UCLouvain

epl

École polytechnique de Louvain

Building High-Quality Datasets of Packed Executables

Enhancing Static Detection Models via Curated Packed Binary Datasets

Author: **Clément Alloin**
Supervisors: **Ramin Sadre, Alexandre D'Hondt**
Readers: **Charles-Henry Bertrand Van Ouytsel, Irfan Bulut**
Academic year 2024-2025
Master in Cybersecurity

Abstract

Executable packing refers to a set of techniques applied to executable binaries, including compression, encryption, virtualization, and other operations. These techniques have been used for both legitimate purposes, such as software piracy protection and license management, as well as for malicious activities like distributing harmful software. The methods used for packing continue to evolve, becoming increasingly complex and sophisticated, particularly in terms of evading antivirus detection when applied to malicious binaries. As a result, packing poses a significant challenge in malware analysis, particularly when it comes to accurate detection.

The Packing Box tool has been developed to generate datasets through mass-packing bunches of samples and to combine or filter datasets. The main aim of this experimental toolkit is to identify the most effective machine learning models for static detection of executable packing. While many functionalities of the toolkit are already well developed, it still lacks the ability to evaluate the quality of datasets and some functionalities to generate large ones, and studies arising from the use of this framework are still relying on small unevaluated datasets.

The lack of datasets with proven quality for studying packing detection significantly hinders progress in the field. Without reliable and consistent datasets, it becomes challenging to compare different detection methods, especially machine learning models, in a meaningful way. As a result, the effectiveness of packing detection techniques remains uncertain, and progress in the development of more robust and accurate models is slowed down. A standardized, datasets of proven high quality are essential not only for evaluating existing approaches but also for driving innovation in the detection of packed executables.

The aim of this master's thesis is to address the lack of standardized and high-quality datasets by creating and improving dataset generation techniques, implementing robust methods for dataset validation, and assessing the quality of the datasets to ensure their reliability and effectiveness in training and evaluating packing detection models. These techniques are then integrated into the Packing Box for enhancing its capabilities.

Foreword

"Without clean data, or clean enough data, your data science is worthless."

Michael Stonebraker

Executable packing is a problem that has already been extensively covered in the scientific literature. Yet, many proposed approaches rely on heuristics and models calibrated on small- or medium-sized datasets, sometimes using volatile sources such as malware collections that persist only for a few years, as well as samples from fresh Windows installations or manually packed files. Regardless of the quality of the underlying studies, their datasets are sometimes of questionable quality or, at the very least, not proven to be of sufficient quality.

Therefore, to ensure the development of relevant models, it is crucial to demonstrate that we possess clean data relevant to our problem. The literature already describes methods and metrics for evaluating dataset quality in a generic manner, while some studies also propose ideas for executable-specific metrics. By combining these approaches, it becomes possible to automate the assessment of such datasets, allowing them to be refined to maximize their potential for building effective models.

This master's thesis aims to improve the Packing Box, the experimental toolkit for studying packing and its detection, a framework created in 2022 and showcased in the work of D'Hondt *et al.* in 2024 [1]. Furthermore, this work will add some new capabilities to assess the quality of datasets. This assessment will include consistency checks and similarity evaluation, enabling for the correction and refinement of datasets before model training, with the goal of ensuring their soundness and performance.

Alexandre D'Hondt, ir

Acknowledgements

I would like to first express my gratitude to my promotor, Professor Samin Sadre, and my copromotor, Alexandre D'Hondt, whose guidance, encouragement, and insightful advice were invaluable throughout this research. Their expertise and constructive feedback greatly contributed to shaping the direction and outcomes of this thesis.

I also wish to extend my sincere thanks to my readers, Charles-Henry Bertrand Van Ouytsel and Irfan Bulut, for generously devoting their time to review this work and for providing thoughtful comments that helped improve its quality.

Finally, I owe special gratitude to my family for their support, patience, and understanding during this journey. Their encouragement has been a constant source of motivation.

Table of Contents

List of Figures v

List of Tables vi

Chapter 1 Introduction 1

1.1 Problem Statement . . . 2
1.2 Objectives . . . 2
1.3 Research Questions . . . 3
1.4 Content . . . 3
1.5 Reading Advices . . . 4

Chapter 2 Background 5

2.1 Executable . . . 6
2.1.1 Typical Structure . . . 6
2.1.2 Portable Executable File . . . 7
2.2 Machine Learning . . . 8
2.2.1 Definitions & Concepts . . . 8
2.2.2 Unsupervised Learning . . . 9
2.3 Dataset . . . 11
2.3.1 Definitions & Concepts . . . 11
2.3.2 Quality . . . 12
2.3.3 Creation . . . 12
2.4 Similarity . . . 13
2.4.1 Definitions & Concepts . . . 13
2.4.2 Context-Triggered Piecewise Hashing . . . 14
2.4.3 Locality Sensitive Hashing . . . 15
2.4.4 Alternative Hashing Methods . . . 17
2.4.5 Non-Hashing Methods . . . 17

Summary **19**

Chapter 3 State of the Art **20**

3.1 Dataset Creation **21**

3.1.1 Manual Curation 21

3.1.2 Benchmark Datasets 22

3.1.3 Active Collection 23

3.1.4 Synthetic Packing 24

3.1.5 Industry Contributions 26

3.2 Quality Assessment **28**

3.2.1 Data Quality Metrics 28

3.2.2 Automated Tools 29

3.2.3 Impact on Machine Learning 30

3.3 Similarity **31**

3.3.1 CTPH Applications 31

3.3.2 LSH Applications 33

3.4 Literature Review **33**

3.4.1 Trends 34

3.4.2 Limitations 34

3.4.3 Opportunities 36

Summary **38**

Chapter 4 Framework **39**

4.1 Packing Box **40**

4.1.1 Architecture 40

4.1.2 Abstraction 41

4.1.3 Machine Learning Pipeline 41

4.2 Dataset Creation Enhancements **43**

4.2.1 Archive Filtering Tool 44

4.2.2 Similarity Threshold 45

4.3 Dataset Quality Scoring **45**

4.3.1 Generic Metrics 46

4.3.2 Specific Metrics 46

4.3.3 Dataset Assessment 48

4.3.4 Dataset Quality Improvement 50

Summary **52**

Chapter 5 Experiments **53**

5.1 Methodology **54**

5.1.1 Context 54

5.1.2 Approach 54

5.1.3 Datasets 55

5.2 Experiment 1: Evaluation of Hashing Method **55**

5.2.1 Setup 55

5.2.2 Results 57

5.3 Experiment 2: Metrics Calibration **61**

5.3.1 Setup 61

5.3.2 Training and Testing 62

5.3.3 Results 63

5.4 Experiment 3: Dataset Quality Improvements **66**

5.4.1 Baseline 66

5.4.2 Dataset Fixing 68

5.4.3 Results 71

Summary **73**

Chapter 6 Conclusion **74**

6.1 Summary **75**

6.2 Achievements **75**

6.2.1 Objectives 75

6.2.2 Research Questions 76

6.2.3 Contributions 76

6.3 Future Works **77**

6.3.1 Improvements 77

References **78**

Appendix A Hash Methods

List of Acronyms

AI	Artificial Intelligence
ASLR	Address Space Layout Randomization
AUC	Area Under the Curve
COFF	Common Object File Format
CTPH	Context-Triggered Piecewise Hashing
DBSCAN	Density-Based Spatial Clustering of Applications with Noise
DEP	Data Execution Prevention
DLL	Dynamic Link Library
DQ	Data Quality
EDA	Exploratory Data Analysis
ELF	Executable and Linkable Format
ICA	Independent Component Analysis
K-means	K-Means Clustering
LSH	Locality Sensitive Hashing
LZJD	Lempel-Ziv Jaccard Distance
Mach-O	Mach Object
MCC	Matthews Correlation Coefficient
ML	Machine Learning
MRSH-v2	Multi-Resolution Similarity Hashing version 2
OS	Operating System
PCA	Principal Components Analysis
PE	Portable Executable
RPS	Reference Performance Score
SDhash	Similarity Digest Hash
t-SNE	t-Distributed Stochastic Neighbor Embedding
TLS	Thread Local Storage
TLSH	Trend Locality Sensitive Hash

List of Figures

2.1 Typical structure of a binary executable . . . 7
2.2 Structure of a PE File . . . 7
2.3 Unsupervised machine learning pipeline . . . 10
2.4 DBSCAN . . . 11
2.5 Dataset Creation Pipeline . . . 13
2.6 Taxonomy of similarity search methods . . . 14
2.7 LSH simplified . . . 16

4.1 Architecture of Packing Box tool . . . 40
4.2 Packing Box's machine learning pipeline . . . 42
4.3 Dataset Creation - Focused on Collection and Cleaning . . . 44
4.4 Dataset Creation - Focused on Cleaning and Transformation . . . 45
4.5 Result of the `assess` command . . . 49

5.1 TLSH Normalization Curves . . . 57
5.2 Similarity Hashing Methods Benchmark . . . 58
5.3 Numbers of similar pairs detects for each methods . . . 59
5.4 Similarity hashing methods jaccard heatmap . . . 60

A.1 Ssdeep comparative . . . 1
A.2 Similarity Hashing Methods Benchmark with lsh . . . 2

List of Tables

3.1 Timeline of related works for manual collection technique . . . 21
3.2 Timeline of related works using existing datasets . . . 23
3.3 Timeline of related works for active collection technique . . . 24
3.4 Timeline of related works for synthetic collection technique . . . 25
3.5 Timeline of related works using Industry contributors . . . 26
3.6 Timeline of related works using quality metrics . . . 28
3.7 Timeline of related works proposing assessment tools . . . 30
3.8 Timeline of related works on the impact of quality . . . 30
3.9 Timeline of related works using ctph . . . 31
3.10 Timeline of related works using LSH . . . 33
3.11 Packers Mentioned or Used in Reviewed Papers . . . 35
3.12 Detector Mentioned or Used in Reviewed Papers . . . 36

4.1 Quality Assessment Metrics Overview . . . 48

5.1 Overview of the Similarity Hashing Methods . . . 56
5.2 Time comparison for 1000 files . . . 58
5.3 Summary of model performances . . . 63
5.4 Top Weight combinations for data quality metrics . . . 64
5.5 Choosen Weight combinations for data quality metrics . . . 64
5.6 Evaluation metrics for two models across two tests . . . 71
5.7 Evaluation metrics for non-converted model 2 . . . 72

1 Introduction

1.1 Problem Statement 2
1.2 Objectives 2
1.3 Research Questions 3
1.4 Content 3
1.5 Reading Advices 4

This chapter introduces the problematic and states the objectives of this thesis.

It begins by outlining the general problem of packing, followed by the objectives and the research questions that guide this work. Finally, it introduces the structure of the remainder of this document.

Domain	Machine Learning (ML) for Binary Analysis
Scope	Datasets of Portable Executable samples for models of executable packing static detection
Audience	ML Researchers, Reverse Engineers, Binary Analysts
Purpose	Study quality assessment techniques for building curated datasets of executables, integrating new capabilities into the Packing Box experimental toolkit for that purpose

1.1 Problem Statement

Packing remains a long-standing issue in the field of executable analysis, especially within the Windows PE format. Whether used for legitimate purposes such as software protection or for malicious goals like evading detection, packing transforms binaries in ways that complicate both static and dynamic analysis. As malware authors continue to leverage packing to bypass traditional detection mechanisms, research has increasingly focused on developing more robust static detection techniques, particularly those based on Machine Learning (ML).

However, while ML-based approaches have shown promising, their success heavily depends on the datasets used for training. A growing concern in recent years is the **lack of** high-quality or, at least, **quality-proofed datasets** for packed executable detection. Many existing works rely on small, questionable, or poorly documented datasets, or on outdated tools to label data. Our intuition is that the widespread reliance on such datasets and tools over decades of research could have contributed to misleading ground truths, potential bias in evaluations, and limited generalization to real-world scenarios.

This leads to the core **challenge** in the field of packed executable detection: the lack of reliable, **high-quality datasets**. Without them, it becomes difficult to fairly assess or compare detection methods, especially those based on machine learning. Moreover, the literature rarely defines clear standards for what constitutes a trustworthy dataset, leading to inconsistencies in how ground truths are established.

The **Packing Box** [1] was introduced by D'Hondt in 2022 to facilitate the development of more robust static detection machine learning models. In this experimental toolkit, the machine learning pipeline is automated from dataset generation to model training, with many options to tune each step. But it still lacks the ability to control the quality of manipulated datasets. This toolset therefore constitutes an excellent framework to study quality control methods.

Consequently, this thesis addresses that gap by investigating how dataset quality affects detection performance and by proposing practical methods for building curated datasets using clean binaries and controlled packing processes. It also integrates the results of this study into the Packing Box.

1.2 Objectives

This master thesis focuses on the crucial role of dataset quality in the field of packed executable static detection. It shows the evolution of dataset creation practices and assessment strategies over the past decades. Then, it presents how we contribute the quality assessment ability into the Packing Box framework. Finally, it presents a series of experiments that demonstrate how improving dataset quality directly influences model robustness and generalization.

Our objectives are four-fold :

1. State the **background**.

A – Present basics notions of the Portable Executable format.

B – Introduce Machine Learning with a focus on unsupervised learning technique.

C – Expose the importance of dataset quality.

D – Introduce existing similarity hashing methods.

2. Draw a **state-of-the-art**.

A – Present related work about dataset creation techniques.

B – Give an overview of existing assessment methods.

C – Examine prior research on similarity detection techniques.

3. Present the **framework**.

A – Show the tools from the Packing Box that will be used for experiments.

B – Present the additions designed to improve Packing Box we integrated.

C – Present the quality assessment metrics we added to the framework.

4. Conduct **experiments**.

A – Experiment with different hashing methods.

B – Show the importance of the weighting of the quality metrics.

C – Analyze performance of models when undergoing quality improvement technique.

1.3 Research Questions

In parallel of the objectives, we want to contribute answers to the following research questions:

Question 1 How can we extend the Packing Box framework with new useful functionalities regarding qualitative dataset generation?

Question 2 What measurable criteria and metrics can be defined to assess the quality of datasets used for packed executable static detection?

Question 3 How does the quality of a dataset, as measured by these metrics, influence the performance of Machine Learning models for static packing detection?

1.4 Content

The remainder of this document is structured as follows :

- **Chapter 2 - Background** provides the necessary background information for addressing the problem of building high quality dataset. It also presents the core principles of Machine Learning with a focus on unsupervised learning.
- **Chapter 3 - State-of-the-Art** attempts to draw an overview of the related work in the field, including past researches and more recent development. It focuses on the methods to create a dataset, the way assessing the quality has evolved and the existing methods of similarity hashing.

- **Chapter 4 - Framework** presents Packing Box [2], the framework developed, including tools for generating and manipulating datasets, studying performance and training Machine Learning models. It focuses on the part of the framework that are of interest to expand it with data quality control functionalities.
- **Chapter 5 - Experiments** proposes a methodology and practical cases to compare similarity hashing methods, determine the importance of each new quality metrics and explore the effect of data quality improvement on datasets.
- **Conclusion** closes this thesis by presenting a general summary, by parsing the achieved objectives and by providing ways ahead and ideas for future works.

1.5 Reading Advices

This document is organized such that it can be read in whole or in part, only reading targeted chapters or even only parsing summaries.



Already familiar with PE format, datasets, unsupervised learning and similarity detections techniques ?
If so, you can safely skip Chapter 2 as it gathers all these basics related to these topics. You can eventually get refreshed by reading the summary at its very end.

Already aware of the latest research in the field of dataset creation, quality assessment and similarity search ?
Then for Chapter 3, you can read the summary at its end.

Do you want to start quickly with the toolbox ?
In this case, you can parse Chapter 4 for showcased terminals, illustrating the most interesting commands to get started using the framework.

Do you want to focus on practical matters ?
Then, you should definitely focus on Chapter 5 and parse it for the emphasized observations and results.

2 Background

THIS chapter introduces the foundational concepts needed to understand the rest of this work.

It first explains the structure and format of executable files, focusing on PE. It then provides an overview of Machine Learning principles relevant to binary analysis. Finally, it discusses key notions related to datasets such as quality, creation and similarity search.

2.1 Executable 6
2.1.1 Typical Structure 6
2.1.2 Portable Executable File . . . 7
2.2 Machine Learning 8
2.2.1 Definitions & Concepts . . . 8
2.2.2 Unsupervised Learning . . . 9
2.3 Dataset 11
2.3.1 Definitions & Concepts . . . 11
2.3.2 Quality 12
2.3.3 Creation 12
2.4 Similarity 13
2.4.1 Definitions & Concepts . . . 13
2.4.2 Context-Triggered Piecewise Hashing 14
2.4.3 Locality Sensitive Hashing . 15
2.4.4 Alternative Hashing Methods 17
2.4.5 Non-Hashing Methods . . . 17
Summary 19

Objectives

A – Present basics notions of the Portable Executable format.

B – Introduce Machine Learning with a focus on unsupervised learning technique.

C – Expose the importance of dataset quality.

D – Introduce existing similarity hashing methods.

2.1 Executable

This section introduces the concept of binary executable with its typical structure. It then focuses on Portable Executable (PE) format.

2.1.1 Typical Structure

An executable is a type of file that contains a program that runs directly as the Operating System (OS) executes the program. Executable files are normally created by compiling code into machine code that can be examined and executed by the processor. In addition to the actual instructions for the program, executables often contain metadata, resources and some instructions for the OS stating how to load and run the code. The formats of the executable will vary depending on the OS. For example we can find:

- **Executable and Linkable Format (ELF)** are files commonly used by Linux systems.
- **Mach Object (Mach-O)** are used by macOS.
- **Portable Executable (PE)** are used by Windows.

Executable formats, although differing across operating systems, generally share a common internal structure designed to provide the necessary information for the OS loader to correctly map the file into memory and begin execution. This structure typically includes the following components:

- **Header**: This part contains metadata required to interpret the executables. It usually includes magic number or format identifiers, the Entry Point address (where the executions begins). It can be divided into multiple headers to separate file and section information.
- **Section**: Blocks of the file that contain specific types of data or code. Each section typically serves a distinct purpose, such as storing executable code, static data, or resource information. For example, in the PE format, common sections include *.text*, *.data* and *.rdata*.
- **Relocation Table**: When an executable is loaded into memory, it may not always be placed at the preferred base address specified in the header. The relocation table lists all locations within the code or data that require adjustment if the executable is loaded at a different address. This ensures that internal memory references remain correct regardless of where the executable is loaded.
- **Symbol Table**: Symbols are metadata that associate names with their corresponding addresses within a program. They are organized into a structure called the Symbol Table, which facilitates linking during compilation.

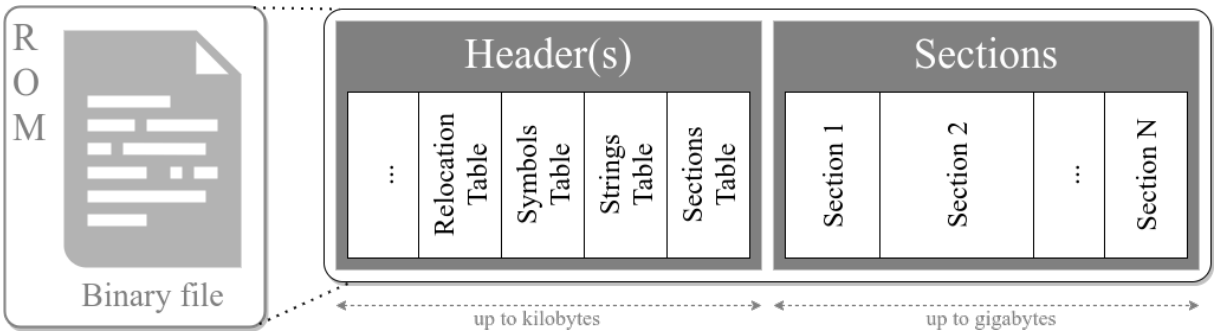


Figure 2.1: Typical structure of a binary executable [3]

2.1.2 Portable Executable File

The Portable Executable (PE) was originally derived from the Common Object File Format (COFF) used in UNIX systems. The format has been adapted and extended to meet the needs of the Windows environment. This file format is used not only for standard application binaries (such as *.exe* files), but also for dynamic-link libraries (*.dll*), system drivers (*.sys*) among others.



Figure 2.2: Structure of a PE File [3]

As illustrated in Figure 2.2 a typical PE file consists of the following major elements:

- **DOS Header**: This header ensures backward compatibility to be recognized by a DOS environment. It starts the PE file and the first byte are "*MZ*", known as **magic byte**". This header does not play a significant role in modern Windows execution. It does store the offset to find the location of the PE Header.
- **PE Header**: This header starts the windows specific portion of the executable. It starts with the **PE Signature** consisting of the byte "*PE\0\0*" which shows the valid PE format. It also includes some metadata such as the machine type (most common types are I386 and AMD64), the number of sections, timestamps, and pointers to optional headers.
- **Optional Header**: Contains fields for execution portability, such as the minimum required operating system version, subsystem type (console vs GUI), and flags for security and memory management features like Address Space Layout Randomization (ASLR), aimed to randomize memory addresses of key data areas to hinder exploitation, and Data Execution Prevention (DEP), used for preventing execution of code from non-executable memory regions.
- **Section Table**: This table follows the PE Header provides a summary of each sections in the executable. Each entry describes a specific section's name, virtual address size in memory, size on disk, and various attributes.
- **Sections**: A section contains all the content required for the program's execution. Some common sections are:

- **.text** contains the executable code. The program's **Entry Point** is typically located within this section.
- **.data** & **.bss** store initialized and uninitialized global and static variables, respectively.
- **.rdata** holds read-only data, such as constants and string literals.
- **.rsrc** stores the PE file's resources such as icons among others.
- **.reloc** holds relocation information, used mainly when the executable or DLL cannot be loaded at its preferred base address. More common in DLL.

PE binaries may also include digital signatures, which provide authenticity verification and may influence whether the file can run without warnings in certain environments. Some may reference absolute paths or contain platform-specific imports, which can restrict portability.

There exist other, less common sections in the PE format, such as *.edata* for exported symbols, *.idata* for imported functions and DLLs, *.pdata* for exception handling, *.debug* for storing debugging information. While these are all standard section names, PE binaries are flexible and can include custom-named sections as well, allowing developers to create specialized areas for specific purposes.

2.2 Machine Learning

This section introduces the fundamental concepts of Machine Learning, providing the necessary background to understand the techniques used in this thesis. It begins by defining Machine Learning and its main categories, followed by a focused on unsupervised learning, dimensionality reduction technique and clustering algorithms.

2.2.1 Definitions & Concepts

Machine Learning (ML) is a subfield of Artificial Intelligence (AI) focused on developing algorithms that can learn from and make decisions or predictions based on data. These algorithms are called **model**. The internal parameters can be adjusted by the model to optimize its performance and their hyperparameters can also be tuned. This is the learning phase. The core objective of ML is to build systems that generalize well on a defined task and goal. We can identify two main goals of ML :

- **Prediction**: using the model to make accurate forecasts or decisions based on input data.
- **Description**: understanding the underlying structure or distribution of the data.

Machine learning tasks can generally be grouped into the following categories:

- **Classification**: Assigning inputs to discrete categories (e.g., spam vs. not spam). It can be either *binary classification* if only two classes are involved or *multiclass classification* if it involves more than two classes.
- **Regression**: Predicting continuous numerical values based on input features (e.g., house price estimation). Regression is essential for tasks where outputs are real-valued and often used in finance, forecasting, and resource optimization.
- **Clustering**: Grouping a set of objects in such a way that objects in the same group, called *cluster*, are more similar to each other than to those in other groups. Clustering helps in identifying hidden structures in datasets, supporting EDA and pattern discovery.

- **Anomaly Detection**: Identifying rare or unusual instances that deviate significantly from the majority of the data (e.g., fraud detection or network intrusion). These methods can be used in monitoring systems for security breaches, equipment failures or unusual behavior detection.

Depending on how the data is presented to the learning algorithm, Machine Learning is generally divided into several key types, each with its own approach to learning and problem-solving: **supervised learning**, where models learn from labeled data; **unsupervised learning**, where models find structure in unlabeled data; **semi-supervised learning**, which combines labeled data with unlabeled data; **self-supervised learning**, where the model generates its own labels from the data; and **reinforcement learning**, where an agent learns to make decisions through trial and feedback.

- In **supervised learning**, the model is trained on labeled dataset, meaning each input is associated with a known output. The algorithm learns a mapping from inputs to outputs by minimizing a predefined loss function. This method is widely used for tasks such as classification and regression, where the goal is to predict an output based on input features. An example of a simple supervised learning algorithm is One Rule (OneR), which builds a single rule based on a single feature to make predictions. This approach is dominant in applications like spam detection, image classification, medical diagnosis, and credit scoring.
- **Unsupervised learning** deals with data that has no explicit labels. Instead of learning from predefined outputs, the goal is to uncover hidden structures, patterns, or relationships within the data. Common techniques include clustering, anomaly detection, and dimensionality reduction. Since there are no labeled outputs to guide the learning process, evaluation often requires domain-specific expertise or indirect measures of success. Algorithms like K-Means Clustering or Density-Based Spatial Clustering of Applications with Noise (DBSCAN) and methods like Principal Components Analysis (PCA) fall into this category.
- **Semi-supervised learning** is a hybrid approach that combines both labeled and unlabeled data. A small set of labeled data is used to guide learning, while a larger set of unlabeled data is leveraged to improve the model's generalization ability. This technique is useful when acquiring labeled data is expensive or time-consuming, but vast amounts of unlabeled data are available. Semi-supervised learning is frequently applied in areas like image recognition and speech processing, where labeled data can be scarce but unlabeled data is abundant.
- **Self supervised learning** is a subset of unsupervised learning. It enables models to train themselves on unlabeled data, instead of requiring labeled datasets. These algorithms, also called predictive or pretext learning algorithms, learn one part of the input from another part, automatically generating labels and transforming unsupervised problems into supervised ones.
- **Reinforcement learning** involves an agent that learns by interacting with an environment. The agent performs actions and receives feedback in the form of rewards or penalties. Over time, it learns a strategy, called policy, that maximizes long-term rewards. This method is widely used in robotics, autonomous control, and games.



In the next section, we go for a deeper discussion on the unsupervised learning type since it is the approach most relevant to this work.

2.2.2 Unsupervised Learning

As stated in the previous section, unsupervised learning refers to a class of Machine Learning techniques where the model is provided with data **without explicit labels** and with the goal to uncover hidden structures or patterns. The typical workflow for an unsupervised learning task involves the following steps.

Firstly, raw data is prepared through cleaning, normalization, or feature extraction techniques. Then, as Figure 2.3 shows, the training process begins by giving the model the unlabeled dataset along with some initial parameters. The chosen Machine Learning algorithm then processes them to produce a model that will be used to describe data.

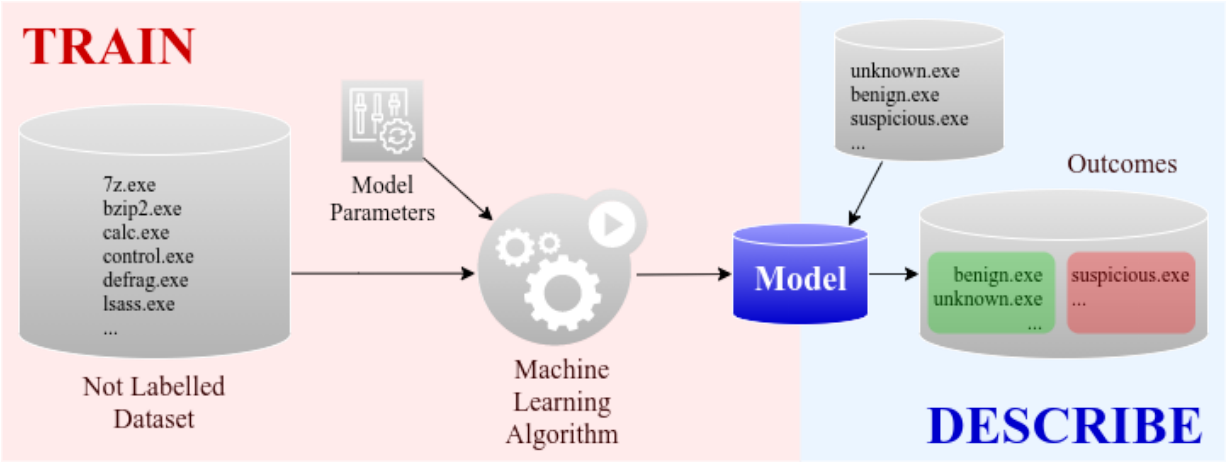


Figure 2.3: Unsupervised machine learning pipeline [3]

Below, we describe the main tasks of unsupervised learning dimensionality reduction, clustering, and anomaly detection and briefly explain their roles and importance.

- **Dimensionality Reduction** A high-dimensional datasets could suffers from issues such as the **curse of dimensionality**, causing the available data to become sparse and increase the difficulty for algorithms to detect pattern. To avoid this growth in computational costs and difficulties in visualization and interpretation, we can use dimensionality reduction technique. The aims of these is to transform high-dimensional data into a lower-dimensional representation while preserving as much of the significant structure as possible to improve the efficiency of clustering or anomaly detection algorithms. Some of these techniques are Principal Components Analysis (PCA), which projects data onto directions of maximum variance Independent Component Analysis (ICA),which separates a multivariate signal into additive, independent components or t-SNE, which is particularly effective for visualizing complex, high-dimensional datasets in two or three dimensions.

- **Clustering** One of the most fundamental problems in unsupervised learning is the clustering task. Different algorithms are used depending on the assumptions and properties of the dataset. Among the most popular techniques are K-means, which groups data into a fixed number of clusters by assigning each point to the nearest cluster center; hierarchical clustering, which builds a tree-like structure of nested clusters and DBSCAN, which groups data points based on density.
  Density-Based Spatial Clustering of Applications with Noise (DBSCAN), in particular, relies on two parameters: **epsilon** , which defines the radius for neighborhood search, and **MinPts**, the minimum number of neighbors a point must have to be considered a core point. Points that do not meet the density requirements are labeled as noise, allowing DBSCAN to effectively find clusters.

- **Anomaly Detection** Anomaly detection refers to the identification of data points that deviate significantly from the general pattern. These points can be called **outliers** or **noise**. They may correspond to rare events or errors. Techniques for anomaly detection can be based on distance measurements, density estimation or clustering alogrithm such as **DBSCAN**.

Figure 2.4 illustrate the concepts and parameters of **DBSCAN**. If a point does not belong to any cluster, it is classified as noise. Detecting these outliers is important because they can disrupt the patterns that Machine Learning models try to capture, resulting in lower accuracy and less reliable generalization. Ignoring noise may also introduce bias into the model. Overall, this degrades the quality of the dataseta critical aspect that will be further discussed in the following section.

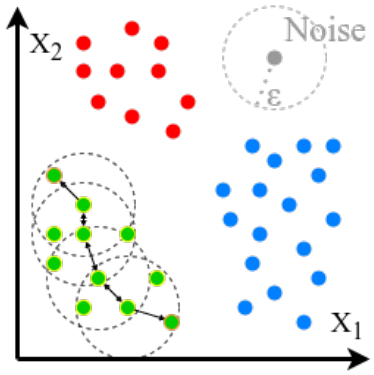


Figure 2.4: DBSCAN



Readers interested in a deeper understanding of other unsupervised learning methods and techniques are encouraged to consult the master's thesis of Martinez titled "*Static Features Exploration for Executable Packing with Unsupervised Learning*" [4].

2.3 Dataset

A dataset is an organized collection of data used for analysis or Machine Learning tasks. This section presents the definitions and concepts of a dataset, discusses the importance of maintaining high data quality and the challenges associated with it, and concludes with an overview of how datasets are commonly created.

2.3.1 Definitions & Concepts

Datasets can take various forms depending on the application. The most common structure is tabular, where data is arranged into rows and columns, each row representing an observation and each column a feature. However, in many contexts, datasets may adopt alternative structures. For example, in natural language processing, a dataset may consist of a collection of text documents; in computer vision, it could take the form of image matrices; in audio processing, datasets often contain waveforms or spectrograms; and in time series analysis, data may be ordered over time.

Datasets can also differ based on their labeling. As discussed in the previous section, supervised learning relies on **labeled** datasets, where each input is associated with a known output, whereas unsupervised learning works with **unlabeled** datasets.

Additionally, datasets often contain **metadata** which are information about the dataset itself, such as feature descriptions, data types, source of data collection, timestamps, and usage guidelines. Metadata is essential for ensuring proper understanding and reproducibility of experiments.

Datasets may also be split into multiple parts to facilitate Machine Learning workflows:

- **Training set**: Used to train the model.
- **Validation set**: Used for hyperparameter tuning and model selection.
- **Testing set**: Used to evaluate the final model's performance.

2.3.2 Quality

The quality of the dataset plays a fundamental role in the performance of the trained Machine Learning models and is often seen as the limiting factor for algorithmic development and scientific progress. A high-quality dataset ensures that models can learn the intended patterns and generalize effectively to new data. Inversely, a poor dataset quality can often lead to biased or underperforming models. The poor quality can be characterized by incomplete labels, duplicated entries, corrupted samples, or unbalanced distributions.

Several key dimensions are critical regarding dataset quality. These include **completeness** (having all necessary data available), **consistency** (the uniformity and coherence of the data), **relevance** (how well the data supports the intended task) and **accuracy** (the correctness of the data values). Other important factors may include **class balance**, the **uniqueness** of records, **representativeness** of the data for the problem domain, and the **absence of noise or outliers**. Ensuring these quality properties is essential for building reliable Machine Learning systems and obtaining trustworthy results.

The quality of a dataset can be lowered by common pitfalls, which in turn degrade the model's efficiency and performance. Some of the most frequent issues that can be observed are:

- **Label noise**: It occurs when data samples are incorrectly labeled. It can happen either due to human error, ambiguous data, or inconsistencies in the labeling process. Since models in supervised learning heavily rely on the correctness of labels, if they are inaccurate, it can lead to reduced predictive accuracy and poor generalization to unseen data. There are two types of possible label noise: *random noise*, where errors happen unpredictably and *systematic noise*, where consistent mislabeling creates biased patterns.
- **Class imbalance**: it happens when the number of samples for each class is significantly unequal. In such cases, the models tend to favor the majority classes, leading to biased predictions and poor detection of minority classes.
- **Dataset drift**: This can happen when the real-world environment evolves, causing the model to face data distributions that are different from those seen during training. As a result, model performance deteriorates because the assumptions it learned from the training data are no longer valid. This dataset drift can be caused by a significant change in the user behavior or even updates when it comes to a model operating on the output of a software.

Moreover, datasets must be sufficiently diverse to reflect the variability of real-world data. When models are trained over an overly uniform dataset, they may achieve high performance during testing but struggle to reach the same performance once deployed in practical settings. As a result, careful curation, thorough validation, and ongoing monitoring of dataset quality are essential components of a robust Machine Learning pipeline.

2.3.3 Creation

The creation of a high-quality dataset is a foundational step in any Machine Learning project, directly influencing the effectiveness of the resulting models. The process typically follows several key stages shown in Figure 2.5.

It begins with **data collection**, where raw data is gathered from various sources such as sensors, logs, user interactions, public repositories, websites or even through existing datasets. Early Machine Learning datasets were often assembled manually through simple collection methods, resulting in relatively small and sometimes noisy collections. Modern practices leverage automated pipelines to amass much larger and more diverse datasets.

Following collection, the dataset undergoes **data cleaning**. This step is important for removing irrelevant entries known as noise, handling missing, correcting inconsistencies, duplicated values, and standardize

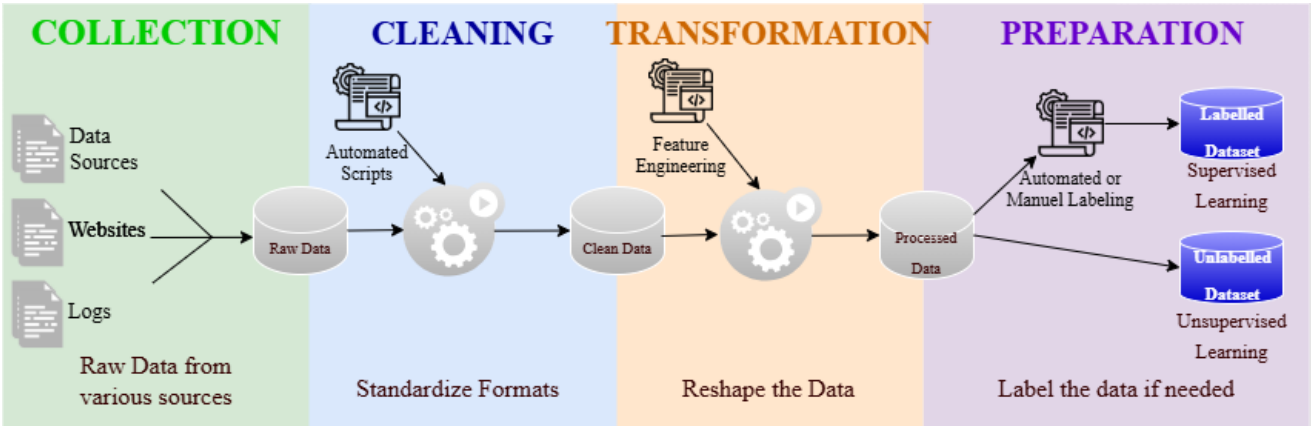


Figure 2.5: Dataset Creation Pipeline

formats. A clean dataset is essential for ensuring that the model won't be biased or hindered by poor data hygiene.

Once the dataset is cleaned, feature engineering plays a critical role in preparing the data for Machine Learning. Through transformations, combinations, or the creation of new features, the raw data is reshaped into forms that better expose underlying patterns. Effective feature engineering enhances the model's ability to detect meaningful signals and can significantly improve performance by presenting the data in a more informative and accessible form.

For **supervised learning** tasks, the next stage is **data labeling**. In this step, each sample is annotated with the correct output. Labeling can be done manually by human experts or be automated with labeling techniques.

In some cases, augmentation techniques can also be applied. These methods are used to artificially expand the dataset and improve model generalization without collecting new data. The technique used will depend on the type of data composing the dataset.



While this section provides a general overview of dataset creation methods, a dedicated and more detailed analysis of approaches specific to binary executable datasets are presented in the State of the Art in Chapter 3.

2.4 Similarity

Identifying and measuring similarity between executable files is an essential step when building, cleaning, and analyzing large datasets of malware or packed binaries, as it helps to sanitize and deduplicate the data. This section introduces the general concept of similarity search and describes various approaches, including both hashing-based and non-hashing methods.

2.4.1 Definitions & Concepts

Similarity search aims to identify files that share content or structural features, even if they are not completely identical. In the context of packed executables, this process provides ways to detect duplicates, near-duplicates, or variants produced by similar packing or obfuscation methods, ultimately improving the dataset quality and thus the model efficiency. Broadly, similarity search methods fall into two families: hashing methods and non-hashing methods. Figure 2.6 depicts an overview of such similarity search methods.

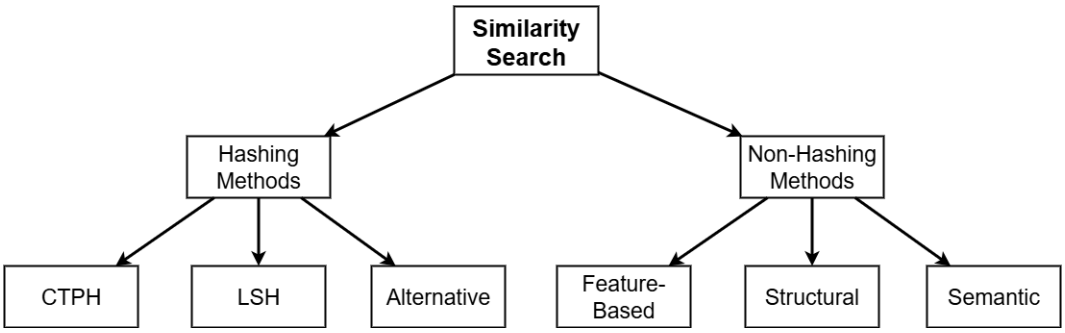


Figure 2.6: Taxonomy of similarity search methods

Hashing methods include approaches like Context-Triggered Piecewise Hashing (CTPH) and Locality Sensitive Hashing (LSH). These methods produce compact representations, called hashes, that preserve some notion of similarity, making it fast to detect near-duplicates or related files at scale.

Non-hashing methods directly compare richer representations of executables without reducing them to hashes. These can include feature-based similarity, which compares extracted characteristics like opcode n-grams or entropy histograms; structural similarity, which analyzes program structures such as control-flow graphs or function call graphs; and semantic similarity, which leverages dynamic execution traces or learned embeddings to capture deeper behavioral likeness.

This section first details the principles behind CTPH and LSH, which are the primary methods used in this work due to their balance of effectiveness and efficiency. It then outlines other non-hashing techniques that can also play a role in measuring similarity between binaries.

2.4.2 Context-Triggered Piecewise Hashing

Context-Triggered Piecewise Hashing (CTPH) is a family of similarity-preserving hashing techniques, often referred to as fuzzy hashing. Unlike traditional cryptographic hash functions such as SHA-256 or MD5, which are designed to produce completely different outputs for even the slightest change in input, CTPH methods aim to detect and quantify partial similarity between files. This makes them particularly useful for packed binary analysis where files often share substantial overlapping content despite being modified, obfuscated, or repacked. The key principle behind CTPH is that files are divided into multiple content-defined blocks and hashes are computed for these blocks independently to form together a fuzzy hash. The boundaries of these blocks are not fixed but are dynamically determined based on the file's content, often by scanning for specific patterns called triggers. This ensures that small modifications (like insertions, deletions, or packing-induced alterations) only affect certain localized regions of the hash while preserving the comparability of the remaining sections.

- **ssdeep** is the most well-known implementation of CTPH. It employs a rolling hash algorithm that continuously computes hashes over a sliding window of bytes. This approach ensures that if data is inserted, removed, or slightly changed, only some local blocks are affected while the rest of the hash remains comparable. As a result, ssdeep remains robust against small edits, packing variations, or the insertion of junk instructions. Once block boundaries are established, each block is hashed, and the resulting sequence of hash values forms the file's fuzzy hash signature. The comparison of two files involves computing the edit distance (Levenshtein distance) between their respective hash strings. This distance is then normalized into a similarity score ranging from 0 (completely different) to 100 (identical).
- Beyond ssdeep, other tools such as **Similarity Digest Hash (SDhash)** have been developed. This tool approaches the problem differently. Instead of using rolling hashes and trigger points, SDhash identifies statistically significant substrings within a byte sequence that are likely to be unique or

meaningful in distinguishing similar files. These substrings are selected based on criteria such as entropy and frequency of occurrence, ensuring that trivial or overly common patterns are disregarded. Each of these significant substrings is hashed and inserted into a Bloom filter (a probabilistic data structure optimized for set membership tests). The collection of these Bloom filters forms the file's similarity digest. To compare two files, SDhash computes the feature set overlap between their digests, quantifying how many significant substrings they share. The resulting similarity score, expressed as a percentage, indicates how much meaningful content the files have in common. One of SDhash's main strengths is its robustness to insertions, deletions, and reordering of file sections, as the substring selection process is unaffected by local shifts in alignment. This makes SDhash particularly suitable for detecting similarities in packed executables, where inserted junk code or padding is common. However, SDhash tends to be computationally heavier than ssdeep, especially during comparison, due to the need for evaluating substring overlaps using Bloom filter intersections.

- Another tool is **Multi-Resolution Similarity Hashing version 2 (MRSH-v2)**. It is an evolution of fuzzy hashing designed to address known alignment and collision-handling weaknesses. At its core, MRSH-v2 retains the same fundamental principle as ssdeep, relying on a rolling hash function to scan through a file and detect suitable block boundaries. However, MRSH-v2 introduces modifications to the hashing algorithm to improve sensitivity and robustness against alignment shifts. In practice, mrshv2 still relies on a rolling hash function to identify trigger points and therefore block boundaries in the file, similar to ssdeep. However, its trigger logic is refined to reduce hash collisions and better align with meaningful content in executablessuch as code or structured data sectionsrather than arbitrary shifts caused by padding or junk data. This helps maintain meaningful comparisons even when files include content alterations typical of packed binaries.

In summary, CTPH methods offer a lightweight yet powerful approach to detecting partial similarities between files. By focusing on content-defined block boundaries and localized hashing, they can uncover relationships that traditional hash functions would miss.

2.4.3 Locality Sensitive Hashing

Locality Sensitive Hashing (LSH) is a probabilistic approach to fuzzy hashing, designed to detect partial similarity between files without requiring full pairwise comparisons. Like other fuzzy hashing methods, it aims to preserve and quantify approximate similarity between inputs but it uses a different mechanism. Instead of computing a digest to be compared via the edit distance as in CTPH, LSH indexes inputs in a way that ensures similar items are mapped into the same bucket with high probability. This indexing mechanism enables LSH to efficiently perform similarity search in high-dimensional feature spaces, which is especially valuable when analyzing large collections of packed executables where exact matching techniques might be ineffective. By preserving approximate similarity, LSH can be applied to feature vectors extracted from binaries – such as n-gram distributions, opcode histograms, or other static characteristics – to identify related samples without exhaustive comparisons.

Traditionally, finding similar files would require comparing each file against every other file, leading to a computational cost of $O(n^2)$, which becomes impractical for large datasets. LSH avoids this by grouping similar files into buckets, as depicted in Figure 2.7, so that only items within the same bucket need to be compared directly. As a result, it becomes much faster to detect duplicates or near-identical files, which is particularly relevant when building or cleaning large datasets.

The typical LSH workflow consists of three main steps. First, a file (or its extracted features) is broken down into n-grams, which are small overlapping sequences that capture local content patterns. These n-grams are then represented as high-dimensional sparse vectors. Next, a hashing process is applied to compress these vectors into compact signatures while preserving similarity information. Finally, these signatures are inserted into multiple hash tables using hash functions specifically designed to ensure that similar signatures are grouped into the same buckets. Candidate pairs found within the same buckets can

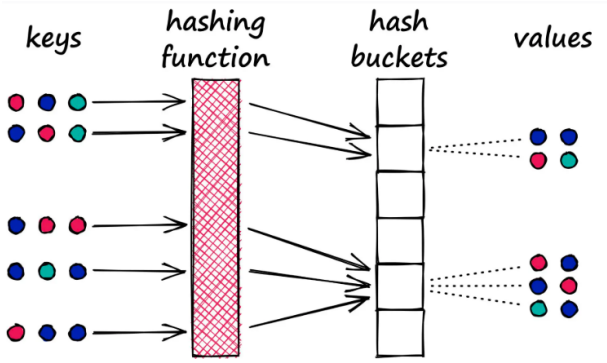


Figure 2.7: LSH simplified [1]

then be directly compared to verify similarity. This pipeline, *n*-grams division, hashing and bucketing, enables scalable detection of near-duplicates or content-related variants, even across datasets with millions of samples.

Two of the most widely hashing techniques used with LSH are MinHash and SimHash. While they both aim to enable efficient similarity search, they operate on different data representations and are tailored to distinct similarity metrics.

- **MinHash**, or Minimum Hashing, is a hashing technique specifically designed to approximate the Jaccard similarity between sets. In practice, a binary file is first transformed into a set of featurestypically n-grams representing sequences of bytes or opcodes. The Jaccard similarity between two files is defined as the size of the intersection of their feature sets divided by the size of their union. Calculating Jaccard similarity directly can be expensive for large datasets, so MinHash compresses each feature set into a compact signature vector. This is achieved by applying multiple hash functions to the feature set and recording the minimum hash value observed for each function. These signature vectors retain the probabilistic property that the chance of two sets having the same hash value at a given position is equal to their Jaccard similarity. Once the MinHash signatures are generated, LSH is applied to group similar signatures into the same buckets, allowing for efficient similarity detection. This method is particularly useful for identifying files with substantial content overlap, even when some sections are inserted, deleted, or packed differently
- **SimHash** is another prominent LSH algorithm but is designed to approximate the cosine similarity between high-dimensional vectors rather than Jaccard similarity between sets. This makes SimHash well-suited for scenarios where file features are best expressed as weighted vectorssuch as opcode frequency distributions, byte histograms, or other statistical profiles derived from binary files. The process begins by converting a file into a high-dimensional vector where each component reflects the weight or frequency of a particular feature. SimHash then transforms this vector into a fixed-length binary fingerprint. This is done by projecting the vector into a series of dimensions and deciding, for each bit position in the fingerprint, whether the weighted combination of features is positive or negative. When comparing two files, SimHash calculates the Hamming distance between their fingerprints. The Hamming distance simply counts how many bits are different between the two fingerprints. A smaller Hamming distance indicates greater similarity between the files, as their overall feature distributions are alike.

Both MinHash and SimHash provide scalable techniques to perform similarity search on large binary datasets, but each excels depending on how the file's characteristics are represented.

[1] https://www.pinecone.io/learn/series/faiss/locality-sensitive-hashing/

2.4.4 Alternative Hashing Methods

While traditional fuzzy hashing techniques such as Context-Triggered Piecewise Hashing and Locality Sensitive Hashing pipelines like MinHash and SimHash have proven effective for similarity detection, several modern algorithms have emerged that do not fit neatly into either category. These alternative hashing methods combine characteristics from digest-based hashing and locality-sensitive techniques, providing robust similarity detection even in the presence of file modifications, obfuscation, or packing.

Two existing methods in this category are Trend Locality Sensitive Hash (TLSH) and Lempel-Ziv Jaccard Distance (LZJD). Both are designed to produce lightweight, similarity-preserving digests while maintaining scalability and resilience against common binary transformations.

- **TLSH** is a similarity digest algorithm designed to detect files with partial structural resemblance, even after transformations like insertions, deletions, or block reordering. Unlike CTPH techniques, TLSH does not rely on detecting block boundaries or context-trigger points. Instead, it builds a histogram of byte frequencies over sliding windows across the entire file, capturing localized patterns that remain stable even under structural changes. Once the byte histogram is computed, TLSH applies a series of hashing and normalization steps to create a compact digesta fixed-length string encoding the file's content structure. To compare two TLSH digests, a custom distance function is used, which evaluates differences in their byte frequency histograms while also accounting for the relative ordering of content blocks within the digest. The resulting TLSH distance score is symmetrical and normalized, with smaller values indicating stronger similarity between files. Unlike other fuzzy hashing methods such as ssdeep, which output a similarity score ranging from 0 (different) to 100 (identical), TLSH produces a distance score where 0 means the files are identical, and the distance increases with dissimilarity. There is no fixed upper limit to this scoreit can grow indefinitely depending on how structurally different the files are.
- **LZJD** offers an alternative approach by combining ideas from data compression and set-based similarity metrics. It leverages the Lempel-Ziv (LZ77) compression algorithm to parse a file and extract a dictionary of significant substrings (LZ-set), representing repetitive or meaningful content within the binary. Once a file is represented as a set of substrings, LZJD measures similarity using the Jaccard index, which compares the intersection over the union of two files' LZ-sets. This method is resilient to insertions, deletions, and reordering of content blocks, as it focuses on shared substrings rather than their exact position within the file. Unlike MinHash, which approximates Jaccard similarity via randomized projections, LZJD computes it directly based on substrings derived from compression.



While other similarity hashing algorithms do exist, the methods presented here were selected for their practical relevance, robustness, and widespread recognition in binary analysis research. Expanding the scope to lesser-known or experimental techniques was intentionally avoided to maintain a focused and applicable comparison.

2.4.5 Non-Hashing Methods

While hashing-based techniques such as CTPH and LSH provide efficient mechanisms for approximate similarity detection, they may overlook deeper structural or semantic relationships between packed binaries. To address these limitations, a range of non-hashing similarity methods exist, which operate on richer representations of binaries. These approaches generally fall into three categories: feature-based, structural, and semantic similarity.

Feature-based similarity involves comparing manually or automatically extracted characteristics from binary files. These features can include statistical descriptors such as byte entropy histograms, opcode or

instruction n-gram distributions, imported API call frequencies, and section sizes. By encoding these characteristics into numeric vectors, standard distance metrics such as cosine similarity, Euclidean distance, or Manhattan distance can be used to quantify pairwise similarity.

Structural similarity methods go beyond flat features by analyzing the internal layout and control structure of the executable. This includes representations such as Control Flow Graphs (CFGs), Function Call Graphs (FCGs), or other inter-procedural relationships. Graph matching algorithms, graph embeddings, and subgraph isomorphism tests can be used to compare these structures, identifying binaries that share similar logic or packing strategies, even when the underlying bytecode differs. Some systems encode these graphs into vector representations using graph neural networks (GNNs), which allows them to be compared at scale.

Semantic similarity attempts to capture the behavior of the binary by analyzing how it executes. This can involve collecting dynamic execution traces (e.g., system calls, memory writes, or taint propagation) through sandboxing or emulation, and comparing these traces across samples. Alternatively, deep learning models can be trained to learn embeddings from raw binary content, decompiled code, or disassembled instructions, allowing semantically related samples to be mapped to nearby points in an embedding space. These models often leverage pre-training and transfer learning to generalize across different packing schemes or binary families.

Compared to hashing-based approaches, non-hashing methods typically offer greater precision, making them well-suited for detecting variant-packed executables or samples altered through polymorphic packing. However, this increased accuracy comes at a significant computational cost. Unlike fuzzy hashes, which condense file content into compact signatures for fast comparison, non-hashing methods often require processing the entire file or its high-dimensional representations, such as control-flow graphs, feature vectors, or execution traces, and then performing pairwise comparisons across all samples. This can result in substantial overhead, particularly when dealing with large datasets. Furthermore, many of these techniques depend on expensive preprocessing steps, including disassembly, deobfuscation, or dynamic analysis, making them less scalable in time-constrained or resource-limited scenarios.

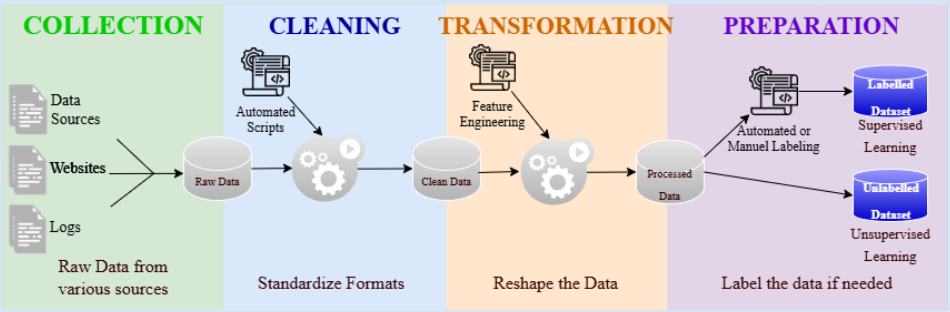
SUMMARY

Executables, regardless of their specific formats, share a common structure with similar components. The Portable Executable (PE) format is used by Windows applications, including executables, DLLs, and system drivers, consists of several key components: such as the DOS Header, followed by the PE Header, and a Section Table that describes the sections that organizing the executable code, data, and resources.



Machine Learning (ML) is a broad field that can be divided based on how data is presented to the algorithm, including supervised, unsupervised, semi-supervised, self-supervised, and reinforcement learning. In **unsupervised** learning, models seek to uncover hidden structures without labeled outputs. Among the important unsupervised tasks are clustering, dimensionality reduction, and anomaly detection. **DBSCAN** plays an essential role in identifying outliers that could lead to lowering the model's performance.

The **Dataset quality** plays a critical role in the success of Machine Learning models. Without high-quality data, models are prone to poor generalization, biases, and reduced overall performance. The quality of a dataset can be evaluated through several key dimensions, including **completeness**, **consistency**, **relevance**, and **accuracy**. Importantly, quality must be considered at every phase of the dataset **creation** process to ensure the reliability and robustness of the final model.



Similarity-based techniques like **LSH** and **CTPH**, with tools such as **ssdeep** and **SDhash**, further complement traditional Machine Learning and data curation processes. LSH helps efficiently identify and cluster similar or near-duplicate executables by grouping them into hash buckets based on shared features, dramatically reducing the computational cost of large-scale similarity search. Meanwhile, CTPH methods provide a lightweight, content-aware fuzzy hashing method that can detect partial similarities between files even after modifications. Both approaches are especially useful when building or cleaning large datasets of packed executables, ensuring higher dataset quality and better detection performance. In addition, alternative similarity hashing methods such as **TLSH** and **LZJD** offer robust detection capabilities by capturing structural similarities through histogram-based digests or compression-derived substrings. These techniques provide valuable alternatives to traditional hashing approaches, further enhancing the ability to curate reliable and diverse datasets for static detection models.

3 State of the Art

THIS chapter reviews the current landscape of dataset construction and data quality assessment for packed malware detection.

It describes the different ways datasets are built how data quality has become a critical factor based on the past research. It traces the evolution of data quality assessment from early conceptual frameworks to modern tools, and highlights evidence on the practical impact of data quality issues. It then identifies trends, limitations and opportunities which stem from the current literature.

3.1 Dataset Creation 21
3.1.1 Manual Curation 21
3.1.2 Benchmark Datasets 22
3.1.3 Active Collection 23
3.1.4 Synthetic Packing 24
3.1.5 Industry Contributions . . . 26
3.2 Quality Assessment. 28
3.2.1 Data Quality Metrics 28
3.2.2 Automated Tools 29
3.2.3 Impact on Machine Learning 30
3.3 Similarity 31
3.3.1 CTPH Applications 31
3.3.2 LSH Applications 33
3.4 Literature Review. 33
3.4.1 Trends 34
3.4.2 Limitations 34
3.4.3 Opportunities 36
Summary 38

Objectives

A – Present related work about dataset creation techniques.

B – Give an overview of existing assessment methods.

C – Examine prior research on similarity detection techniques.

3.1 Dataset Creation

The effectiveness of static and dynamic analysis techniques in binary classification, malware detection, or packer identification is highly dependent on the nature and quality of the datasets used for training and evaluation. Over the years, researchers have adopted a variety of strategies to construct such datasets, each reflecting different trade-offs between realism, control, and scalability. This section categorizes and reviews the principal approaches found in the literature for creating datasets of Windows binary executables, particularly in contexts involving packed or obfuscated files. Each category is introduced below, followed by a chronological overview of representative research papers.

3.1.1 Manual Curation

This approach involves manually assembling datasets by collecting benign binaries from known, clean environments, such as fresh Windows installations, and malware from publicly available sources, for instance VX Heaven[1], VirusTotal[2] or VirusShare[3]. Researchers typically curate these datasets to ensure control over content and labeling, making them well-suited for reproducible experiments, though potentially less reflective of real-world diversity. The following timeline presents key works that utilize this approach to dataset construction.

2011	Detection of packed executables using support vector machines	Wang *et al.*
2012	Detecting malicious code by applying classification techniques	Shabtai *et al.*
2015	Mal-EVE: Static detection model for evasive malware	Lim *et al.*
2017	Packer identification based on metadata signature	Minh Hai *et al.*
2021	A Benchmark API Call Dataset for Windows PE	Ozgur Catak *et al.*
2022	Multi-feature Dataset for Windows PE Malware Classification	Yousuf *et al.*
2024	Static Malware Analysis Using Low-Parameter ML Models	Baker del Aguila *et al.*
	Assemblage: Automatic Binary Dataset Construction for ML	Liu *et al.*

Table 3.1: Timeline of related works for manual collection technique

In 2011, Wang *et al.* [5] proposed a generic packing detection framework. They created their dataset by collecting from two websites ; 3376 malicious programs from VX Heaven and 408 benign programs obtained from PChome Online[4]. They verified the packed proportion using PEiD. They also collected non-packed samples from a clean installations of Windows XP Professional, Windows Vista, Windows Server 2003 and Windows Server 2008.

[1] https://web.archive.org/web/20170817143838/http://vxheaven.org
[2] https://www.virustotal.com
[3] https://virusshare.com
[4] https://www.pchome.com.tw

In 2012, Shabtai *et al.* [6] proposed a method to detect unknown malware using OpCode *n*-gram patterns as features. They created their dataset by gathering 7688 malicious files from the VX Heaven website. They also gathered 22735 benign files from machines running the Windows XP operating system on their campus. Their dataset contain some packed or compressed files.

In 2015, Lim *et al.* [7] published Mal-EVE, a static detection model aimed at identifying evasive malware. They created their dataset by collecting 1603 malware samples from the VirusSign[5] service and 407 benign files from a Windows 7 installation.

In 2017, Minh Hai *et al.* [8] proposed an identification method for packed code based on metadata signature. They performed their evaluation on a dataset composed of 5374 malware from VX Heaven and 7440 malware from VirusShare. They did not apply any additional packing techniques on these samples.

In 2021, Catak *et al.* [9] published a benchmark dataset aiming to standardize research in static malware classification. They created the dataset by collecting Windows PE binaries from VirusTotal and labeled them based on antivirus detection rates.

In 2022, Yousuf *et al.* [10] published a multi-feature dataset for training machine learning classifiers for detecting malicious PE files. To compose this dataset they collected over 20000 samples from MalwareBazaar[6]. They then used VirusTotal on the samples to label them.

In 2024, Baker del Aguila *et al.* [11] proposed low-parameter machine learning models for static malware detection. They created their dataset by collecting malware samples from VirusShare and extracting features directly from the binaries without applying additional modifications.

In 2024, Liu *et al.* [12] published a study introducing ASSEMBLAGE, an automated method for binary dataset construction aimed at machine learning applications. They created their dataset by collecting binaries from large GitHub repositories and applied automated processing steps to filter, label, and prepare the samples without manual intervention.

While these approaches seem to provide well-labeled and reproducible datasets (notwithstanding the volatility of some sources used, e.g. VX Heaven has disappeared), they may under-represent the variability and complexity seen in real-world binary samples.

3.1.2 Benchmark Datasets

Benchmark datasets are pre-compiled collections released by the research community or data science platforms. Examples include EMBER[7][13], [8], and Kaggle[9] malware challenges. These datasets provide standardized baselines for evaluation and comparison across studies, enabling faster experimentation by eliminating the need for manual collection and labeling. However, they may have limitations in terms of transparency, adaptability to niche tasks (such as packer detection), or temporal relevance due to aging sample content. The studies below demonstrate the use of already existing datasets.

2008	Classification of packed executables for accurate virus detection	Perdisci *et al.*
2009	Mining Structural Information to Detect Malicious Executables	Shafiq *et al.*
2016	Detecting Packed Executable File	Hubballi *et al.*
2020	Limits of ML Classifiers Based on Static Analysis Features	Aghakhani *et al.*
2021	Pipeline for Detecting and Classifying Malware through ML	Loi *et al.*

[5] https://www.virussign.com
[6] https://bazaar.abuse.ch/browse
[7] https://github.com/elastic/ember
[8] https://software.imdea.org/projects/malicia/dataset.html
[9] https://www.kaggle.com/datasets

2023	ML-Based Approach for Malware Detection and Categorization	Buriro *et al.*

Table 3.2: Timeline of related works using existing datasets

In 2008, Perdisci *et al.* [14] published a paper in which they apply pattern recognition techniques for fast detection between packed and non-packed executables. Their dataset had 5498 executables. They collected 2598 of these from the Malfease Project[10] dataset. Their benign files were collected 2231 executables from a clean Windows XP Home installation. They packed some of these benign executables using 17 unspecified packers. The author used the same dataset in a later work about McBoost, a tool intended to improve the scalability of existing malware collection and analysis approaches, published the same year [15].

In 2009, Shafiq *et al.* [16] presented a PE-miner framework[11] that extracts features from PE executables to detect zero-day malware. The dataset they used in their study was build by collecting 1447 benign files from the local network of their virology lab and by collecting 10,339 malware from VxHeavens and an additional 5586 unlabeled malware from the Malfease dataset.

In 2016, Hubballi *et al.* [17] proposed a semi-supervised technique and an anomaly based detection method to identify packed executable files. In their study, they used two existing datasets. The first, referred as the Georgia-Tech dataset, is composed of 2598 packed malware, 2231 non-packed benign files and 669 packed benign files. The second was the Malicia dataset.

In 2020, Aghakhani *et al.* [18] published a study exploring the limitations of static machine learning classifiers when dealing with packed malware. They built their dataset by collecting a large scale of real-world malware and goodware files from a commercial vendor of advanced malware protection products that were collected during a timeframe of 4 months mid-2017. They combined this dataset composed of 29573 executables with the EMBER dataset [13]. They built a second dataset, based on the first one, by packing all executables with commercial and free packers known as Obsidium, PELock, Themida, PECompact, PEtite, UPX, kkrunchy, MPRESS, and tElock.

In 2021, Loi *et al.* [19] proposed a malware taxonomic classification pipeline able to classify Windows Portable Executable files as either malicious or benign. They used the EMBER dataset [13] for their work and more specifically the 600000 labeled samples.

In 2023, Buriro *et al.* [20] published a study proposing MALWD&C, a machine learning-based approach for efficient malware detection and categorization. The authors evaluated their method using the publicly available BODMAS[12] dataset, which contains a large collection of labeled PE malware and benign samples. Although they also referenced the EMBER dataset [13] as a well-known benchmark in the field, they opted to use BODMAS for their experiments without providing an explicit justification for this choice.

Although benchmark datasets might accelerate experimentation and facilitate comparison across studies, they may lack adaptability for specific tasks or suffer from aging sample content.

3.1.3 Active Collection

This category includes samples obtained through automated or semi-automated mechanisms such as honeypots, or sandbox environments interacting with Command-and-Control servers. These techniques yield realistic, up-to-date samples, often including packed or polymorphic variants. However, the benign sample side is typically limited or separately sourced, and labeling may require additional effort. This timeline presents research that employed some of the previously mentioned active collection techniques.

[10] https://web.archive.org/web/20141221153307/http://malfease.oarci.net
[11] https://github.com/kmuinfosec/Pe-miner
[12] https://github.com/whyisyoung/BODMAS

2006	Automating the Hidden-Code Extraction of Malware	Royal *et al.*
2009	Packed PE File Detection for Malware Forensics	Han *et al.*
2010	A Fast Flowgraph Based Classification System	Cesare *et al.*
2013	Analysis of Drive-by Download Operations and Abuse Reporting	Nappa *et al.*
2023	Method for analysis and detection of malicious executables	Subrahmanyam *et al.*

Table 3.3: Timeline of related works for active collection technique

In 2006, Royal *et al.* [21] published PolyUnpack, a framework to automate the extraction of hidden code from self-unpacking malware. They created their dataset by collecting 3467 real malware samples from OARC[13] repository. These samples were captured from September 2005 to January 2006 by mail traps, user submissions, honeypots.

In 2009, Han *et al.* [22] published a paper where they detect packed PE file using the entropy of the entry point section and some other essential characteristic of packed PE file. Their dataset was created by collecting normal 400 PE files from a Windows installation (version unspecified) and 400 additional packed PE files from a honeypot.

In 2010, Cesare *et al.* [23] proposed an approach to detect polymorphic variants using flow graphs and determine similarity between programs by identifying the underlying isomorphic flow graphs. To evaluate their approach, they created a dataset by using honeypots in the mwcollect Alliance network. They collected a total of 809 unique malware samples between 29-04-2009 and 17-05-2009. The mwcollect Alliance, started around 2005, was a distributed network of honeypots designed to automatically gather real-world malware samples from the internet.

In 2013, Nappa *et al.* [24] conducted a large-scale analysis of drive-by download operations across the internet. To facilitate this, they constructed a dataset of malware samples by leveraging honeypots and automated web crawlers that simulated user browsing behavior during 11 months. These systems visited potentially malicious websites and collected binaries that were silently delivered (drive-by) to users without interaction.

In 2023, Subrahmanyam *et al.* [25] proposed a novel hybrid detection model designed to analyze and detect malicious executables, including those that are obfuscated or packed. To evaluate their approach, they collected 2210 malware samples and 935 benign samples using an IoT honeypot. Additionally, they manually packed some of the samples using UPX and a custom packer to test the model's robustness against packing techniques.

Datasets gathered through active collection offer realism and temporal relevance but often require additional effort for labeling and may focus predominantly on malicious content.

3.1.4 Synthetic Packing

In synthetic setups, researchers construct controlled datasets by applying known packers, such as UPX, Themida, ASProtect, to benign or malicious binaries. This enables targeted experimentation with packing-related detection mechanisms. Although the samples may not reflect the complexity of real-world adversarial techniques, this method offers precise ground truth and reproducibility. The following timeline includes studies where researchers deliberately applied packers to construct datasets tailored to packing detection or unpacking evaluation.

[13] https://www.dns-oarc.net

2012	SPADE: Signature based PAcker DEtection	Naval *et al.*
	Countering entropy measure attacks on packed software detection	Ugarte-Pedrero *et al.*
2014	Feature set reduction for the detection of packed executables	Burgess *et al.*
2017	A control flow graph-based signature for packer identification	Saleh *et al.*
2019	All-in-One Framework for Detection, Unpacking, and Verification	Choi *et al.*
	CEG-Based Approach for Malware Packer Identification	Li *et al.*
2020	Detection of Metamorphic Malware Packers Using Multilayered LSTM Networks	Bergenholtz *et al.*
2021	Self-Evolving PAcker Classifier against rapid packers evolution	Noureddine *et al.*
2022	Research on the Construction of Malware Variant Datasets	Lu *et al.*
2024	Identifying Malware Packers through Multilayer Feature Engineering in Static Analysis	Alkhateeb *et al.*

Table 3.4: Timeline of related works for synthetic collection technique

In 2012, Naval *et al.* [26] proposed a novel technique, SPADE (Signature-based PAcker DEtection), to identify packers. They prepared their dataset by collecting malware through VX Heaven and allied user agencies. They selected 500 native (non-packed) samples, which they divided into training and testing sets. They packed both sets using these following 6 packers : ASPack, exe32pack, NsPack, PeCompact, PEtite and UPX. The packers were chosen based on the top packers listed on the website VirusTotal.

In 2012, Ugarte-Pedrero *et al.* [27] proposed a method to obtain an alternative entropy measure more resilient to adversarial techniques. To performed their experimentation, they created a dataset of 3000 files composed of 1000 not-packed executables, 1000 executable files protected with known packers, and 1000 samples protected with custom packers. They used VX Heaven to gather 1000 malware and a clean Windows XP installation to collect 1000 goodware. They then packed 500 files from both sources with 10 packers : Armadillo, ASProtect, FSG, MEW, PackMan, RLPack, SLV, Telock, Themida and UPX. The last 1000 were malware samples from the Zeus family.

In 2014, Burgess *et al.* [28] published a packing detection method using entropy analysis. They detailed the creation of their dataset as follows. For their training dataset,they collected 150 non-packed executable files from a clean Windows XP install. They then packed them using 2 packers, ASPACK and UPX, resulting in 450 executables. Concerning their validation set, they selected 170 different non-packed files from Windows XP and used 4 additional packers known as Themida, Obsidium , RLPack and Telock.

In 2017, Saleh *et al.* [29] proposed automatic identification of malware packers using a compact signature for the control flow graph. To conduct their experiment , they used a dataset composed of 20 files on which they used 7 know packers: : UPX, EXEcryptor, Themida, FSGv1.33, FSGv2.0, eXpressor and Yodas Protector. They did not state the source of the 20 files.

In 2019, Choi *et al.* [30] published an all-in-one framework for detecting and unpacking malware. They created their dataset by manually packing 130 known clean executables, collected from Putty and provided by Microsoft, using 19 different packers, resulting in 2600 packed samples. The 19 packers used are UPX, ASPack, NSPack, MPRESS, Yodas Protector, MEW, Packman, RLPack, BeRoEXEPacker, PECompact, Petite, JDpack, Molebox, eXpressor, Yodas Crypter, FSG, exe32pack, WinUpack and Neolite.

In 2019, Li *et al.* [31] proposed a graph-based method for identifying the specific packer used on an executable, leveraging consistently executing graph mining. For their experiments, they constructed three datasets. The first consisted of 10 benign executables from a Windows 7 Ultimate installation, packed using 15 different commercial and open-source packers. The second and third datasets were built from over 50,000 malware samples collected from VirusShare and VX Heaven repositories.

In 2020, Bergenholtz *et al.* [32] presented a method using recurrent neural networks to detect metamorphic malware packers. They collected 1904 executable files from a Windows installation and used 42 different packers on each file. After that, they added 13,002 DLL files found in the same windows installation.

In 2021, Noureddine *et al.* [33] presented SE-PAC, A Self-Evolving PAcker Classifier, that provides an effective and robust solution to deal with the evolution of packers. They created their dataset using two sources: (1) real malware samples provided by Cisco, and (2) a synthetic dataset constructed by manually packing Windows binaries using 31 different packers. The Cisco samples were collected between February and October 2017, similar to the dataset used in a 2019 study by Biondi *et al.* [34], suggesting both works relied on the same or overlapping dataset. Since Cisco provide some sample, this work could also fall under the next category of Subsection 3.1.5.

In 2022, Lu *et al.* [35] published a study on constructing malware variant datasets to improve malware detection. They created the dataset by collecting real malware samples (source not precisely specified), then generating synthetic variants through controlled code transformations such as packing, encryption, and adding dummy operations.

In 2024, Alkhateeb *et al.* [36] introduced a static analysis-based approach for detecting both known and unknown malware packers employing multilayer feature engineering. Their dataset comprised over 7000 samples collected from two sources. The first one, [14], was used to gather 2162 benign executables. Malware samples were obtained from VirusTotal, all of which were manually packed using various tools resulting in 5116 samples. The packers used were Aspack, UPX, PECompact, NSpack and Aegis.



The key distinction in this category is that packing is deliberately and manually applied to files within the dataset. While the original binaries, whether benign or malicious, may be sourced in ways similar to other categories (e.g. from clean Windows installations or malware repositories), the defining feature here is that packing is manually or programmatically applied or modified for experimental purposes.

3.1.5 Industry Contributions

Some datasets are obtained through collaborations with antivirus vendors, threat intelligence companies, or industry research groups. These may offer expert labeling, proprietary metadata, or large-scale and highly varied samples. However, they are often not publicly accessible, making replication and third-party validation difficult. The following shows the few references we could find on this topic. The studies listed here rely on datasets contributed by security vendors or industry partners, often providing access to large-scale or specialized collections.

2008	PE File Header Analysis-Based Packed PE File Detection Technique	Choi *et al.*
2010	Pattern Recognition Techniques for Classification of Malware Packer	Sun *et al.*
2014	Instructions-Based Detection of Sophisticated Obfuscation and Packing	Saleh *et al.*
2019	Effective, efficient, and robust packing detection and classification	Biondi *et al.*
2020	Open Dataset for Learning based Temporal Analysis of PE Malware	Yang *et al.*
2021	Analysis of Machine Learning Approaches to Packing Detection	Van Ouytsel *et al.*

Table 3.5: Timeline of related works using Industry contributors

[14] https://www.softpedia.com

In 2008, Choi *et al.* [37] published a method for detecting packed PE files based on analyzing anomalies in PE header structures. Their dataset was constituted of 1027 PE files where 437 were packed and 590 non-packed. The files where gathered from a Windows XP system from internet downloads (no specific source mentioned). Additional samples containing viruses and computer worms were provided by an antivirus vendors.

In 2010, Sun *et al.* [38] proposed a simple, fast, and effective packer classification framework that applies pattern recognition techniques to automatically extracted randomness profiles. The dataset used in their study consisted of 17336 real malware samples, each labeled with its corresponding packer type. These samples were prepared by a third party known as Computer Associates (CA) Threat Management Team from Melbourne, Australia. These samples were collected over January and February 2009 and labeled using multiple antivirus scanners and tools like PEiD.

In 2014, Saleh *et al.* [39] published a technique for detecting sophisticated packing and obfuscation by analyzing instruction-level features. They use several set of files for their experiments. The first one was created by collecting 250 clean files from a clean Windows XP machine. The second set was constitute of 250 malicious packed files, packed by both commercial and custom packers but no specific packers were mentioned. They also collected 423 clean non-obfuscated files from a clean Windows 7 32-bit Home Basic Edition. The last one, the largest set, was provided by a security firm and was composed of 10171 malicious files but no details about packing or obfuscation were provided.

In 2019, Biondi *et al.* [34] published a study on how to train packing detection and classification algorithms based on machine learning to be both highly effective and efficient. They constructed a large dataset of 281,344 malware samples, collected from file streams between February and October 2017. These samples were obtained via collaboration with Cisco Systems and academic partners.

In 2020, Yang *et al.* [40] released BODMAS, an open PE malware dataset. The goal of this dataset is to facilitate research efforts in machine learning based malware analysis. Their dataset contains a total of 134435 samples, 57293 malware and 77142 benign. The malware were collected randomly from a security companys internal malware database during a while year, from 29-08-2019 to 30-09-2020. The benign sample were collected from the same database from 01-01-2007 to 30-09-2020. This work could also fall under the category of already existing dataset.

In 2021, Bertrand Van Ouytsel *et al.* [41] published a paper exploring different machine learning approaches to understand which features and algorithms were the most efficient, significant and economical. They created two different dataset to conduct their experiment, both of them from samples provided by Cisco. The first dataset had 43092 samples collected between June and July 2019; the second included 95876 samples collected from October 2019 to February 2020. Both datasets had a packed-to-unpacked ratio of approximately 1:3.

Industry-sourced datasets can offer depth, diversity, and expert annotations, though limited public access may restrict reproducibility or peer validation.



While we categorize prior works based on the primary method of dataset construction, it is important to note that many research efforts could reasonably span multiple categories. For instance, a dataset may be partially collected from public repositories but augmented with synthetic samples or enriched with vendor-provided data. In this survey, we classify works according to their dominant dataset creation method for clarity, but acknowledge that boundaries between categories are often blurred.

3.2 Quality Assessment

The construction of high-quality datasets is essential to the development of effective and reliable malware detection systems. However, creating a dataset is only the first step; ensuring and validating its quality is equally critical. Poor data quality can lead to misleading results, biased models, and reduced generalizability in real-world scenarios. As such, the field of data quality assessment has emerged as a key research area across many domains.

This section surveys the state of the art in dataset quality assessment, identifying the most prominent research directions. Three main categories are discussed, for each, a brief conceptual introduction is followed by a timeline of relevant works.

3.2.1 Data Quality Metrics

The first major direction in Data Quality (DQ) assessment research centers around the definition and interpretation of quality metrics. These works aim to answer the foundational question: *What does it mean for a dataset to be of "**high quality**"?*

Researchers in this area have proposed various **dimensions** (such as accuracy, completeness, consistency, timeliness, and reliability) to describe the quality of a dataset from different perspectives. Some metrics are objective and quantifiable, while others depend on subjective judgment or use case context. This line of work forms the theoretical basis for all later approaches in quality assessment.

The following timeline outlines key contributions that introduced and formalized core data quality metrics over the last three decades:

1996	Beyond Accuracy: What Data Quality Means to Data Consumers	Wang *et al.*
2002	Data Quality Assessment	Pipino *et al.*
2009	Methodologies for Data Quality Assessment and Improvement	Batini *et al.*
2015	The Challenges of Data Quality and Data Quality Assessment	Cai *et al.*
2023	Data Quality for Software Vulnerability Datasets	Croft *et al.*
2024	A Framework for Current and New Data Quality Dimensions	Miller *et al.*
	The Five Facets of Data Quality Assessment	Mohammed *et al.*

Table 3.6: Timeline of related works using quality metrics

In 1996, Wang *et al.* [42] laid the foundation for modern DQ assessment by introducing a consumer-oriented perspective. They categorized data quality into four main **categories**, each comprising specific **dimensions**: Intrinsic (e.g. accuracy, objectivity), Contextual (e.g., relevance, timeliness, completeness), Representational (e.g. interpretability, ease of understanding), and Accessibility (e.g., access security). This model provided one of the first practical and structured frameworks for measuring, analyzing, and improving DQ based on how data is used and perceived by end users.

In 2002, Pipino *et al.* [43] built directly upon the foundational **dimensions** proposed by Wang *et al.* in 1993 by emphasizing the dual nature of data quality assessment: it must account for both objective measures (e.g., completeness, accuracy) and subjective perceptions (e.g., relevance, interpretability). They argued that a data value could be technically correct yet still considered poor quality by users if it lacks meaning or usefulness in context. This work helped refine the conceptual framework for data quality by showing that evaluation must reflect user needs, expectations, and situational context, not just intrinsic properties of the data.

In 2009, Batini *et al.* [44] advanced the field by proposing concrete methodologies for assessing and improving data quality across real-world systems. Building on earlier frameworks that defined what quality means (Wang *et al.*, 1993) and how to evaluate it (Pipino *et al.*, 2002), this work focused on operationalizing data quality assessment through structured phases: data profiling, rule-based evaluation, metric aggregation, and iterative improvement. It emphasized the importance of context-aware quality requirements, integration into data governance, and the continuous nature of quality management. This marked a shift from theoretical understanding toward scalable, actionable practices.

In 2015, Cai *et al.* [45] extended the conversation around data quality into the big data era, where traditional assessment techniques encounter scalability and complexity limitations. Building upon the structured methodologies of Batini *et al.* (2009), this work examined how volume, velocity, variety, and veracity -the defining characteristics of big data- pose new challenges to measuring and ensuring quality. The authors highlighted the need for automated, real-time, and scalable approaches, and emphasized that existing models often fail to accommodate unstructured data, dynamic schemas, and incomplete ground truth. This paper marked an important evolution, pushing the field toward more adaptive quality assessment techniques.

In 2023, Croft *et al.* [46] examined the inherent data quality of software vulnerability datasets, focusing on five key attributes from the ISO/IEC 25012 framework: **Accuracy**, **Uniqueness**, **Consistency**, **Completeness** and **Currentness**. Their analysis revealed severe duplication issues, 17% to 99% of records, were duplicates-causing data leakage and artificially inflated performance in predictive models. While their study centers on textual vulnerability records rather than binary executables, the findings are equally relevant to packed binary datasets, where duplicate or near-duplicate samples can similarly distort evaluation results. The authors emphasize the critical **importance of deduplication**, as duplicates can lead to inflated model performance, a concern that can be addressed in binary datasets through techniques such as similarity search.

In 2024, Miller *et al.* [47] revisited existing data quality taxonomies and proposed a revised framework of dimensions tailored to modern data-intensive environments. Building on the original Wang & Strong model, they mapped dimensions from standards like ISO/IEC 25012 and proposed new qualities such as **trust**, **governance**, and **usefulness**, which had previously been overlooked. This work bridged the gap between traditional DQ concepts and emerging needs in machine learning, AI, and large-scale data analytics, reinforcing the necessity of expanding and adapting quality definitions as usage contexts evolve.

In 2024, Mohammed *et al.* [48] introduced a new model for Data Quality assessment based on five key facets: **Data**, **Source**, **System**, **Task**, and **Human**. Each facet includes specific quality attributes, allowing for a more complete and flexible way to evaluate DQ. This model builds on earlier frameworks by combining ideas from both traditional metrics (like **accuracy** and **completeness**) and newer concerns (like **trust** and **relevance** to specific tasks). It reflects a shift toward a more comprehensive and practical approach to assessing whether data is truly useful and reliable in different real-world settings.

Over time, data quality assessment has moved from basic checks to more flexible and user-focused approaches. As data has become bigger and more complex, the ways we measure its quality have had to adapt to stay useful and relevant.

3.2.2 Automated Tools

As datasets grow in size and complexity, automated tools have become indispensable for efficient data quality assessment. While many are general-purpose, some frameworks are designed specifically to address issues relevant to classification tasks such as label noise, feature inconsistencies, or preprocessing standardization. This section presents tools that, while not originally designed for binary executables, offer useful ideas or techniques that can help improve the quality of datasets.

2021	Automatic assessment of DQ and remediation for ML datasets	Gupta *et al.*

2023	Automated data preprocessing framework for supervised ML	Santos *et al.*
2025	ML-Oriented Interactive Tabular DQ Dashboard	Abdelaal *et al.*

Table 3.7: Timeline of related works proposing assessment tools

In 2021, Gupta *et al.* [49] presented the Data Quality Toolkit, an extensible platform for automated data quality assessment aimed at machine learning pipelines. The system purpose is to detect, explain and remediate issues in the data. The toolkit places particular emphasis on pipeline integration and usability for data scientists. Although designed primarily for tabular or image classification datasets, the principles of automatic validation and cleaning are relevant to packed classification pipelines, where mislabeled or ambiguous samples are common.

In 2023, Santos *et al.* [50] proposed Atlantic, an automated preprocessing open-source Python package for supervised machine learning. It integrates data cleaning steps such as missing value handling, normalization, feature selection, and encoding, all customizable and pipeline-friendly. While general purpose, its focus on ML-ready data pipelines makes it a strong candidate for integration in workflows where binary features are extracted.

In 2025, Abdelaal *et al.* [51] introduced DataLens, a dashboard that allows interactive dataset profiling, error detection, and repair. It allows users to explore quality issues visually and manually validate or correct the systems suggestions, integrating human feedback into the process. Although originally applied to tabular data, its support for custom metrics and real-time feedback makes it potentially adaptable to binary datasets especially when working with extracted features from executables.



The tools discussed in this section were primarily developed for structured or general-purpose machine learning datasets. While they were not evaluated directly on binary executable datasets or packing detection tasks, they are included here to highlight existing solutions and inspire transferable ideas for data quality assessment in our context.

3.2.3 Impact on Machine Learning

While theoretical frameworks and tools provide the means to assess and improve data quality, empirical studies offer insights into how data quality directly affects machine learning performance. This category examines research that quantifies the impact of various data quality dimensions on model outcomes, emphasizing the practical implications of data quality in real-world applications.

2017	Effects of Dataset Characteristics on the Performance	Oreski *et al.*
2021	Estimating the Extent of the Effects of DQ	Foroni *et al.*
2022	How Important is Data Quality?	Morán-Fernández *et al.*
2025	The Effects of DQ on Machine Learning Performance	Mohammed *et al.*

Table 3.8: Timeline of related works on the impact of quality

In 2017, Oreski *et al.* [52] conducted a large-scale empirical study exploring how different dataset characteristics influence the effectiveness of feature selection algorithms. The authors emphasizes that DQ, reflected through **noise**, **sparsity**, **correlation**, and **homogeneity**, plays a vital role in the success of feature selection techniques. The study provides empirical evidence that no single feature selection technique is

universally best; instead, the optimal choice depends on the specific data quality and characteristics of the dataset. Their findings show that poor Data Quality can reduce the benefit of even the best feature selection methods, reinforcing the importance of curating high-quality datasets for downstream performance in ML tasks.

In 2021, Foroni *et al.* [53] examined how different types of data defects (e.g., missing values, duplicates, inconsistent labels) affect ML models across a variety of real-world datasets. They proposed a method for estimating performance degradation without modifying the dataset itself. This work supports the idea that quality issues can be quantified and predicted, offering a foundation for prioritizing data cleaning efforts in ML pipelines.

In 2022, Morán-Fernández *et al.* [54] empirically evaluated whether improving DQ (through better features and labels) or using more advanced classifiers had a greater impact on **accuracy**. Their findings showed that **data quality** had a stronger effect on model performance than classifier choice, especially in noisy or imbalanced datasets. This highlights the practical importance of dataset reliability where false positives can be costly.

In 2025, Mohammed *et al.* [55] presented a comprehensive benchmark examining how different DQ issues (e.g., label noise, missing values, skewed distributions) impact a wide range of ML algorithms. They found that **label noise** and **outliers** consistently had the most negative effects, and that simpler models were often more robust to poor-quality data than complex ones. While the study focused on tabular data, the results are transferable to any ML pipeline, including static analysis of binaries where features may be sparse, noisy, or mislabeled.

These empirical studies collectively demonstrate that Data Quality plays a decisive role in the effectiveness of machine learning models, often outweighing the algorithm choice itself. Issues such as **label noise**, **imbalance**, and **feature inconsistency** can significantly degrade performance, emphasizing the need for careful dataset construction and validation.

3.3 Similarity

Building on the similarity concepts defined in Section 2.4, this section reviews relevant academic literature that applies those methods to the analysis of packed binaries and their use in dataset preparation.

3.3.1 CTPH Applications

This category includes studies that focus on Context-Triggered Piecewise Hashing. Many of these works aim either to evaluate and compare existing CTPH algorithms or to introduce new methods.

2015	Experimental Study of Fuzzy Hashing in Malware Analysis	Li *et al.*
2018	Lempel-Ziv Jaccard Distance, an Alternative to ssdeep and sdhash	Raff *et al.*
	Beyond Precision and Recall: Understanding Uses of Similarity Hashes	Pagani *et al.*
2019	Improving file-level fuzzy hashes	Shiel *et al.*

Table 3.9: Timeline of related works using ctph

In 2015, Li *et al.* [56] conducted an extensive evaluation of fuzzy hashing tools-ssdeep, sdhash, and mvHash-B-on millions of malware binaries to assess their effectiveness in clustering and deduplicating samples in realistic operational environments. They analyzed the internal behavior of each algorithm and evaluated their performance with respect to scalability, precision, and sensitivity to content modifications.

The results showed that ssdeep, while fast and compact, is overly sensitive to minor byte-level changes such as those introduced by packing. No single fuzzy hashing tool consistently offered optimal performance across different malware families or use cases. To address these shortcomings, the authors proposed a conceptual design called NextGen-hash, which aims to improve block alignment and matching robustness by combining strengths from existing approaches. However, this proposal was not implemented or released as a tool, and therefore cannot be experimentally tested. The authors emphasized that while fuzzy hashing has value for tasks such as malware triage and clustering, it should be applied thoughtfully, with awareness of its limitations and variability across datasets.

In 2018, Raff *et al.* [57] introduced Lempel-Ziv Jaccard Distance as a more accurate and efficient alternative to popular fuzzy hash algorithms such as ssdeep and sdhash. The goal of their work was to improve the detection of similarity between binary files, especially in cases where files may have been modified, packed, or obfuscated. LZJD works by breaking each file into a set of unique substrings using the Lempel-Ziv 77 compression algorithm, and then measuring similarity based on the Jaccard index-the proportion of shared substrings between two files. To evaluate their method, the authors compared LZJD with ssdeep and sdhash across several tasks, including malware family classification, file fragment matching, and general binary similarity detection. Their results showed that LZJD consistently provided better matching accuracy than both ssdeep and sdhash, particularly for smaller files and structurally varied data. While LZJD's hashing speed and digest size were not as minimal as ssdeeps, it outperformed sdhash in both efficiency and compactness. Overall, LZJD strikes a balance between performance and accuracy, offering a robust and scalable solution for similarity detection in binary datasets, especially when minor file changes or packing artifacts are involved.

Also in 2018, Pagani *et al.* [58] conducted a rigorous empirical and theoretical evaluation of four fuzzy hashing algorithms: ssdeep, sdhash, TLSH, and mrshv2, across multiple binary analysis scenarios, including library identification, compiler variation, and executable version comparison. They found that minor structural changes such as a single instruction shift can drastically affect similarity scores in ssdeep, often reducing it to zero. The comparative results showed that no single fuzzy hashing method consistently delivers accurate similarity detection across all contexts. Overall, methods like sdhash and TLSH frequently outperformed the de facto standard ssdeep, particularly in code-level similarity tasks. The authors argue that the choice of fuzzy hashing technique should be driven by the specific binary comparison goal, as there is no universally optimal algorithm.

In 2019, Shiel *et al.* [59] conducted a detailed study evaluating the effectiveness of section-level fuzzy hashing for malware variant classification in Windows PE files. By splitting executables into headers and sections and applying a similarity digest separately to each chunk, they significantly improved accuracy, precision, and recall relative to traditional file-level hashing. The experiments and framework presented in the paper specifically used ssdeep for generating and comparing similarity hashes at both the file and section levels. Their findings show that section-wise hashing is far more robust against obfuscation and structural rearrangements, making it a promising strategy for dataset deduplication and malware family clustering in packed binary collections. However, the authors also noted that this approach introduces additional computational overhead – particularly in the comparison phase – due to the need to evaluate similarity across multiple sections rather than a single file-level hash.



Although this section focuses on CTPH, LZJD is not a CTPH method. It was included since it was evaluated against CTPH algorithms such as ssdeep.
Later in this work, we compared multiple fuzzy hashing methods including for instance ssdeep and LZJD using their available Python implementations. Concerning NextGen-hash, it was not included, as it remains a conceptual proposal without a public implementation.

3.3.2 LSH Applications

This second category includes research that exploresLocality Sensitive Hashing techniques. These works typically focus on selecting the most suitable LSH scheme for specific tasks and comparing its performance against other similarity hashing methods.

2014	In Defense of MinHash Over SimHash	Shrivastava *et al.*
	Locality-Sensitive Hashing Optimizations	Oprisa *et al.*
2023	An Evaluation of Malware Triage Similarity Hashes	Liu *et al.*

Table 3.10: Timeline of related works using LSH

In 2014, Shrivastava *et al.* [60] conducted a theoretical and empirical comparison between MinHash and SimHash, two widely used Locality Sensitive Hashing schemes for similarity detection. The paper aims to answer the important practical question of which method is preferable for near neighbor search when the data are binary, a question that had not been conclusively addressed in prior literature. The authors use six large-scale binarized datasets with predefined query and training partitions. Their results demonstrated that MinHash consistently outperforms SimHash in providing accurate and reliable similarity estimations, even in scenarios where cosine similaritySimHash's intended strengthshould theoretically excel. These findings show MinHash as a more robust choice for set-based similarity tasks, including deduplication and clustering in binary data domains.

In 2014, Opria *et al.* [61] proposed an optimized malware clustering approach using LSH to efficiently group similar binaries at large scale. Their method uses MinHash-based LSH to estimate Jaccard similarity between malware samples, which allows the system to avoid comparing every file with every other file directly. This greatly reduces the time and resources needed to process large binaries collections. Each binary is represented as a set of features (such as byte sequences or instruction patterns), which are converted into MinHash signatures. LSH is then used to quickly find pairs of similar files, which are grouped using single-linkage hierarchical clustering. The authors showed that their system achieves clustering results nearly identical to exact methods, but runs much faster and can scale to millions of samples. However, their runtime evaluation focuses solely on the clustering phase, assuming that feature sets have already been extracted. The computational cost of the initial feature extraction processsuch as generating shingles or n-grams from raw binariesis not detailed, leaving the preprocessing overhead unquantified. While the paper focuses on clustering, the LSH-based similarity detection they use is also well-suited for similarity search.

In 2023, Liu *et al.* [62] evaluated two widely used similarity hashing methods: TLSH and LZJD. The study systematically compared both methods in terms of detection accuracy, error rates, and runtime performance, aiming to determine which approach offers the best balance between efficiency and similarity detection capability.

A key finding of the study was the significant performance gap in runtime efficiency: TLSH completed both indexing and similarity search operations in under one second, while LZJD required several minutes for equivalent tasks on datasets of similar size. This substantial difference is attributed to TLSH's lightweight digest and faster distance computations, whereas LZJDs substring-set comparison, while more accurate, incurs heavier computational overhead.

3.4 Literature Review

From our literature review, we could identify *trends* based on the research threads we showed in Sections 3.1 and 3.2, *limitations* for some of these approaches and *opportunities* resulting from gaps we could see through these limitations. This section consecutively enumerates each of them.

3.4.1 Trends

Among the literature parsed here before, we could identify some trends regarding the technique for composing datasets and assessing data quality:

- **Windows installations**: We can observe the reliance on clean installations of Windows operating systems as sources of benign executables [5–7, 14, 22, 27, 28, 31, 32, 39]. These environments are often freshly installed, providing a controlled and reproducible baseline for collecting benign software samples. While this approach ensures a clean separation from potentially unwanted or malicious software, it may introduce a bias by omitting the diversity found in real-world benign executables and hinder the generalization of detection models trained on these datasets.
- **Data Quality importance**: Recent studies [54, 55] have demonstrated that the quality of a dataset often outweighs the quality of the machine learning model applied to it. Key issues like label noise, missing values, skewed distributions, and inconsistent feature extraction can all degrade model reliability. This has led to a growing recognition of DQ as a first-order concern in ML workflows.
- **Theory to Practice**: Data Quality research has evolved from abstract frameworks to practical, scalable methods. Early work [42, 43] defined core quality dimensions but lacked implementation guidance. Later studies [44] introduced structured workflows to assess and improve data quality in real systems. These methods are adapted to modern challenges such as big data and real-time analytics. This reflects a broader shift toward making data quality both actionable and relevant in todays complex data environments.
- **Common packers across studies**: As shown in Table 3.11, a handful of packers like UPX, ASPACK, or PECompact dominate the literature. This reflects a focus on widespread or easy-to-identify packers, potentially neglecting more advanced or custom packing methods increasingly used in modern malware.
- **Growing Recognition of ssdeep Limitations**: Recent research [56–58] has highlighted the practical limitations of commonly used fuzzy hashing algorithms, particularly ssdeep, which is based on CTPH. These methods have shown high sensitivity to structural noise introduced by packing, encryption, or simple obfuscation, leading to unreliable similarity scores across binary variants. This trend reflects a growing recognition that traditional fuzzy hashing techniques, while still useful for initial triage, are no longer sufficient for deeper analysis or dataset-level operations without refinement or augmentation.
- **MinHash when using LSH**: Both studies [60, 61] reinforce the trend that MinHash is the preferred choice for similarity detection in binary datasets when using LSH due to its higher accuracy, better theoretical properties for set similarity, and proven practical performance in large-scale deduplication and clustering scenarios. While SimHash remains useful for certain text or vector-space applications, it is less reliable when applied to binary data where set overlap (Jaccard similarity) is the more meaningful similarity metric.

3.4.2 Limitations

- **Creation methodologies**: There is an absence of a standardized methodology for dataset construction. Each study typically defines its own protocol for collecting, labeling, and processing samples. Criteria for labeling executables as "benign", "malicious", or "packed" can vary significantly across studies, and the choice of packers, file types, or execution environments is left to author discretion.

Packer	Royal (2006)	Shafiq (2009)	Shabtai (2012)	Lim (2015)	Perdisci (2008)	Sun (2010)	Ugarte-Pedrero (2012)	Naval (2012)	Burgess (2014)	Hubballi (2016)	Saleh (2017)	Choi (2019)	Aghakhani (2020)	Noureddine (2021)	Subrahmanyam (2023)	Alkhateeb (2024)
UPX	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
ASPACK	✓	✓	–	–	✓	–	–	✓	✓	✓	–	✓	–	✓	–	✓
PECompact	✓	✓	–	–	?	✓	–	✓	–	–	–	✓	✓	✓	–	✓
NsPack	–	–	–	–	?	✓	–	✓	–	–	–	✓	–	✓	–	✓
FSG	✓	–	–	–	?	✓	✓	–	–	–	✓	✓	–	✓	–	–
Themida	–	–	✓	–	✓	–	✓	–	✓	–	✓	–	✓	✓	–	–
PEtite	–	–	–	–	?	✓	–	✓	–	–	–	✓	✓	✓	–	–
tElock	–	–	–	–	?	–	✓	–	✓	✓	–	–	✓	✓	–	–
RLPack	–	–	–	–	?	–	✓	–	✓	✓	–	✓	–	–	–	–
Obsidium	✓	–	–	–	✓	–	–	–	✓	✓	–	–	✓	–	–	–
MEW	✓	–	–	–	?	✓	✓	–	–	✓	–	–	–	✓	–	–
PELock	–	–	–	–	?	–	–	–	–	✓	–	–	✓	✓	–	–
eXpressor	–	–	–	–	?	–	–	–	–	–	✓	✓	–	✓	–	–
Yoda	✓	–	–	–	?	–	–	–	–	–	✓	✓	–	✓	–	–
MPRESS	–	–	–	–	?	–	–	–	–	–	–	✓	✓	✓	–	–
Packman	–	–	–	–	?	–	✓	–	–	–	–	✓	–	✓	–	–
exe32pack	–	–	–	–	?	–	–	✓	–	–	–	✓	–	–	–	–
ASProtect	✓	–	–	–	?	✓	✓	–	–	–	–	–	–	✓	–	–
Armadillo	✓	✓	✓	–	?	–	–	–	–	–	–	–	–	–	–	–

Table 3.11: Packers Mentioned or Used in Reviewed Papers

- **Non-specialized datasets**: Benchmark datasets such as EMBER or Malicia are frequently repurposed for tasks like packer detection or obfuscation classification, even though they were originally developed for broader malware classification tasks. As a result, these datasets may lack the annotations, class balance, or diversity needed for effective evaluation of unpacking or clustering approaches.

- **Weak or unverified labels**: Many studies rely on unlabeled samples from online repositories primarily focused on malware, further compromising the reliability of ground truth. Without manual validation or robust analytical methods, the labels obtained through basic detection techniques can mislead model training and introduce bias into the results.

- **Over-reliance on PEiD**: The Table 3.12 summarizes the packer detection tools mentioned or used across the reviewed literature. It includes both standalone packer detectors that offer built-in packer detection capabilities. Despite the availability of multiple modern tools for packer detection , the literature overwhelmingly relies on PEiD, a legacy tool whose signatures have not been actively maintained in recent years. Its widespread use-spanning nearly two decades of research-introduces the risk of outdated or inaccurate packer identification, particularly as newer or custom packers may evade its detection.

- **Lack of Tools for Binary Data**: Despite the growing importance of data quality assessment, most available tools and frameworks are designed for structured or tabular data, leaving a significant gap in support for binary datasets. General-purpose tools like Atlantic or DataLens offer valuable inspiration but lack the ability to parse, analyze, or visualize low-level binary formats.

- **Lack of Attention to Preprocessing Overhead**: While several of these papers propose improved similarity hashing techniques or efficient approximate search algorithms [61, 62], none provide detailed measurements or analysis of the preprocessing costssuch as feature extraction, shingle generation, or MinHash digest computation. Opria *et al.* explicitly report their clustering runtime assuming features are already extracted, leaving the overall end-to-end performance impact of large-scale deduplication workflows partially unaddressed.

Detector	Royal (2006)	Choi (2008)	Perdisci (2008)	Shafiq (2009)	Han (2009)	Sun (2010)	Wang (2011)	Ugarte-Pedrero (2012)	Shabtai (2012)	Naval (2012)	Burgess (2014)	Lim (2015)	Hubballi (2016)	Saleh (2017)	Choi (2019)	Aghakhani (2020)	Nourredine (2021)	Alkhateeb (2024)
PEiD	✓	✓	✓	✓	✓	✓	✓	✓	–	✓	✓	✓	✓	✓	✓	✓	✓	✓
F-Prot	–	–	✓	–	–	–	–	–	–	–	–	–	–	–	–	✓	–	–
Kaspersky AV	–	–	–	–	–	✓	–	–	✓	–	–	–	–	–	–	–	–	–
exeInfo	–	–	–	–	–	–	–	–	–	✓	–	–	–	–	–	✓	–	–
yara	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	✓	✓	–
DIE	–	–	–	–	–	–	–	–	–	✓	–	–	–	–	–	–	✓	–
RDG	–	–	–	–	–	–	–	–	–	✓	–	–	✓	–	–	✓	–	–
ProtectionID	–	–	–	✓	–	–	–	–	–	✓	–	–	–	–	–	–	–	–
MRC	–	–	–	–	✓	–	–	–	–	–	–	–	–	–	–	–	–	–
Microsoft AV	–	–	–	–	–	✓	–	–	–	–	–	–	–	–	–	–	–	–
CA AV	–	–	–	–	–	✓	–	–	–	–	–	–	–	–	–	–	–	–
Manalyze	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	✓	–	–
Packerid	–	–	–	–	–	–	–	–	–	–	–	–	✓	–	–	–	–	–
Qunpack	–	–	–	–	–	–	–	–	–	–	–	–	✓	–	–	–	–	–
PHAD	–	✓	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–

Table 3.12: Detector Mentioned or Used in Reviewed Papers

3.4.3 Opportunities

- **Standardization of detection tools**: Developing or adopting a shared benchmark for packer detection tools could reduce discrepancies across studies and improve reproducibility. Future work may focus on establishing consensus toolchains or combining multiple detectors for higher label confidence.
- **Diverse and modern packers**: Research can benefit from targeting newer or less commonly studied packers (e.g., custom obfuscators, virtualization-based packers) to reflect evolving malware behaviors. This would address the overrepresentation of older, well-known packers like UPX.
- **Assessing Quality in Binary Datasets**: While data quality is increasingly recognized as a critical factor in data-driven workflows, most existing tools are designed for structured or textual data, leaving binary datasets without adequate support. There is a clear need for methods that can evaluate and clean binary data by considering its unique characteristics. Extending data quality assessment to binaries through well-established data quality dimensions such as accuracy, completeness and relevance would address a significant methodological gap and enable more reliable use of binary data in large-scale analytical pipelines.

- **Comparative Evaluation of Similarity Techniques**: While existing studies often emphasize the accuracy and runtime of similarity search methods, the computational costs of the preprocessing phase such as feature extraction, shingle generation, and hash digest computation-are rarely quantified or compared across techniques. Given the diversity of methods explored in the literature there is an opportunity to conduct a comprehensive evaluation that explicitly measures and contrasts the preprocessing overhead and similarity search efficiency of these methods. Since deduplication is a critical step in ensuring dataset quality, such a comparison would directly inform the selection of the most suitable technique for large-cale binary datasets.



Table 3.11 focuses on packers referenced in multiple papers but several packers were only mentioned in a single study and therefore are not included in it. These include packers such as Aegis, kkrunchy, WinUpack, Neolite, UPACK, PESpin, FishPE, Shrinker, AutoIt, PCGuard, UPX-Scrambler, Morphine and ActiveMARK. The `?` symbol for Perdisci (2008) [14] indicates that the paper did not specify exactly all the packers that were used, it only mentioned that a total of 17 packers were involved.



While this analysis highlights commonly referenced packers and detectors, it does not aim to provide an exhaustive review of all existing tools. For a more comprehensive survey and deeper technical discussion of various packing techniques and detection tools, we refer the reader to the work of D'Hondt [3]. This study offers broader coverage and more detailed descriptions of many packers and detectors.

SUMMARY

The literature reviews has showed that the **construction** of datasets for packed malware detection typically relies on clean Windows installations to gather benign executables and on public malware repositories to source malicious samples. While this ensures controlled and reproducible data, it often leads to limited diversity and biases that can reduce a models generalizability. Studies also frequently rely on widely known packers like UPX and ASPack to label samples, which, while practical, neglects more advanced or custom packing techniques increasingly seen in real-world threats. Across the literature, there is still no standardized methodology for dataset construction; researchers define their own protocols for collecting, labeling, and processing samples, which complicates reproducibility and comparability.

Parallel to dataset creation, the field of data **quality assessment** has evolved significantly over the past three decades. Early works established foundational dimensions like accuracy, completeness, and consistency, framing data quality both as an intrinsic property and as something shaped by context and user perception. Over time, this theoretical groundwork gave rise to operational methodologies and, later, to tools designed to automate and scale quality assessmentespecially in the face of big data and dynamic environments. More recent contributions have expanded classical taxonomies to include trust, governance, and usefulness, reflecting modern challenges in machine learning and AI. Although these toolssuch as DataLens or Atlanticare primarily built for tabular data, they highlight principles that could inspire quality-focused workflows for binary datasets.

An important dimension that emerges from recent studies is the role of **similarity detection** in enhancing dataset quality, particularly for deduplication and variant identification. Traditional fuzzy hashing methods like ssdeep and SDhash, while still widely used, have been shown to lack robustness against packing-induced changes and structural noise. This has led to a growing interest in other methods, such as LZJD, TLSH or LSH techniques. However, a limitation persists in the literature: the computational overhead of preprocessing steps, including feature extraction and digest computation, is often overlooked in performance evaluations. Addressing this gap through comprehensive comparisons of preprocessing and search-phase costs across different methods could inform more efficient dataset curation workflows.

Empirical studies reinforce that **data quality issues**, including label noise, missing values, and skewed distributions, often impact model performance more than the choice of learning algorithm itself. This underscores the importance of careful dataset design, consistent labeling, and robust validation, especially in security contexts where false positives or undetected threats carry significant consequences. Yet, the literature still shows overconfidence on legacy tools like PEiD for packer detection.

Overall, the field has moved from abstract definitions of what makes data "*high quality*" to practical, sometimes automated approaches to ensure it, but significant gaps remain. There is a clear **opportunity to develop specialized tools for binary data**, diversify datasets with modern or custom packers, and standardize protocols for building and labeling datasets. Addressing these challenges could make packing detection not only more reproducible but also better aligned with evolving malware practices and the complex nature of real-world data.

4 Framework

THIS chapter presents the framework used to conduct the experiments: the Packing Box.

It provides a comprehensive toolkit for analyzing executable packing. The chapter begins by describing the overall architecture of the framework, followed by the Machine Learning workflow it supports. It also details the various tools integrated at the different levels. Finally, the chapter outlines the contributions made to assess and improve dataset quality.

4.1 Packing Box 40
4.1.1 Architecture 40
4.1.2 Abstraction 41
4.1.3 Machine Learning Pipeline . 41
4.2 Dataset Creation Enhancements . 43
4.2.1 Archive Filtering Tool 44
4.2.2 Similarity Threshold 45
4.3 Dataset Quality Scoring 45
4.3.1 Generic Metrics 46
4.3.2 Specific Metrics 46
4.3.3 Dataset Assessment 48
4.3.4 Dataset Quality Improvement 50
Summary 52

Objectives

A – Show the tools from the Packing Box that will be used for experiments.

B – Present the additions designed to improve Packing Box we integrated.

C – Present the quality assessment metrics we added to the framework.

4.1 Packing Box

4.1.1 Architecture

The Packing Box [2], developed by D'Hondt *et al.* [3], was created to tackle some of the limitations and explore the opportunities identified in their state-of-the-art review. This experimental toolkit provides a command-line environment packaged in a Docker container, including executable analyzers, packing detectors, packers and unpackers, as well as tools for dataset generation and manipulation, executable structure visualization, and model training and testing. Its main strength lies in being open source, making it easy to adapt, extend, and contribute to, while bundling a range of commonly used utilities.

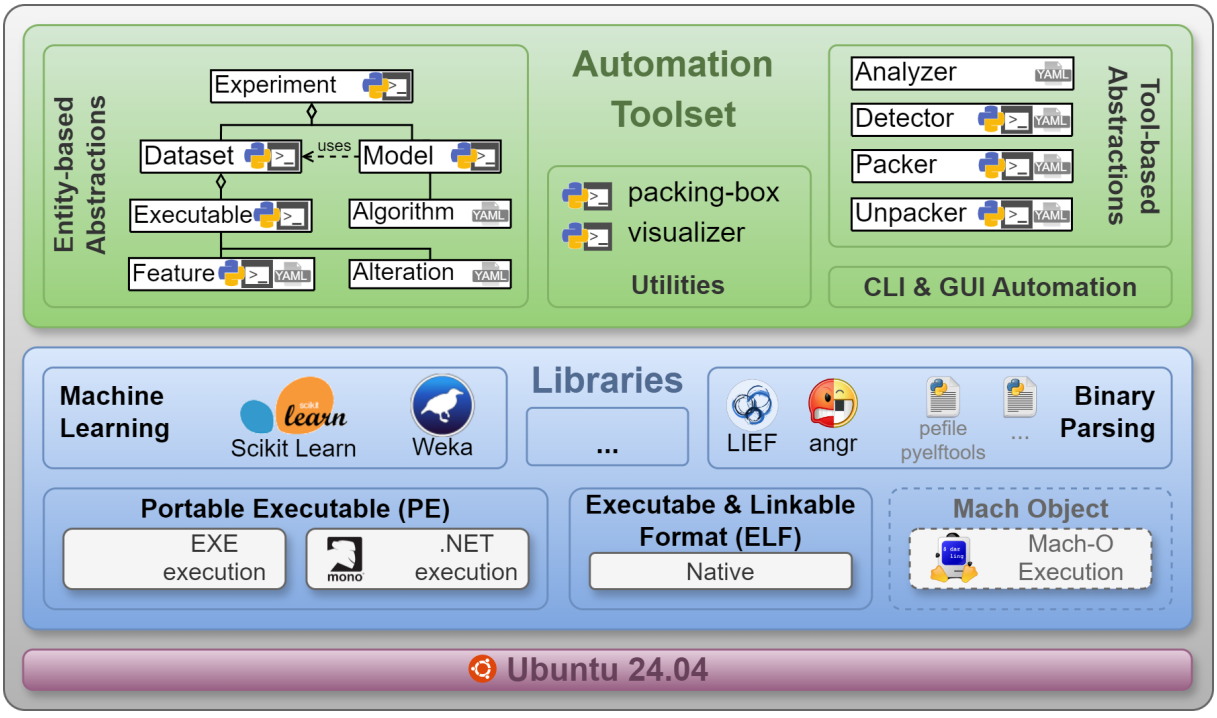


Figure 4.1: Architecture of Packing Box tool [3]

The architecture of the Packing Box tool is shown in Figure 4.1. We can see that the solution is built upon Ubuntu 24.04, with support for multiple executable formats including PE, ELF, and Mach-O, enabled through relevant libraries and tools. Above this foundation lies an automation layer, providing a comprehensive toolset for command-line operations such as packing, unpacking, detection, and analysis. At this layer, we can find abstractions, which organizes entities and tools in a modular way, allowing new executable formats or tools to be integrated easily and consistently without modifying the underlying workflow. This modular and layered architecture provides the flexibility required to incorporate new features, as described in the following sections.

4.1.2 Abstraction

As shown in the architecture overview , the Packing Box ensures modularity and scalability through a system of abstractions that defines how data and processes are structured within the framework. These abstractions encompass *executables*, *datasets*, and *models*, ensuring a consistent way to store and organize files, apply processing steps and integrate with the framework's Machine Learning components.

Executables allows the framework to support multiple file formatscurrently PE and ELF, with Mach-O planned for future integration. Executable objects encapsulate both basic file properties and computed values, including attributes such as the executable format, original file path, creation and modification timestamps, and cryptographic hashes like SHA-256. This abstraction simplifies interactions with binaries by exposing a consistent structure, regardless of their underlying format. Having an abstraction layer for executable enables format-agnostic operations across the toolchain. Certain tasks, such as computing quality metrics like portability, require format-specific logic. Through this abstraction, each file format can implement its own behavior internally, while higher-level components can simply invoke a unified method without needing to know the format details. This design promotes modularity and extensibility, making it easy to support additional formats or behaviors in the future.

Datasets within the Packing Box follow a standardized folder structure, which can slightly vary depending on the dataset format. Two formats are supported:

- `Dataset`: This format includes the actual binary files stored in a `files` directory, a `data.csv` file containing per-file metadata, and a `metadata.json` file that describes global dataset properties.
- `FilelessDataset`: This format does not store the binaries themselves. Instead, it retains only the extracted features of each file, which are saved in the `data.csv` file, while a `features.json` file provides descriptions of these features. This fileless format significantly reduces disk space usage, offering a practical advantage for large-scale datasets where retaining the original binaries is not necessary for certain analysis tasks.

Models also have two formats :

- `Model`: This format consist of folder that contains a `metadata.json` file, storing the chosen classifier and its parameters, a `features.json` file, describing the features used, a `performance.csv` file, tracking the model's test results and the actual trained model saved in a `joblib` file.
- `DumpedModel`: This format is used when only a `joblib` model is available without additional metadata. This approach allows to import and evaluate models trained outside the Packing Box.

By formalizing executables, datasets and models into clearly defined abstractions, Packing Box ensures consistent handling of different data types, simplifies toolchain integration and provides a foundation for extending the framework with new formats and functionalities.

4.1.3 Machine Learning Pipeline

These abstractions are integrated throughout the Machine Learning pipeline, which in the Packing Box framework is organized into four main phases: visualizing the input data, preparing the dataset, training the model and predicting labels for new data instances.

Figure 4.2 illustrates the end-to-end pipeline implemented in the Packing Box framework. Each stage of the process comes with its dedicated tool to explore and refine the data. The `visualizer` tool aims to provide graphical representations of the executable samples. The `dataset` tool helps to create and manipulate the datasets. The last tool, `model`, gives the ability to choose algorithms and their parameters to train models from the created datasets.

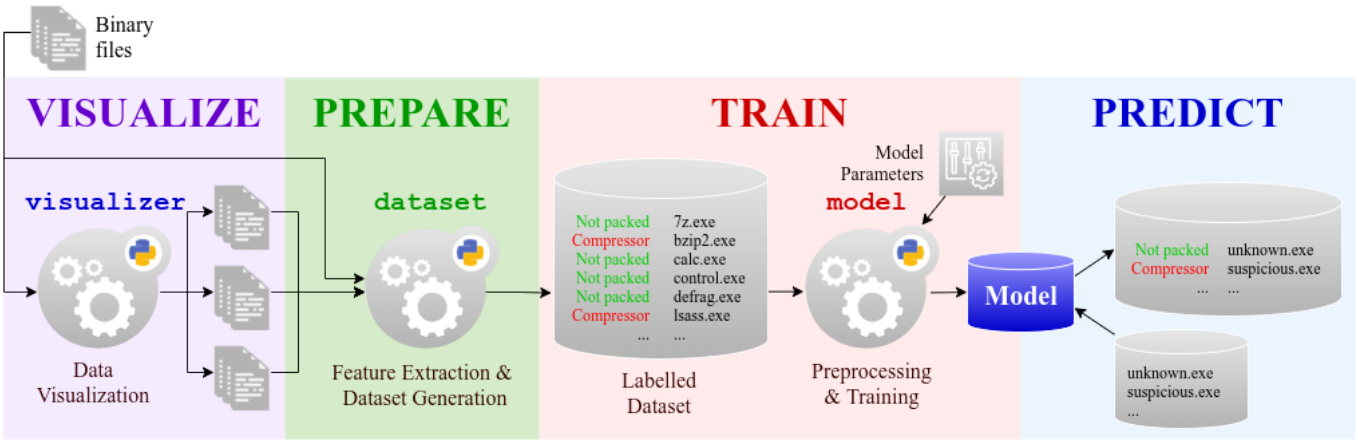


Figure 4.2: Packing Box's machine learning pipeline [3]

The **visualizer** tool is used to inspect binary samples by providing graphical representations, enabling sample comparisons or plotting entropy.

The **dataset** tool provides functionalities to create new datasets and manipulate existing ones in multiple ways. Users can modify datasets, merge several datasets into a single collection, and apply essential operations such as renaming, removing, exporting datasets and much more. In the following, we present and explain the most important features offered by this tool.

Firstly, the `make` command of the dataset tool enables users to create new datasets. It accepts several parameters, such as the input folder, the desired number of executable samples, their format, and the specific packer to be applied to the binary files. Additionally, the `merge` command allows users to combine two existing datasets into a single, unified dataset.

```
┌──[user@packing-box]──[/mnt/share]─────────
$ dataset make new-data -p upx -f PE -n 300
00:00:01.151 [INFO] Source directories:
    ~/.wine64/drive_c/windows,~/.wine32/drive_c/windows
00:00:01.152 [INFO] Considered formats: MSDOS,PE64,PE32
00:00:01.152 [INFO] Selected packers:    UPX
00:00:01.233 [INFO] Searching for executables...
100%|                          | 300/300 samples  0:01:03  new-data
00:01:05.731 [INFO] Used packers: upx
00:01:05.731 [INFO] #Executables: 300

┌──[user@packing-box]──[/mnt/share]─────────
$ dataset merge new-data big-data
00:00:02.601 [INFO] Merging rows from big-data into new-data...
100%|                          | 1474/1474 samples  0:00:45  new-data
```

In this terminal example, we demonstrate the use of the two previously mentioned commands. First, we create a new dataset named new-data, specifying the UPX packer and requesting 300 samples in the Portable Executable format. Next, we merge this newly created dataset with a larger existing dataset to produce an extended, combined dataset.

Another important command provided by the dataset tool is `convert`. As explained in Subsection 4.1.2, Packing Box proposes two formats for datasets to be stored: **Dataset**, which includes the actual binary files, and **FilelessDataset**, which only contains the extracted features without the binaries themselves. This conversion is useful because, without it, feature extraction would only happen later in the pipeline. By transforming datasets into the fileless format earlier, we can directly analyze and assess dataset quality using feature-based methods, as will be discussed later.

```
┌──[user@packing-box]──[/mnt/share]─────────
$ dataset convert new-data -n new-fileless-data
00:01:46.490 [INFO] Converting to fileless dataset...
00:01:46.514 [INFO] Size of dataset:     952MB
00:01:46.515 [INFO] Computing features...
100%|                    | 1474/1474 samples  0:03:52  new-fileless-data
00:05:40.589 [INFO] Size of new dataset: 1MB (compression factor: 637)
```

Here, we convert the previously created dataset into a **FilelessDataset**. By using the `-n` parameter, we create a new dataset instead of overwriting the original one, which allows us to keep both the binary files and the extracted features. As shown, the resulting dataset is also significantly smaller in size.

The **model** tool is used to train Machine Learning models, offering a variety of algorithms to choose from. Once a model has been trained on a dataset, the tool also provides the option to evaluate it against target datasets. A short example is presented below, as this tool will be used in the experiments described later in this work.

```
┌──[user@packing-box]──[/mnt/share]─────────
$ model train my-upx-dataset -A knn
00:00:02.414 [INFO] Selected algorithm: k-Nearest Neighbors
00:00:02.414 [INFO] Reference dataset:  my-upx-dataset
00:00:02.428 [INFO] Computing features...
 100%|                    | 300/300 samples  0:00:25
00:00:29.646 [INFO] Making pipeline...
00:00:29.670 [INFO] Training model...
00:00:34.890 [INFO] [1/3] StandardScaler
00:00:34.899 [INFO] [2/3] PCA with n_components=2
00:00:34.910 [INFO] [3/3] k-Nearest Neighbors
Name: my-upx-dataset_pe32-pe64_300_rf_f126
       .     Accuracy   Precision   Recall    F-Measure     MCC        AUC
   Train    100.00%    100.00%     100.00%   100.00%     100.00%   100.00%
   Test     100.00%    100.00%     100.00%   100.00%     100.00%   100.00%
00:00:44.076 [INFO] Parameters:
- leaf_size = 30
- metric = minkowski
- n_neighbors = 1
- p = 1
- weights = distance
```

In this example, the dataset named `my-upx-dataset` is used to train a model with the Random Forest algorithm. The process begins by computing the features of the files, followed by the actual training phase. Upon completion, the tool outputs the model name, the classification metrics for both the training and test sets and the parameters of the trained model. We can then test it against another dataset.

4.2 Dataset Creation Enhancements

Now that the Packing Box framework's overall architecture and workflow have been presented, we turn to the enhancements that extend its dataset creation capabilities. These improvements are best understood in the context of the dataset creation pipeline illustrated in Figure 2.5 of Subsection 2.3.3.

This pipeline outlines the four main stages of dataset preparation: **COLLECTION**, **CLEANING**, **TRANSFORMATION**, and *Preparation*. The first new features described in this section focus primarily on the early stages - **COLLECTION** and **CLEANING** - where the archive filtering tool `filter-archive` and similarity thresholding mechanisms have been introduced.

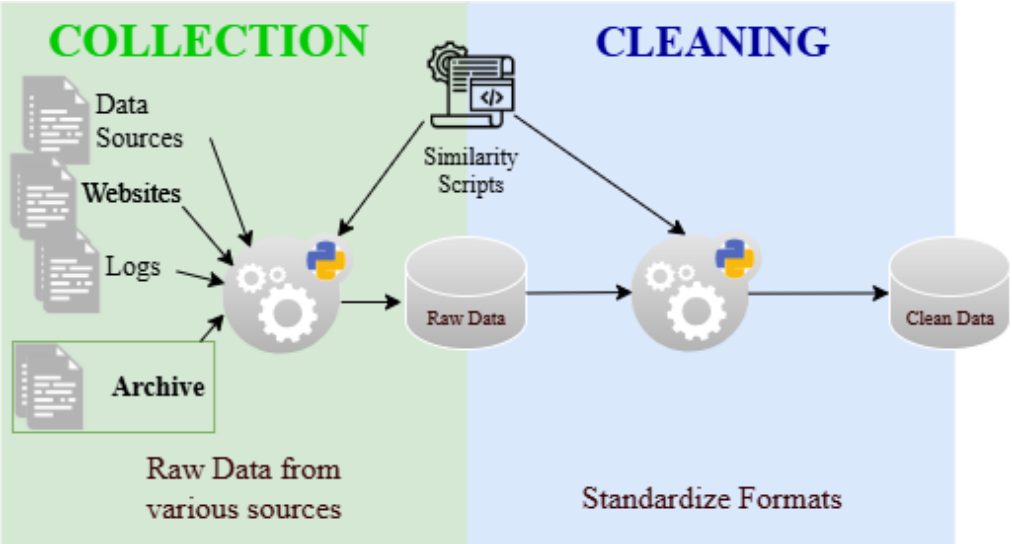


Figure 4.3: Dataset Creation - Focused on Collection and Cleaning

Figure 4.3 highlights these two stages in the pipeline, along with the newly introduced components. The **COLLECTION** stage now supports direct extraction from archives such as Windows ISO images, with the option to apply similarity-based deduplication during this stage or later in **CLEANING**, using hashing techniques.

4.2.1 Archive Filtering Tool

As highlighted in Section 3.4, our study revealed a recurring trend in the literature: many works rely on clean installations of Windows to build datasets for their experiments. This observation motivated the development of the `filter-archive` utility, designed to extend Packing Box's ingestion capabilities and better accommodate such use cases. The primary purpose of this utility is to enable selective extraction from archives or disk images in a variety of formats, including nested archives. When extracting files, the tool can apply filters to target specific formats such as ELF, PE, or more narrowly defined categories like `.exe` files.

Its usage is straightforward: the user provides an archive (for example, a Windows ISO), optionally specifies a filter, and the tool outputs only the files matching the desired criteria. While inspired by the need to handle Windows installation images, the Filter-Archive utility is not limited to ISO archives and can process a wide range of archive formats.

```
┌──[user@packing-box]──[/mnt/share]──────
$ filter-archive Win10_22H2_English_x64v1.iso win10_output.iso -f pe

100%|                          | 42793/? samples  1:08:04    win10_output.iso

22:20:06 [INFO] Filtered 14421 files out of Win10\_22H2\_English.iso
```

Out of more than 40000 files, the `filter-archive` utility retained approximately 14000. This significant reduction is largely due to the similarity threshold, which excludes redundant or near-identical files. For example, within a Windows ISO, many files are repeated across different directories to support multiple system architectures. By applying the similarity filter, these near-duplicates are discarded. Once the archive has been processed, the remaining files can be used directly to build the dataset by specifying the output folder as the source path when running the `make` command. The next section details how the similarity threshold is defined and its impact on the filtering process.

4.2.2 Similarity Threshold

Deduplication is a critical step in the cleaning and processing of a dataset. Previous studies have repeatedly highlighted that non-cleaned datasets often contain a large proportion of similar or duplicate files, which can negatively impact subsequent analysis and model training.

To address this, a similarity filter has been integrated into Packing Box. By default, the system uses `ssdeep`, as it is widely referenced in the literature as a standard similarity hashing method. Additional similarity hashing methods were later chosen to be potentially added ; the rationale for their selection and a comparison of their performance are presented in Section 5.2.

The similarity threshold can be applied at multiple stages in the Packing Box workflow. It can be used as a parameter when invoking the `make` command in the dataset tool, and it can also be applied during the **COLLECTION** stage when using the newly introduced `filter-archive` utility. This flexibility allows deduplication to occur early in the dataset creation process or later during cleaning, depending on the requirements of the experiment. The threshold value must be specified in the range 0100, and any file whose similarity score exceeds this value is discarded.

```
┌──[user@packing-box]──[/mnt/share]─────────
$ dataset make my-dataset -f PE -similarity-threshold 80
«snipped»
00:01:19.994 [INFO] Executables:  100
```

As it will be explained in the next section, the similarity threshold is closely tied to the `similarity` metric in use, which also plays a role in the improved `fix` command introduced as part of the dataset quality improvement process.

4.3 Dataset Quality Scoring

Figure 4.4 highlights the **CLEANING** and **TRANSFORMATION** stages of the dataset creation pipeline, which are the primary focus of the tools and capabilities introduced in this section. These enhancements enable the framework to evaluate prepared datasets and improve their quality in a systematic manner, at any point during the two targeted phases.

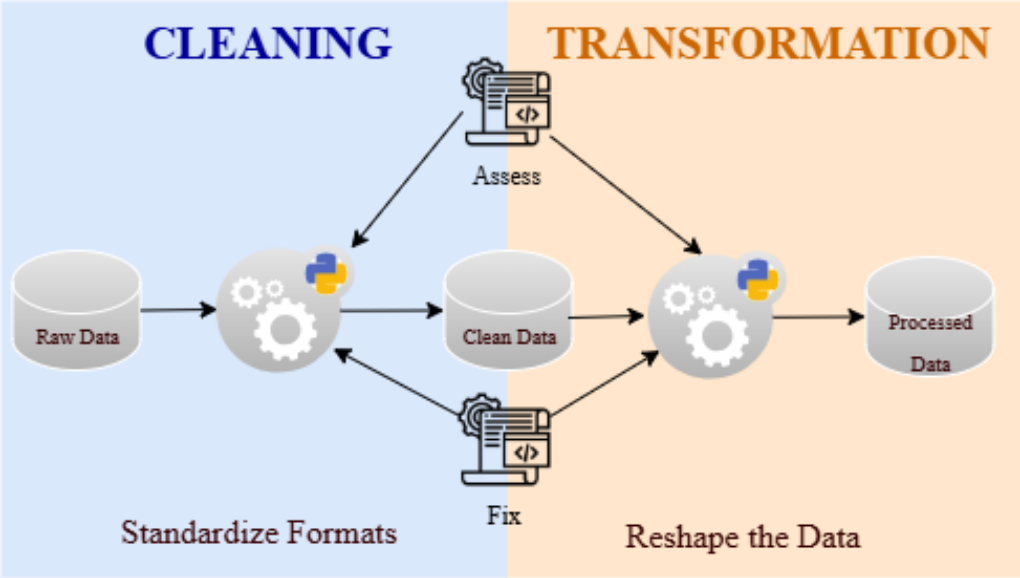


Figure 4.4: Dataset Creation - Focused on Cleaning and Transformation

Two new functionalities have been integrated into Packing Box for this purpose:

- **Assessment :** measuring the quality of a dataset using a set of defined metrics.
- **Fixing :** automatically applying transformations to improve the dataset's quality score.

This subsection will start with the set of assessment metrics integrated into the dataset tool, detailing their purpose and relevance. Each metric has been selected based on established data quality frameworks and the specific challenges associated with datasets of packed binary samples. To better reflect the constraints of different dataset formats, these metrics are divided into two categories: *generic* and *specific* metrics.

We will then show how these metrics can be used to assess the overall quality of a dataset, as well as how the improved `fix` command leverages them to enhance dataset quality.

4.3.1 Generic Metrics

Generic metrics can be computed without requiring direct access to the binary files themselves, as their scores do not depend on information specific to the file content. Instead, they operate on metadata or structural information available in the dataset's `data.csv` file. For each metric presented in this subsection, we explain its purpose, the aspect of dataset quality it addresses, and the method used for its evaluation.

- **Completeness** refers to the extent to which all required data is present in a dataset. For binary datasets, this focuses on metadata completeness, ensuring every sample has associated metadata fields fully populated (e.g. labels, architecture type, file size, packer information, timestamps). Missing metadata severely limits the capacity to perform accurate analysis, feature extraction or label-driven Machine Learning tasks. According to Batini *et al.* [44], completeness is a foundational aspect because incomplete datasets can bias conclusions or impede replicability of research findings. In the domain of packed binaries, completeness ensures that no sample lacks critical contextual information necessary for static analysis workflows. The computation of this metric is straightforward: every dataset in Packing Box is accompanied by a `data.csv` file. The evaluation process inspects this file and, for each missing value in the metadata, applies a penalty to the completeness score.
- **Consistency** checks whether data entries adhere to the same format and constraints across the dataset. For binary samples, this involves verifying that all files are present, conform to a valid formats and have coherent metadata-label alignment. A binary dataset with inconsistent file structures or erroneous labels would yield misleading analysis results. Cai *et al.* [45] emphasize consistency as vital for maintaining data reliability, especially in automated processing pipelines where malformed files could break analysis tools. Consistency is particularly critical for packed binaries where the boundary between legitimate and tampered formats is subtle.
- **Label Balance** measures how evenly distributed the dataset labels are between the packed and not packed categories. Imbalanced label distributions can severely affect classification models, leading to biased or overfitted systems. Batini *et al.* [44] and Miller *et al.* [47] both identify balance as a dimension of fairness and bias control in data-driven systems, making this metric fundamental in datasets intended for security research. When computing label balance in Packing Box, it is possible to define a tolerance margin, meaning that a perfectly even 50/50 distribution is not strictly required. Instead, the balance score accounts for deviations within the acceptable margin while penalizing more severe imbalances.

4.3.2 Specific Metrics

Specific metrics require direct access to the executable files in the dataset. Unlike generic metrics, which can be computed solely from metadata, these metrics rely on analyzing the actual binary content to assess quality.

- **File Balance** refers to the distribution of various file types (EXE, DLL, SYS), architectures (x86, x64), and file size ranges. A well-balanced dataset ensures that the diversity present in real-world packed binaries is represented. This is essential because ML models trained on imbalanced file types may exhibit poor generalization. Mohammed *et al.* [48] highlight "representativeness" as a core facet of data quality-file balance directly addresses this by preventing skewness towards certain file types, thus enabling more robust analysis tools and detectors that generalize beyond a narrow subset of binaries. When computing this metric in Packing Box, a tolerance margin can be specified, similar to the approach used for Label Balance. This allows for natural variation while still penalizing significant imbalances.

- **Outliers** detection aims to identify samples with anomalous properties, such as unusually large file sizes, suspicious timestamps (e.g. future-dated builds), or malformed PE headers. Outliers may indicate corrupt data, noise, or rare but critical cases (e.g., advanced packers). Cai *et al.* [45] associate outliers management with improving data reliability and robustness, emphasizing the need to distinguish between beneficial and detrimental anomalies. For packed binaries, outliers could distort statistical analyses or model training, so identifying and handling them is crucial for dataset curation. For `FilelessDataset` where direct access to extracted features is available, we apply DBSCAN to detect these outliers. Each detected outlier incurs a penalty to the metric score, reflecting its negative impact on overall dataset quality.

- **Portability** assesses whether the binary samples are executable across different environments, considering format-specific attributes such as relocation information, imports/exports, and platform-dependent flags. Packed binaries often contain embedded environment checks or platform constraints that can limit their analyzability or execution outside their intended context. Portability is linked to the "usability" dimension of data quality discussed by Miller *et al.* [47], ensuring that samples are not just structurally valid but are also operationally meaningful across diverse analysis and testing platforms. This is vital for reproducibility in static analysis experiments conducted in heterogeneous environments. Since the criteria for assessing portability differ across executable formats , this metric is computed using format-specific logic encapsulated within the abstraction layer, which allows the framework to seamlessly apply the appropriate method based on the file type. For PE files, the following attributes, that are presented in Section 2.1.2, are evaluated: architecture (penalizing targets outside the common I386 and AMD64 set), declared operating system version (penalizing unusually high minimum versions), subsystem (restricting to Windows Console or Windows GUI), presence of relocation information, absence of suspicious imports such as absolute paths, network shares, or OCX files, security flags enabling Address Space Layout Randomization (ASLR) and Data Execution Prevention (DEP), inclusion of a manifest resource, absence of elevated privilege requirements, presence of a valid digital signature, absence of TLS callbacks and avoidance of hardcoded absolute paths. Each of these checks deducts a small penalty from an initial perfect score (1.0). The final score is normalized between 0 and 1. This format-specific approach is encapsulated within the framework's abstraction layer, allowing different executable formats (e.g., ELF, Mach-O) to use equivalent, format-appropriate checks while maintaining a consistent interpretation of the portability metric.

- **Similarity** measures the degree of near-duplication between samples using techniques like fuzzy hashing. Packed binary datasets are prone to artificial diversity, where minor packing variations result in a large number of near-identical files. Detecting such redundancies helps ensure the dataset represents a wide range of unique behaviors rather than superficial diversity. Mohammed *et al.*'s framework [48] considers redundancy and information entropy as critical to ensuring dataset efficiency and non-redundancy, making similarity analysis a direct application of this principle. Using the similarity threshold presented in Section 4.2.2, the metric compares files within the dataset using a default threshold of 90 (configurable by the user). For each pair of files exceeding the threshold, a penalty is applied to the similarity score.

- **Uniqueness** validates that all samples are distinct by performing cryptographic hash checks (SHA-256) to identify exact duplicates. This metric prevents inflating dataset size with redundant copies, which can distort metrics and experimental results. Unlike similarity (which handles near-duplicates), uniqueness addresses bitwise-identical files. Maintaining uniqueness aligns with Batini *et al.*'s *data integrity* dimension [44] and is a basic yet crucial step for dataset deduplication and quality assurance. In addition to file-level checks with SHA-256, uniqueness can also be assessed at the feature level. It is possible for two files with very different cryptographic hashes to yield exactly the same feature vector, particularly when the extracted features are limited in scope. In such cases, the dataset would still contain redundant representations, which reduces its effective diversity. To mitigate this, the metric incorporates a way to identifies duplicate feature vectors in the metadata. Together, these two validation layers – bitwise verification at the file level and redundancy detection at the feature level – ensure that uniqueness is enforced effectively. While some exact duplicates may already be flagged by the **similarity** metric, uniqueness is retained as a distinct measure to give additional weight to deduplication, since redundant or overly similar samples remain a significant issue in binary datasets.

Table 4.1 provides an overview of the metrics, including a brief description of each with their type.

Name	Description	Type
Completeness	Verifies that all metadata fields are properly populated, ensuring no missing or null values.	Generic
Consistency	Ensures all files physically exist, validates their PE format, and checks that labels are correctly assigned.	Generic
Label Balance	Measures the distribution of labels to check whether classes are balanced.	Generic
File Balance	Evaluates the diversity of architectures (x86/x64), file types (exe, dll, sys) and file sizes.	Specific
Outliers	Detects files with unusual sizes, suspicious timestamps, or other anomalous characteristics.	Specific
Portability	Analyzes format-specific features to assess if files are portable across different environments.	Specific
Similarity	Identifies near-duplicate files within the dataset using similarity hashing.	Specific
Uniqueness	Detects duplicates both at the file level (via SHA-256) and at the feature level (metadata).	Specific / Generic

Table 4.1: Quality Assessment Metrics Overview

4.3.3 Dataset Assessment

Individually, the metrics described in the previous subsections provide valuable insights into specific aspects of dataset quality, but none of them alone offers a complete picture. For this reason, all metrics are considered together to compute an **overall quality score**. This score is calculated using the assess command, which evaluates each individual metric and combines the results into a single value ranging from 0 (lowest quality) to 1 (highest quality).

To account for the fact that not all metrics contribute equally to dataset quality, each metric is assigned a weight. Initially, these weights were determined intuitively based on the perceived importance of each metric in the context of packed binary datasets. Later, the weighting scheme was refined through a more systematic calibration process, described in Section 5.1.

The `assess` command also supports several parameters. As previously explained, the similarity threshold used in the Similarity metric can be adjusted by the user. Additionally, the command can generate visual representations of the results, providing a clearer understanding of how each metric contributes to the overall score.

```
┌──[user@packing-box]─[/mnt/share]─────────
$ dataset assess PE_upx -similarity-threshold .7 --plot
Dataset quality

      Name          Score    Weight
──────────────────────────────────────
  Completeness      1.000    0.5
  Consistency       1.000    0.5
  File Balance      0.767    1.0
  Label Balance     0.940    1.0
  Outliers          1.000    1.0
  Portability       0.512    0.5
  Similarity        0.950    1.0
  Uniqueness        1.000    0.5

     Overall        0.834
```

This terminal illustrates how the overall quality score is derived from the combination of individual metric scores and their respective initial weights.

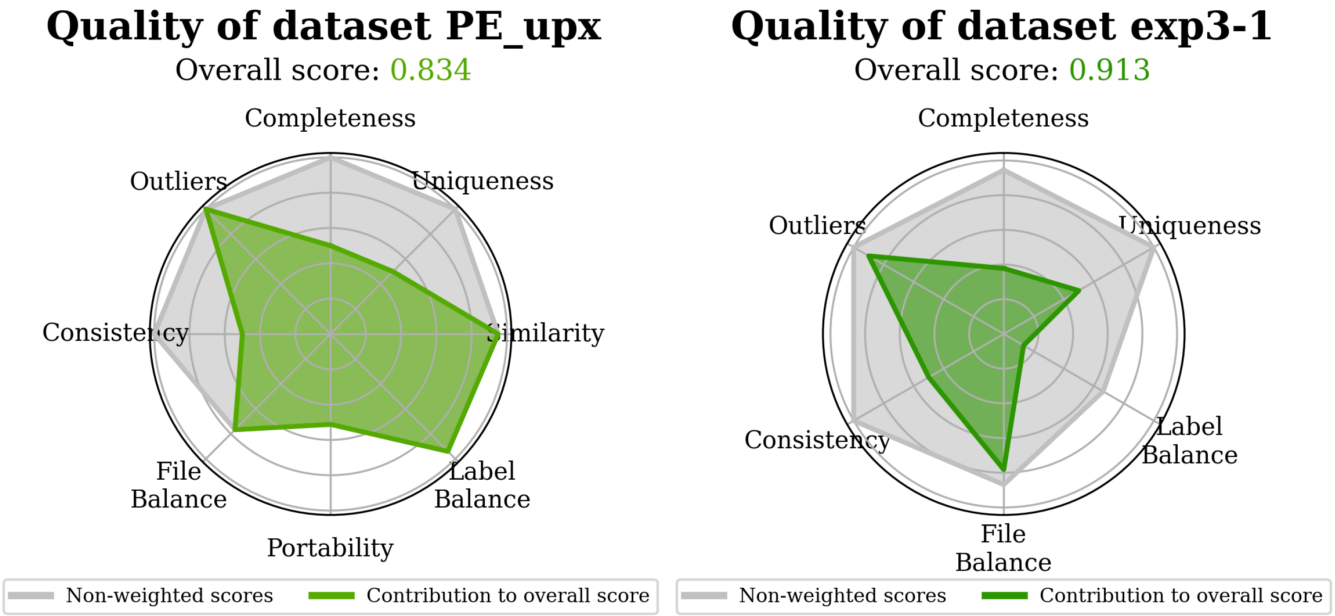


Figure 4.5: Result of the `assess` command

Figure 4.5 shows the resulting plot produced when using the `-plot` parameter on a dataset containing all the original binaries. The plot on the left shows the results of applying the command to a `Dataset`, whereas the plot on the right shows the results for a `FilelessDataset`. These are distinct datasets.

4.3.4 Dataset Quality Improvement

The quality score of a dataset is intended to reflect, as closely as possible, the expected performance of any model trained on it. In other words, improving this quality should directly contribute to improving model performance. This creates a clear need for a mechanism to enhance dataset quality in a systematic way. To address this, the existing `fix` command of the dataset tool was extended to specifically target the defined quality metrics, enabling automated improvements to the overall dataset quality score.

The `fix` command targets all metrics with a score below 1. Since a perfect score cannot be improved further, only underperforming metrics are addressed. The general principle is to remove files that most negatively affect the given metric, thereby raising its score. To prevent excessive deletions, a global deletion threshold is defined, which is then distributed across the different metrics. To illustrate the process, the following terminal output shows an example execution of the `fix` command on the dataset previously assessed. Once the command has completed, the dataset can be reassessed to compare the updated scores with the initial results.

```
┌──[user@packing-box]──[/mnt/share]─────────
$ dataset fix my-dataset

00:00:48.120 [INFO] Outliers cannot be fixed
00:00:48.120 [INFO] Completeness cannot be fixed
00:00:56.709 [INFO] Fix removed 36 samples.
```

- **Uniqueness**
  This metric does not use a deletion threshold, as exact duplicates are both rare and highly detrimental to dataset quality. Every duplicate identified is automatically removed, ensuring the dataset remains unique. Checks are performed at either the file level or the feature level.

- **Completeness and Consistency**
  These metrics are generally strong, with only occasional missing or inconsistent metadata. For this reason, they are allocated a very small deletion budget. In practice, this ensures completeness and consistency are improved when necessary, without removing a significant portion of the dataset.

- **Portability**
  Files with the lowest portability scores are removed, within the deletion quota allocated to this metric. This process prioritizes eliminating binaries that are the least portable, thereby improving the dataset's representativeness across platforms.

- **Similarity**
  As redundancy is a critical issue in packed binary datasets, the similarity metric is given the largest share of the deletion threshold. Using the similarity threshold, files with the highest similarity scores are removed first. The process continues until either the allocated deletion quota is reached or no further near-duplicates remain.

- **Outliers**
  If features are already computed, DBSCAN is used to detect and remove potential outliers. If not, the system checks for abnormal file sizes or unexpected creation timestamps.

- **Label Balance and File Balance**
  These metrics are fixed by removing files from the majority class or majority group (e.g., packed vs. unpacked for label balance, or file types/architectures for file balance). The same margin used during assessment is applied here, ensuring balance is restored without overfitting to a perfectly equal distribution.

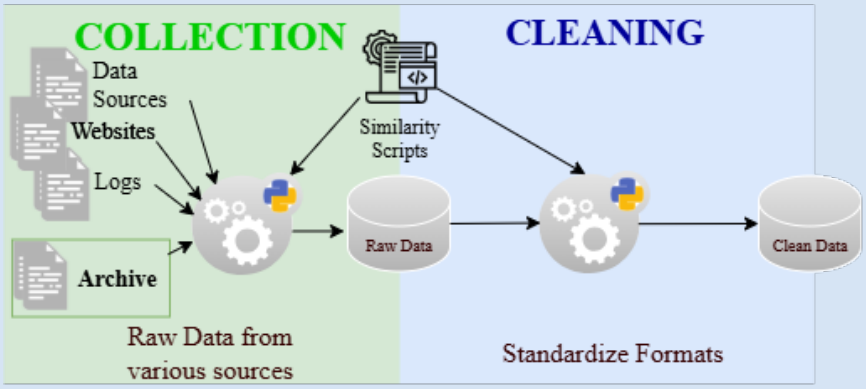
The fix command can be applied at different stages of the dataset lifecycle:

- After the dataset creation, to ensure the first version already meets quality standards.
- Before training, to fine-tune dataset balance and reduce redundancy.
- Iteratively during experimentation, adjusting the dataset composition to achieve optimal model performance.

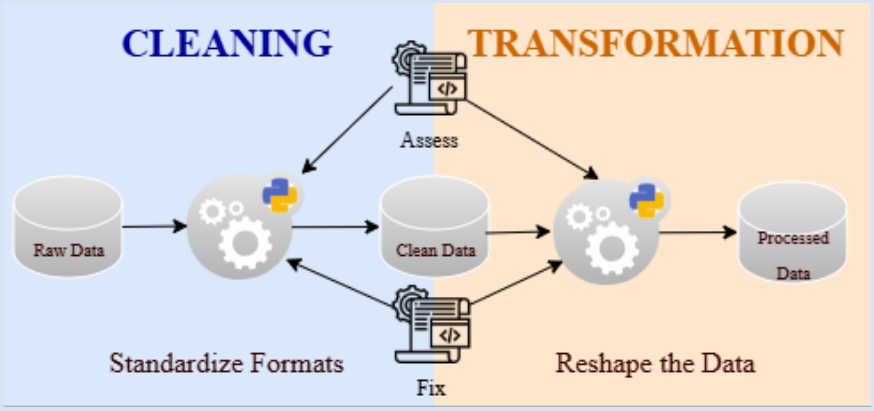
SUMMARY

The **Packing Box** is an open-source command-line environment distributed as a Docker container. It integrates a comprehensive set of executable analyzers, packing detectors, packers and unpackers, as well as utilities for generating and manipulating datasets. The framework provides a complete **pipeline for training** and testing **models**, supported by several dedicated tools. The dataset tool enables extensive dataset manipulation and preparation; the visualizer tool offers data visualization capabilities; and the model tool supports the training and evaluation of ML models.

Two main areas of the dataset creation pipeline were extended in this work. The first group of enhancements targets the **COLLECTION** and **CLEANING** stages. The **archive filtering utility** `filter-archive` enables selective extraction from archives or disk images, including nested archives, with filtering options for specific formats such as ELF, PE, or `.exe`. The **Similarity Threshold** mechanism uses fuzzy hashing (ssdeep by default) to detect and remove near-duplicate files. This threshold can be applied both during archive filtering and during dataset cleaning to minimize redundancy early in the process.



The second group of enhancements can be applied during both the **CLEANING** and **TRANSFORMATION** stages. A **dataset quality scoring** system was added, comprising eight metrics, divided into generic and specific, that evaluate datasets against established quality dimensions. The new `assess` command computes all metrics, applies configurable weights, and produces an overall quality score between 0 and 1, with optional visualization and adjustable parameters such as the similarity threshold.



Finally, the `fix` command was extended to directly target these metrics. It identifies low-scoring areas, applies **corrective actions** such as balancing labels and removing redundant files to improve the overall quality score. Together, these tools enable datasets created within the Packing Box framework to be both high quality and better suited for downstream model performance.

5 Experiments

THIS chapter presents the experiments we conducted, in order to answer our research questions defined in the introduction.

We begin by comparing different methods for detecting similarity among binary samples. Next, we investigate how quality metrics behave across multiple datasets to determine their appropriate weight. Finally, we evaluate the impact of dataset quality on model performance, assessing whether improvements in quality metrics translate into measurable gains in predictive accuracy.

5.1 Methodology 54
5.1.1 Context 54
5.1.2 Approach 54
5.1.3 Datasets 55
5.2 Experiment 1: Evaluation of Hashing Method 55
5.2.1 Setup 55
5.2.2 Results 57
5.3 Experiment 2: Metrics Calibration 61
5.3.1 Setup 61
5.3.2 Training and Testing 62
5.3.3 Results 63
5.4 Experiment 3: Dataset Quality Improvements 66
5.4.1 Baseline 66
5.4.2 Dataset Fixing 68
5.4.3 Results 71
Summary 73

Objectives

A – Experiment with different hashing methods.

B – Show the importance of the weighting of the quality metrics.

C – Analyze performance of models when undergoing quality improvement technique.

5.1 Methodology

This section addresses the methodology used to lead the experiments. We first present relevant information gathered in Chapter 3. We then present our approach and how our experiments relate to the research questions.

5.1.1 Context

In Chapter 3 , we noted the following opportunities and limitations.

- While many studies evaluate the accuracy and search runtime of similarity methods, they often overlook the computational costs of preprocessing steps such as feature extraction, shingle generation, and hash digest computation. This preprocessing overhead is rarely measured or compared across techniques, leaving a gap in understanding the true end-to-end performance of these methods.
- The importance of data quality is becoming increasingly recognized as a critical factor in the success of Machine Learning workflows, often outweighing the impact of the model itself. Issues such as label noise, missing values, distribution skews, and inconsistent feature extraction can severely degrade model reliability, highlighting the need for robust data quality assessments
- Current data quality tools are mainly designed for textual or structured data, with little consideration for binary datasets. Developing a methodology that can assess and clean binary data would address this gap and improve dataset reliability.

5.1.2 Approach

To address the limitations identified in the previous section, we propose the following experiments:

- **Evaluation of Hashing Method:** We conduct a comparative evaluation of several similarity hashing techniques using multiple datasets that vary in size, structure and packers selection. The experiment measures both the preprocessing time (digest computation) and the comparison time for each method. The results are analyzed to assess how these methods perform relative to each other in terms of efficiency and similarity accuracy, providing insights into which techniques are best suited for Packing Box.
- **Metrics Calibration:** We develop a methodology to calibrate data quality metrics by training multiple models across diverse datasets. By correlating the quality scores of different metrics with the actual performance of these models, we aim to determine the optimal weighting of each metric. This calibration ensures that the aggregated quality score accurately reflects the real-world impact of data quality on model reliability and performance.

- **Dataset Quality Improvements:** We build a new, diverse dataset, assess its quality metrics, and evaluate model performance when trained on it. We then iteratively improve the dataset by addressing identified quality issues and remeasure performance, quantifying how targeted improvements affect model accuracy, robustness and general performance.

5.1.3 Datasets

For **Experiment 1**, which evaluates the hashing performance of different methods, the primary concern was the amount of samples rather than their diversity. These datasets were composed by using the make command to select a fixed number of samples from the default source of Packing Box. Additional samples from the dataset-packed-pe repository were included to increase the dataset size and ensure meaningful results.

For **Experiment 2** and **Experiment 3**, more diverse and controlled datasets were required. These were built using the default source of Packing Box, the dataset-packed-pe repository, and additional samples extracted from Windows images via the newly implemented Filter-Archive Tool. A more detailed explanation of the commands used to generate these datasets is provided in the corresponding experiment sections.



The Dataset of packed PE files is a forked extension of PackingData and was extended created by D'Hondt. The process used to compose this dataset is documented in [3].

5.2 Experiment 1: Evaluation of Hashing Method

In this experiment, we aim to evaluate various fuzzy hashing and similarity digest methods to determine their relative efficiency and accuracy in practical scenarios. Fuzzy hashing methods are widely used for similarity detection in cybersecurity, digital forensics, and data deduplication. Despite the prevalence of certain tools like ssdeep, recent literature suggests that alternative algorithms may offer superior performance in specific contexts. However, these claims are often made without thorough, head-to-head comparisons that encompass the entire hashing and similarity search process.

5.2.1 Setup

For this experiment, we choose six widely recognized similarity hashing techniques [5.1] to compare and subsequently determine which methods are worth integrating as default options in the packing box workflow for efficient and accurate similarity detection.

Some of the tools evaluated - such as ssdeep, TLSH, LZJD, and MinHash - offer Python libraries or bindings that allow direct manipulation of their hashing and comparison functionalities within Python scripts. Conversely, tools like SDhash and mrshv2 do not provide such libraries, requiring interaction via subprocess calls to their command-line interfaces (CLI).

For tools where both options were available (e.g., ssdeep), I conducted a preliminary comparison to measure the performance difference between using the Python library directly and invoking the CLI through subprocess calls. Theoretically, avoiding a subprocess should result in better performance due to reduced inter-process communication overhead. Empirically, this expectation was confirmed: hashing and comparison operations performed using the Python library were consistently faster than their subprocess-based counterparts.

Name	Year	Implementation	Similarity Metric	Score Range
LSH with MinHash	1997	Python package (datasketch)	Jaccard Similarity Approximation	0 to 1
ssdeep	2006	CLI tool (C), Python bindings	Edit Distance on Rolling Hash substrings	0 to 100
SDhash	2010	C library, CLI tool	Feature Count Overlap Score	0 to 100
MRSH-v2	2013	Python & C implementations	Edit Distance on Hash Blocks	0 to 100
TLSH	2013	C/C++ library, Python API	TLSH Distance Metric (Histogram comparison)	0 to ∞ (lower is better)
LZJD	2017	Java & Python libraries	Jaccard Similarity on Lempel-Ziv substrings	0 to 1

Table 5.1: Overview of the Similarity Hashing Methods

Given these results, it is logical to use direct library calls wherever possible to minimize overhead. However, for tools like mrshv2 and SDhash, which lack native Python bindings, subprocess invocation remains the only viable integration method.

To ensure a fair and consistent evaluation, the same operational process was applied across all methods. The workflow consists of two distinct phases: (1) hashing/digest computation and (2) similarity comparison. First, the digests for all files are computed and stored. Then, each file is compared against the existing digests to identify the first matching pair. Once a match is found, the comparison process for that file is terminated, and the workflow proceeds to the next file. This simulates a practical scenario where only the first relevant match is needed, rather than performing exhaustive pairwise comparisons.

Both phases - digest computation and similarity comparison - are timed separately to isolate their individual performance characteristics. Additionally, the total runtime encompassing both phases is recorded to assess the overall efficiency of each method. This consistent measurement strategy allows for a meaningful comparison of the preprocessing overhead and similarity matching performance across all evaluated techniques.

An important aspect of this evaluation is the normalization of similarity scores across all methods to ensure a fair comparison. Since each method produces scores on different scales, a unified range of 0 to 100% similarity is adopted for consistency. For some methods, this is straightforward. Tools like ssdeep already output scores within this range, while methods such as LZJD and MinHash, which produce similarity scores between 0 and 1, can be easily scaled by multiplying the result by 100.

However, the normalization of TLSH scores presents additional challenges. As shown in Table 5.1, TLSH produces a distance metric ranging from 0 to infinity, where a score of 0 indicates identical files. Unlike other methods, TLSH's scoring does not inherently map to a bounded similarity percentage. To address this, we refer to the normalization approach proposed by Pagani *et al* [58]. They noted that although TLSH distances can increase indefinitely, the original authors of TLSH highlighted a threshold value of 300 as a point beyond which further differences in score yield diminishing returns in practical similarity detection.

Based on this observation, a normalization formula was proposed:

$$y = max\{0, (300 - x)/3\}$$

Where x is the TLSH distance and y is the mapped similarity score on a 0-100 scale. This mapping compresses TLSH's score into a comparable range while accounting for its diminishing discriminative capacity beyond the 300 threshold.

However, initial tests revealed that TLSH, even when normalized, tends to produce a significantly larger number of matches compared to other methods. While this aggressive matching behavior may be useful for broad triage tasks, it poses challenges in workflows where precision is critical, as excessive matches can lead to false positives and unnecessary validation overhead. To address this, we explored alternative normalization strategies aimed at reducing match density and better aligning TLSH's behavior with the other evaluated methods.

Our intuition was to adopt a curve with a steeper initial drop, thereby minimizing the proportion of matched pairs at higher similarity thresholds. An exponential decay function was chosen over a linear scaling because it more effectively penalizes small increases in distance, ensuring that only very close matches are preserved at high thresholds. Among the various candidate functions, we selected the curve hereafter, an arbitrary yet practical choice that provided the desired attenuation in match density.

$$y = \max\left(0,\ 100 \times e^{-\frac{d}{100}}\right)$$

Figure 5.1 illustrates the chosen function alongside the original linear normalization and two other candidate curves that were ultimately deemed too steep.

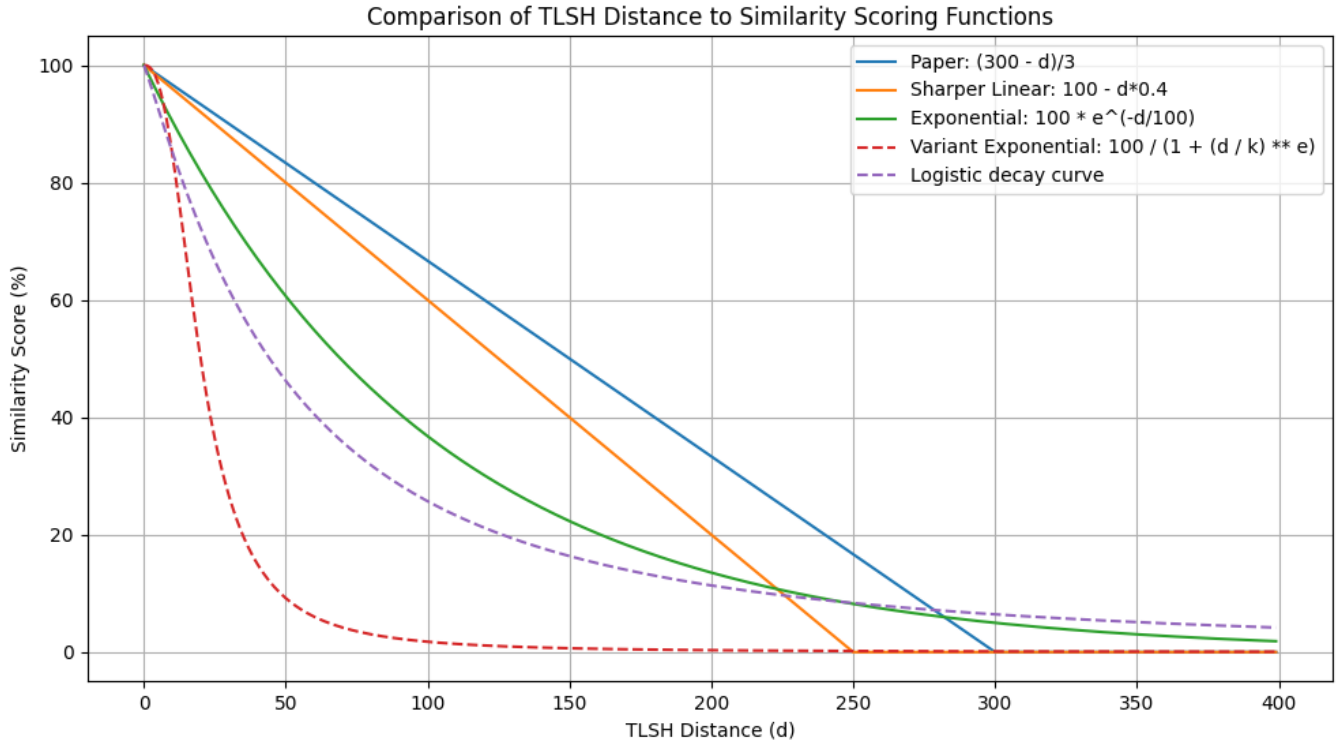


Figure 5.1: TLSH Normalization Curves

5.2.2 Results

We can now execute all the evaluated methods across the different datasets and analyze their performance. The first step is to measure the overall processing timeincluding both hashing and comparison phasesas the number of input files increases. The following graph illustrates how the total processing time scales with dataset size for each method.

One method, LSH, was excluded from this plot because preliminary experiments revealed that its processing time was significantly higher than the others, making the scale of the graph less informative. The time taken by LSH can still be seen in the comparative results table, and a plot showing its performance on a smaller subset is provided in the appendix.

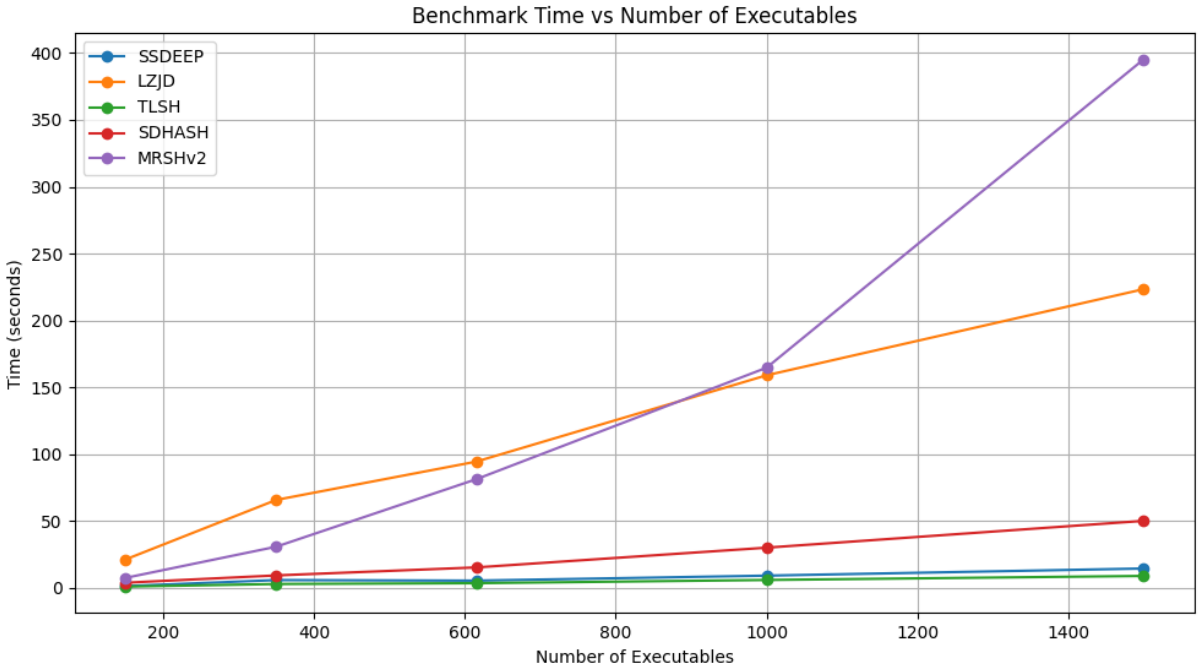


Figure 5.2: Similarity Hashing Methods Benchmark

As expected, all methods show an increase in processing time as the number of files grows, but the rate of this increase differs substantially between them. TLSH and ssdeep exhibit near-linear scaling, suggesting lower computational overhead and efficient handling of larger datasets. SDhash also scales roughly linearly but with a slightly steeper slope than TLSH and ssdeep. In contrast, MRSH-v2 and LZJD display much steeper curves, indicative of higher algorithmic complexity or less efficient scaling.

In absolute terms, TLSH is consistently the fastest method, closely followed by ssdeep. SDhash occupies an intermediate position, while TLSH and especially MRSH-v2 incur significantly higher runtime costs. Based solely on time performance, TLSH and ssdeep emerge as the most practical choices, with SDhash remaining a viable option. The high computational cost of LZJD and MRSH-v2 may limit their suitability for integration into the Packing Box framework. However, runtime alone does not determine a method's overall utilitypair detection performance must also be considered before making a final decision.

The table 5.2 reports the processing times for each method, broken down into pre-processing and comparison phases. The total time is also given, along with a relative speed ratio referenced to ssdeep, which is arbitrarily chosen as the baseline method in this context. Unlike the earlier runtime graph, this table also includes LSH, which was excluded from the plot due to its much higher overall runtime.

Name	Hashing Time(s)	Comparison Time(s)	Total Time(s)	Ratio
ssdeep	9.32	0.78	10.10	1
TLSH	5.56	0.83	6.39	0.63
SDhash	17.99	20.88	38.87	3.85
LZJD	124.83	3.09	127.92	12.67
MRSH-v2	12.26	154.16	164.58	16.48
LSH	480.15	0.02	480.17	47.54

Table 5.2: Time comparison for 1000 files

Clear patterns emerge from this breakdown. TLSH, the fastest overall, achieves its speed through a hashing phase nearly twice as fast as ssdeep's, with negligible comparison cost. Ssdeep offers a balanced profile, dominated by hashing but with minimal time spent in comparison. SDhash splits its runtime between a moderately expensive hashing stage and a comparison phase that alone exceeds the total runtime of TLSH or ssdeep. LZJD's runtime is almost entirely dictated by hashing, whereas MRSH-v2 shows the opposite behavior: hashing is relatively quick, but comparison is extremely costly. Finally, LSH is dominated almost exclusively by hashing, with comparison cost negligible-consistent with its design, where bucket indexing greatly reduces the need for pairwise comparisons.

Beyond runtime performance, it is essential to examine the consistency of similarity detection across methods. Specifically, we want to assess whether different approaches flag the same file pairs as similar or if discrepancies arise. We begin by analyzing the total number of similar pairs detected by each method, providing an overview of their relative sensitivity. Following this, a heatmap illustrates the pairwise Jaccard similarity between methods, quantifying the degree of overlap in the file pairs they classify as similar. Higher values indicate stronger agreement in similarity detection, whereas lower values highlight divergences in matching behavior.

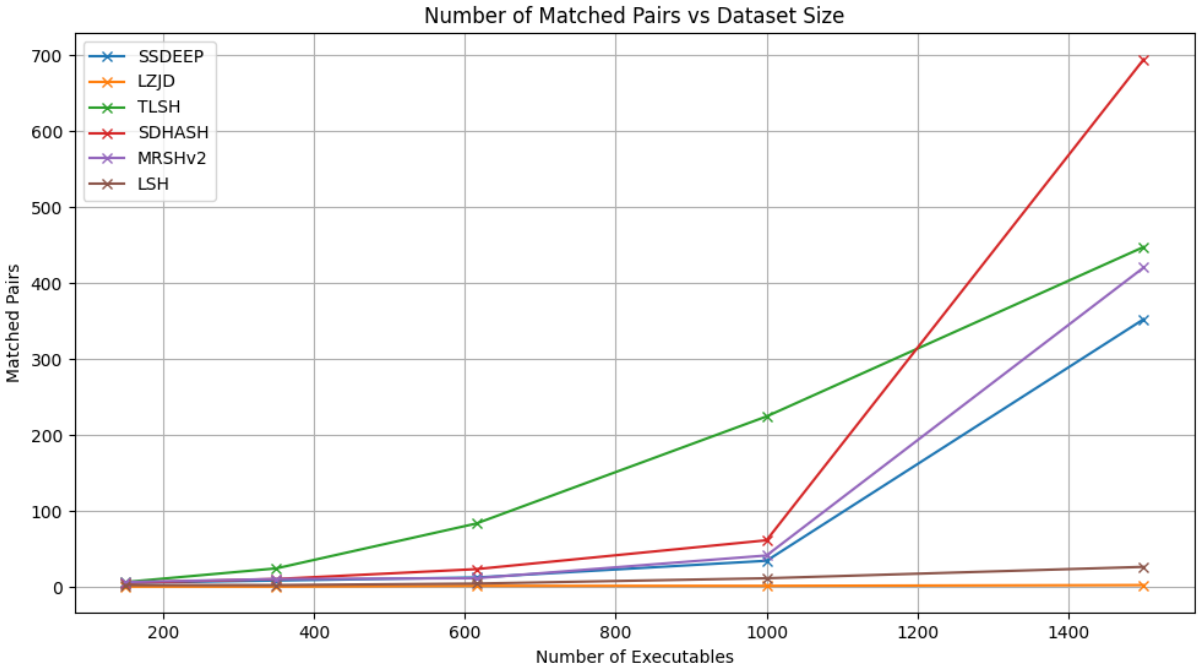


Figure 5.3: Numbers of similar pairs detects for each methods

Across the first four datasets, most methods exhibit a gradual increase in the number of matched pairs as the number of executables grows. TLSH consistently produces a relatively high and steadily increasing match count compared to other methods, which is expected given its scoring behavior and the need for normalization.

In the final dataset – deliberately enriched with more similar files to test each method's sensitivity – SDhash shows by far the most pronounced jump in detected pairs. At this point, the gap between TLSH and methods like ssdeep or MRSH-v2 narrows, although TLSH still produces more matches overall. LZJD, on the other hand, shows no significant increase in matches, suggesting that it failed to capture these additional similarities.



Figure 5.4: Similarity hashing methods jaccard heatmap

SSDEEP shows moderate-to-high overlap with SDhash and MRSH-v2, indicating that these methods often identify a similar subset of matches, even if their total match counts differ. TLSH, by contrast, has only moderate agreement with ssdeep and SDhash, largely because TLSH tends to detect a greater number of pairs overall. MRSH-v2 and SDhash share a relatively high similarity, reinforcing the idea that they frequently capture a comparable set of strong matches. At the other extreme, LZJD exhibits minimal overlap with all other methods, consistent with its near-zero match counts. LSH also has a fearly low overlap.

FINDINGS

This experiment confirms that, in terms of runtime and detection consistency, ssdeep remains a strong performer, explaining its status as a widely adopted standard. TLSH matches ssdeep's speed and can serve as a viable alternative, provided that an optimal normalization function is applied. SDhash also delivers competitive performance with a high overlap in detected pairs, while MRSH-v2, though slower, still shows strong correspondence in its matches.

Results from prior studies from the State of the Art in Chapter 3 on precision and recall further support the reliability of these methods for similarity detection in packed executables. Consequently, we decided to **keep ssdeep as the default** choice, while TLSH, SDhash, and MRSH-v2 have been structured to allow easy integration into the Packing Box if needed.

In contrast, methods such as LSH and LZJD combine long processing times with very low numbers of detected pairs, making them unsuitable for integration into the framework.



Additional details and graphs on this experiment are provided in Appendix A for interested readers.

5.3 Experiment 2: Metrics Calibration

The purpose of the dataset quality score is to estimate how well a model trained on a given dataset is likely to perform. For this score to be meaningful, it must correlate strongly with actual model performance. It is important to note, however, that a model trained on a high-scoring dataset will not necessarily achieve high results on every test dataset; performance also depends on the composition of both the training and test sets. For example, if the test dataset contains binaries packed with different packers than those present in the training set, the resulting score will likely be lower than when both sets share the same packers.

Ultimately, the expectation is that improving a dataset's overall quality score should, on average, improve the performance of models trained on it. Demonstrating this relationship will be the focus of the next experiment. The goal of the present experiment is to determine the optimal weights for the individual quality metrics that are combined to compute the overall dataset quality score.

5.3.1 Setup

The first step in this experiment is to create the datasets required for model training and evaluation. Multiple training datasets are generated, each containing between ∼400 and ∼600 samples. These datasets are designed to vary in their composition, differing in the packers or obfuscators used and in their label distributions.

In addition to the training datasets, a larger reference dataset of approximately 1500 samples is prepared. This dataset is used exclusively for testing in order to evaluate how well each model generalizes. All models are trained on the smaller datasets but evaluated on the same reference dataset, ensuring consistent and comparable results across the experiment.

```
┌──[user@packing-box]─[/mnt/share]─────────
$ dataset make exp2-1 -f PE -p upx -n 500
«snipped»
┌──[user@packing-box]─[/mnt/share]─────────
$ dataset make exp2-2 -n 250 -p mpress
«snipped»
┌──[user@packing-box]─[/mnt/share]─────────
$ dataset update exp2-2 -s dataset-packed-pe/packed/MPRESS -l
   dataset-packed-pe/labels.json
«snipped»
┌──[user@packing-box]─[/mnt/share]─────────
$ dataset make exp2-3 -s Win10_output.iso -n 500
```

The terminal output shows examples of commands used to compose the datasets. In the first example, the make command applies UPX to samples collected from the default source of Packing Box. The second and third commands create datasets using both Packing Box default source and the Dataset of packed PE files repository. The last command composes a dataset from files extracted from a Windows ISO using the `filter-archive` tool.

All datasets used in this experiment were composed using a combination of these methods. The following terminal output lists all the created datasets, providing a quick overview of the diversity of packers included in each. We can sse that some datasets are well-balanced, while others skew toward non-packed samples. Several include multiple packers while others have only one.

```
┌──[user@packing-box]──[/mnt/share]─────────
$ dataset list --hide-files

 Datasets (11)
  Name        #Executables    Size     Formats             Packers
 ──────────────────────────────────────────────────────────────────────
  exp2-0      823             266MB    MSDOS,PE32,PE64     mew{116},rlpack{115}
  exp2-1      500             269MB    PE32,PE64           upx{257}
  exp2-2      476             193MB    .NET,PE32,PE64      mpress{271}
  exp2-3      608             192MB    PE                  upx{122},exe32pack{86}
  exp2-4      547             153MB    PE                  fsg{117},aspack{80}
  exp2-5      472             470MB    PE                  themida{123},molebox{119}
  exp2-6      449             270MB    PE                  petite{147},upx{102}
  exp2-7      427             140MB    .NET,PE             winupack{120}
  exp2-8      546             227MB    PE                  neolite{116}
  exp2-9      424             193MB    PE                  amber{152},upx{122}
  exp2-ref    1544            640MB    PE,PE32             upx{319},amber{152},
   pecompact{124},aspack{118},exe32pack{58}
```

5.3.2 Training and Testing

Once all datasets have been created, the next step is to collect the data required for calibrating the metrics. This involves obtaining both the score of each metric and a performance score from a model, since both must be correlated. The process begins by assessing each dataset and recording the individual metric scores. Next, models are trained and evaluated. For each of the training datasets, a model is trained using the Random Forest algorithm. Random Forest is chosen as the common algorithm for this experiment because it is reliable, interpretable, and offers a good balance between performance and computational complexity. This makes it particularly well suited for comparing datasets on an equal footing.

```
┌──[user@packing-box]──[/mnt/share]─────────
$ model train exp2-0 -A rf
«snipped»

┌──[user@packing-box]──[/mnt/share]─────────
$ model train exp2-1 -A rf
«snipped»

┌──[user@packing-box]──[/mnt/share]─────────
$ model train exp2-9 -A rf
```

Once trained, each model is evaluated on the same reference dataset prepared in the first step. This approach ensures that all models are tested under identical conditions, allowing for a direct comparison of the relationship between dataset quality and model performance.

```
┌──[user@packing-box]──[/mnt/share]─────────
$ model test exp2-0_pe32-pe64_500_rf_f122 exp2-ref-fileless
«snipped»
Test results for:  exp2-ref-fileless
Classification metrics

  Accuracy    Precision    Recall     F-Measure     MCC       AUC
 ──────────────────────────────────────────────────────────────────
  76.05%      59.35%       99.22%     74.27%        60.76%    92.03%
```

```
┌──[user@packing-box]──[/mnt/share]─────────
$ model test exp2-1_pe32-pe64_500_rf_f122 exp2-ref
«snipped»
```

Once a model is tested, the six performance metrics are collected: Accuracy, Precision, Recall, F1-score, Matthews Correlation Coefficient (MCC), and Area Under the Curve (AUC). These are then combined to compute a Reference Performance Score (RPS) for each dataset. The RPS is a single aggregated performance indicator, obtained by taking the simple average of all six metrics. This approach provides a balanced measure that captures multiple aspects of model performance while avoiding bias toward any single metric. Additionally, it simplifies the subsequent correlation analysis by reducing the evaluation results to one representative score per dataset.

Name	Accuracy	Precision	Recall	F-Measure	MCC	AUC	RPS
exp2-0	65.58%	59.20%	100.00%	74.37%	43.01%	85.14%	71.22
exp2-1	86.38%	79.72%	97.52%	87.73%	74.64%	92.92%	86.49
exp2-2	71.64%	63.78%	100.00%	77.89%	52.59%	94.83%	76.79
exp2-3	91.62%	92.85%	91.62%	91.58%	84.49%	0.45%	75.44
exp2-4	82.92%	74.56%	99.87%	85.38%	70.00%	88.37%	83.52
exp2-5	68.38%	61.23%	100.00%	75.95%	47.50%	82.23%	72.55
exp2-6	86.57%	78.81%	100.00%	88.15%	75.94%	99.78%	88.21
exp2-7	70.99%	63.25%	100.00%	77.49%	51.58%	84.69%	74.50
exp2-8	53.13%	51.58%	100.00%	68.06%	18.14%	84.78%	62.28
exp2-9	69.49%	69.66%	68.93%	69.29%	38.98%	86.36%	67.62

Table 5.3: Summary of model performances

The table 5.3 summarizes the model's performance for each dataset, as well as the computed RPS. For convenience, the displayed name is the name of the dataset, but the values shown represent the model's performance.

5.3.3 Results

With all the necessary data collected, the next step is to calibrate the dataset metric weights. This is done using a grid search, an exhaustive search method that evaluates all possible combinations of weights from a predefined set of values. For each dataset, its combined quality score is computed multiple times, each time using a different set of weights for the eight dataset-quality metrics. For every weight configuration, the datasets are ranked according to their computed quality score. This ranking is then compared to the RPS-based ranking using the Spearman rank correlation coefficient.

The Spearman correlation measures the strength and direction of a monotonic relationship between two ranked variables. A value of +1 indicates perfectly aligned rankings, 0 indicates no correlation, and 1 represents a perfectly inverse ranking. In this context, a higher Spearman value means that the dataset-quality score ranking is consistent with the actual model performance ranking, indicating that the chosen weight configuration better reflects real-world predictive performance.

Completeness	Consistency	File Balance	Label Balance	Outliers	Portability	Similarity	Uniqueness
0.2	0.6	0.9	0.2	0.8	0.2	0.8	0.1
0.2	0.7	0.9	0.2	0.8	0.2	0.8	0.4
0.6	0.2	0.9	0.2	1.0	0.3	0.4	0.2
0.6	0.6	0.8	0.2	0.9	0.2	0.6	0.6

Table 5.4: Top Weight combinations for data quality metrics

The table 5.4 presents the four weight combinations that achieved the highest correlation with model performance. By examining these results, we can interpret the relative importance of each metric.

- **Completeness and Consistency** consistently receive low weights across the best combinations. This outcome is reasonable since metadata issues are rare in the way Packing Box composes datasets, and thus have little impact on downstream performance. In fact, in our experiments, these metrics were almost always perfect or nearly perfect. Because they offer little variation, they naturally contribute less to explaining performance differences.
- **Balance metrics** show more surprising results. **File Balance** receives relatively high weights (0.80.9), despite the tested datasets scoring consistently well on this dimension. This suggests that diversity in file types, architectures, and size ranges plays an important role in generalization. In other words, even if our datasets already scored well on this metric, its influence on model robustness appears substantial. In contrast, **Label Balance** is consistently given low weights (0.2), which was unexpected. Across our datasets, label distributions varied considerably, giving this metric a wide range. Nevertheless, the low weight indicates that variations in packed/unpacked ratios did not significantly alter model performance in practice.
- **Outliers** stand out with higher weights compared to many other metrics. This finding highlights that the presence of atypical samples does indeed affect model quality more than anticipated. Outliers are relatively rare, but their impact may come from distorting decision boundaries or introducing noise into the training process.
- **Portability**, like Label Balance, tends to receive low weights, which aligns with expectations. While this metric was among the most variable across datasets, its low correlation with predictive performance confirms our intuition that portability is less critical for the Machine Learning tasks explored here.
- **Similarity** is consistently assigned high weights, underscoring the strong effect of near-duplicate files on dataset quality and model performance. This confirms that deduplication is an essential step. **Uniqueness**, by contrast, receives lower but non-negligible weights. This is in line with expectations, as exact duplicates are rare in practice, yet removing them still contributes positively to dataset quality and should not be ignored.

Completeness	Consistency	File Balance	Label Balance	Outliers	Portability	Similarity	Uniqueness
0.4	0.5	0.9	0.2	0.9	0.2	0.65	0.3

Table 5.5: Choosen Weight combinations for data quality metrics

Since several weight configurations achieved the same optimal Spearman correlation, we adopted a consensus approach by averaging across the best solutions, ensuring robustness and avoiding arbitrary preference for a single combination. The resulting consensus weights are shown in table 5.5 and will serve as the final configuration in the following experiments. While some of the observed weight distributions were expected and others more surprising, their relative importance will be further examined in the next experiment to validate and better understand their impact on model performance.

FINDINGS

Overall, the results indicate that metrics capturing dataset **diversity** (File Balance) and **redundancy** (Similarity, Uniqueness) dominate the calibration process, strongly influencing alignment with model performance. In contrast, metadata-related dimensions (Completeness, Consistency) and structural properties (Portability, Label Balance) play a more limited role, as they were generally stable across datasets and had little effect on predictive accuracy. Interestingly, **Outliers** emerged as highly influential, suggesting that even a small number of atypical samples can disproportionately impact downstream learning outcomes.

Since multiple weight configurations achieved the same optimal Spearman correlation, we adopted a consensus approach by averaging across them. This provides a robust and non-arbitrary final weight configuration.



These calibrated weights should not be regarded as definitive. Their validity is inherently dependent on the datasets and evaluation conditions used in this experiment. Applying the calibration procedure to alternative or more diverse datasets may yield different results, particularly if hidden biases or unobserved characteristics are present. Consequently, these weights represent a robust configuration for the current setting but remain open to re-evaluation in future applications.

5.4 Experiment 3: Dataset Quality Improvements

The goal of this final experiment is to investigate whether improving a dataset's quality leads to measurable gains in model performance. Using the newly enhanced fix command, which applies targeted transformations to improve individual dataset-quality metrics, we aim to evaluate whether an increase in the dataset's quality score translates into higher predictive performance.

In addition to evaluating standard datasets containing binary files, this experiment also examines the behavior of `FilelessDataset` structures, which can only compute a subset of five metrics due to the absence of the original binaries. The objective is to determine whether the same quality-improvement process can enhance their performance in a manner similar to full datasets.

This experiment provides a direct test of the relationship between the dataset-quality score and the performance of models trained on that dataset. Several datasets from the previous experiments are reused here, along with newly composed ones, ensuring that the case of `FilelessDataset` is also covered in the evaluation.

5.4.1 Baseline

We begin by assessing a set of datasets, including some previously evaluated in earlier experiments. Since the metric weights have been calibrated, reassessing these datasets is necessary to obtain an accurate baseline for this experiment. This updated assessment provides the new reference quality scores for comparison. As a starting point, we consider a dataset composed of 257 UPX-packed binaries and 327 non-packed samples.

```
┌──[user@packing-box]──[/mnt/share]─────────
$ dataset assess exp3-1

Dataset quality

      Name          Score    Weight
──────────────────────────────────────
  Completeness      1.000    0.4
  Consistency       1.000    0.5
  File Balance      0.927    0.9
  Label Balance     0.912    0.2
  Outliers          1.000    0.9
  Portability       0.715    0.2
  Similarity        0.963    0.65
  Uniqueness        1.000    0.5

     Overall        0.961
```

As shown in the assessment results, this dataset already exhibits high overall quality. The lowest score is observed in portability, but given its relatively low weight, the impact on the global quality score is minimal. Minor improvements could be achieved by slightly improving label balance and reducing near-duplicate samples. Beyond these aspects, the remaining metrics are already optimal.

We now proceed to train a model on this dataset and evaluate its performance across different test datasets. First, the model is trained on the selected dataset and tested against another dataset (exp2-3) composed of samples packed with the same packer, UPX, and exe32pack.

```
┌──[user@packing-box]──[/mnt/share]─────────
$ model train exp3-1 -A rf
«snipped»
┌──[user@packing-box]──[/mnt/share]─────────
$ model test exp3-1_msdos-pe32-pe64_674_rf_f137 exp2-3
«snipped»
  Accuracy    Precision    Recall    F-Measure    MCC       AUC
──────────────────────────────────────────────────────────────────
  83.70%      87.23%       83.70%    82.67%       68.73%    99.78%
```

The model achieves relatively high predictive performance, which can be attributed to the fact that it was tested against another dataset containing samples packed with the same packer.

To further examine generalization, we also test the model against a dataset that includes compressed samples packed with methods other than UPX. This setup allows us to compare performance when the training and test datasets share the same packer versus when they differ.

```
┌──[user@packing-box]──[/mnt/share]─────────
$ model test exp3-1_msdos-pe32-pe64_674_rf_f137 exp2-4
«snipped»
  Accuracy    Precision    Recall    F-Measure    MCC       AUC
──────────────────────────────────────────────────────────────────
  78.53%      74.89%       100.00%   85.64%       54.94%    99.91%
```

The model generalizes less effectively in this setting-achieving perfect detection coverage (recall) but suffering from precision loss, as expected when testing on previously unseen packers. Nonetheless, overall performance remains relatively strong.

In addition, we also need to establish a baseline for the `FilelessDataset` format. To this end, one of the datasets from the previous experiment is converted into a `FilelessDataset` and then assessed using the calibrated weights to obtain its quality score.

```
┌──[user@packing-box]──[/mnt/share]─────────
$ dataset convert exp2-7 -n exp3-2
«snipped»
┌──[user@packing-box]──[/mnt/share]─────────
$ dataset assess exp3-2

Dataset quality

      Name          Score    Weight
──────────────────────────────────────
  Completeness      0.945    0.4
  Consistency       1.000    0.5
  File Balance      0.869    0.9
  Label Balance     0.662    0.2
  Outliers          0.981    0.9
  Uniqueness        1.000    0.5

     Overall        0.934
```

For this `FilelessDataset`, the quality score is also relatively high, though slightly lower than the previous dataset. The completeness metric shows a minor drop, indicating that some features were not computed for certain samples. Label balance also presents a noticeable weakness, which is precisely why this dataset was selected-to examine whether improving this gap can enhance downstream model performance.

The outlier metric is reduced as well, reflecting that several anomalies were detected by DBSCAN. Aside from these aspects, the dataset performs strongly, with both uniqueness and consistency reaching optimal levels.

We also evaluate the performance of the `FilelessDataset` on the same test datasets used for the standard `Dataset`. This direct comparison allows us to see whether improvements in its limited set of quality metrics can still lead to better model performance.

```
┌──[user@packing-box]──[/mnt/share]─────────
$ model train exp3-2 -A rf
«snipped»
┌──[user@packing-box]──[/mnt/share]─────────
$ model test exp3-2_pe_427_rf_f139 exp2-3
«snipped»
  Accuracy    Precision    Recall    F-Measure    MCC       AUC
──────────────────────────────────────────────────────────────
  83.70%      87.23%       83.70%    82.67%       68.73%    80.85%
```

For this first test with different packers, the model achieves relatively good results overall, though the MCC is noticeably weaker. To further examine its generalization ability, we now test it on the second dataset that also contains different packers.

```
┌──[user@packing-box]──[/mnt/share]─────────
$ model test exp3-2_pe_427_rf_f139 exp2-4
«snipped»
  Accuracy    Precision    Recall    F-Measure    MCC       AUC
──────────────────────────────────────────────────────────────
  64.04%      64.04%       100.00%   78.08%       0.00%     99.98%
```

The results show that while recall remains perfect, precision and overall accuracy drop significantly, leading to an MCC of 0.00. This indicates that the model tends to over-predict the packed class, misclassifying many clean samples.

Now that we have gathered baseline results for both dataset structures across different test sets, we can try to fix the datasets and examine whether targeted improvements in dataset quality can lead to more balanced and reliable performance.

5.4.2 Dataset Fixing

With the baseline established - including the quality scores of various datasets and their corresponding model performance on different target datasets - we proceed to apply the `fix` command. This command applies targeted transformations aimed at improving individual quality metrics, as described in Section 4.3.4. After applying the command, we reassess each dataset to verify whether the intended improvements in the metrics have been achieved.

```
┌──[user@packing-box]──[/mnt/share]─────────
$ dataset fix exp3-1
«snipped»
┌──[user@packing-box]──[/mnt/share]─────────
$ dataset assess exp3-1

Dataset quality

      Name          Score    Weight
──────────────────────────────────────
  Completeness      1.000    0.4
  Consistency       1.000    0.5
  File Balance      0.949    0.9
  Label Balance     0.935    0.2
  Outliers          1.000    0.9
  Portability       0.729    0.2
  Similarity        1.000    0.65
  Uniqueness        1.000    0.5

     Overall        0.973
```

Since this dataset already exhibited a high quality score, the improvements achieved by the `fix` command are relatively modest, with the overall score increasing only slightly from 0.961 to 0.973. Minor adjustments can be observed in file balance, label balance, and portability, though these changes are not substantial. The similarity metric, however, has reached an optimal value, indicating that all near-duplicate pairs above the default threshold were successfully removed.

Following these adjustments, models are retrained on the improved datasets using the same procedure and parameters as in the baseline phase. This controlled setup ensures that any observed performance differences can be attributed directly to changes in dataset quality rather than model configuration. The newly trained models are then evaluated on the same set of target datasets as before, allowing a direct before-and-after comparison of predictive performance.

```
┌──[user@packing-box]──[/mnt/share]─────────
$ model test exp3-1_pe32-pe64_647_rf_f137 exp2-3
«snipped»
  Accuracy    Precision    Recall     F-Measure     MCC       AUC
──────────────────────────────────────────────────────────────────
  83.70%      87.23%       83.70%     82.67%        68.73%    99.14%
```

For this first test, the results remain identical across all performance metrics except for a marginal decrease in AUC (99.78% to 99.14%). This slight decline is negligible in practice and does not meaningfully affect the overall outcome. We now turn to the second test dataset to examine whether more substantial changes can be observed.

```
┌──[user@packing-box]──[/mnt/share]─────────
$ model test exp3-1_pe32-pe64_647_rf_f137 exp2-4
«snipped»
  Accuracy    Precision    Recall     F-Measure     MCC       AUC
──────────────────────────────────────────────────────────────────
 83.12%      79.14%       100.00%    88.35%        64.80%    99.92%
```

This time, we can observe a clear improvement across all performance metrics. Accuracy, precision, F-measure, and MCC all show noticeable gains, while recall remains at its maximum and AUC increases

slightly. These results suggest that the dataset adjustments made through the `fix` command had a tangible positive effect.

The same procedure is also applied to the `FilelessDataset`s, in order to examine whether improvements in their more limited set of quality metrics likewise translate into measurable gains in predictive accuracy.

```
┌──[user@packing-box]──[/mnt/share]─────────
$ dataset fix exp3-2
«snipped»
┌──[user@packing-box]──[/mnt/share]─────────
$ dataset assess exp3-2

Dataset quality

      Name          Score    Weight
──────────────────────────────────────
  Completeness      0.952    0.4
  Consistency       1.000    0.5
  File Balance      0.876    0.9
  Label Balance     0.683    0.2
  Outliers          1.000    0.9
  Uniqueness        1.000    0.5

     Overall        0.943
```

We can observe a modest improvement in the overall quality score. The outliers detected by DBSCAN have been removed, while both file and label balance show slight gains. Completeness has also increased. Although the changes are not drastic, they indicate that the `fix` command successfully addressed the main weaknesses identified in the baseline assessment. We will now train and test the newly fixed dataset.

```
┌──[user@packing-box]──[/mnt/share]─────────
$ model test exp3-2_pe_398_rf_f139 exp2-3
«snipped»
  Accuracy    Precision    Recall     F-Measure     MCC       AUC
──────────────────────────────────────────────────────────────────
  83.70%      87.23%       83.70%     82.67%        68.73%    82.67%
```

For the first test dataset, results remain consistent with previous runs, showing stable performance across all metrics with no significant changes.

```
┌──[user@packing-box]──[/mnt/share]─────────
$ model test exp3-2_pe_398_rf_f139 exp2-4
«snipped»
  Accuracy    Precision    Recall     F-Measure     MCC       AUC
──────────────────────────────────────────────────────────────────
  64.04%      64.04%       100.00%    78.08%        0.00%     100.00%
```

For the second test dataset, the model again achieves perfect recall but struggles with precision and MCC, despite reaching a maximal AUC.

5.4.3 Results

The results of this experiment are presented by comparing the performance of the model pre- and post-fix performance for each dataset, with particular attention to the differences between `Datasets` and `FilelessDataset`

Model / Test	Accuracy	Precision	Recall	F-Measure	MCC	AUC
Model 1-Test 1 (Pre)	83.70%	87.23%	83.70%	82.67%	68.73%	99.78%
Model 1-Test 1 (Post)	83.70%	87.23%	83.70%	82.67%	68.73%	99.14%
Model 1-Test 2 (Pre)	78.53%	74.89%	100.00%	85.64%	54.94%	99.91%
Model 1-Test 2 (Post)	83.12%	79.14%	100.00%	88.35%	64.80%	99.92%
Model 2-Test 1 (Pre)	83.70%	87.23%	83.70%	82.67%	68.73%	80.85%
Model 2-Test 1 (Post)	83.70%	87.23%	83.70%	82.67%	68.73%	82.67%
Model 2-Test 2 (Pre)	64.04%	64.04%	100.00%	78.08%	0.00%	99.98%
Model 2-Test 2 (Post)	64.04%	64.04%	100.00%	78.08%	0.00%	100.00%

Table 5.6: Evaluation metrics for two models across two tests

The table 5.6 summarizes the performance of the models trained on the `Dataset` (Model 1) and the `FilelessDataset` (Model 2) across both test sets.

For Model 1, when tested on the first dataset (containing samples with the same packer), the results remained essentially unchanged, with only a minimal decrease in AUC. This marginal variation is not expected to have any real impact on predictive capability. This suggests that when the training and test datasets share very similar characteristics, further improving dataset quality does not necessarily yield measurable performance gains since the baseline dataset was already sufficient for this controlled case. It could also suggest that the improvements in quality were too minimal to produce a noticeable impact.

In contrast, the second test dataset-composed of samples packed with different packers than those used for training-shows clear improvements across all metrics after applying the `fix` command. Accuracy, precision, and F-measure increased notably, while recall remained at 100%. The MCC, a strong indicator of overall model reliability, also increased significantly. These improvements suggest that the dataset refinements reduced redundancy and imbalance, yielding a training set that better generalized to previously unseen packing methods. In particular, the gains in precision point to fewer false positives, which is consistent with the removal of near-duplicates and the rebalancing of labels and files.

For Model 2, the results are more underwhelming. Across both tests, all metrics except AUC remain unchanged, showing neither improvement nor decline. The observed increases in AUC are marginal-reaching 100% in Test 2 and showing a slightly larger but still modest gain in Test 1.

Since only a subset of quality metrics can be computed (and therefore improved), the `fix` command has less leverage to meaningfully alter the datasets composition. As a result, performance gains are naturally more constrained compared to standard datasets. In particular, the similarity metric cannot be improved within the `FilelessDataset` structure. Yet, as highlighted in the literature review, deduplication is considered a critical factor for ensuring dataset diversity and reliability.

To further investigate this limitation, we conduct one final experiment using the dataset `exp3_2` prior to its conversion into a `FilelessDataset`, allowing us to act directly on the similarity metric.

```
┌──[user@packing-box]──[/mnt/share]─────────
$ dataset fix exp2-7
«snipped»
┌──[user@packing-box]──[/mnt/share]─────────
$ model train exp2-7 -A rf
«snipped»
┌──[user@packing-box]──[/mnt/share]─────────
$ model test exp2-7_net-pe32-pe64_401_rf_f139 exp2-3
«snipped»
┌──[user@packing-box]──[/mnt/share]─────────
$ model test exp2-7_net-pe32-pe64_401_rf_f139 exp2-4
«snipped»
```

Model / Test	Accuracy	Precision	Recall	F-Measure	MCC	AUC
Model 2-Test 1 (Pre)	83.70%	87.23%	83.70%	82.67%	68.73%	80.85%
Model 2-Test 1 (Post)	83.70%	87.23%	83.70%	82.67%	68.73%	81.34%
Model 2-Test 2 (Pre)	64.04%	64.04%	100.00%	78.08%	0.00%	99.98%
Model 2-Test 2 (Post)	64.04%	64.04%	100.00%	78.08%	0.00%	99.98%

Table 5.7: Evaluation metrics for non-converted model 2

Once again, we observe virtually no difference across the metrics, with only a marginal increase in AUC for the first test set. This outcome was unexpected, as improving similarity by removing near-duplicates was anticipated to provide clearer benefits. A possible explanation is that the baseline dataset already contained relatively few redundant samples, meaning that deduplication had little practical effect. Another possibility is that the test datasets themselves introduced stronger biases (e.g., different packers) that overshadowed the marginal quality improvements. In other words, the impact of similarity cleaning may only become visible in larger or more redundant datasets, whereas here the effect is too minimal to meaningfully shift the results.

FINDINGS

In this experiment, we investigated whether applying the `fix` command to improve dataset-quality metrics translates into measurable gains in model performance. Both standard `Datasets` and `FilelessDatasets` were evaluated, before and after fixing, across multiple test sets.

Results showed that **quality improvements** can, in some cases, lead to **better predictive performance** – most clearly when training and testing involve different packers. However, in other cases, especially when training and test datasets were already highly similar, the impact was negligible. Attempts to isolate the effect of similarity cleaning (deduplication) also yielded limited improvements, contrary to expectations.

These mitigated results suggest several possibilities: the effect of deduplication may only become significant in larger or more redundant datasets, minor quality improvements may be insufficient to shift performance in measurable ways, and the datasets generated by Packing Box might already achieve relatively high quality, leaving little room for further gains.

SUMMARY

The first experiment assessed six **similarity hashing techniques** to determine their **suitability for detecting near-duplicate** packed binaries. Both runtime efficiency and detection consistency were evaluated. Results showed that ssdeep remains a strong and reliable choice, combining speed with consistent pair detection. TLSH demonstrated comparable runtime performance and could serve as a viable alternative when properly normalized. SDhash and MRSH-v2 also produced useful results, though MRSH-v2 incurred higher processing times. In contrast, LZJD and LSH were found unsuitable due to their high cost and limited detection rates. Based on these findings, ssdeep was selected as the default hashing method for the Packing Box, with TLSH, SDhash, and MRSH-v2 offered as optional alternatives.

The second experiment focused on **calibrating the dataset-quality metrics** to ensure they reflect their actual impact on predictive performance. Random Forest classifiers were trained across multiple datasets, with the Reference Performance Score (RPS) serving as a unified performance benchmark. A grid search over metric weight configurations was conducted, with Spearman correlation used to compare metric-based rankings against RPS-based rankings. The results highlighted that File Balance-capturing dataset diversityand redundancy-related metrics (Similarity and Uniqueness) dominate predictive performance, while metadata-based metrics (Completeness, Consistency) and structural properties (Portability, Label Balance) play a more limited role. Outliers also showed to be influential. Since multiple weight sets achieved the same optimal correlation, a consensus averaging strategy was adopted, yielding the final configuration used in subsequent experiments.

The final experiment investigated whether **improving dataset quality** through the new `fix` command could translate into measurable model performance gains. Both standard `Datasets` and `FilelessDatasets` were tested before and after fixing. Results indicated that improvements are most visible when training and testing involve different packers, where models trained on improved datasets showed stronger generalization. Conversely, when training and test sets shared very similar characteristics, the impact of fixing was negligible, as baseline quality was already sufficient. For `FilelessDatasets`, improvements were minimal, reflecting the smaller set of metrics that can be optimized. An additional test explored the effect of deduplication, but the expected gains from similarity cleaning did not materialize, possibly due to the relatively high quality of the datasets already produced by the Packing Box.

6 Conclusion

6.1 Summary **75**

6.2 Achievements **75**

6.2.1 Objectives 75

6.2.2 Research Questions 76

6.2.3 Contributions 76

6.3 Future Works **77**

6.3.1 Improvements 77

THIS chapter wraps up this work, stating the objectives reached and concluding with points of improvement and future works.

It summarizes our contributions to the toolbox and the key findings from our experiments. It enumerates the objectives and how they were reached; it also answers our research questions. It finally presents some ideas for improvements.

6.1 Summary

The use of packing, a collection of binary transformations such as compression, encryption, anti-reversing protections, and other runtime-preserving operations, has steadily increased since the early 2000s. While packing serves legitimate purposes like software protection, it is also widely exploited for malicious goals, particularly malware evasion and obfuscation. Detecting packed binaries remains a difficult task due to the sophistication of packers, and this challenge is compounded by the lack of reliable and consistent datasets. Without such datasets, it becomes difficult to meaningfully compare different detection methods, especially those based on Machine Learning.

The current literature has explored various ways of collecting samples and proposed dimensions for reasoning about dataset quality, but so far lacked concrete methods for systematically assessing quality in binary datasets. A deeper review reveals clear trends, such as the reliance on Windows installations for data collection and the significant impact that dataset quality has on model performance. These insights can serve as guidance for shaping our own approach and refining our goal.

The Packing Box, introduced in 2022, addresses part of the observed gap by providing an open-source framework for dataset generation, model training, and evaluation. Building upon this foundation, our work extended the framework with new capabilities, including an archive filtering utility, similarity search through hashing methods, a set of dataset quality metrics, and finally, a quality improvement procedure. Using these tools, our experiments evaluate whether applying the quality improvement procedure enhances dataset performance. This analysis highlights the impact of dataset quality on model outcomes.

6.2 Achievements

This section focuses on the achieved goals and how we contributed to the Packing Box framework.

6.2.1 Objectives

This subsection is a review of the achieved objectives (as defined in Section 1.2):

1. Chapter 2 stated the **necessary information** to understand packing, dataset and similarity hashing.
   - A – We described the fundamental structure of executable files, with particular emphasis on the Portable Executable format.
   - B – We introduced the principles of Machine Learning, focusing on unsupervised learning techniques.
   - C – We highlighted the critical role of dataset quality.
   - D – We described the main similarity hashing methods currently in use.
2. Chapter 3 reviews **related work** in the field of dataset creation, quality metrics and similarity hashing.
   - A – We presented related literature about dataset creation techniques.
   - B – We outlined current assessment methods applied in similar research contexts.
   - C – We examined prior work on similarity detection techniques, identifying their trends.
3. Chapter 4 introduced the **Packing Box** and detailed our contributions to it.
   - A – We presented the architecture, pipeline and tools from the Packing Box that will be used.
   - B – We integrated and presented new tool designed to enhance the Packing Box's capabilities.
   - C – We implemented and applied quality assessment metrics to strengthen the framework.

4. Chapter 5 presents our **experiments**, which were designed to answer our research questions and demonstrate the utility of the new tools we developed.
   A – We evaluated multiple hashing methods under various conditions.
   B – We illustrated how weighting quality metrics impacts overall results.
   C – We analyzed model performance resulting from the application of quality enhancement techniques.

6.2.2 Research Questions

In parallel of the objectives, we defined some research questions that we can now answer:

Question 1 **Can we extend the Packing Box framework with new useful features regarding dataset generation?**
Yes, we extended the framework with features that improved dataset creation and early curation. The first addition, Filter-Archive, enables selective extraction from large archives and disk images, such as Windows ISOs, while applying format filters, extending the possible source for binary samples. A second addition introduced similarity thresholding, based on fuzzy hashing methods, which allows near-duplicate filtering to be applied either during collection or during later cleaning stages.

Question 2 **What measurable criteria and metrics can be defined to assess the quality of datasets used for packed executable detection?**
To address this question, eight quality metrics were defined, grounded both in the literature on data quality and in the specific challenges of datasets for packed binary analysis. Completeness measures whether required metadata fields in the dataset are fully populated, while Consistency verifies coherence between entries and files. Label Balance evaluates the class distribution , while File Balance captures diversity across the samples. Outliers are detected through anomalous characteristics. Portability examines format-specific properties related to cross-environment robustness, while Similarity penalizes near-duplicates using fuzzy hashing. Finally, Uniqueness detects exact duplicates at the file level and can also capture duplicate metadata entries. These metrics are integrated into the Packing Box workflow through the assess and fix commands.

Question 3 **How does the quality of a dataset, as measured by these metrics, influence the performance of machine learning models for static packing detection?**
The experiments showed that dataset quality does seem to influence model performance, though the effect depends on context. Improvements from the fix command were most visible when training and test datasets differed in packer families. In contrast, when datasets were very similar, gains were negligible. Overall, this indicates that quality improvements are most valuable for enhancing generalization rather than fine-tuning performance on closely aligned datasets.

6.2.3 Contributions

Our contributions are the following:

✓ Through our contributions to the **Packing Box**, improving its ingestion capabilities and enabling more efficient dataset generation.

✓ We defined a set of quality metrics that provide a structured way to assess the quality of datasets for packed executable static detection.

✓ Through our experiments, we demonstrated that improving these quality metrics can lead to gains in model performance.

6.3 Future Works

This section closes this work by providing ideas we collected along this work for improvements.

6.3.1 Improvements

File Balance While the current metrics are defined, some could benefit from alternative or extended computation methods. For example, the file balance metric is currently based on arbitrarily chosen parameters. A more robust approach would derive this metric from the extracted features of each file, providing a data-driven measure of balance within the dataset. This would more accurately capture the true diversity of samples, rather than relying on fixed thresholds.

Fix Command Currently, the command improves the quality metric by removing files that negatively impact overall quality. A possible improvement would be to incorporate an external dataset, allowing the method not only to remove low-quality samples but also to add higher-quality ones. This dual strategy could lead to a more robust and balanced dataset.

Additional Sources The current implementation includes the new filter-archive utility, which allows collecting samples from sources like Windows ISO files. A potential improvement would be to integrate additional tools capable of harvesting large volumes of samples from other sources-for example, public repositories or platforms. This would significantly expand the dataset's coverage and diversity.

References

[1] Alexandre D'Hondt, Charles Henry Bertrand Van Ouytsel, and Axel Legay. "Experimental toolkit for manipulating executable packing". In: *Risks and security of internet and systems*. Ed. by Abderrahim Ait Wakrime et al. Springer Nature Switzerland, 2024.

[2] Alexandre D'Hondt. *packing-box/docker-packing-box*. 2025.

[3] Alexandre D'Hondt. "Experimental Toolkit for Studying Executable Packing - Analysis of the State-of-the-Art Packing Detection Techniques". EN. MA thesis. Université Catholique de Louvain, 2022.

[4] Sébastien Martinez Balbuena. "Static Features Exploration for Executable Packing with Unsupervised Learning". MA thesis. Université Catholique de Louvain, 2023.

[5] Tzu-Yen Wang and Chin-Hsiung Wu. "Detection of packed executables using support vector machines". In: *2011 International Conference on Machine Learning and Cybernetics*. 2011.

[6] Asaf Shabtai et al. "Detecting unknown malicious code by applying classification techniques on Op-Code patterns". In: *Security Informatics* 1 (2012).

[7] Charles Lim and Nicsen. "Mal-EVE: Static detection model for evasive malware". In: *2015 10th International Conference on Communications and Networking in China (ChinaCom)*. 2015.

[8] Nguyen Minh Hai, Mizuhito Ogawa, and Quan Thanh Tho. "Packer identification based on metadata signature". en. In: *Proceedings of the 7th Software Security, Protection, and Reverse Engineering / Software Security and Protection Workshop*. ACM, 2017.

[9] Ferhat Ozgur Catak and Ahmet Faruk Yaz. *A Benchmark API Call Dataset for Windows PE Malware Classification*. 2021.

[10] Muhammad Irfan Yousuf et al. *Multi-feature Dataset for Windows PE Malware Classification*. 2022.

[11] Ryan Baker Del Aguila et al. "Static Malware Analysis Using Low-Parameter Machine Learning Models". en. In: *Computers* 3 (2024).

[12] Chang Liu et al. *Assemblage: Automatic Binary Dataset Construction for Machine Learning*. 2024.

[13] H. S. Anderson and P. Roth. *EMBER: An open dataset for training static PE malware machine learning models*. 2018.

[14] Roberto Perdisci, Andrea Lanzi, and Wenke Lee. "Classification of packed executables for accurate computer virus detection". In: *Pattern Recognition Letters* 14 (2008).

[15] Roberto Perdisci, Andrea Lanzi, and Wenke Lee. "McBoost: Boosting Scalability in Malware Collection and Analysis Using Statistical Classification of Executables". In: *2008 Annual Computer Security Applications Conference (ACSAC)*. 2008.

[16] M. Zubair Shafiq et al. "PE-Miner: Mining Structural Information to Detect Malicious Executables in Realtime". In: *Recent Advances in Intrusion Detection*. Ed. by Engin Kirda, Somesh Jha, and Davide Balzarotti. Springer Berlin Heidelberg, 2009.

[17] Neminath Hubballi and Himanshu Dogra. "Detecting Packed Executable File: Supervised or Anomaly Detection Method?" In: *2016 11th International Conference on Availability, Reliability and Security (ARES)*. 2016.

[18] Hojjat Aghakhani et al. "When Malware is Packin' Heat; Limits of Machine Learning Classifiers Based on Static Analysis Features". en. In: *Proceedings 2020 Network and Distributed System Security Symposium*. Internet Society, 2020.

[19] Nicola Loi, Claudio Borile, and Daniele Ucci. *Towards an Automated Pipeline for Detecting and Classifying Malware through Machine Learning*. 2021.

[20] Attaullah Buriro et al. "MalwD&C: A Quick and Accurate Machine Learning-Based Approach for Malware Detection and Categorization". en. In: *Applied Sciences* 4 (2023).

[21] Paul Royal et al. "PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware". In: *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*. 2006.

[22] Seungwon Han, Keungi Lee, and Sangjin Lee. "Packed PE File Detection for Malware Forensics". In: *2009 2nd International Conference on Computer Science and its Applications*. 2009.

[23] Silvio Cesare and Yang Xiang. "A Fast Flowgraph Based Classification System for Packed and Polymorphic Malware on the Endhost". In: *2010 24th IEEE International Conference on Advanced Information Networking and Applications*. 2010.

[24] Antonio Nappa, M. Zubair Rafique, and Juan Caballero. "Driving in the Cloud: An Analysis of Drive-by Download Operations and Abuse Reporting". en. In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Ed. by Konrad Rieck, Patrick Stewin, and Jean-Pierre Seifert. Springer, 2013.

[25] Singam Sai Bala Subrahmanyam et al. "A hybrid method for analysis and detection of malicious executables in IoT network". en. In: *Computers & Security* (2023).

[26] Smita Naval et al. "SPADE: Signature based PAcker DEtection". en. In: *Proceedings of the First International Conference on Security of Internet of Things*. ACM, 2012.

[27] Xabier Ugarte-Pedrero et al. "Countering entropy measure attacks on packed software detection". In: *2012 IEEE Consumer Communications and Networking Conference (CCNC)*. 2012.

[28] Colin Burgess et al. "Feature set reduction for the detection of packed executables". In: *25th IET Irish Signals & Systems Conference 2014 and 2014 China-Ireland International Conference on Information and Communications Technologies (ISSC 2014/CIICT 2014)*. 2014.

[29] Moustafa Saleh, E. Paul Ratazzi, and Shouhuai Xu. "A control flow graph-based signature for packer identification". In: *MILCOM 2017 - 2017 IEEE Military Communications Conference (MILCOM)*. 2017.

[30] Mi-Jung Choi et al. "All-in-One Framework for Detection, Unpacking, and Verification for Malware Analysis". en. In: *Security and Communication Networks* (2019).

[31] Xingwei Li et al. "A Consistently-Executing Graph-Based Approach for Malware Packer Identification". In: *IEEE Access* (2019).

[32] Erik Bergenholtz et al. "Detection of Metamorphic Malware Packers Using Multilayered LSTM Networks". en. In: *Information and Communications Security*. Ed. by Weizhi Meng et al. Springer International Publishing, 2020.

[33] Lamine Noureddine et al. "SE-PAC: A Self-Evolving PAcker Classifier against rapid packers evolution". en. In: *Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy*. ACM, 2021.

[34] Fabrizio Biondi et al. "Effective, efficient, and robust packing detection and classification". In: *Computers & Security* (2019).

[35] Faming Lu et al. "Research on the Construction of Malware Variant Datasets and Their Detection Method". en. In: *Applied Sciences* 15 (2022).

[36] Ehab Alkhateeb, Ali Ghorbani, and Arash Habibi Lashkari. "Identifying Malware Packers through Multilayer Feature Engineering in Static Analysis". en. In: *Information* 2 (2024).

[37] Yang-seo Choi et al. "PE File Header Analysis-Based Packed PE File Detection Technique (PHAD)". In: *International Symposium on Computer Science and its Applications*. 2008.

[38] Li Sun et al. "Pattern Recognition Techniques for the Classification of Malware Packers". en. In: *Information Security and Privacy*. Ed. by David Hutchison et al. Springer Berlin Heidelberg, 2010.

[39] Moustafa Saleh, E. Paul Ratazzi, and Shouhuai Xu. "Instructions-Based Detection of Sophisticated Obfuscation and Packing". In: *2014 IEEE Military Communications Conference*. 2014.

[40] Limin Yang et al. "BODMAS: An Open Dataset for Learning based Temporal Analysis of PE Malware". In: *2021 IEEE Security and Privacy Workshops (SPW)*. 2021.

[41] Charles-Henry Bertrand Van Ouytsel, Khanh Huu The Dam, and Axel Legay. "Analysis of machine learning approaches to packing detection". EN. In: *Computers & Security* (2023).

[42] Richard Y. Wang and Diane M. Strong. "Beyond Accuracy: What Data Quality Means to Data Consumers". en. In: *Journal of Management Information Systems* 4 (1996).

[43] Leo L. Pipino, Yang W. Lee, and Richard Y. Wang. "Data quality assessment". en. In: *Communications of the ACM* 4 (2002).

[44] Carlo Batini et al. "Methodologies for data quality assessment and improvement". en. In: *ACM Computing Surveys* 3 (2009).

[45] Li Cai and Yangyong Zhu. "The Challenges of Data Quality and Data Quality Assessment in the Big Data Era". In: *Data Science Journal* 0 (2015).

[46] Roland Croft, M. Ali Babar, and Mehdi Kholoosi. *Data Quality for Software Vulnerability Datasets*. 2023.

[47] Russell Miller et al. "A Framework for Current and New Data Quality Dimensions: An Overview". en. In: *Data* 12 (2024).

[48] Sedir Mohammed et al. "A Data Quality Glossary". en. In: (2024).

[49] Nitin Gupta et al. *Data Quality Toolkit: Automatic assessment of data quality and remediation for machine learning datasets*. 2021.

[50] Luís Santos and Luís Ferreira. "AtlanticAutomated data preprocessing framework for supervised machine learning". en. In: *Software Impacts* (2023).

[51] Mohamed Abdelaal et al. *DataLens: ML-Oriented Interactive Tabular Data Quality Dashboard*. 2025.

[52] Dijana Oreski, Stjepan Oreski, and Bozidar Klicek. "Effects of dataset characteristics on the performance of feature selection techniques". en. In: *Applied Soft Computing* (2017).

[53] Daniele Foroni, Matteo Lissandrini, and Yannis Velegrakis. "Estimating the extent of the effects of Data Quality through Observations". In: *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2021.

[54] Laura Morán-Fernández, Verónica Bólon-Canedo, and Amparo Alonso-Betanzos. "How important is data quality? Best classifiers vs best features". en. In: *Neurocomputing* (2022).

[55] Sedir Mohammed et al. "The effects of data quality on machine learning performance on tabular data". en. In: *Information Systems* (2025).

[56] Li Zhi, Chaki Sushil, and Dong Xu. "Experimental study of fuzzy hashing in malware clustering analysis". In: USENIX, 2015.

[57] Edward Raff and Charles K. Nicholas. *Lempel-Ziv Jaccard Distance, an Effective Alternative to Ssdeep and Sdhash*. 2018.

[58] Fabio Pagani, Matteo Dell'Amico, and Davide Balzarotti. "Beyond Precision and Recall: Understanding Uses (and Misuses) of Similarity Hashes in Binary Analysis". en. In: *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*. ACM, 2018.

[59] Ian Shiel and Stephen O'Shaughnessy. "Improving file-level fuzzy hashes for malware variant classification". en. In: *Digital Investigation* (2019).

[60] Anshumali Shrivastava and Ping Li. *In Defense of MinHash Over SimHash*. 2014.

[61] Ciprian Oprisa, Marius Checiches, and Adrian Nandrean. "Locality-sensitive hashing optimizations for fast malware clustering". In: *2014 IEEE 10th International Conference on Intelligent Computer Communication and Processing (ICCP)*. IEEE, 2014.

[62] Haoping Liu et al. "An Evaluation of Malware Triage Similarity Hashes". In: *Proceedings of the 25th International Conference on Enterprise Information Systems - Volume 1: ICEIS*, INSTICC. SciTePress, 2023.

A Hash Methods

A.1 Introduction . . . . . 1 A.2 Time performance 1

A.1 Introduction

In this appendix, we present additional results, details, and graphs collected during Experiment 1 that were omitted from the main section to avoid redundancy.

A.2 Time performance

As mentioned in the experiment, a comparison was conducted to measure the performance difference between using the Python library directly and invoking subprocess calls. The results, shown in the next figure A.1, indicate that avoiding subprocess calls yields better performance.

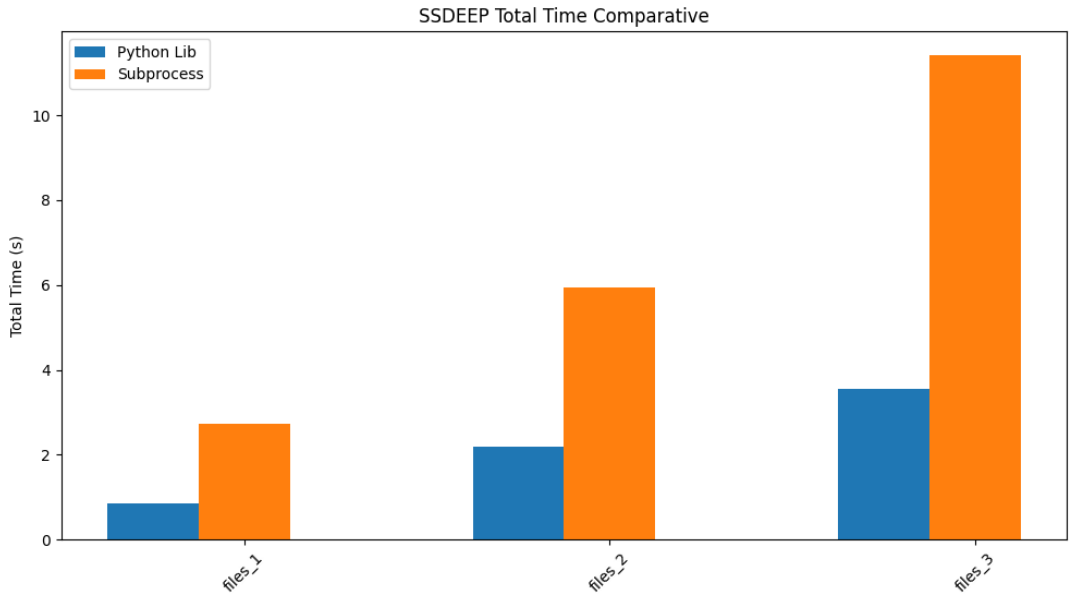


Figure A.1: Ssdeep comparative

It was also stated that LSH was excluded from the overall time performance graph 5.2, as its large processing time would make the graph less readable. The next figure illustrates this point-LSHs runtime already begins to overshadow the other results, even when showing data for only 600 files.

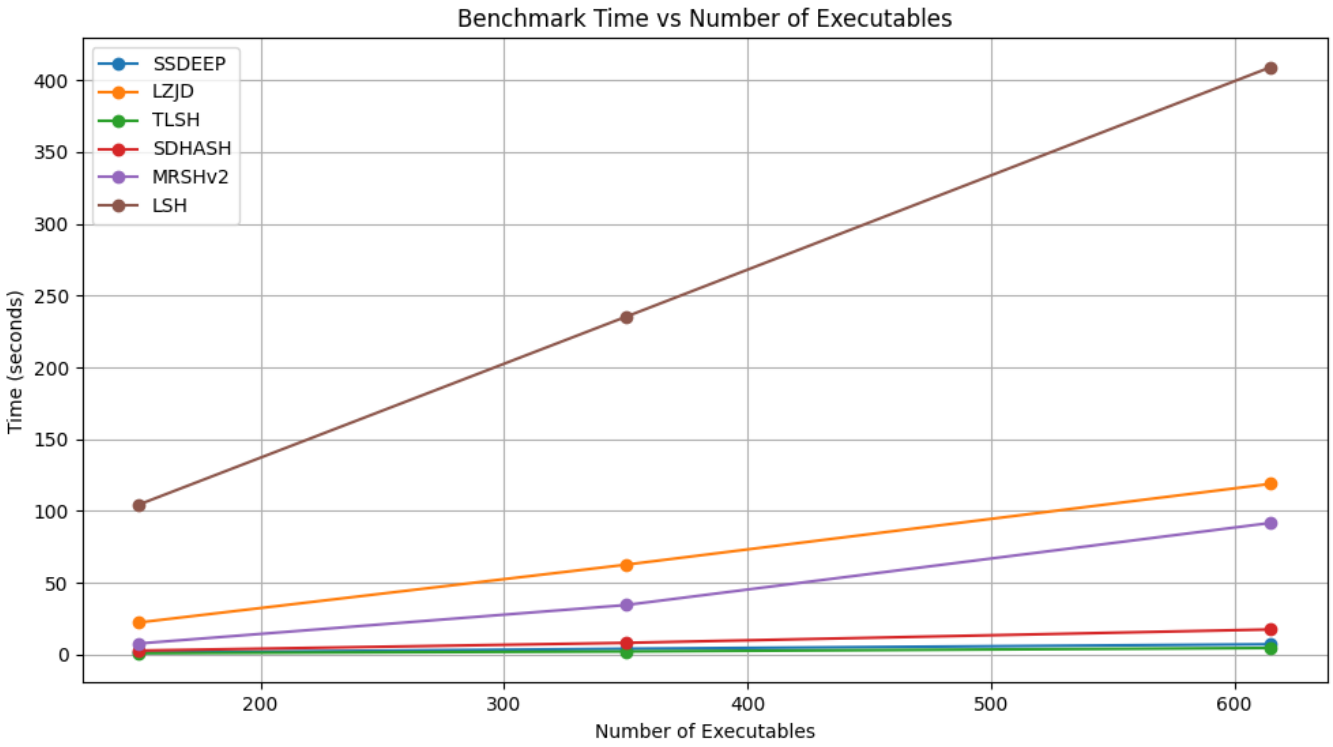


Figure A.2: Similarity Hashing Methods Benchmark with lsh

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