, polymorphism.

Other Sections. This component includes default sections with a predefined purpose (e.g., `.idata`, `.bss`, .rdata). But it is possible to create custom sections as well. This is the case of packers, which usually create

one section for accomodating the unpacked payload. For one family (*ezsoftwareupdater*) the only difference in the whole executable is a single 32-byte string in the .rdata section, containing hexadecimal characters, likely a unique identifier, and in another family (*stormattack*) the difference was also in the .rdata section where a few hexadecimal strings were changing.

Overlay. To measure how much hidden data has been added to samples with an overlay, we define the *excess ratio* as the quotient of the size on disk over the expected size (i.e., without the overlay). The median excess ratio is 1.73x, an additional 73% content over the expected length. However, the mean is 130x, because some samples contain a vast amount of additional data. The extreme case is a sample with an expected size of 2.56KB but an overlay of 454MB.

We removed the overlays from all samples in the dataset and re-computed their SHA256 hash (which we will now refer to as "no-overlay hash"). About one-fourth (18,315/66,160) of the samples share the same no-overlay hash with at least another sample, indicating that the overlay was the only difference between them. In particular, 13 families only contain samples with the same no-overlay hash, and 46 families have at least 75% of their samples with the same no-overlay hash. Interestingly, 156 no-overlay hashes are shared across different families, with one matching samples in 12 families. This hash corresponds to `7zS.sfx`, a template for 7-zip self-extracting archives. These archives are created by combining `7zS.sfx`, a configuration file, and a compressed archive, with the overlay holding the latter two. Another hash matches 6 families and corresponds to `Default.sfx`, used by WinRAR for self-extracting files. These cases highlight the frequent use of self-extracting archives in malware for distributing multiple files, and likely also for obfuscation as analyzing them requires inspecting the overlay's compressed data.

Four families had overlay differences that stem solely from strings. In three cases, the overlay was entirely ASCII text, while the fourth had one meaningless differing string. However, examining the overlay revealed strings resembling CA names. Despite the Certificate Table fields being set to 0 in the Data Directory, treating the overlay as a Certificate Table revealed valid PKCS#7 signatures.

Finally, we analyzed overlay content using DiE. In 43.67% of cases (13,537/31,096), no file type was detected. Among the rest, 6.08% (1,890/31,096) were archives (RAR, ZIP, 7ZIP), 8.87% (2,750/31,096) were PE files, and 18.17% (5,650/31,096) contained ASCII text. Of these, 7.88% (445/5,650) were valid Base* encodings with no recognizable file types when decoded.

Certificate Table. For the 11,257 samples (17.0%) with a Certificate Table, we extracted the Authentihash and its hash algorithm. The Authentihash is computed from the file's content at signing, excluding the `Checksum` and `Certificate Table Data Directory`. We failed to extract a signature for 1,044 samples due to truncation or corruption (e.g., incorrect offset). For the remaining 10,213, a mismatched Authentihash indicated modification or unrelated signatures in 1,845 samples across 211 families. Thus, 8,368 samples (12.6%) had valid signatures when signed, though some may now be invalid (e.g., revoked). Among these, 1,098 samples had an overlay after the Certificate Table.

We computed the Authentihash for all samples using SHA256, resulting in 63,404 unique values, with 492 shared by multiple samples. Samples with identical Authentihash must belong to the same program (family and version), differing only in their Certificate Table and checksum. For instance, all the samples in the *amigo* family share the same Authentihash, checksum, and a chain of 5 certificates, indicating polymorphism in hidden parts of the Certificate Table.

Other Components. Of the remaining six components (DOS header, COFF and Optional Header, DOS Stub, Section Table, Data Directory), all except the DOS Stub have predefined structures in the PE format but can still be manipulated for polymorphism. For example, in *bebloh*, the only difference is the Optional Header's version values, while the code and data are identical. In *lebreat*, standard UPX section names (.upx0, .upx1, .rsrc) are replaced with random strings, though the content remains unchanged. We also analyzed the COFF Header's creation timestamp, which can be faked. It differs in 622 (83.7%) families, but 130 (11.3%) families share the same timestamp in over 75% of comparisons. Interestingly, *obit* has the same timestamp across all samples (Saturday, June 1, 2019 5:56:28 AM), likely fabricated, as the content differs.

5.4.4 File Infectors

File infectors, or viruses, infect benign executables with malicious code, creating samples with a combination of malicious and benign content. Furthermore, file infector families tend to be highly polymorphic since a single sample may infect many executables stored in the compromised host.

We investigated file infectors on our dataset using a combination of static and dynamic analysis. We first used AVClass [112] to obtain tags for all samples in the dataset. Using the tags, we identified 70 *likely-virus* families

where the CLASS:virus tag appeared in more than half of the samples. For each of these families, we randomly selected 5 samples, dynamically executed them in a virtual machine (VM), and identified those that modified executables that already existed in the VM prior to the execution, i.e., the same filepath in the VM pointing to an executable file had different hashes before and after the execution. For those samples, we used *PEdiff* to examine the component differences between the original executable and the modified one produced during the execution.

Using this approach we identified 20 virus families. Of those, 16 are pre-pender viruses where the PE executable contains the malicious code and the infected (benign) executable is in the overlay: *lamer*, *induc*, *neshta*, *shodi*, *sinau*, *sivis*, *soulclose*, *xiaobaminer*, *memery*, *pidgeon*, *detroie*, *gogo*, *lmir*, *stihat*, *xolxo*, *xorer*. For all those 16 families, our analysis identifies the overlay as a component that changes. For two families (*gogo*, *soulclose*) the overlay is the only component that changes, i.e., the malicious executable has no polymorphism itself, but obtains it from the infected executable in the overlay. In the other 14 families, polymorphism is also added to other components. The remaining 4 families are appender viruses. Two of these (*expiro*, *wlksm*) extend one of the sections of the infected executable with the malicious code. The other two families (*triusor*, *wapomi*) add new sections at the end of the infected executable with malicious code. For all these four families, our analysis outputs that no component is similar across the family samples.

5.5 Final Remarks

A Complex Picture. Our study aimed to identify the main causes of polymorphism and assess their prevalence across a large dataset of malware families. Through our experiments, we identified several causes, summarized in the following section. More importantly, we found that a single factor is rarely sufficient to explain the diversity of samples within a family. In fact, only 12.8% of the families (95 out of 743) exhibited polymorphism due to a single cause. For the remaining 87.2%, polymorphism arose from multiple overlapping factors. This is not a failure but a key finding, highlighting that attributing polymorphism solely to repacking is an oversimplification. It also suggests that no holistic solution exists to address the problem. For example, while removing or normalizing certain components may help in comparing samples, any approach addressing only one or a few causes will have limited success in explaining and mitigating the dissimilarities within a family.

Truncation. While truncated files are rarely, if ever, mentioned in malware studies, we observed that 3.6% of the files in the initial datasets are truncated, with 99.3% coming from the Balanced Dataset. Since that dataset was collected from the VirusTotal file feed, a similar ratio of truncated PE executables might affect other studies using the VT feed [126]. Truncated samples are distributed among nearly half of the dataset families, indicating this is not an issue specific to some families but likely a common error that occurs during sample collection. Truncated samples pollute malware feeds and waste resources such as storage and sandbox time if they are queued for execution. Therefore, we suggest filtering out these samples, as we did, to avoid biasing the results.

Overlays. Similar to truncation, the impact and role of overlays are rarely mentioned in malware studies. However, our experiments show that they are extremely prevalent, affecting a stunning 47.0% of all samples in our dataset, being the most prevalent cause of polymorphism that we find. These overlays often contain a considerable amount of data, on average over a hundred times larger than the main executable alone. Despite this, previous works sometimes purposefully excluded overlays when extracting features for static analysis [97]. This is fine if the overlay contains useless data simply added to achieve polymorphism, but our analysis shows that this is not the case: 6.1% of the overlay data are compressed archives, and 8.9% are PE files.

Packing. Packing is a pervasive phenomenon in our dataset: while it is difficult to measure with precision, it might affect between 40% to 90% of our samples. This is not surprising since it is one of the most effective methods to counter static analysis. However, one would expect a significant difference in the components between the families where packing was most prevalent, but the distributions of packed samples and the negligible correlations we found did not confirm this expectation. We also discover that malware authors, in a trivial but effective way, achieve high polymorphism by using many different packers.

Other polymorphism. Beyond packing, our study reveals that malware families introduce polymorphism into a variety of components. Among others, we observe families that modify PE headers to generate polymorphic variants such as *bebloh* that varies the version fields in the Optional Header. We also observe families that reorder resources without modifying them (e.g. *moarider*, *winloadsda*), introduce random bytes in the padding (*blackshades*,

umbra, *virfire*), and introduce hidden data in the certificate table (*amigo*). There are also families whose differences are limited to some specific strings (e.g., *lolbot*). The range of techniques we observe shows that a structural analysis of the PE file format is a powerful tool for analyzing the reasons behind malware family polymorphism.

Dataset limitations. Our analysis is limited by the datasets used. One issue is the quality of family labels. Despite dataset authors' efforts to refine labeling (e.g., identifying aliases and generic tokens), we found some errors such as *privateexeprotector* being considered a family. Also, the MOTIF and Malicia datasets are highly imbalanced, with few families having more than 10 samples. Still, our use of three datasets should help ameliorate selection bias.

Conclusions. Our large-scale analysis of 743 malware families offers a comprehensive understanding of the factors driving polymorphism in malware. The study reveals that in about 90% of cases, polymorphism results from multiple overlapping factors, rather than a single cause. This complexity underscores the inadequacy of simplistic solutions, such as attributing polymorphism solely to repacking, and highlights the need for multifaceted approaches.

Chapter 6

Structural Overlaps and the Precision Boundaries of Malware Clustering

While Chapter 5 focused on investigating the structural diversity that exists *within* malware families, highlighting the many reasons why samples from the same family may differ, this chapter addresses the complementary problem: understanding the structural similarities that emerge *across* different malware families. These inter-family similarities can severely affect the *precision* of malware clustering techniques based on static features, leading to situations where unrelated samples are mistakenly grouped together. Beyond measuring this loss of precision, we also aim to explain *why* these features fail, identifying the structural artifacts and confounding factors most responsible for false similarities.

Given the critical role of clustering in malware analysis pipelines, understanding these precision limitations is essential. Clustering malicious executables into families enables analysts and researchers to group samples originating from the same malicious codebase, allowing for scalable investigation, labeling, and threat tracking. This step is commonly used for tasks such as malware classification [39], labeling support [122], new threat discovery [25], malware triage [55], and lineage analysis [44, 32].

To group malware samples belonging to the same family, clustering approaches may use static features extracted directly from executables, dynamic features extracted from the runtime behavior, or a combination of both. Static features are cheaper to extract and thus are more commonly used in applications that require the analysis of a large numbers of samples. They can also avoid the common mistake of grouping unrelated samples

only because they exhibit a similar runtime behavior, e.g., putting together different downloader families in the same cluster.

Different classes of static features exist. For example, binary code similarity approaches [123] extract features from the executable code. However, these approaches require a correct disassembly of the malware code, which is very challenging in presence of obfuscation [64], and may also require complex analysis of control and data flows. This is why most of the existing approaches focus instead on features that can be easily and efficiently extracted, and thus can be applied to very large numbers of samples. Most popular amongst these are whole-file fuzzy hashes like SSDeep [61] and TLSH [86], which produce similar digests for similar input files. Another highly scalable approach is to compute a hash over a subset of an executable such as the import table (*imphash*) [1], selected fields in the PE headers (*pehash*, *richpe*) [135, 134], or the certificate table [63, 59]. We will focus on these *scalable* features in our study.

Researchers have proposed multiple supervised malware classifiers based on machine learning techniques [138, 40, 128, 136, 34]. However, supervised classifiers can only accurately classify samples of families observed during model training. Their performance significantly degrades when applied to out-of-distribution data where samples from new families may appear. In contrast, malware clustering approaches can handle previously unknown malware families by capturing similarity between samples. An accurate malware clustering approach should optimize both *precision* and *recall*, or the F1-score that combines both. However, a key observation about malware clustering is that, while *recall* is an important metric (i.e., we do not want samples of a given family to be sub-divided in many clusters), *precision* is always paramount (i.e., it is rarely acceptable when samples of different families are erroneously combined within the same cluster). In fact, while a lower recall might result in more manual work for the analysts (e.g., to label the different clusters), a poor precision often leads to wrong results and conclusions.

To date, little is known about the practical limitations that affect the precision of popular static features. In particular, both *when* they fail and, most importantly, *why* they fail are two aspects still poorly understood.

Since precision is instrumental in identifying similarities between different families, while recall is more focused on uncovering diversity within samples from the same family, the primary objective of this chapter is to investigate the former aspect. We examine the limitations of the precision of widely-used, highly scalable, static similarity features. Our emphasis is on understanding the underlying causes of errors that result in the mis-

classification of samples from different malware families that are incorrectly placed in the same cluster.

It is important to emphasize that our aim is not to propose novel malware clustering approaches, nor to critically evaluate or challenge existing ones. On the contrary, we contend that the insights derived from this investigation will offer valuable contributions to the broader field of malware clustering research. Specifically, for each static feature analyzed, we explore the factors that lead to the erroneous grouping of samples from distinct families, thereby reducing the overall precision.

To this end, we leverage three public datasets of malicious Windows executables labeled with their family [34, 53, 83]. We cluster the 66,160 unique samples using separately each of 11 popular static similarity features. We selected these features for their proven ability to effectively cluster malware samples, utilizing established methods from both industry and academic research. These features are commonly available through online malware analysis services, such as VirusTotal [129], and have been used in previous studies for clustering based on static features [37, 86, 69, 63, 59, 60, 15, 85, 26, 139]. Then, we compute the clustering accuracy obtained by each feature individually, with an emphasis on precision. Finally, we examine the *mixed clusters* containing samples from different families according to the ground truth labels. For this examination, we leverage 8 analysis features that capture possible reasons why samples from different families may be grouped such as being generated by different EXE-building tools (e.g., packers, installers, compressors), having overlays, or being truncated. Using this approach, we answer two research questions.

RQ1: What is the precision of the most commonly-adopted static features when used for malware family clustering? Our analysis identifies three groups of features regarding precision. The first group contains three features related to code signing with nearly perfect precision (over 99%). However, their ability to group samples is limited, making them especially suitable for verifying the correctness of ground truth labels during dataset construction. The second group achieves high precision in the range of 94% to 97% and comprises structural hashes such as *pehash* and VirusTotal's proprietary *vhash* and whole-file fuzzy hashes (*SSDeep* and *TLSH*), In this group, *pehash* is particularly effective, offering the highest precision and average cluster size. The third group has features with lower precision, below 90% and includes *imphash*, *richpe*, and icon-based features. These features exhibit frequent collisions in their values introducing significant errors in identifying samples from the same family.

Dataset	Samples	Families	Collection	CV	KL
Balance Dataset [34]	67,000	670	08/2021 - 03/2022	0.0	0.0
MOTIF [53]	3,095	454	01/2016 - 12/2020	1.69	0.71
Malicia [83]	9,908	53	03/2012 - 02/2013	4.56	2.71
Superset	79,993	1,125	03/2012 - 03/2022	2.77	0.72

Table 6.1: Datasets used and imbalance metrics: Coefficient of Variation (CV) and Kullback-Leibler Divergence (KL). Lower means more balanced.

RQ2: What are the main limits of those features for malware family clustering? When should analysts be careful with their use? The main limits of the analyzed similarity features stem from their sensitivity to EXE-building tools, which often introduce similarities unrelated to malware family characteristics. The impact of EXE-building tools is small in the first group of features (high precision), but significant for the other two groups where the largest clusters are mixed, i.e., contain samples from different families. For some features like *SSDeep* and *TLSH*, up to 45% of the mixed clusters may be caused by EXE-building tools. Beyond EXE-building tools, features in the third group (lowest precision) also suffer from inherent weaknesses, such as collisions caused by short import and rich header tables and generic icons. We identify cases where the precision is lowered, not by feature limitations, but due to erroneous ground truth labels.

Finally, we evaluate whether pre-processing techniques such as avoiding to group samples known to be generated by EXE-building tools, with invalid certificate chains, or with short import tables can solve the issues. The precision increase is highest for features in the third group, especially for *imphash*, where precision improves 10.3%. However, even with preprocessing, some limitations persist, such as those related to generic icons. These findings emphasize the importance of careful feature selection and preprocessing to mitigate errors and maximize the precision in malware family clustering.

6.1 Datasets

In their seminal work on the evaluation of malware clustering, Li et al. [67] showed that the approach used to assemble ground truth (GT) data and the balance between different families introduced a considerable bias in the accuracy of previous clustering experiments, often leading to vastly different

results when the same technique was used on different malware datasets. To mitigate these problems, we use three datasets of malware samples as summarized in Table 6.1: the Balanced Dataset [34], MOTIF [53], and Malicia [83]. Detailed descriptions of their collection processes, family coverage, and labeling methodologies are provided in Chapter 3.

VT reports. We collected reports on all samples through the VirusTotal (VT) API in October 2024. All samples were known to VT as dataset authors had submitted the samples. We use the VT reports to extract features for the samples such as the sample hashes (MD5, SHA1, SHA256), scan results with AV engines, and the date the sample was first submitted to VT.

Superset. To enable joint analysis across datasets, we unified the sample identifiers using SHA256 hashes obtained from the VT reports. This yielded 66,160 unique samples across the three datasets, covering 1,125 distinct family names. These samples constitute our *Superset* dataset.

At the sample level, the three datasets are largely disjoint. However, 8 samples appear both in Balance Dataset and MOTIF and 2 samples in both Balance Dataset and Malicia. There are no common samples between MOTIF and Malicia. Of the 8 samples in Balance Dataset and MOTIF, 5 have the same family in both datasets and three have different families. One sample is assigned *disttrack* in Balance Dataset and *shamoon* in MOTIF, which are aliases according to AVClass. Of the 2 samples in Balance Dataset and Malicia, one has the same family in both datasets and the other has different families. Thus, we find 3 samples with incompatible families in two datasets. When analyzing one dataset, we include all dataset samples. When analyzing the Superset dataset, we consider the 66,160 unique samples. One family (*ramnit*) appears in all three datasets and 51 families appear in two datasets. The largest overlap is between MOTIF and Balance Dataset with 46 families in common. Balance Dataset and Malicia share 6 families and MOTIF and Malicia two.

6.2 Features

After a careful review of the static features used in the literature, detailed in Section 2.6, we selected 11 features for clustering the samples (described in Section 6.2.1), 8 features for analyzing the clustering results (described in Section 6.2.2), and the AVClass [111] labeling results as a baseline for comparison.

Feature	Type	Source	Samples	Values	Clustering
authentihash	cryptohash	PE	79,993 (100.0%)	77,222	FVG
cert_subject	string	VT	8,906 (11.13%)	1,610	FVG
cert_thumbprint	cryptohash	VT	8,906 (11.13%)	2,004	FVG
icon_dhash	fuzzyhash	VT	45,820 (57.28%)	9,138	FVG
icon_hash	cryptohash	VT	45,820 (57.28%)	15,166	FVG
imphash	cryptohash	PE	77,245 (96.56%)	20,023	FVG
pehash	cryptohash	PE	79,993 (100.0%)	31,367	FVG
richpe	cryptohash	PE	49,815 (62.27%)	11,651	FVG
tlsh	fuzzyhash	PE	79,993 (100.0%)	79,236	HAC-T
ssdeep	fuzzyhash	PE	79,993 (100.0%)	75,535	Single linkage
vhash	structhash	VT	79,823 (99.79%)	27,446	FVG
avclass	string	VT	78,067 (97.59%)	1,270	FVG
die_packer	string list	PE	16,520 (20.65%)	190	-
die_installer	string	PE	3,359 (4.20%)	24	-
die_archive	string list	PE	5,015 (6.27%)	15	-
die_script	string list	PE	2,227 (2.78%)	7	-
packgenome	string list	PE	25,961 (32.45%)	100	-
truncated	Boolean	PE	2,520 (3.15%)	2	-
overlay_sha256	cryptohash	PE	35,461 (36.27%)	30,690	-
no_overlay_sha256	cryptohash	PE	79,993 (100.0%)	61,366	-

Table 6.2: Features used and presence in Superset dataset. On top the 12 similarity features used for clustering samples, below the 8 analysis features used to analyze clustering results.

6.2.1 Similarity Features

The top part of Table 6.2 summarizes the 11 similarity features used to cluster samples in the datasets. The list includes six crypto hashes, calculated over different parts of a PE file: *imphash* covers the import table [1]; *pehash* covers selected fields in the PE headers [135]; *richpe* covers the optional Rich Header that contains information about the compilation of modules in the PE executable [134]; *cert_thumbprint* covers the leaf certificate for signed samples; *authentihash* covers the full PE executable excluding code signing fields (e.g., checksum and Certificate Table) and overlays; and *icon_hash* covers the optional icon image. All cryptographic hashes are compared using equality, i.e., two files have the same hash or not.

The list also includes three fuzzy hashes, which produce similar values for inputs that share some similarities. We use two fuzzy hashes computed over the whole file: *tlsh* and *ssdeep*. *tlsh* uses Hamming distance on the digests to

determine if two inputs are similar while *ssdeep* uses edit distance. We also include *dhash*, a perceptual hash applied over the embedded icon image. Perceptual hashes are a special type of fuzzy hashes that produce similar digests for images that look similar to a human (e.g., resized, color changes). We use equality to compare *dhash* digests, thus grouping images only if they are nearly identical. It is possible to use Hamming distance with a threshold (e.g., 2 bits) on the *dhash* digest to identify more dissimilar icons, but that reduces the precision.

Another type of hash included is *vhash*, VT's proprietary structural hash. According to VT's minimal documentation on it [131], this hash takes into account properties such as imports, exports, sections, and file size. VT provides daily results of files clustered by *vhash*. Samples in the same cluster always have the same *vhash* value, so we use equality to compare two *vhashes*.

Finally, our list of features includes the leaf certificate subject distinguished name (*cert_subject*), which is useful for identifying signed samples that use different certificates for the same entity.

Most features can be extracted directly from the PE executables, except VT's proprietary *vhash*. However, as shown in the *Source* column in Table 6.2, we also extract the certificate and icon features from the VT reports. The reason for this is that different ways exist for extracting these pieces of information from an executable, and it is therefore more convenient for users to leverage the VT reports to avoid implementing their own method. For example, signed samples contain an unordered sequence of certificates and identifying the leaf certificate requires carefully ordering them. Similarly, an executable may include multiple resources of type icon, and VT selects among those the one it considers the main icon.

As shown in Table 6.2, some features are optional and may not exist in all samples. For example, only 11% of samples are signed, 57% have icons, 62% have a Rich Header, and 96% have an import table.

AVClass. We also consider the malware family obtained by feeding the VT reports to the AVClass malware labeling tool [111, 112]. While the 11 similarity features can be used for clustering similar samples, they do not provide a family name to the clusters. In contrast, AVClass is mainly a labeling tool, that aims to assign a family name to each input sample. However, its output can also be used for clustering, by grouping together samples assigned the same family name. In contrast to the similarity features, we do not know exactly how the AVClass family was generated since it is obtained from a majority voting between the labels assigned by different

AV engines, each using their own proprietary techniques (e.g., signatures, machine learning). We use AVClass families as a baseline to compare the clusters obtained by individual similarity features.

6.2.2 Analysis Features

The bottom part of Table 6.2 summarizes the 8 analysis features that we use to examine the clustering results. These features capture potential reasons for two different programs to share some binary similarity. We will study how these analysis features correlate with the clusters created using the similarity features.

EXE building tools. Beyond compilers and linkers, there exist other off-the-shelf tools that malware authors can use to build their malicious executables. We consider four classes of such tools: software protectors, installers, self-extracting archives, and tools that embed scripts and their interpreter in an executable. Protectors are used by malware authors for evasive purposes [121, 76]. For instance, protectors may compress or encrypt the code and data of the original program and uncompress or decrypt them at runtime. They may also inspect the runtime environment to identify the presence of analysis tools or security products, refraining from running the malicious behavior in case any are detected. For simplicity, in this work, we refer to software protectors as *packers*. Popular packers include UPX, PECompact, ASPack, and Themida. *Installers* are responsible for performing all installation steps for a target program. They are used when a program requires multiple files beyond the main executable or when the installation requires actions such as creating folders or setting up registry keys. Popular installer builders include InnoSetup, NSIS, InstallShield, WIX, and InstallMate. *Self-extracting (SFX) archives* embed a compressed archive and the decompression routine required to automatically extract the archive when the executable is run. Most popular compressors (e.g., 7-zip, WinRAR, and WinZip) can build SFX archives. Finally, executables can be generated from scripts by embedding the script's code with its corresponding interpreter. Popular tools in this category include aut2exe for AutoIt and bat2exe for BAT.

To identify samples built using the above classes of tools, we leverage the signatures provided by Detect-it-Easy (DiE) [5]. To identify packed samples we also leverage YARA rules for 20 packers built with PackGenome [68]. Other packing detection tools exist (e.g., PEiD [3]), but DiE and PackGenome are the most up-to-date (e.g., PEiD [3] has not been updated in 7 years).

According to DiE, one-third of the samples in the Superset dataset have been produced by EXE-generating tools: 16,520 (20.6%) samples packed with well-known packers, 5,015 (6.2%) SFX archives, 3,359 (4.2%) installers, and 2,227 (2.7%) samples generated from scripts. PackGenome detects 32.4% packed samples.

Overlay. A PE executable has an overlay if it contains additional data appended to its end, whose presence is not revealed by the PE header fields. Overlays can be detected because the size of the file on disk is larger than its expected size (i.e., the sum of the start offset and size of the last section). A stunning 36.2% (35,461) of our samples include an overlay. For each of them, we computed the hash of the overlay's content (*overlay_sha256*) and the hash of the executable without the overlay (*no_overlay_sha256*).

Truncation. Samples may get truncated as part of the malware collection process. From a security analysis perspective, truncated samples can be considered corrupted and cannot be executed. Truncated samples can be identified because their expected size (i.e., the sum of the start offset and size of the last section) is larger than the real size of the file on disk. We identify 2,521 (3.1%) truncated samples: 2,486 (3.7%) in Balance Dataset, 24 (0.8%) in MOTIF, and 11 (0.1%) in Malicia. Although it is especially prevalent in Balance Dataset (and thus in the VT file feed), this confirms that truncation is a widespread phenomenon that affects all datasets. Truncated samples should arguably not be included in malware datasets, as they do not correspond to fully functional programs.

6.3 Analysis Approach

This section summarizes our approach for analyzing the limits of the similarity features. It comprises three steps: clustering the samples, computing the clustering accuracy, and analyzing the clusters with samples from multiple families.

Clustering. We perform a separate clustering of the samples in each dataset by using each similarity feature in isolation. Depending on the feature, we employ two different clustering approaches. For 10 of the similarity features, we perform a simple feature value grouping (FVG) which places samples with the same feature value in the same cluster, i.e., one cluster per feature value. This approach is used for cryptographic hashes, structural hashes, strings, and boolean features. We also use FVG for the

icon_dhash fuzzy hash to only group samples with nearly identical icons. In the event that a sample lacks a particular feature, the FVG clustering places the sample in a singleton cluster on its own.

For the other two fuzzy hashes, we employ an agglomerative clustering approach. For *tlsh*, we use HAC-T, a hierarchical agglomerative clustering technique specially designed for TLSH digests [85]. The main advantage of HAC-T is that it is more efficient than traditional hierarchical clustering, which has a quadratic complexity. For our experiments, we use the suggested clustering hyper-parameter value, i.e., $C_{dist} = 30$ [85]. For *ssdeep*, we rely instead on the built-in single-linkage clustering provided by its official implementation [117]. In this case, clusters are formed based on the presence of at least one pairwise connection above a threshold t. We choose a conservative threshold $t = 70$ since this value appears to provide the best balance between sensitivity and specificity for malware classification [80, 81].

Clustering accuracy. We compute the clustering accuracy[1] using an external clustering validation approach that compares the clustering results with the reference clustering provided by the GT. The external validation evaluates if the two sets of clusters group samples in a similar way. It is important to note that only the structure of the clusters is compared, i.e., the family names in the reference clusters are not used in the evaluation. For each experiment, we report precision, recall, and F1 score, as computed in prior malware clustering works [25, 103, 83, 93]. A clustering has perfect precision if every cluster is *pure*, i.e., contains only samples from a single family. In other words, there are no *mixed clusters* (MCs) containing samples from multiple families. A high recall means that most samples of each family belong to a single cluster, while a low recall indicates that a family is fragmented over multiple clusters.

We examine how often each similarity feature makes errors that reduce its precision, i.e., link samples in different families. We focus on precision because features with high precision can be combined into more sophisticated clustering approaches with minimal errors. Also because when clusters have perfect precision, i.e., contain only samples of one family, then an analyst can simply label one of the samples in the cluster and propagate the label to other samples. While we also provide recall and F1 score metrics, it is important to stress that our goal is not to examine *how good* each individual feature is at clustering samples, as real-world malware clustering approaches

[1]Henceforth, we use the term *accuracy* as an overarching designation for measures of predictive performance, rather than the *Accuracy* metric itself.

would most likely combine multiple features to achieve better results. We focus instead on analyzing the limits of each feature and the reasons they might produce mixed clusters.

Mixed cluster analysis. None of the features has perfect precision, i.e., they all incorrectly group together some samples belonging to different families. To identify the causes behind the errors, we analyze the mixed clusters (MCs), which contain samples from different families and are responsible for lowering the overall clustering precision. For each MC, we compute a distribution of each of the 8 analysis features. For example, we count how many samples are packed with each specific packer identified by DiE or PackGenome, how many samples use each specific installer software, and how many samples are truncated. If any of the analysis features affects all samples in the cluster, then we conclude that the analysis feature offers a *possible explanation* for the MC. For example, if a cluster has 100 samples and DiE identifies that all 100 samples are generated by the InnoSetup installer, we consider that samples in the cluster may have been grouped together (despite belonging to different families) due to being generated using that installer software.

It is worth noting that this approach captures correlation rather than causality. However, while we cannot claim that a given analysis feature necessarily is the cause behind the errors, our results often point to a possible causal link. For instance, the use of a given installer tool does in fact introduce common parts in the resulting program binaries, and this similarity is picked up by some of the features and thus results in erroneously combining otherwise different samples in the same cluster. In any case, in order to gain further insight into the underlying causes of the errors introduced by each feature, we complement the correlation analysis with a manual investigation of the clusters.

6.4 Analysis

This section first presents the clustering results in Section 6.4.1 and then analyzes the limits of the similarity features in Section 6.4.2.

	Clusters							
Feature	**All**	**Singl.**	**> 1**	**Max**	**Avg**	**Prec.**	**Recall**	**F1**
authentihash	77,221	76,721	500	185	6.5	0.997	0.026	0.050
cert_subject	72,696	72,003	693	160	11.5	0.991	0.050	0.096
cert_thumbprint	73,090	72,232	858	160	9.0	0.994	0.040	0.077
icon_dhash	43,310	40,618	2,692	1,651	14.6	0.856	0.203	0.328
icon_hash	49,338	46,281	3,057	1,521	11.0	0.888	0.186	0.307
imphash	22,771	17,133	5,638	547	11.1	0.873	0.321	0.470
pehash	31,367	25,342	6,025	345	9.1	0.963	0.230	0.372
richpe	41,524	37,847	3,677	715	11.5	0.887	0.211	0.340
ssdeep	41,367	34,866	6,501	310	6.9	0.967	0.172	0.292
tlsh	40,025	33,412	6,613	342	7.0	0.959	0.174	0.295
vhash	27,615	19,726	7,889	715	7.6	0.939	0.183	0.306
avclass	3,195	2,241	954	2,595	81.5	0.909	0.872	0.890

Table 6.3: Clustering results on Superset dataset. For each similarity feature, it shows the number clusters, singleton clusters, non-singleton clusters, largest cluster size, average cluster size excluding singletons, and the accuracy metrics.

6.4.1 RQ1 – Similarity Feature Precision

Table 6.3 summarizes the clustering accuracy of each feature over the Superset dataset. While all individual features provide reasonably high precision, we can clearly identify three groups: three features (*authentihash*, *cert_thumbprint*, *cert_subject*) have a precision above 99%, four features (*pehash*, *ssdeep*, *tlsh*, *vhash*) have a precision between 94% and 97%, and four features (*imphash*, *richpe*, *icon_hash*, *icon_dhash*) have a precision below 90%.

On the other hand, individual features fail to group many samples together: the largest cluster contains 1,651 samples (*icon_dhash*) but the average cluster size (excluding singletons) is always lower than 14.6 samples (icon_dhash). The trend of groups we observed for the precision is reverted, e.g., the group of features with higher precision has the lowest recall. For instance, the feature that is most effective at creating large clusters (*icon_dhash*) is also the one with the lowest precision, i.e., the one that makes more errors.

The comparison with the AVClass labeling tool used as baseline shows that individual similarity features have precision in the same range as a popular malware labeling tool, but the recall (and thus the F1 score) is

much lower. This makes sense as malware clustering approaches typically combine multiple features to increase the overall accuracy [22, 25, 92].

Feature	Mixed	None	Packer	Archive	Script	Inst.	No-Over.	Trunc.
authentihash	7	2	2	1	-	2	-	-
cert_subject	141	113	12	3	1	12	-	-
cert_thumbprint	122	91	13	4	1	13	-	-
icon_dhash	890	764	82	17	11	13	3	-
icon_hash	725	609	75	15	10	12	4	-
imphash	790	434	234	44	35	22	19	2
pehash	418	172	92	36	38	35	38	7
richpe	656	492	93	39	5	16	9	2
ssdeep	443	240	97	41	27	12	23	3
tlsh	504	267	117	40	28	24	22	6
vhash	1,191	387	427	110	88	87	45	47
avclass	361	346	3	1	4	7	-	-
All	6,248	3,917	1,247	351	248	255	163	67

Table 6.4: Number of mixed clusters (MCs) per feature in Superset and possible reasons.

6.4.2 RQ2 – Similarity Feature Limits

Table 6.4 reports the number of MCs obtained with each *similarity feature*, along with the number of such clusters in which all samples share the same *analysis feature*. For instance, a value of 2 in the *Packer* column means that two of the MCs contained all their samples packed with the same packer. Next, we examine the reasons behind the MCs for each feature in order to identify their limits for family clustering.

Authentihash. This feature exhibits the highest precision (0.997) because samples with the same *authentihash* are nearly identical and can only differ in the checksum field, certificate table, and overlay (if these parts exist). On the other hand, it also groups the least number of samples with over 99% of the clusters containing only one sample, and the remaining containing 6.5 samples on average. There are only 7 *authentihash* MCs, all from the Balance Dataset dataset. None of the samples in the MCs include an overlay. Looking at the certificate, two clusters have all samples signed using the same leaf certificate, three have a mixture of signed and unsigned samples but all signed samples use the same leaf certificate, and two have two unsigned samples each. Since the only differences are in the signature-related fields, samples in these MCs have the same code and data, and thus

they necessarily belong to the same family. Thus, the reason behind the MCs are instead errors in the GT. In particular, we identify that 4 MCs are due to different Balance Dataset families being in reality aliases. In particular, we identify two alias groups (*ascentive*, *speedcat*, *gamini*, *deceptpcclean*) and (*winwrapper*, *spigot*). We approached the AVClass team about these two groups and they concluded they are indeed aliases that should be incorporated into AVClass. The other MCs are due to three samples being mislabeled in the Balance Dataset GT.

Thus, we conclude that while *authentihash* does not group much, it has nearly perfect precision and therefore we suggest researchers to use it while constructing malware datasets to identify GT errors.

Certificates. The certificate features boast the second and third highest precision with 0.994 for *cert_thumbprint* and 0.991 for *cert_subject*. The latter groups more as it can identify samples with different certificates for the same entity, but at the expense of lower precision. The number of MCs (122–141) is the lowest behind *authentihash*. The most common likely explanations are *packers* and *installers*, which cover 10% of MCs each. Installers are a much more common explanation than other features likely because installers are prevalent among PUPs, and a larger fraction of PUP is signed compared to malware [63]. This is interesting as it shows a clear correlation without causality, as the installer per se does not affect the certificates. Of the 10 largest clusters for *cert_thumbprint*, 6 are pure and the other 4 are MCs due to GT errors already described, thus no real collisions are observed between families in the top 10 clusters. However, we identify a few smaller MCs that are due to invalid certificate chains, including leaf certificates of benign entities such as "Mozilla Corporation", "Corel Corporation", and "Opera Software AS". To avoid such errors we can change the certificate features to only be extracted for properly signed samples. This change increases the precision of both features to 0.996.

We conclude that certificate features have high precision and are rarely affected by EXE-building tools. However, to handle the misuse of benign certificates by different families, we suggest certificate features to be extracted only for properly signed samples.

Whole-file fuzzy hashes. Both SSDeep and TLSH capture whole-file binary-level similarity and achieve similar results, so we discuss them together. Both features achieve high precision (0.967 for SSDeep and 0.959 for TLSH) and limited grouping (6.9 and 7.0 average samples per cluster). SSDeep shows slightly better precision, generating 443 MCs compared

Table 6.5: Of the 10 largest Superset clusters using SSDeep, 9 are MCs and 6 are likely due to EXE-generating tools.

#	Samples	MC	Families	Likely Reason
1	310	✓	5	installer:inno
2	185	✓	4	GT errors
3	173	✓	19	script:aut2exe
4	167	✓	2	archive:sfx
5	162		1	-
6	157	✓	2	no_overlay_sha256
7	157	✓	2	packer:dxpack
8	149	✓	8	packer:upx
9	132	✓	3	GT errors
10	125	✓	3	archive:sfx

to 504 for TLSH. Our analysis features provide a possible explanation for 46%–47% of these MCs, with the most common causes being packing (22%–23%), SFX archives (8%–9%), script-generating tools (5%–6%), overlays (4%–5%), installers (3%–5%), and file truncation (0.7%–1%). Table 6.5 shows the 10 largest SSDeep clusters. Of those, 9 are MCs and 6 are likely caused by EXE-generating tools such as SFX archives (2 MCs), the InnoSetup installer (1), the aut2exe script-building tool (1), and the UPX (1) and dxpack (1) packers. For example, the MC at rank 3 comprises 173 samples from 19 families written in AutoIt. The top families in this MC are *autinject* (70 samples), *autoitinject* (37), and *aitinject* (21), but the cluster also contains families that may wrap samples in AutoIt scripts such as *remcos* [133]. Another two MCs are due to the GT errors already described in the *authentihash* paragraph. The other MC contains 157 samples from two families (*spesr*, *vbinder*) all with an overlay and the exact same content when ignoring the overlay (*no_overlay_sha256*). This behavior is characteristic of prepender viruses that store benign infected program in the file overlay [49]. Interestingly, there is an almost one-to-one mapping with the 10 largest TLSH clusters, the exceptions being the top SSDeep being broken in two with TLSH and the top TLSH cluster broken in three by SSDeep. This suggests that both fuzzy hashes perform similar mistakes and that those mistakes are likely caused by the same underlying reasons.

We conclude that up to 45% of the clusters generated by both SSDeep and TLSH, and most of the largest ones, only capture similarity introduced

by EXE-building tools, rather than the similarity of the malicious code and data.

PEhash. This feature achieves high precision (0.963) and less MCs (418) than the whole-file fuzzy hashes. In addition, the use of *PEhash* also tends to group more samples, with an average cluster size of 9.1. The distribution of possible reasons reported in Table 6.4 is similar to the one we discussed for fuzzy hashes: packers (19%), script-building tools (9%), SFX archives (9%), overlays (9%), installers (8%), and truncation (1.6%). Out of its 10 largest clusters, 6 are pure and 4 are MCs. The MCs correspond to ranks 1, 2, 4, and 10 in Table 6.5, two of which are due to SFX archives, one to InnoSetup, and the other to GT errors.

We conclude that *pehash* is slightly better than fuzzy hashes for grouping samples into families, but is similarly affected by EXE-building tools.

Vhash. The proprietary *vhash* has medium precision (0.939), lower than *pehash*, *ssdeep*, and *tlsh*. It groups more samples than *ssdeep* and *tlsh*, but less than *pehash*. The use of this feature results in a stunning 1,191 MC clusters, which correspond to 15% of its non-singleton clusters. This is twice the ratio of other features: *ssdeep* (6.8%), *pehash* (6.9%), and *tlsh* (7.6%). The possible reasons for MCs are dominated by packers (36%). This feature also seems more affected by truncation (4%) than others. Of the largest 10 clusters, 6 are pure and 4 MCs. The 4 MCs correspond to rank 1 (split into two MCs), 2, and 6 in Table 6.5.

We conclude that *vhash* is worse than *pehash* for grouping PE executables into families, achieving less precision and grouping less. It seems to generate a higher number of collisions too, although its proprietary design makes it difficult to assess the exact reasons.

Imphash. This feature has the lowest precision (0.873) and second highest number of MCs (790). On the other hand, it has the highest recall (0.321) and an average cluster size of 11.1 samples. Compared to other features, this feature is more affected by packers (30%). This makes sense as packers often hide the import table of the original code, replacing it with a potentially smaller table. This is reflected in the 10 largest clusters, which are all MCs. Five of them are likely due to known EXE-building tools: InnoSetup (2), aut2exe (1), and the UPX (1) and Themida (1) packers. The import table for the UPX cluster has 21 imports and the one from Themida only two. Another four clusters have very small import tables with 1–4 imports and contain detections for multiple packers. Thus, different packers

Figure 6.1: Icons responsible for the 9 largest MCs using *icon_hash*.

may generate the same small import table. The final cluster is one of the previously mentioned with GT errors.

We conclude that *imphash* is not very good at grouping samples into families. Not only it is affected by EXE-building tools, but it also groups samples built using different packers that produce the same small import tables. And since packers are very common among malware authors, this is a very severe limitation. Removing samples with very small import tables can ameliorate the impact of packers, but it would remove a large number of samples and would not affect MCs caused by other tools, e.g., the mentioned aut2exe MC has 522 imports, and the InnoSetup MCs 99–137.

Richpe. This feature has the third lowest precision (0.887). Only 25% of its 656 MCs have a possible explanation with packers (14%) being the most common explanation. Out of the 10 largest clusters, all are MCs and all have a small number of entries (1–13) in the RichPE header. One MC is likely caused by the NSIS installer. For the rest, there is no likely cause and we observe a mixture of packers and other tools. This likely indicates collisions where different EXE-building tools happen to have used the same compilers to build their modules. It is interesting to note that in this case collisions are also observed with samples where no EXE-building tool is detected.

Similar to *imphash*, we conclude that the *richpe* hash is not very good at grouping malware samples into families, suffering from collisions between different EXE-building tools and other malicious samples.

Icons. *icon_dhash* has lower precision (0.856) and higher recall (0.203) than *icon_hash* (0.888 and 0.186, respectively), as it groups samples that have visually similar, but not identical, icons, incorrectly grouping unrelated samples. A small ratio of MCs has a likely explanation: 16% for *icon_hash* and 14% for *icon_dhash*. Of the 10 largest clusters for *icon_hash*, all are MCs, although one seems to identify *zbot* (352 samples) with one incorrectly labeled *virut* sample. The other nine are due to the icons in Figure 6.1. These are common icons, not specific to a single family. Thus, their MCs contain samples from many (25–100) families identified as using different

EXE-building tools and no tools at all.

We conclude that *icon_hash* works better than *icon_dhash* for grouping samples into families, but both suffer from common icons that are not specific to a family and are responsible for 9 out of 10 of the largest clusters.

AVClass. AVClass has the lowest ratio (4%) of MCs with a possible explanation. This makes sense as AVClass produces much larger clusters (81.5 samples on average) due to its ability to link samples of the same family considered dissimilar by individual features. Thus, the analysis features should rarely be able to explain an AVClass MC. Most likely, the possible explanations for those 4% MCs capture correlation rather than causality.

6.5 Discussion

This section discusses the impact of our results on malware family clustering and potential avenues for improvements.

Feature effectiveness. The similarity features analyzed can broadly be divided into three groups. First are the *authentihash* and the leaf certificate features, which have nearly perfect precision (over 99%) but limited grouping ability. We suggest that these features be used during dataset construction to spot potential errors in the GT labels. Next come structural hashes (*pehash*, *vhash*) and whole-file fuzzy hashes (*ssdeep*, *ssdeep*) with precisions 94%–97%. These features are impacted by EXE-building tools, which may cause them to capture similarity not due to family characteristics but rather to the EXE-building technology itself, thus causing samples from unrelated malware families to be placed together in MCs. For instance, for SSDeep and TLSH up to 45% of MCs were likely caused by EXE-building tools. In this group, *pehash* seems the best choice for grouping samples into families as it has the highest precision and average cluster size. Finally, the remaining 4 features (*imphash*, *richpe*, both icon features) have precision below 90%. They are affected by EXE-building tools, but also show collisions in values due to other factors (e.g., short tables or generic icons). These features can introduce significant errors in family clustering.

Feature pre-processing. A potential fix to address the limits of the similarity features would be to avoid using them in cases known to be problematic. Table 6.6 measures the improvement in feature precision if we place in singleton clusters samples identified as being produced by EXE-building tools. This is a generic fix that can be applied to all similarity features.

Table 6.6: Clustering precision on the Superset dataset before (Original) and after (NoBuilt) placing the 35,766 samples identified as being built with EXE-generating tools in singleton clusters.

Feature	Original	NoBuilt	Delta
authentihash	0.997	1.000	0.003
cert_subject	0.991	0.998	0.007
cert_thumbprint	0.994	0.999	0.005
icon_dhash	0.856	0.949	0.093
icon_hash	0.888	0.963	0.075
imphash	0.873	0.976	0.103
pehash	0.963	0.988	0.025
richpe	0.887	0.959	0.072
ssdeep	0.967	0.991	0.024
tlsh	0.959	0.987	0.028
vhash	0.939	0.982	0.043
avclass	0.909	0.942	0.033

The top three features by precision (*authentihash* and the certificate features) show very limited improvement (<1%) since they are originally little affected by EXE-building tools. In contrast, *imphash* shows a precision improvement of 10.3% The icon features and *richpe* still maintain lower precision (94.9%–96.3%) indicating that collisions unrelated to EXE-building tools still happen often.

Another option is applying specific pre-processing to selected features. For example, we showed that if we avoid extracting leaf certificate features from samples with invalid certificate chains, the precision of the certificate features increased by 2% for *cert_thumbprint* and 4% for *cert_subject*. Similarly, if we place in singletons samples with an import table with less than 10 entries, the *imphash* precision improves by 3.4% from 0.873 to 0.907. However, this is lower than the precision obtained by especially handling samples produced with EXE-building tools (0.976) showing that collisions also happen on larger import tables. Another potential fix is ignoring generic icons in both icon features. However, identifying generic icons is a challenge in itself.

Ground truth errors. We identified several cases where MCs are likely due to GT errors, rather than to limits of the similarity features. Most cases are aliases identifying the same family, which can happen within a dataset and across datasets. For example, in the *authentihash* analysis we observed two groups of aliases within Balance Dataset and we also spotted likely aliases in MOTIF, e.g., *ligsetrac* and *skimer*. But, we also observe aliases across datasets such as MOTIF using *kronos* where Balance Dataset uses *kronosbot*. We have been reporting such cases to the AVClass developers and hope that the identified aliases will be incorporated into that tool. We also spot cases where individual samples seem incorrectly labeled. However, verifying these errors is harder and may require significant manual analysis.

6.6 Conclusion

The findings of this study have significant implications for malware clustering methodologies. Future work should focus on measuring the presence of EXE building tools in their datasets, to ensure the generalizability of findings. Additionally, the systematic refinement of preprocessing techniques, such as the exclusion of known problematic elements (e.g., short import tables, common icons), holds promise for improving feature robustness.

Chapter 7

Conclusion

This thesis presented a multi-faceted investigation into malware diversity and its implications for malware classification and similarity analysis. Each of the three studies included in this work focused on a distinct, yet complementary, aspect of the problem: machine learning-based classification, intra-family polymorphism, and inter-family structural similarity.

7.1 Summary of Findings

The first study focused on understanding the key factors that influence the performance of machine learning models for malware detection and family classification. The experimental results revealed several critical insights. First, the study provided a quantitative assessment of how dataset characteristics, such as the number of families and their distribution, affect classification performance. The findings showed that models trained on small, imbalanced datasets often report inflated accuracy scores and generalize poorly to different family distributions. Second, the study highlighted that static features dominate detection and classification for samples belonging to known families, largely unaffected by common techniques such as packing. Dynamic features, while more costly and error-prone to extract, demonstrated better generalization to unknown families and improved binary classification tasks. However, their overall impact on family classification for known families was marginal. Finally, the experiments demonstrated that all machine learning models experienced significant performance drops when tested on out-of-distribution samples. The data-driven nature of these models makes their performance heavily dependent on the quality and coverage of the training data. Nevertheless, incorporating dynamic features helped alleviate some of the performance degradation when classifying previously

unseen malware, especially for binary classification tasks, although this improvement came with higher false positive rates.

The second study examined the causes and prevalence of polymorphism within malware families. The analysis revealed that polymorphism is rarely the result of a single factor. In fact, for nearly 90% of the families studied, multiple overlapping causes contributed to intra-family diversity. This finding challenges the common assumption that polymorphism can be explained solely by repacking. Among the observed causes, truncation affected 3.6% of the samples, with truncated files distributed across nearly half of the families. This suggests that truncation is a widespread artifact in datasets collected from large-scale malware feeds and highlights the importance of filtering such samples to avoid biasing results. Overlays emerged as the most prevalent cause of polymorphism, affecting 47.0% of all samples. Contrary to assumptions in prior research, many overlays contained meaningful data, including compressed archives and even entire PE files. Packing remained widespread in the dataset, though its correlation with intra-family component differences was lower than expected. Malware authors often used multiple different packers to increase diversity within the same family. Additionally, several other forms of polymorphism were observed, including modifications to PE headers, resource reordering, random padding insertion, and hidden data in certificate tables. To enable this fine-grained analysis, we developed a dedicated structural comparison tool for PE files, which allowed us to locate and quantify differences across specific binary components. These findings underscore the complexity of structural polymorphism in malware and the limitations of simplistic, single-cause explanations.

The third study investigated inter-family structural similarity and its impact on malware clustering methodologies. The findings demonstrated that shared artifacts introduced by common build environments and EXE-building tools could lead to false similarities between unrelated samples. This structural overlap risks distorting clustering results and undermining the reliability of feature-based analysis. The study also highlighted the importance of measuring the presence of such build artifacts within datasets to ensure the generalizability of future research findings.

In conclusion, this thesis exposes both the promise and the limits of current learning-based malware analysis. Moreover, a deep dive into polymorphism further dispelled the myth of any single root cause, revealing many PE manipulations that together drive intra-family diversity. Finally, we demonstrated how ubiquitous build-environment artifacts can manufacture illusory inter-family links, threatening the integrity of clustering re-

sults. Collectively, these findings argue for richer, better-curated datasets, multi-modal feature pipelines that balance static speed with dynamic robustness, and rigorous controls for structural noise. Only by embracing this multifaceted perspective can future research and defenses hope to achieve reliable, real-world malware detection and classification.

7.2 Future Work

The findings presented in this thesis open several directions for future research. One important aspect is exploring how to mitigate the impact of missing features in dynamic analysis, for example through feature selection techniques aimed at identifying the most informative and consistently available features across samples. Reducing the dependency on features that are often incomplete or unavailable could help improve classification robustness, especially in real-world deployment scenarios.

Another promising line of work involves investigating the reasons behind the poor detectability of certain malware families. The results observed in this thesis suggest that these detection challenges may arise from a combination of factors, including the use of custom packing strategies, the inclusion of benign-like functionality that complicates behavioral detection, or overly generic labeling practices that blur family boundaries. Conducting targeted studies on these hard-to-detect families could help clarify the underlying causes and inform the development of more specialized or adaptive detection approaches.

A deeper understanding of build environment artifacts and EXE-building tools also emerges as an important objective. Measuring their presence within malware datasets could help ensure that future analysis findings generalize beyond the specific build environments observed during training and evaluation. This would also support more accurate interpretation of similarity measurements by reducing the risk of feature-driven false relationships between unrelated samples.

Finally, the systematic refinement of preprocessing techniques, such as excluding known problematic elements like short import tables or common icons, holds promise for improving the robustness of static similarity features. Such refinements could help reduce feature noise and improve the reliability of similarity-based clustering results.

Appendices

Table 1: Summary of the features extracted from PE headers

Feature name	Header	Description
ImageBase	Optional	The address of the memory mapped location of the file
AddressOfEntryPoint	Optional	The address where the loader will begin execution
SizeOfImage	Optional	The size (in bytes) of the image in memory
SizeOfCode	Optional	The size of the code section
BaseOfCode	Optional	The address of the first byte of the entry point section
SizeOfInitializedData	Optional	The size of the initialized data section/s
SizeOfUninitializedData	Optional	The size of the uninitialized data section/s
BaseOfData	Optional	The address of the first byte of the data section
SizeOfHeaders	Optional	The combined size of the MS-DOS stub, PE headers, and section headers
SectionAlignment	Optional	The alignment of sections loaded in memory
FileAlignment	Optional	The alignment of the raw data of sections
NumberOfSections	COFF	The number of sections
SizeOfOptionalHeader	COFF	The size of the optional header
Characteristics bit	COFF	16 Boolean values - one for each bit of the Characteristics bit [31]

Table 2: Summary of the features extracted from PE sections. *Processed resources* and *Processed sections* reflect max, mean, and min values computed among all the resources and sections in the PE file. The last block of features are computed for each i^{th} section and for the section containing the binary entry point, and then only features that show variability and are present in more than 1% of the samples retained.

Feature name	**Description**
Processed_resources_nb	Resource number in the PE
Processed_resourcesMaxEntropy	Max Shannon entropy among resources
Processed_resourcesMaxSize	Max size among resources
Processed_resourcesMeanEntropy	Mean Shannon entropy among resources
Processed_resourcesMeanSize	Mean size among resources
Processed_resourcesMinEntropy	Min Shannon entropy among resources
Processed_resourcesMinSize	Min size among resources
Processed_sectionsMaxEntropy	Max Shannon entropy among sections
Processed_sectionsMaxSize	Max size among sections
Processed_sectionsMaxVirtualSize	Max virtual size among sections
Processed_sectionsMeanEntropy	Mean Shannon entropy of all the sections
Processed_sectionsMeanSize	Mean size of all the sections
Processed_sectionsMeanVirtualSize	Mean virtual size of all the sections
Processed_sectionsMinEntropy	Min Shannon entropy of all the sections
Processed_sectionsMinSize	Min size of all the sections
Processed_sectionsMinVirtualSize	Min virtual size of all the sections
Section_i_exists	True if the i^{th} section exists in the binary
Section_i_name_is_standard	True if the i^{th} section has a standard name [31]
Section_i_size	Size of the i^{th} section
Section_i_phisicalAddress	The starting address of the i^{th} section in the file
Section_i_virtualSize	The total size of the i^{th} section in memory
Section_i_entropy	The Shannon entropy of the i^{th} section
Section_i_numberOfRelocations	The number of relocation entries in the i^{th} section
Section_i_pointerToRelocations	The address of the first byte of the relocation entries in the i^{th} section
Section_i_characteristics bit	32 Boolean values - one for each bit of the section Characteristics bit [31]

References

[1] Tracking Malware with Import Hashing | Mandiant — mandiant.com. https://www.mandiant.com/resources/blog/tracking-malware-import-hashing. January 16, 2026.

[2] Decodingmlsecretsofwindowsmalwareclassification, 2023. https://anonymous.4open.science/r/DecodingMLSecretsOfWindowsMalwareClassification-E60C.

[3] Pe identifier (peid), 2024. https://github.com/wolfram77web/app-peid.

[4] Chocolatey, the package manager for windows. https://chocolatey.org/, Accessed January 16, 2026.

[5] Detect-it-easy. https://github.com/horsicq/Detect-It-Easy, Accessed January 16, 2026.

[6] Find malware detection names for Microsoft Defender for Endpoint. https://learn.microsoft.com/en-us/microsoft-365/security/intelligence/malware-naming, Accessed January 16, 2026.

[7] Juanlespin. https://github.com/Maff1t/JuanLesPIN-Public, Accessed January 16, 2026.

[8] Lordnoteworthy/al-khaser. https://github.com/LordNoteworthy/al-khaser, Accessed January 16, 2026.

[9] MOTIF Dataset. https://github.com/boozallen/MOTIF, Accessed January 16, 2026.

[10] PEdiff. https://github.com/im-overlord04/PEDiff, Accessed January 16, 2026.

[11] Pefile, portable executable reader module. https://pypi.org/project/pefile/, Accessed January 16, 2026.

[12] Proxmox virtual environment. https://www.proxmox.com/en/proxmox-ve, Accessed January 16, 2026.

[13] Yara patterns of retdec. https://github.com/avast/retdec/tree/master/support/yara_patterns, Accessed January 16, 2026.

[14] Hojjat Aghakhani, Fabio Gritti, Francesco Mecca, Martina Lindorfer, Stefano Ortolani, Davide Balzarotti, Giovanni Vigna, and Christopher Kruegel. When malware is packin'heat; limits of machine learning classifiers based on static analysis features. In *Network and Distributed Systems Security (NDSS) Symposium 2020*, 2020.

[15] Muqeet Ali, Josiah Hagen, and Jonathan Oliver. Scalable malware clustering using multi-stage tree parallelization. In *2020 IEEE International Conference on Intelligence and Security Informatics (ISI)*, pages 1–6. IEEE, 2020.

[16] Hyrum S Anderson and Phil Roth. Ember: an open dataset for training static pe malware machine learning models. *arXiv preprint arXiv:1804.04637*, 2018.

[17] Simone Aonzo, Yufei Han, Alessandro Mantovani, and Davide Balzarotti. Humans vs. machines in malware classification. In *To appear in Usenix Security 2023*, 2022.

[18] Asad Arfeen, Zunair Ahmed Khan, Riaz Uddin, and Usama Ahsan. Toward accurate and intelligent detection of malware. *Concurrency and Computation: Practice and Experience*, 34(4):e6652, 2022.

[19] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. Dos and don'ts of machine learning in computer security. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3971–3988, Boston, MA, August 2022. USENIX Association.

[20] AV-TEST Institute. AV-TEST Security Report: Malware Statistics for 2025, 2025. Accessed: June 2025.

[21] Ahmad Azab, Robert Layton, Mamoun Alazab, and Jonathan Oliver. Mining malware to detect variants. In *Cybercrime and Trustworthy Computing Conference*, 2014.

[22] Michael Bailey, Jon Oberheide, Jon Andersen, Zhuoqing Morley Mao, Farnam Jahanian, and Jose Nazario. Automated Classification and

Analysis of Internet Malware. In *International Symposium on Recent Advances in Intrusion Detection*, 2007.

[23] Márton Bak, Dorottya Papp, Csongor Tamás, and Levente Buttyán. Clustering iot malware based on binary similarity. In *IEEE/IFIP Network Operations and Management Symposium*, 2020.

[24] Federico Barbero, Feargus Pendlebury, Fabio Pierazzi, and Lorenzo Cavallaro. Transcending transcend: Revisiting malware classification in the presence of concept drift. In *IEEE Symposium on Security and Privacy (Oakland)*, 2022.

[25] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. Scalable, Behavior-Based Malware Clustering. In *Network and Distributed System Security Symposium*, 2009.

[26] Marcus Botacin, Vitor Hugo Galhardo Moia, Fabricio Ceschin, Marco A Amaral Henriques, and André Grégio. Understanding uses and misuses of similarity hashing functions for malware detection and family clustering in actual scenarios. *Forensic Science International: Digital Investigation*, 38:301220, 2021.

[27] Frank Breitinger and Harald Baier. Similarity preserving hashing: Eligible properties and a new algorithm mrsh-v2. In *International Conference on Digital Forensics and Cyber Crime*, 2013.

[28] Capstone. Capstone - The ultimate disassembly framework, 2022. https://www.capstone-engine.org/.

[29] Silvio Cesare, Yang Xiang, and Wanlei Zhou. Malwise—an effective and efficient classification system for packed and polymorphic malware. *IEEE Transactions on Computers*, 62(6):1193–1206, 2012.

[30] Macdonald Chikapa and Anitta Patience Namanya. Towards a fast off-line static malware analysis framework. In *2018 6th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)*, pages 182–187. IEEE, 2018.

[31] Microsoft Corporation. PE Format, 2022. https://docs.microsoft.com/en-us/windows/win32/debug/pe-format.

[32] Emanuele Cozzi, Mariano Graziano, Yanick Fratantonio, and Davide Balzarotti. Understanding Linux Malware. In *IEEE Symposium on Security and Privacy*, 2018.

[33] George E. Dahl, Jack W. Stokes, Li Deng, and Dong Yu. Large-Scale Malware Classification using Random Projections and Neural Networks. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, 2013.

[34] Savino Dambra, Yufei Han, Simone Aonzo, Platon Kotzias, Antonino Vitale, Juan Caballero, Davide Balzarotti, and Leyla Bilge. Decoding the Secrets of Machine Learning in Malware Classification: A Deep Dive into Datasets, Feature Extraction, and Model Performance. In *ACM Conference on Computer and Communications Security*. ACM, November 2023.

[35] Jake Drew, Tyler Moore, and Michael Hahsler. Polymorphic malware detection using sequence classification methods. In *IEEE Security and Privacy Workshops*, 2016.

[36] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. Blanket execution: Dynamic similarity testing for program binaries and components. In *USENIX Security Symposium*, 2014.

[37] David French and William Casey. 2 fuzzy hashing techniques in applied malware analysis. *Results of SEI Line-Funded Exploratory New Starts Projects*, page 2, 2012.

[38] Nicola Galloro, Mario Polino, Michele Carminati, Andrea Continella, and Stefano Zanero. A systematical and longitudinal study of evasive behaviors in windows malware. *Computers & Security*, 113:102550, 2022.

[39] Marius Gheorghescu. An automated virus classification system. In *Virus bulletin conference*, 2005.

[40] Daniel Gibert, Carles Mateu, Jordi Planes, and Ramon Vicens. Using convolutional neural networks for classification of malware represented as images. *Journal of Computer Virology and Hacking Techniques*, 15:15–28, 2019.

[41] Google. BinDiff. https://github.com/google/bindiff, Accessed January 16, 2026.

[42] Weijie Han, Jingfeng Xue, Yong Wang, Lu Huang, Zixiao Kong, and Limin Mao. Maldae: Detecting and explaining malware based on correlation and fusion of static and dynamic characteristics. *computers & security*, 83:208–233, 2019.

[43] Weijie Han, Jingfeng Xue, Yong Wang, Zhenyan Liu, and Zixiao Kong. Malinsight: A systematic profiling based malware detection framework. *Journal of Network and Computer Applications*, 125:236–250, 2019.

[44] Irfan Ul Haq, Sergio Chica, Juan Caballero, and Somesh Jha. Malware lineage in the wild. *Computers & Security*, 78:347–363, 2018.

[45] Xin Hu, Tzicker Chiueh, and Kang G. Shin. Large-scale Malware Indexing Using Function-call Graphs. In *ACM Conference on Computer and Communications Security*, 2009.

[46] Xin Hu, Kang G. Shin, Sandeep Bhatkar, and Kent Griffin. MutantX-S: Scalable Malware Clustering Based on Static Features. In *USENIX Annual Technical Conference*, 2013.

[47] Wenyi Huang and Jack W. Stokes. MtNet: A Multi-Task Neural Network for Dynamic Malware Classification. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2016.

[48] AV-TEST The Independent IT-Security Institute. AV-ATLAS - Malware & PUA — portal.av-atlas.org. https://portal.av-atlas.org/malware. [Accessed 30-06-2025].

[49] Lorenzo Ippolito. A Framework for the Analysis of File Infection Malware. Master's thesis, Politecnico Di Torino, Torino, Italy, March 2024.

[50] Jiyong Jang, David Brumley, and Shobha Venkataraman. Bitshred: feature hashing malware for scalable triage and semantic analysis. In *ACM conference on Computer and Communications Security*, 2011.

[51] Chani Jindal, Christopher Salls, Hojjat Aghakhani, Keith Long, Christopher Kruegel, and Giovanni Vigna. Neurlux: Dynamic Malware Analysis without Feature Engineering. In *Annual Computer Security Applications Conference*, 2019.

[52] Roberto Jordaney, Kumar Sharad, Santanu Kumar Dash, Zhi Wang, Davide Papini, Ilia Nouretdinov, and Lorenzo Cavallaro. Transcend: Detecting concept drift in malware classification models. In *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, page 625–642, USA, 2017. USENIX Association.

[53] Robert J Joyce, Dev Amlani, Charles Nicholas, and Edward Raff. MOTIF: A large malware reference dataset with ground truth family labels. In *Workshop on Artificial Intelligence for Cyber Security*, 2022.

[54] Robert J Joyce, Kevin Bilzer, and Seamus Burke. Malware attribution using the rich header, 2019.

[55] Fabian Kaczmarczyck, Bernhard Grill, Luca Invernizzi, Jennifer Pullman, Cecilia M Procopiuc, David Tao, Borbala Benko, and Elie Bursztein. Spotlight: Malware Lead Generation at Scale. In *Annual Computer Security Applications Conference*, 2020.

[56] Kesav Kancherla and Srinivas Mukkamala. Image visualization based malware detection. In *IEEE Symposium on Computational Intelligence in Cyber Security*, 2013.

[57] ElMouatez Billah Karbab and Mourad Debbabi. Maldy: Portable, data-driven malware detection using natural language processing and machine learning techniques on behavioral analysis reports. *Digital Investigation*, 28:S77–S87, 2019.

[58] Kaspersky. The Cyber Surge: Kaspersky Detected 467,000 Malicious Files Daily in 2024, 2024. Accessed: June 2025.

[59] Doowon Kim, Bum Jun Kwon, and Tudor Dumitraş. Certified malware: Measuring breaches of trust in the windows code-signing pki. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1435–1448, 2017.

[60] Jun-Seob Kim, Wookhyun Jung, Sangwon Kim, Shinho Lee, and Eui Tak Kim. Evaluation of image similarity algorithms for malware fake-icon detection. In *2020 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 1638–1640. IEEE, 2020.

[61] Jesse Kornblum. Identifying Almost Identical Files Using Context Triggered Piecewise Hashing. *Digital Investigation*, 3:91–97, September 2006.

[62] Jesse Kornblum. Identifying almost identical files using context triggered piecewise hashing. *Digital investigation*, 3:91–97, 2006.

[63] Platon Kotzias, Srdjan Matic, Richard Rivera, and Juan Caballero. Certified pup: abuse in authenticode code signing. In *Proceedings of*

the 22nd ACM SIGSAC Conference on Computer and Communications Security, pages 465–478, 2015.

[64] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static Disassembly of Obfuscated Binaries. In *USENIX Security Symposium*, 2004.

[65] Alexander Kuechler, Alessandro Mantovani, Yufei Han, Leyla Bilge, and Davide Balzarotti. Does Every Second Count? Time-based Evolution of Malware Behavior in Sandboxes. In *Network and Distributed System Security (NDSS) Symposium*, NDSS 21, February 2021.

[66] Shinho Lee, Wookhyun Jung, Wonrak Lee, Hyung Geun Oh, and Eui Tak Kim. Android malware dataset construction methodology to minimize bias–variance tradeoff. *ICT Express*, 2021.

[67] Peng Li, Limin Liu, Debin Gao, and Michael K Reiter. On challenges in evaluating malware clustering. In *International Symposium on Recent Advances in Intrusion Detection*, 2010.

[68] Shijia Li, Jiang Ming, Pengda Qiu, Qiyuan Chen, Lanqing Liu, Huaifeng Bao, Qiang Wang, and Chunfu Jia. Packgenome: Automatically generating robust yara rules for accurate malware packer detection. In *ACM SIGSAC Conference on Computer and Communications Security*, 2023.

[69] Yuping Li, Sathya Chandran Sundaramurthy, Alexandru G Bardas, Xinming Ou, Doina Caragea, Xin Hu, and Jiyong Jang. Experimental study of fuzzy hashing in malware clustering analysis. In *Workshop on Cyber Security Experimentation and Test*, 2015.

[70] Chia Chin Lip and Dzati Athiar Ramli. *Comparative Study on Feature, Score and Decision Level Fusion Schemes for Robust Multibiometric Systems*, pages 941–948. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[71] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. αDiff: Cross-version Binary Code Similarity Detection with DNN. In *ACM/IEEE International Conference on Automated Software Engineering*, 2018.

[72] Weitang Liu, Xiaoyun Wang, John D. Owens, and Yixuan Li. Energy-based out-of-distribution detection. In *Proceedings of the 34th In-*

ternational Conference on Neural Information Processing Systems, NIPS'20, Red Hook, NY, USA, 2020. Curran Associates Inc.

[73] Nicola Loi, Claudio Borile, and Daniele Ucci. Towards an automated pipeline for detecting and classifying malware through machine learning, 2021.

[74] Robert Lyda and James Hamrock. Using entropy analysis to find encrypted and packed malware. *IEEE Security & Privacy*, 5(2):40–45, 2007.

[75] Lorenzo Maffia, Dario Nisi, Platon Kotzias, Giovanni Lagorio, Simone Aonzo, and Davide Balzarotti. Longitudinal study of the prevalence of malware evasive techniques. *arXiv preprint arXiv:2112.11289*, 2021.

[76] Alessandro Mantovani, Simone Aonzo, Xabier Ugarte-Pedrero, Alessio Merlo, and Davide Balzarotti. Prevalence and impact of low-entropy packing schemes in the malware ecosystem. In *Network and Distributed System Security*, volume 20, 2020.

[77] Brad Miller, Alex Kantchelian, Michael Carl Tschantz, Sadia Afroz, Rekha Bachwani, Riyaz Faizullabhoy, Ling Huang, Vaishaal Shankar, Tony Wu, George Yiu, Anthony D. Joseph, and J. D. Tygar. Reviewer Integration and Performance Measurement for Malware Detection. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2016.

[78] Najmeh Miramirkhani, Mahathi Priya Appini, Nick Nikiforakis, and Michalis Polychronakis. Spotless sandboxes: Evading malware analysis systems using wear-and-tear artifacts. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 1009–1024. IEEE, 2017.

[79] Nitin Naik, Paul Jenkins, and Nick Savage. A ransomware detection method using fuzzy hashing for mitigating the risk of occlusion of information systems. In *International Symposium on Systems Engineering*, 2019.

[80] Nitin Naik, Paul Jenkins, Nick Savage, Longzhi Yang, Tossapon Boongoen, Natthakan Iam-On, Kshirasagar Naik, and Jingping Song. Embedded yara rules: strengthening yara rules utilising fuzzy hashing and fuzzy rules for malware analysis. *Complex & Intelligent Systems*, 7:687–702, 2021.

[81] Nitin Naik, Paul Jenkins, Nick Savage, Longzhi Yang, Kshirasagar Naik, Jingping Song, Tossapon Boongoen, and Natthakan Iam-On. Fuzzy hashing aided enhanced yara rules for malware triaging. In *IEEE Symposium Series on Computational Intelligence*, 2020.

[82] Anitta Patience Namanya, Irfan U Awan, Jules Pagna Disso, and Muhammad Younas. Similarity hash based scoring of portable executable files for efficient malware detection in iot. *Future Generation Computer Systems*, 110:824–832, 2020.

[83] Antonio Nappa, M. Zubair Rafique, and Juan Caballero. The MALICIA Dataset: Identification and Analysis of Drive-by Download Operations. *International Journal of Information Security*, 14(1):15–33, February 2015.

[84] Lakshmanan Nataraj, Vinod Yegneswaran, Phillip Porras, and Jian Zhang. A comparative assessment of malware classification using binary texture analysis and dynamic analysis. In *Proceedings of the 4th ACM Workshop on Security and Artificial Intelligence*, 2011.

[85] Jonathan Oliver, Muqeet Ali, and Josiah Hagen. Hac-t and fast search for similarity in security. In *International Conference on Omni-layer Intelligent Systems*, 2020.

[86] Jonathan Oliver, Chun Cheng, and Yanggui Chen. Tlsh–a locality sensitive hash. In *Cybercrime and Trustworthy Computing Workshop*, 2013.

[87] Fernando C Colon Osorio, Hongyuan Qiu, and Anthony Arrott. Segmented sandboxing-a novel approach to malware polymorphism detection. In *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 59–68. IEEE, 2015.

[88] Fabio Pagani, Matteo Dell'Amico, and Davide Balzarotti. Beyond precision and recall: understanding uses (and misuses) of similarity hashes in binary analysis. In *ACM Conference on Data and Application Security and Privacy*, 2018.

[89] Marek Pawlicki, Michał Choraś, Rafał Kozik, and Witold Hołubowicz. Missing and incomplete data handling in cybersecurity applications. In Ngoc Thanh Nguyen, Suphamit Chittayasothorn, Dusit Niyato, and Bogdan Trawiński, editors, *Intelligent Information and Database Systems*, pages 413–426, Cham, 2021. Springer International Publishing.

[90] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder, and Lorenzo Cavallaro. TESSERACT: Eliminating Experimental Bias in Malware Classification across Space and Time. In *USENIX Security Symposium*, 2019.

[91] Roberto Perdisci, Andrea Lanzi, and Wenke Lee. McBoost: Boosting Scalability in Malware Collection and Analysis using Statistical Classification of Executables. In *Annual Computer Security Applications Conference*, 2008.

[92] Roberto Perdisci, Wenke Lee, and Nick Feamster. Behavioral Clustering of HTTP-Based Malware and Signature Generation Using Malicious Network Traces. In *USENIX Symposium on Networked Systems Design and Implementation*, 2010.

[93] Roberto Perdisci and U. ManChon. VAMO: Towards a Fully Automated Malware Clustering Validity Analysis. In *Annual Computer Security Applications Conference*, 2012.

[94] Marco Pontello. TrID - File Identifier, 2021. http://marko.net/soft-trid-e.html.

[95] Michal Poslušnỳ and Peter Kálnai. Rich headers: Leveraging this mysterious artifact of the pe format. *Virus Bulletin*, October 2019.

[96] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[97] Erwin Quiring, Lukas Pirch, Michael Reimsbach, Daniel Arp, and Konrad Rieck. Against all odds: Winning the defense challenge in an evasion competition with diversification. Technical report, 2020.

[98] Dima Rabadi and Sin G Teo. Advanced windows methods on malware detection and classification. In *Annual Computer Security Applications Conference*, 2020.

[99] Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles K Nicholas. Malware Detection by Eating a Whole EXE. In *Workshops at the AAAI Conference on Artificial Intelligence*, 2018.

[100] M. Zubair Rafique and Juan Caballero. FIRMA: Malware Clustering and Network Signature Generation with Mixed Network Behaviors. In *International Symposium on Research in Attacks, Intrusions and Defenses*, 2013.

[101] Matilda Rhode, Pete Burnap, and Kevin Jones. Early-stage malware prediction using recurrent neural networks. *computers & security*, 77:578–594, 2018.

[102] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Düssel, and Pavel Laskov. Learning and Classification of Malware Behavior. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2008.

[103] Konrad Rieck, Philipp Trinius, Carsten Willems, and Thorsten Holz. Automatic Analysis of Malware Behavior using Machine Learning. *Journal of Computer Security*, 19(4), 2011.

[104] Christian Rossow, Christian J Dietrich, Chris Grier, Christian Kreibich, Vern Paxson, Norbert Pohlmann, Herbert Bos, and Maarten Van Steen. Prudent practices for designing malware experiments: Status quo and outlook. In *2012 IEEE symposium on security and privacy*, pages 65–79. IEEE, 2012.

[105] Vassil Roussev. Data fingerprinting with similarity digests. In *IFIP International Conference on Digital Forensics*, 2010.

[106] Vassil Roussev and Candice Quates. Content triage with similarity digests: The m57 case study. *Digital Investigation*, 9:S60–S68, 2012.

[107] Zahra Salehi, Ashkan Sami, and Mahboobe Ghiasi. Maar: Robust features to detect malicious activity based on api calls, their arguments and return values. *Engineering Applications of Artificial Intelligence*, 59:93–102, 2017.

[108] Igor Santos, Felix Brezo, Xabier Ugarte-Pedrero, and Pablo G Bringas. Opcode sequences as representation of executables for data-mining-based unknown malware detection. *Information Sciences*, 231:64–82, 2013.

[109] Igor Santos, Jaime Devesa, Felix Brezo, Javier Nieves, and Pablo Garcia Bringas. OPEM: A Static-Dynamic Approach for Machine-learning-based Malware Detection. In *International joint conference CISIS'12-ICEUTE´ 12-SOCO´ 12 special sessions*, 2012.

[110] Joshua Saxe and Konstantin Berlin. Deep Neural Network Based Malware Detection using two Dimensional Binary Program Features. In *International Conference on Malicious and Unwanted Software*, 2015.

[111] Marcos Sebastian, Richard Rivera, Platon Kotzias, and Juan Caballero. Avclass: A tool for massive malware labeling. In *Research in Attacks, Intrusions, and Defenses*, 2016.

[112] Silvia Sebastián and Juan Caballero. Avclass2: Massive malware tag extraction from av labels. In *Annual Computer Security Applications Conference*, pages 42–53, 2020.

[113] Kimin Seo, Kyungsoo Lim, Jaemin Choi, Kisik Chang, and Sangjin Lee. Detecting similar files based on hash and statistical analysis for digital forensic investigation. In *International Conference on Computer Science and Its Applications*, 2009.

[114] M Zubair Shafiq, S Momina Tabish, Fauzan Mirza, and Muddassar Farooq. Pe-miner: Mining structural information to detect malicious executables in realtime. In *International workshop on recent advances in intrusion detection*, pages 121–141. Springer, 2009.

[115] Ian Shiel and Stephen O'Shaughnessy. Improving file-level fuzzy hashes for malware variant classification. *Digital Investigation*, 28:S88–S94, 2019.

[116] Michael R Smith, Nicholas T Johnson, Joe B Ingram, Armida J Carbajal, Bridget I Haus, Eva Domschot, Ramyaa Ramyaa, Christopher C Lamb, Stephen J Verzi, and W Philip Kegelmeyer. Mind the gap: On bridging the semantic gap between machine learning and malware analysis. In *Proceedings of the 13th ACM Workshop on Artificial Intelligence and Security*, pages 49–60, 2020.

[117] ssdeep - Fuzzy hashing program. https://ssdeep-project.github.io/ssdeep/index.html.

[118] Asghar Tajoddin and Saeed Jalili. Hm 3 ald: Polymorphic malware detection using program behavior-aware hidden markov model. *Applied Sciences*, 8(7):1044, 2018.

[119] Nazgol Tavabi, Andres Abeliuk, Negar Mokhberian, Jeremy Abramson, and Kristina Lerman. Challenges in forecasting malicious events from incomplete data. In *Companion Proceedings of the Web Conference 2020*, WWW '20, page 603–610, New York, NY, USA, 2020. Association for Computing Machinery.

[120] G. V. Trunk. A problem of dimensionality: A simple example. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-1(3):306–307, 1979.

[121] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G Bringas. SoK: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers. In *IEEE Symposium on Security and Privacy*, 2015.

[122] Xabier Ugarte-Pedrero, Mariano Graziano, and Davide Balzarotti. A close look at a daily dataset of malware samples. *ACM Transactions on Privacy and Security (TOPS)*, 22(1):1–30, 2019.

[123] Irfan Ul Haq and Juan Caballero. A Survey of Binary Code Similarity. *ACM Computing Surveys*, 54(3), April 2021.

[124] Jason Upchurch and Xiaobo Zhou. Variant: a malware similarity testing framework. In *International Conference on Malicious and Unwanted Software*, 2015.

[125] Jason Upchurch and Xiaobo Zhou. Malware provenance: code reuse detection in malicious software at scale. In *International Conference on Malicious and Unwanted Software*, 2016.

[126] Kevin van Liebergen, Juan Caballero, Platon Kotzias, and Chris Gates. A Deep Dive into the VirusTotal File Feed. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, 2023.

[127] Danish Vasan, Mamoun Alazab, Sobia Wassan, Hamad Naeem, Babak Safaei, and Qin Zheng. Imcfn: Image-based malware classification using fine-tuned convolutional neural network architecture. *Computer Networks*, 171:107138, 2020.

[128] Sitalakshmi Venkatraman, Mamoun Alazab, and R Vinayakumar. A hybrid deep learning image-based analysis for effective malware detection. *Journal of Information Security and Applications*, 47:377–389, 2019.

[129] VirusTotal. https://www.virustotal.com/, Accessed January 16, 2026.

[130] Antonino Vitale, Simone Aonzo, Savino Dambra, Nanda Rani, Lorenzo Ippolito, Platon Kotzias, Juan Caballero, and Davide

Balzarotti. The Polymorphism Maze: Understanding Diversities and Similarities in Malware Families. In *Proceedings of the 30th European Symposium on Research in Computer Security (ESORICS)*, 2025. To appear.

[131] 2021. https://developers.virustotal.com/reference/files.

[132] Virustotal api 2.0 reference: File feed. https://developers.virustotal.com/v2.0/reference/file-feed, Accessed January 16, 2026.

[133] Amged Wageh. Automating The Analysis Of An AutoIT Script That Wraps A Remcos RAT. https://amgedwageh.medium.com/analysis-of-an-autoit-script-that-wraps-a-remcos-rat-6b5b66075b87, January 2022.

[134] George D Webster, Bojan Kolosnjaji, Christian von Pentz, Julian Kirsch, Zachary D Hanif, Apostolis Zarras, and Claudia Eckert. Finding the needle: A study of the pe32 rich header and respective malware triage. In *International conference on detection of intrusions and malware, and vulnerability assessment*, pages 119–138. Springer, 2017.

[135] Georg Wicherski. pehash: A novel approach to fast malware clustering. *LEET*, 9:8, 2009.

[136] Guoqing Xiao, Jingning Li, Yuedan Chen, and Kenli Li. Malfcs: An effective malware classification framework with automated feature extraction based on deep convolutional neural networks. *Journal of Parallel and Distributed Computing*, 141:49–58, 2020.

[137] jiezhong xiao, qian han, and yumeng gao. Hybrid classification and clustering algorithm on recent android malware detection. In *2021 5th International Conference on Computer Science and Artificial Intelligence*, CSAI 2021, page 249–255, New York, NY, USA, 2022. Association for Computing Machinery.

[138] Jinpei Yan, Yong Qi, and Qifan Rao. Detecting malware with an ensemble method based on deep neural network. *Security and Communication Networks*, 2018(1):7247095, 2018.

[139] Wang Yang, Mingzhe Gao, Ligeng Chen, Zhengxuan Liu, and Lingyun Ying. Recmal: Rectify the malware family label via hybrid analysis. *Computers & Security*, 128:103177, 2023.

[140] Miuyin Yong Wong, Matthew Landen, Manos Antonakakis, Douglas M Blough, Elissa M Redmiles, and Mustaque Ahamad. An inside look into the practice of malware analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 3053–3069, 2021.

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

[142] Hao Zhang, Wenjun Zhang, Zhihan Lv, Arun Kumar Sangaiah, Tao Huang, and Naveen Chilamkurti. Maldc: a depth detection method for malware based on behavior chains. *World Wide Web*, 23(2):991–1010, 2020.

[143] Zhaoqi Zhang, Panpan Qi, and Wei Wang. Dynamic malware analysis with feature engineering and feature learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2020.