BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

MULTI-CRITERIA CLUSTERING OF FILES

MULTIKRITERIÁLNÍ SHLUKOVÁNÍ SOUBORŮ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. MATÚŠ JASNICKÝ**
AUTOR PRÁCE

SUPERVISOR **Ing. LUKÁŠ ZOBAL**
VEDOUCÍ PRÁCE

BRNO 2021

Brno University of Technology
Faculty of Information Technology

Department of Information Systems (DIFS) Academic year 2020/2021

Master's Thesis Specification

Student: **Jasnický Matúš, Bc.**
Programme: Information Technology and Artificial Intelligence Specialization: Cybersecurity
Title: **Multi-Criteria Clustering of Files**
Category: Security

Assignment:

1. Familiarize yourself with the topic of clustering. Focus on clustering of files based on their shared properties using multiple criteria at once.
2. Review Clusty, an internal Avast system used for clustering of files. Concentrate on its shortcomings, particularly with respect to its single-criteria method of clustering, performance, and availability.
3. Design a system that will be able to accept a stream of information about files on its input and produce clusters of these files based on their shared properties. Focus primarily on using multicriteria clustering methods and overcoming other shortcomings of Clusty. Experiment with various methods to see which performs the best. The system should be targeted at clustering; analysis of files, classification of clusters, web, etc. are outside the scope of this work.
4. Implement the system designed in the previous point.
5. Verify correctness of the implementation via a suite of unit, integration, and end-to-end tests.
6. Evaluate the implemented system with respect to the quality of created clusters and overcoming Clusty's shortcomings.
7. Assess your work and discuss possible future improvements.

Recommended literature:

- B. Everitt, S. Landau, M. Leese, D. Stahl: Cluster Analysis, Wiley (2011, 5th edition), ISBN 978-0470749913
- Internal Avast documentation
- Additional literature as recommended by the supervisor or consultant

Requirements for the semestral defence:

- The first three items of the assignment and a part of the fourth item.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Zobal Lukáš, Ing.**
Consultant: Zemek Petr, Ing., Avast
Head of Department: Kolář Dušan, doc. Dr. Ing.
Beginning of work: November 1, 2020
Submission deadline: May 19, 2021
Approval date: October 26, 2020

Abstract

This work aims to create the clustering part of a new version of the clustering tool named Clusty, which is developed by Avast Software. Clusty is a tool for automatic analysis and online clustering of all incoming samples. The most notable shortcomings are using a single criterion for clustering, vertical scalability, and lack of support for achieving high availability. Among the good features belong a good performance, interpretability of clusters' origin, and an ability to use other techniques like YARA rules.

The designed tool overcome the shortcomings while keeping the features. None of the existing clustering methods is being used because none of them had satisfied the requirements. Instead, three new methods are proposed. They are based on the method in the current version of Clusty and the standard methods. The tool uses so-called rules to allow using multiple clustering methods concurrently.

The clustering results can be considered better compared to the results from the current version. This work proposes a solution for the shortcomings and shows the usable clustering methods.

Abstrakt

Cieľom tejto práce je vytvorenie zhlukovacej časti novej verzie nástroja Clusty, ktorý je vyvíjaný spoločnosťou Avast Software. Nástroj slúži na automatickú analýzu a zhlukovanie rozličných typov súborov. Jeho najväčšími nedostatkami sú zhlukovanie súborov na základe jediného kritéria, zlá škálovateľnosť a dostupnosť v prípade poruchy. Medzi prínosy patria výkonnosť, vysvetliteľnosť vzniku zhlukov a možnosť používať techniky ako YARA pravidlá.

Navrhnuté riešenie rieši nedostatky súčasnej verzie, pričom ponecháva požadované vlastnosti. Na zhlukovanie nepoužíva žiadnu z existujúcich metód, pretože žiadna zo zvažovaných metód nespĺňala kladené požiadavky. Namiesto toho sú predstavené tri nové metódy založené na metóde použitej v aktuálnej verzii nástroja Clusty a štandardných metódach. Pri zhlukovaní používa systém tzv. pravidiel, ktorý umožňuje používanie viacerých metód súčasne a s rôznymi konfiguráciami.

Výsledné zhluky je možné považovať za lepšie ako pri použití súčasnej verzie. Práce navrhuje riešenie problémov nástroja Clusty, a predstavuje použiteľné metódy na zhlukovanie.

Keywords

clustering, malware, file clustering, malware clustering, online clustering, multi-criteria clustering

Kľúčové slová

zhlukovanie, malware, zhlukovanie suborov, zhlukovanie malware, priebezne zhlukovanie, multi-kriterialne zhlukovanie

Reference

JASNICKÝ, Matúš. *Multi-Criteria Clustering of Files.* Brno, 2021. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Lukáš Zobal

Rozšírený abstrakt

Termín škodlivý softvér (angl. *malware* zo slov *malicious* a *software*) sa používa na popis softvéru, ktorý je vytvorený za účelom škodenia. Môže byť použitý na ničenie, získavanie alebo rozširovanie informácii bez vedomia používateľa. Podľa jeho činnosti a spôsobu akým sa šíri ho môžeme rozdeliť do kategórii, ako napríklad trójsky kôň, vírus, ransomware alebo adware. Nie je nezvyčajné pokiaľ softvér spadá do viacerých kategórii súčasne.

Vďaka používaniu rozličných techník, ako napríklad polymorfizmus, obfuskácia, zabalovanie alebo šifrovanie, sú autori škodlivého softvéru schopní vytvárať stále nové varianty, čím sťažujú jeho detekciu antivírovými spoločnosťami. Podľa AV-Test, za rok 2020 bolo zaznamenaný nárast variánt škodlivého softvéru o 62% oproti predošlému roku.

Kvôli veľkému množstvu nových vzoriek každý deň nie sú analytici schopní manuálne skúmať každú vzorku. Sú potrebné technológie, ako zhlukovanie alebo klasifikácia, ktoré zoskupujú vzorky do skupín. Skupina by mala obsahovať vzorky softvéru z jednej rodiny, teda rôzne varianty toho istého škodlivého softvéru. Zoskupovanie vzoriek do skupín s podobnými vlastnosťami môže viesť k lepšiemu porozumeniu stavby softvéru, napríklad odhalenie spoločných častí kódu. Okrem toho môžu analytici vynechať známe vzorky a venovať sa novým neznámym vzorkám. Taktiež pozorovanie tvorby zhlukov a ich zmien môže viesť ku detekcií anomálii.

Avast Software je jedna z antivírových spoločností. Denne jej prichádzajú milióny nových vzoriek. Jedným z nástrojov, ktoré firma používa na vysporiadanie sa s veľkým množstvom vzoriek, je nástroj Clusty, ktorého úlohou je automatická analýza a zhlukovanie vzoriek. Je to interný nástroj využívajúci statickú a dynamickú analýzu na zhlukovanie vzoriek. Každý zhluk je vytvorený na základe jedného kritéria (vlastnosti vzorky), ako napríklad PDB cesta v prípade PE kategórie. Clusty podporuje binárne súbory, dokumenty, textové súbory, a iné. Zvyčajne najdôležitejšie kritérium je YARA pravidlo, ktoré je buď vytvorené manuálne analytikom, alebo automatizovane pomocou interného alebo externého nástroja. YARA pravidlá umožňujú popísať škodlivý softvér na základe textových a binárnych vzorov. Každý zhluk má minimálne jednu vlastnosť, ktorá je spoločná pomedzi všetkými vzorkami v zhluku.

Cieľom tejto práca je navrhnúť zhlukovaciu časť pre novú verziu nástroja Clusty, pričom je potrebné sa zamerať na nedostatky súčasnej verzie a zároveň ponechať jej dobré vlastnosti. Identifikácia a analýza vzoriek, ani ďalšie jeho časti, nie sú súčasťou tejto práce. Je však potrebné zohľadniť ich pre prípadné doplnenie v budúcnosti. Nedostatky súčasnej verzie sa týkajú jednak použitej zhlukovacej metódy a jednak architektúry.

Zhlukovacia metóda je založená na manuálne vytvorenom zozname vlastností, ktorý sa postupne prechádza od najprioritnejšej, až po kým nie je možné aktuálnu vlastnosť použiť. Podmienka, ktorá určuje použiteľnosť vlastnosti, je často založená na prítomnosti danej vlastnosti vo vzorke. Napríklad ak má vzorka PDB cestu, tak sa použije práve táto cesta a vzorka sa zazhlukuje do zhluku s rovnakou cestou. Použitie jediného kritéria pri zhlukovaní sa ukázalo byť náchylné na podvrhnutie. Napríklad náhodne generované hodnoty by viedli ku zhlukom o veľkosti jedinej vzorky. Opačným prípadom by bolo generovanie hodnôt, ktoré sa nachádzajú vo vzorkách považovaných za bezpečný súbor. Použitie viacerých kritérií súčasne by mohlo tento problém vyriešiť.

Najvýraznejšie nedostatky architektúry nástroja Clusty sú čiste vertikálna škálovateľnosť a zlá odolnosť voči poruchám. Clusty nie je schopný bežať vo viacerých inštanciách súčasne. Nová verzia by mala podporovať škálovateľnosť a odolnosť voči výpadkom.

Clusty má však aj veľa užitočných vlastností, ktoré je potrebné zachovať. Napríklad nekvalitné vzorky a zhluky môžu byť dodatočne zakázané. Pri vzorkách to znamená, že

vzorka sa nezazhlukuje. Pri zhlukoch to znamená, že zhluk už viac nevznikne. Taktiež je možné vzorky a zhluky opätovne zazhlukovať. Je to užitočné najmä v prípade, keď sú dostupné nové informácie o vzorke. V neposlednom rade je užitočné poznať vlastnosť, na základe ktorej zhluk vznikol. Táto znalosť môže byť využitá analytikmi alebo automatizovanými nástrojmi pri hľadaní spoločných vlastností škodlivého softvéru.

Navrhnutý nástroj bol pracovne pomenovaný Rusty (spojenie slov *Rust* a *Clusty*). Jeho architektúra je navrhnutá tak, aby bol škálovateľný a bolo ho možné spustiť vo viacerých spolupracujúcich inštanciách na rovnakom alebo odlišnom servery. Ako databázu používa viac-modelovú databázu ArangoDB. Databáza je použitá na ukladanie dokumentov, vzťahov a aj na synchronizáciu. Na distribúciu práce je použitý RabbitMQ server, ktorý rovnomerne rozdeľuje vzorky jednotlivým *workerom*. ArangoDB aj RabbitMQ taktiež podporujú škálovateľnosť a vysokú dostupnosť. Rusty používa systém tzv. pravidiel, ktoré podobne ako v nástroji Clusty umožňujú definovať zoznam priorít. Každé pravidlo môže definovať ľubovoľnú podmienku použiteľnosti a taktiež podmienku pre vloženie do zhluku. Tento systém nijako nebráni použitiu jedinej zhlukovacej metódy pri použití jediného pravidla, kde by podmienka použitia pravidla bola vždy splnená. Týmto použitím by bolo možné systém pravidiel zanedbať. Pravidlá však najmä umožňujú použitie vlastností ako napríklad YARA pravidlá, ktoré sa používajú v nástroji Clusty s veľmi vysokou prioritou.

V práci sú zvážené a popísané najznámejšie typy a reprezentanti zhlukovacích metód, ako aj súčasné metódy zhlukovania zamerané špecificky na škodlivý softvér. Žiadna z týchto metód sa však neukázala byť vhodná, a to najmä z dôvodu nemožnosti zachovania súčasných vlastností nástroja Clusty, ako napríklad kontinuálne zhlukovanie. Ako náhrada sú predstavené tri nové zhlukovacie metódy, založené na znalostiach z nástroja Clusty a existujúcich metódach. Tieto metódy sú implementované do nástroja Rusty a otestované pomocou rôznych metrík, ako napríklad počet zhlukov so zmiešanými klasifikáciami vzoriek. Výsledky zhlukovaní sú taktiež porovnané s výsledkami nástroja Clusty. Konkrétne sa jedná o výsledky za použitia všetkých metód v nástroji Clusty a za vynechania určitých špecifických metód ako YARA. Pre každú z predstavených metód boli rovnako vykonané rýchlostné testy zhlukovania. Následne bola na základe výsledkov vybraná najlepšia z metód. Táto metóda vykazuje dobré výsledky, istým spôsobom lepšie ako nástroj Clusty. Hodnotenie a porovnávanie výsledkov prebiehalo komplexným spôsobom, keďže hodnotiť kvalitu zhlukov nie je samo o sebe postačujúce. Napríklad celkový počet zhlukov je ďalší kľúčový aspekt, ktorý je potrené pri vyhodnotení a porovnávaní zohľadniť. Výsledná metóda je však zároveň aj najpomalšia. V budúcnosti je možné vylepšiť jej výkonnosť, alebo využiť potenciál ďalšej z metód, ktorá si vyžaduje viacej času a experimentov.

Implementácia je otestovaná sadov jednotkových, integračných a koncových testov. Sú prekonané nedostatky nástroja Clusty a ponechané jeho dobré vlastnosti. Nástroj je možné v budúcnosti rozšíriť napríklad o nové kategórie alebo zhlukovacie metódy. Pre kompletnosť novej verzie je potrebné doimplementovať ďalšie časti nástroja Clusty, napr. identifikáciu a analýzu vzoriek.

Multi-Criteria Clustering of Files

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Ing. Lukáš Zobal. The supplementary information was provided by Ing. Petr Zemek, Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Matúš Jasnický
May 17, 2021

Acknowledgements

I would like to thank my supervisor Lukáš Zobal as well as Petr Zemek for their guidance and valuable feedback.

Contents

1 Introduction **4**

2 Clustering **6**
2.1 Data Types 6
2.2 Properties of Methods 7
2.3 Types and Algorithms 8
2.3.1 Partitioning Methods 8
2.3.2 Hierarchical-Based Methods 9
2.3.3 Density-Based Methods 10
2.3.4 Grid-Based Methods 11
2.3.5 Model-Based Methods 11
2.3.6 High-Dimensional Methods 12
2.4 State of the Art of Malware Clustering 12

3 Malicious Software **14**
3.1 Malware Types 15
3.2 Analysis 15
3.2.1 Static Analysis 16
3.2.2 Dynamic Analysis 16
3.3 YARA 16
3.3.1 YARA Language 17
3.3.2 YARA Library 17

4 Clusty **18**
4.1 Architecture 18
4.2 Clusty v1 Clustering 19
4.3 Clusty v2 Clustering 20
4.4 Cluster Description 21
4.5 Sample Description 22

5 Design **24**
5.1 Expectations 24
5.2 Architecture 25
5.2.1 Structure 26
5.2.2 Database 28
5.3 Clustering 31
5.3.1 Standard Methods 31
5.3.2 State of the Art 31

5.3.3 New Methods . . . 32

6 Implementation **39**
6.1 Rusty Library . . . 39
6.1.1 Cluster . . . 41
6.1.2 Sample . . . 42
6.1.3 Database . . . 42
6.1.4 Method . . . 44
6.1.5 Rule . . . 44
6.1.6 Placer . . . 45
6.2 Rusty Tool . . . 46
6.3 Clustering Methods . . . 47
6.3.1 Gradual Selection . . . 47
6.3.2 Ssdeep and Experimental Selection . . . 48
6.4 Classify Tool . . . 49
6.4.1 Design and Implementation . . . 49
6.4.2 Usage and Example Output . . . 50
6.5 Auxiliary Scripts . . . 51

7 Experiments **53**
7.1 Dataset . . . 53
7.2 Clustering Results Overview . . . 57
7.3 Current Clusty . . . 58
7.4 Gradual Selection . . . 59
7.4.1 Selecting of Attributes and Parameter E . . . 60
7.4.2 Parameter T . . . 60
7.4.3 Parameter R . . . 60
7.4.4 Speed Test . . . 64
7.4.5 Evaluation . . . 64
7.5 Ssdeep Selection . . . 64
7.5.1 Generating of Groups . . . 66
7.5.2 Clustering . . . 66
7.5.3 Speed Test . . . 66
7.5.4 Evaluation . . . 69
7.6 Experimental Selection . . . 69
7.6.1 Clustering . . . 69
7.6.2 Speed Test . . . 69
7.6.3 Evaluation . . . 69

8 Testing **72**
8.1 Rust Library . . . 72
8.1.1 Unit Tests . . . 72
8.1.2 Integration Tests . . . 73
8.1.3 End-to-End Tests . . . 73
8.2 Scalability and High Availability . . . 73

9 Evaluation **75**
9.1 Design . . . 75
9.1.1 Preserved Features . . . 75

9.1.2 Design Shortcomings of Clusty 76
9.2 Multi-criteria Clustering 76
9.2.1 Preserved Features 77

10 Conclusion **79**

Bibliography **80**

A Crate Configuration **84**

B Auxiliary Scripts **85**

C Division of Attributes into Groups **88**

D Gradual Selection Experiments **89**

E Tests **91**

F Web Preview **92**

G Content of Attached Media **94**

Chapter 1

Introduction

Since the notion of malware in 1971, it has become a major threat in modern society. Increasing digitalisation also increased interest to empower the rise of dedicated teams and cybercriminal organisations. The Internet has become home to large black markets to purchase and distribute malware [9]. Malware is not related only to personal computers. It is also prevalent in conventional servers, mobiles, and IoT devices [3].

The term malware is used to describe malicious software intentionally designed to cause harm in various ways. It can be used to destroy, gather, or distribute information without the user's knowledge [33]. Malware can be divided into non-exclusive categories like trojan, virus, ransomware, and adware [6].

Using different techniques like polymorphism, obfuscation, packing, or encryption, malware authors can create many different variations of the same malware, making it even harder to detect by anti-malware companies. Developing a signature for one version does not have to be sufficient to detect other versions [9]. According to Symantec, the number of detected malware variants increased by 62% in 2020 compared with the previous year [17].

Because of the a sheer amount of new samples each day, malware analysts cannot analyse them manually. Hence, automated techniques like classification or clustering are required to categorise malware into malware strains [12]. Grouping samples into groups with similar characteristics can help better understand relevant information, such as code reuse and malware evolution. Furthermore, analysts can discard all samples belonging to known strains and focus on new ones [23]. Monitoring of created clusters could lead to a better understanding of activities and detecting anomalies [9].

Avast Software is one of the anti-virus companies, and it receives millions of new samples each day. To be able to process them, a tool for automatic analysis and clustering of newly arrived samples named Clusty was developed. Clusty is an Avast internal tool that uses both static and dynamic analysis for the clustering of samples. Each cluster is based on a single criterion (samples' property), like the PDB path for the PE category. It supports binaries, documents, text files, and others. The most prioritised criterion is often clustering by YARA [2] rules, which can be written by an analyst or automatically by internal tools. Clustering results can also be classified automatically or by analysts. Each cluster has at least one property (used criterion) shared across all samples inside [19].

The aim of this work is to design a new version of Clusty's clustering part from scratch, focusing on the shortcomings of the current version. Other parts like sample analysis, web preview, or cluster classification are not part of this work, but there should be a possibility to implement them later. The shortcomings of the currently used version concern the used clustering method and the architecture. The clustering method uses a manually created list

of criterion, which are consecutively evaluated, and the first matching condition is used. The condition is usually a presence of a specific attribute in a sample. Using just a single criterion turned out to be prone to adversarial attacks. It can lead to small clusters or clusters having different samples. Using multi-criterion clustering can prevent the creation of such clusters. The most notable shortcoming of the architecture is pure vertical scalability. Clusty is not able to run in multiple instances of clustering within the same clustering result. This inability also leads to a single point of failure. Multiple instances running at the same time would cover outages. But Clusty has a lot of features as well, which have to be kept. For example, clusters can be blacklisted (they will not be created again) or reclustered on demand. Also, it is clear to see which criterion the cluster was created from. This knowledge can be further used by analysts or automated tools to find specific malware signatures.

The work is organised as follows. Types of clustering methods with some representatives algorithms and the state of the art malware clustering approaches are described in Chapter 2. Description of malware, its variants, and the methods of malware analysis are described in Chapter 3. Clusty, its architecture, and both older versions are described in Chapter 4. It also contains a description of a sample used in clustering. The design of the new version is presented in Chapter 5. Expectations of the new version are stated at the beginning. Then, an architecture considering those expectations is presented. In the end, it contains a review of the clustering methods and a description of the new ones. Implementation of the designed tool, proposed clustering methods, and all auxiliary scripts are described in Chapter 6. Experiments, the results, and the used dataset are described in Chapter 7. Chapter 8 describes the tests of the implementation. The whole work is then summarised in Chapter 9. The last Chapter 10 is a conclusion.

Chapter 2

Clustering

The chapter is based on [11,31,32]. Clustering is the process of dividing objects into classes (clusters) based on similarity. Classes are then made up of objects that are similar to each other and different to objects in other classes. The similarity is determined based on the values of these objects' attributes, often using the so-called distance functions.

From the point of view of machine learning, it is unsupervised learning. Clustering does not require any predefined classes or any training set of examples.

The clustering algorithms can require a specific type of data. The most used data types are described in Section 2.1.

Different clustering algorithms lead to different clustering results. Since clustering is not controlled by humans but by the used algorithm, its result can discover new patterns in the data. Dividing algorithms into categories is complicated because categories can overlap, i.e. the method can have characteristics from several categories. In general, however, clustering methods can be divided into several groups. Book [11] divides algorithms into four basic groups: partitioning methods, hierarchical-based methods, density-based methods, and grid-based methods. According to [32] we can add two more groups: model-based methods and high-dimensional methods. The properties required for clustering methods are described in Section 2.2 and the individual groups, together with examples of algorithms, are described in Section 2.3.

2.1 Data Types

Data sets are made up of data objects representing a real-world entity. These objects can also be referred to as *samples*. They are described mostly by their attributes. An attribute (also called *dimension*) is a data field representing a feature of the sample. Based on the type of value an attribute can have, we can divide them into the following categories:

Nominal attributes Nominal attributes, also referred to as *categorical*, have values without a meaningful order. They are a generalisation of binary attributes because of allowing more than two values. But the range of values is predefined and finite. An example of nominal attribute is a *colour*, which can be *red*, *green*, etc. These values can be represented by numbers, but it does not make sense to find their average for example.

Binary attributes A specialisation of nominal attributes is binary attributes allowing only two possible values: 0 and 1. The typical interpretation of 0 is an absence of the

attribute, and 1 is its presence. Binary attributes can also be referred to as boolean values if an interpretation of values is *true* and *false*.

Ordinal attributes Ordinal attributes have defined ordering of values, but the magnitude between them is unknown. For example, dividing size into three categories: *small*, *medium*, and *large*. We can say what is larger but not by how much larger it exactly is.

Numeric attributes Nominal, binary, and ordinal attributes are qualitative. It means that even if they would be represented in a numeric form, the number would mean a code category rather than some measurable value. On the contrary, numeric values are quantitative, i.e. they are measurable. They can be divided further into *Interval-Scaled* attributes (e.g. temperature) and *Ratio-Scaled* attributes (e.g. height).

2.2 Properties of Methods

In addition to dividing the algorithms into certain categories, we can compare algorithms according to their properties. Typical properties required of the clustering method are:

Scalability: Clustering algorithms usually work well on a small amount of data. In practice, however, it is often necessary to process large amounts of data.

Ability to process different types of attributes (versatility): Many algorithms are designed to process only numeric data. However, the processing of data of another type is often required in practice, e.g., binary data or ordinary data.

Creating clusters of arbitrary shape: The most common clustering methods create spherical shape clusters based on the Euclidean or Manhattan distance function. Clusters of arbitrary shape could better match the sought classes.

Minimum knowledge of the problem when determining parameters: Numerous methods require the determination of input conditions (e.g. the required number of clusters). These parameters are often difficult to determine and can have a major impact on the quality of the clusters found.

Ability to deal with data containing noise: Incorrect, unknown, or missing data can significantly reduce the clusters' quality. Hence, methods that are resistant to noise are needed.

Incremental (online) clustering: Some algorithms cannot incrementally process new data, and clustering has to be done repeatedly.

Input record order insensitivity: Some algorithms may find different clusters with different input data arrangements.

Ability to process high dimensional data: Conventional clustering methods process low dimensional data well (data with 2–3 items). Significant algorithms are those that can process even data items with a larger number of attributes.

Constraint-based clustering capability: The task is to find data classes that meet the required constraints.

Explainable and usable clusters: The resulting clusters must be interpretable, comprehensible, and usable.

The partitioning criteria: In some methods, all clusters are at the same level without any hierarchy (flat clustering). Other methods can form clusters in a certain hierarchy, where each level can have a different semantic meaning (hierarchical clustering).

Separation of clusters: Some methods divide objects so that each object belongs to exactly one cluster (hard clustering). Alternatively, an object can be assigned to multiple clusters (soft clustering).

Similarity measure: The similarity between objects can be determined based on the objects' distance, for example, using the Euclidean distance. Another way is to define similarity by connectivity based on density or contiguity of objects, which do not necessarily rely on the distance between objects.

Clustering space: Many clustering methods search for clusters throughout the space. In high-dimensional data, this can be a problem because they may contain some irrelevant attributes.

2.3 Types and Algorithms

It is not generally possible to determine the most appropriate method. The choice of a particular method depends on the type of data analysed and the specific purpose of the application. Sometimes several different methods are used at the same time, and their results are then compared.

2.3.1 Partitioning Methods

These methods divide n objects into k classes where $k \leq n$. Each of the classes must contain at least one object, and in most methods, each object belongs to exactly one class (hard). All clusters are at the same level. The disadvantage is that these methods must specify the number of classes into which the objects are to be divided.

Most partitioning methods are based on object distances. In the initial step, an object is randomly selected to represent each class. Other objects are divided into these classes based on similarity to these class representatives. Subsequently, the most suitable class representatives are iteratively searched for. The objects are moved between the classes so that the similarity of objects of one class is as large as possible, and the similarity of objects from different classes is as small as possible. A mean or medoid can represent the cluster. The K-means and K-medoids heuristics are used to achieve an optimal division of objects into classes. Using these heuristics, clusters of circular shapes are created.

Methods using the class representatives can also be called *Prototype-based* methods [25].

K-means Represents a class using a fictitious central point whose attributes are determined as the mean value of object attribute values. Objects are divided into these classes based on the distance from the centre points. The algorithm has linear complexity and works well as long as the data forms well-separated compact clusters. The disadvantage is that the method does not make it possible to find clusters of non-convex shapes and various sizes. It is also sensitive to noise and outliers in the data, which can significantly distort the resulting cluster distribution.

Bisecting K-means Unlike K-means, it starts with all objects in one cluster. It then proceeds by dividing the largest cluster into two smaller ones using K-means and thus continues until the required number of clusters is reached. Unlike K-means, it tends to form clusters of similar size.

K-medoids In contrast to the K-means method, it makes it possible to reduce the effect of outliers. The method does not use a fictitious centre point, but each class is represented by an object located closest to the middle of the class. It is stronger because it is less affected by noise and outliers in the data, but it is more computationally intensive.

2.3.2 Hierarchical-Based Methods

These methods create a hierarchical decomposition of a given set of objects – a cluster of trees is created. Depending on how the decomposition takes place, they are divided into agglomerative and divisive.

- Agglomerative methods (bottom-up) first place each object in its own class. Subsequently, the most similar classes are merged until all classes are combined into one or until the required level of aggregation is reached.
- Divisive methods (top-down) first place all objects in one class. Subsequently, the classes are divided into smaller ones until each object is in its own class or until the required level is reached.

Hierarchical methods can be distance-based, density-based, or continuity-based. The disadvantages of the methods are the impossibility of going back one step (e.g. merging divided classes backwards) and quadratic complexity.

A bad decision, when merging or splitting, can lead to low-quality clusters. A possible improvement is to combine hierarchical clustering with another technique. This method is called multi-phase clustering, and its representatives are BIRCH and Chameleon.

Hierarchical methods can be further categorised into algorithmic, probabilistic, and Bayesian.

- Algomerative, divisive, and multi-phase are algorithmic, i.e., they consider the data to be deterministic, and they calculate their distance deterministically.
- Probabilistic methods use probabilistic models to capture clusters and evaluate their quality based on a fitness function.
- Bayesian methods return a group of clustering structures and their probabilities instead of a single clustering result.

AGNES - AGglomerative NESting Each object is initially placed in its own class. Subsequently, the most similar classes are merged in two until all classes are combined into one or until the required level of aggregation is reached.

DIANA - DIvisive ANAlysis First, all objects are placed in one class. The classes are then gradually divided into smaller ones until each object is in its own class or until the desired level is reached.

BIRCH - Balanced Iterative Reducing and Clustering using Hierarchies It belongs to a group of multi-phase methods and is designed for a large number of data. It works on numerical data in two phases. The first phase uses hierarchical clustering, and the second phase uses other methods, e.g. iterative partitioning. The advantage over agglomerative clustering is the improvement of scalability and the inability to revert the executed steps.

It uses the so-called clustering feature CF to represent the cluster, and the cluster hierarchy is represented by the clustering feature tree CF-tree. CF summarises information about objects in the cluster. CF is defined as $CF = (n, LS, SS)$, where LS is the linear sum of n points and SS is the square sum of points.

In the first phase of clustering, the database is scanned, and a CF tree is created. In the second phase, the selected clustering algorithm is used to cluster the CF tree leaf nodes. This achieves the reduction of remote clusters and the transfer of objects to larger clusters.

Chameleon Chameleon is a multi-phase clustering algorithm. The first phase generates a nearest neighbour graph (using a graph distribution algorithm) that contains links only between the point and its nearest neighbours. The second phase uses the algorithm of agglomerative hierarchical clustering to find real clusters by merging them into subgroups [7].

2.3.3 Density-Based Methods

Clusters are considered areas with a high density of objects separated by areas with a low density of objects. Objects in a low-density space are considered noise. Using these methods, it is possible to find clusters of different shapes and deal with noise (remote values do not belong to any cluster).

DBSCAN - Density-Based Spatial Clustering of Applications with Noise The algorithm combines clusters into areas with a sufficiently large density of objects. What density is large enough is determined by the parameter ε, which indicates the maximum distance between two objects in the cluster, and the parameter min_pts, which indicates the minimum number of objects in the cluster. The neighbourhood of an object with radius ε is called ε-neighbourhood. The problem with this algorithm is the determination of the input parameters. The principle of operation is as follows: the method first checks the ε-neighbourhood of each object. If the ε-neighbourhood of one of the objects contains at least min_pts objects, a new cluster is created, and its core is this object. Subsequently, iteratively searched for objects located in the ε-vicinity of the cluster cores, and clustering can occur. The process is terminated if no more points can be assigned to any of the noises.

DENCLUE - DENsity-based CLUstEring The method is based on the use of the density distribution function. It is based on three ideas:

- a mathematical function can model the influence of an object in its vicinity
- the total function of data space density can be modeled as the sum of individual functions of the influence of all objects in the data space
- clusters are places where the local maxima of the total density function are located

The influence function can be any function derived from the distance of two objects, e.g. Euclidean distance from which the rectangular function of influence can be derived:

$$f_{square} = \begin{cases} 0 & \text{for } d(x,y) > \delta \\ 1 & \text{for } d(x,y) \leq \delta \end{cases}$$

where δ is the maximum distance between two points. The total density function at point x is given by the sum of the functions of all remaining points' influence on point x. Places, where the overall density function reaches the local maximum, represent the clusters' centres found. The main advantages include that the method makes it possible to describe clusters of different shapes mathematically on better level, even for high-dimensional data, and deal well with noise. The problem is setting the parameter δ.

OPTICS - Ordering Points To Identify the Clustering Structure The principle is similar to DBSCAN. Unlike other algorithms, however, it tries to eliminate the problem of difficult selection of initial parameters, which greatly influences the resulting shrinkage. The output of OPTICS is not clustering, but cluster ordering, a linear list of all objects representing the density-based clustering structure of data. Objects in clusters with a higher density are closer together in the list. This list can obtain clustering information such as cluster centers, but it also represents the cluster visualisation itself.

2.3.4 Grid-Based Methods

The methods use a multilevel grid data structure. Space is divided by a grid into a finite number of cells. All clustering operations are performed over this grid. The main advantage is the low time required, which depended only on the number of cells. Examples of algorithms not described below are STING and CLIQUE.

WaveCluster The method uses wave transformation to transform the data space. It efficiently processes large data sets, allows you to find clusters of various shapes, and can deal with outliers. The resulting clusters are independent of the order of processing objects and do not require input parameters. The procedure for this method is as follows: The data space is first divided by a grid. Each cell then summarises information about the objects that fall into it. Subsequently, a multilevel wave transformation is applied to the values in the grids. Wave transformation emphasises areas where points cluster and conversely suppresses information beyond the clusters. This highlights clusters and removes outliers.

2.3.5 Model-Based Methods

The methods try to find clusters that would correspond as closely as possible to some mathematical model. The data are most often generated according to one of the complex probability distribution functions.

EM - Expectation–Maximization It is assumed that a parametrised probability distribution function can represent each cluster. Based on these probabilistic distribution functions, it is possible to create a density model and then perform clustering according to it. The problem is to find such parameters of the distribution function that the probability of distribution corresponds as much as possible to the data set. The EM method makes it possible to find these parameters. The EM method represents an extension of K-means,

but there are no fixed boundaries between clusters. For each cluster and object, the probability that the object belongs to the cluster in which it is located is determined. These probabilities are then used to calculate the parameters of the new points representing the clusters.

Conceptual clustering It tries to find a classification scheme for objects and, in contrast to common clustering methods, a characteristic description for each cluster. The best-known method is COBWEB, which uses a classification tree, with each node related to a concept and containing a probabilistic description of that concept. A probability description summarises the objects classified under this node.

2.3.6 High-Dimensional Methods

The problem with clustering high-dimensional data is that only a small number of dimensions (attributes) are relevant to clustering. Data in other dimensions can cause too much noise, and with the increasing number of dimensions, more data is scattered. The following methods address these issues:

Feature transformation method: transforms data into a space with fewer dimensions while maintaining relative distances. Attributes of attributes (features) are combined. The features created in this way are difficult to interpret, which reduces the usefulness of the results. The method is suitable for datasets where most attributes are relevant.

Attribute selection method: searches for relevant attributes and removes irrelevant. Relevance is determined using various criteria. Supervised learning is most often used, where the most relevant objects are selected according to the evaluation of individual classes of objects.

2.4 State of the Art of Malware Clustering

This section describes the state of the art of malware clustering approaches. They often aim to find new attributes suitable for clustering or combine the standard clustering methods mentioned in the previous section.

Bayer et al. [5] propose a new approach for clustering similar samples. They use dynamic analysis to create behavioural profiles, which have to be transformed into feature sets. Before clustering, they discard all features of unique samples among the whole set because unique features would not participate in finding similar samples. The feature set is suitable for Locally Sensitive Hashing (LSH), which is then used to compute single-linkage hierarchical clustering.

Perdisci et al. [22] present a novel malware clustering system aimed at network-level behaviour. It consists of three phases: (1) *Coarse-grained Clustering* using statistical features, e.g. the total number of HTTP requests, (2) *Fine-grained Clustering*, which further splits relative large clusters into smaller ones using structural similarities, which allows separating malware with similar statistic but different structures, and (3) *Cluster Merging* of very similar clusters, to derive more generic behaviour models.

Pitolli et al. [23] propose a novel solution for identifying malware strains using the BIRCH algorithm using both static and dynamic features. Feature vector consists of 241

numeric features (strings are converted to numbers via LSH). To speed up computation, all shared features among at least 80% of samples are considered useless and removed.

Asquith [4] proposes the use of a data structure called an *aggregation overlay graph* for clustering malware data. They consider a bipartite graph with samples as nodes on one side and their features on the other side. Malware variants often share a certain subset of features forming a *complete subgraph*. Compression of bipartite graphs with complete subgraphs can be used for finding variants of the same strain, but samples can belong to more strains. The method is not restricted to any type of data and can use both static and dynamic features.

Xin Hu et al. [14] present a novel scalable framework called *MutantX-S* that can efficiently cluster samples based on code instruction sequences. All statically analysed samples are processed in the three steps: (1) *Instruction encoding* for converting instruction into a sequence that is resilient to low-level mutations, (2) *N-gram analysis* to construct a feature vector, and (3) *Hashing Trick* for compressing the feature vectors to improve similarity speed computation. Finally, prototype-based clustering is applied to the feature vector, producing a set of clusters. The prototypes are initially selected randomly, and then samples with the largest distance to the existent prototype are selected each iteration. Using prototypes allows considering only a small subset of samples during the clustering of samples.

Shuwei et al. [27] use the clustering algorithm based on *Shared Nearest Neighbour (SNN)* combined with DBSCAN. They use the frequency of the system calls as the feature vector for clustering. Clustering is divided into four steps: (1) calculate SNN density of each sample, (2) find core samples that have density above the given threshold, (3) merge similar core samples, and (4) cluster samples into existing clusters if they are similar enough. The rest of the samples is considered to be noise.

Zhang et al. [33] combine multiple features from both static and dynamic analysis. Their approach consists of several steps: (1) extraction of multiple categories of static and dynamic information, (2) base clustering using different clustering algorithm (K-means or hierarchical algorithms) for each category, and (3) combining the results of base clustering to get the best ensemble results.

Ye et al. [30] developed an Automatic Malware Categorisation System (AMCS). It uses a cluster ensemble by aggregating the clustering results from partitional and hierarchical clustering methods, like K-medoids. They decided to use two static features, instruction frequency and function-based instruction sequences, for representing PE samples. Besides that, it also supports sample-level constraints by specifying pairs of samples that should or should not be in the same cluster.

Xi Hu et al. [13] propose a novel clustering framework called DUET. DUET uses a clustering ensemble to combine several clustering results into a single one systematically. The base clustering algorithms can use static or dynamic features. It also uses quality metrics to score clusters, which helps the ensemble algorithm with the decision. Using different algorithms and static and dynamic analysis leads to better results because of combining the strength of individual clustering and both types of analysis.

Chapter 3

Malicious Software

The chapter is based on [8, 26, 29].

Malware is a program or a piece of code created on the purpose to achieve the harmful intent of an attacker. Attacker's intentions are gaining control over the system or gathering user sensitive data without user permission. Based on the purpose and the spreading way, malware can be categorised into several categories. Categories are described in Section 3.1.

There are multiple ways malware can spread: (1) *Vulnerable network services* can be used to infect the system automatically over the network, (2) *Drive-by downloads* exploit a web browser vulnerability, which allows the malware to fetch itself from the web and execute it on the user machine. The user needs to visit the infected page at first, which can be achieved for example by spam emails, (3) *Social engineering* is a technique that lures a user into directly executing malware on its computer. It is often hidden behind an innocent and credible occasion like installing a missing video codec or opening an image.

The major defence to protect against malware is to use anti-malware software. Concurrently with the anti-malware companies constantly improving malware detection, malware creators also develop more evasion techniques to prevent malware analysis:

- *Encryption* of malware and providing the decryption algorithm and the key to decrypt malicious components
- *Packing* is a technique of compressing an executable file. The file needs to be unpacked to reveal a original binary structure
- *Obfuscation* aims to hide the program logic by code transformation, e.g., adding garbage code or unnecessary jumps.
- *Polymorphic* malware is a malware, which uses encryption and creates different looking replication each time, leading to unlimited number of variants of the same code
- *Metamorphic* malware changes itself, so each instance will look different.

To detect and classify malware, static or dynamic analysis can be used. While the static analysis incorporates all techniques that analyse software by inspecting it, the dynamic analysis examines software behaviour by running it. Static and dynamic analysis is more described in Section 3.2. Besides that, there is also YARA tool able to match malware samples based on textual or binary patterns. YARA is further described in Section 3.3.

3.1 Malware Types

Based on the different purpose and the way it spreads, malware can be categorised into multiple types. Malware types are not mutually exclusive, i.e. malware samples may share characteristics from multiple types. Some of the malware types are described in this section.

Virus When executed by an already infected "host" program, a virus will infect files with the same or evolved copy of itself. By infecting files shared on a server, it can also spread to other computers.

Worm Unlike a virus, a worm does not need a "host" program. It is a self-replicating program able to spread a full copy of itself across the network.

Trojan Horse A trojan is a software, which pretends to be useful but behaves maliciously in the background. It can distinguish itself as any legitimate program, and once installed, its malicious part can download other malware or anything else an attacker wants. Because of this, the standard way it spread is using social engineering.

Spyware It is malware, which silently spies on users. It can collect sensitive data, for example, track user activities, including its browser history, and then sell them to third parties.

Bot A botnet is a network of infected computers called bots under the control of an attacker. They typically spread by exploiting software vulnerabilities or using social engineering. Botnets are commonly used to send spam emails or in launching Distributed Denial of Service attacks (DDoS).

Adware The main purpose of adware is downloading or displaying advertisements and thus getting revenue to the attacker.

Rootkit A rootkit is a type of malware operating on different system levels designed to hide certain information from a user. It can hide processes or files by manipulating information. Hence it is often used by other malware to hide it.

Ransomware It aims to encrypt all the user data and ask him to pay money as a ransom the get the decryption key allowing him to get the data back. The machine infected by ransomware is unusable, and the information about the payment usually appears as a desktop image.

3.2 Analysis

Both static and dynamic analysis has several advantages and drawbacks. Thus it is suggested to use them altogether. Combining static and dynamic analysis is called *Hybrid analysis*.

3.2.1 Static Analysis

Static analysis investigates samples without executing them. The sample often needs to be preprocessed first to overcome evasion techniques, e.g., decompress/unpack sample. Several tools can be used to modify samples into a readable format or get a different kind of information. For example, in static analysis of Windows PE executable files, there are tools for decompression, unpacking, disassembling, memory dumping, and the following data extraction.

Some static features are common for all or a part of malware samples (e.g. binaries), and some are format-specific. Common features include, for example, *file size*, *file name*, and *included strings*. Common features specific to binaries are, for example, *file resources*, *imports*, *exports*, and *entry point address*. There are also file-type-specific features. For example, *rich header* is specific for Windows PE binaries, Linux ELF binaries do no have such a feature.

Even static analysis suffers from evasion techniques, compared to dynamic analysis, its advantage is that it can explore all possible execution paths, which dynamic analysis cannot do. Call graphs can give an analyst a good overview of what functions are called and what the flow of the program would be. Also, the analyser's machine cannot be infected when using static analysis.

3.2.2 Dynamic Analysis

The dynamic analysis uses debuggers (e.g. GDB[1]), sandboxes (e.g. Cuckoo[2]), or other techniques to observe malware behaviour.

Debuggers are used for analysis on an instruction level. However, malware can easily detect the presence of a debugger by simply watching changes of itself that are necessary for debugging. Newer ways of debugging support stealth breakpoints or use hardware virtualisations to remain invisible.

Sandboxes are tools that run the software in a self-governing virtual computerised technology, so they can pretend to be a potential victim and then monitor all software activities, e.g., *file modifications*, *system calls*, or *registry modifications*. Sandboxes separate virtual machine from the actual machine and the same for the network, making it safer easier to distinguish malware actions [16].

Dynamic analysis is especially useful in analysing malware using evasion techniques like packing because, at some point of analysis, malware will unpack itself and reveal the actual code. The problem of dynamic analysis is that malware can depend on some special conditions (e.g. particular time), and thus its malicious behaviour may stay hidden. Besides that, it requires more resources and more time compared to static analysis.

3.3 YARA

YARA [2] is a tool aimed identify and classify malware samples based on the signature description in the YARA language. It is widely used by malware analysts, creating so-called YARA rules. These rules are used for matching malware samples based on textual or binary patterns (there is no need for any external static or dynamic analysis).

[1] https://www.gnu.org/software/gdb/
[2] https://cuckoosandbox.org/

3.3.1 YARA Language

YARA is also a language for writing YARA rules. A YARA rule consists of three sections: mandatory condition, optional strings, and optional metadata. Condition determines whether a sample matches the YARA rule or not. If the rule wants to refer to a string, the string needs to be put in the strings part. The metadata section provides additional information about the rule alongside its name. An example of a YARA rule describing PDF magic bytes can be seen in Figure 3.1.

```
rule PDF
{
    meta:
        file_type = "PDF"
        version = 1
    strings:
        $pdf_magic = { 25 50 44 46 2D }
    condition:
        $pdf_magic at 0
}
```

Figure 3.1: YARA rule matching PDF magic bytes

3.3.2 YARA Library

YARA as a tool is just a wrapper around `libyara` library, which is used to search for signatures written in YARA in a given file. The output of the YARA scan is a list of YARA rules matching the specific file. Because of searching files for a string occurrence, it does not really matter the type of file. This type of searching lack context.

YARA modules are extensions providing additional context by parsing the file. An example of such a module is *PE* module exposing most of the fields present in a PE header (e.g. `pe.number_of_sections`). Besides that, there are also modules providing general functions like *math*, *hash*, or *magic*.

Another example of a non standard module is *cuckoo* module. It is not a standard module because it does not parse samples itself. Rather it parse JSON output of Cuckoo sandbox. In this way, YARA rules can also include dynamic signatures.

Chapter 4

Clusty

This chapter is based on [19].

The Avast company receives up to several millions of new samples every day. Each sample then needs to be analysed and classified to know whether it is malware or cleanware. In the case of malware, it is also needed to determine type and strain. Malware analysts are not capable of processing samples one by one. It is not even effective for fewer samples as analysts could discover similar samples over and over again. The better solution is to automatically group samples into clusters according to their similarity. This is what Clusty does.

Clusty is an internal tool for automatic analysis and clustering of newly arrived samples. It consumes events about new samples, analyses them, and cluster them into a best-suited cluster. Each cluster is based on a single attribute and can have several attributes sharing the same value among all its samples. These attributes are valuable as they characterise the group. Clusters are then classified either automatically by one of the available classifiers or manually by malware analysts. Clusty supports binaries (e.g., PE, ELF, Mach-O), documents (e.g., PDF, Office), text files (e.g., HTML, email, various scripts), and others. It uses a wide range of properties like static properties, dynamic properties, YARA rules, or antivirus detections to cluster samples.

At this time, the second version of Clusty is already in use. The biggest difference between the first and the second version is the clustering part. Apart from clustering, they are very similar. Because of that, everything described in this chapter (except Section 4.2) is based on the description of the second version but could also be largely implied to the first version as well. Clusty's architecture is described in Section 4.1. Clustering for both versions is described in sections 4.2 and 4.3, and the description of a cluster shown on the web is described in Section 4.4. An example of a JSON representation of the analysed sample alongside the description is described in Section 4.5.

4.1 Architecture

Clusty can run in two modes. The first mode is called *continuous* (or production because Clusty v1 was not actually continuous) and is used in the production, where Clusty continuously analyses and cluster newly arrived samples. The second mode is called *ad-hoc*. This mode is used for development, where the user can cluster samples in a given directory.

Clusty can be divided into three parts: analysis, clustering, and web. This work is focused on the clustering part only.

Analysis

The analysis part is written in Python v3 and is responsible for consuming events, identifying, and analysing samples. It uses RabbitMQ[1] message broker to consume events (e.g. arrival of the new sample) and MongoDB[2] to store analysed samples in a JSON format. Identification and analysis are performed by various internal or external tools, depending on the sample's category. After identification and analysis of the sample, the sample is stored in the database and prepared for clustering. To speed up clustering, a sample can be clustered based on static attributes at first, and reclustered by all attributes once the longer-lasting dynamic analysis results are available.

Clustering

The clustering part is written in C++ because of its huge memory and CPU utilisation. It is responsible for clustering samples, which are already analysed and stored in the database. This part is more described in sections 4.2 and 4.3 for each version of Clusty.

Web

The web part's purpose is to provide clustering results in interpretable form. It provides API to get results (e.g. samples in a specific cluster) in a machine-readable form and a web interface to get clustering results in a more human-readable form. This part is what analysts use. Section 4.4 describes clusters as can be seen on the web.

4.2 Clusty v1 Clustering

Description of Clusty v1 is based on [24].

After every sample is analysed and stored in the database, the clustering part begins. The information used in clustering is compressed (typically by hashing) and clustered using the DBSCAN algorithm separately per category. It divides samples into clusters with a specific minimum size based on the distance. The clustering runs several times, every time using the next attribute in attributes hierarchy, clustering all remaining samples, i.e., samples that were not clustered to any cluster by one of the previous attributes. The attributes and their order were determined on the basis of experiments. The distance is computed as equality or similarity of attributes, depending on the specific attribute. At the end of clustering, common attributes for each cluster are computed.

In production mode, samples are analysed during the day as they arrive. The clustering itself runs only once per day at its end. Only samples from the day are clustered. There is no continuity to any previous day.

Shortcomings

This approach has several shortcomings, mostly because of the selected clustering approach. Because all samples must be present when clustering begins, to cluster new samples, clustering has to run each day and cannot use already existing clusters. This forces analysts to review clusters over and over again. According to internal Avast documentation, even in

[1] https://www.rabbitmq.com/
[2] https://www.mongodb.com/

the case of adding continuous mode, the clusters would remain hardly explainable. That means it is hard to say why a certain cluster was created.

4.3 Clusty v2 Clustering

Unlike Clusty v1, Clusty v2 does not use DBSCAN to cluster samples. It uses the Clusty-unique concept instead. Each category of samples (e.g., PE, HTML) has its own clustering methods ordered by priority. For example, the method based on YARA rules is higher than the method based on the PDB path for the PE category. The clustering method always represents a single attribute. Clusty checks for each method in the given category during clustering whether the sample satisfies all conditions required by the method (e.g. minimal length of the list). The first method which satisfies those conditions is used. If a cluster for the given method, category, and value exists, the sample is placed in that cluster. Otherwise, a new cluster is created, and the sample is added to that cluster. Methods for each category are determined on the basis of experiments. Some methods can be used for each sample regarding its category, e.g. YARA rules. The last method of each category is based on ssdeep[3] hash similarity with a specific threshold. In the case none of the methods could be applied, the sample is marked as unclustered.

If a cluster is incorrect (e.g. containing both clean and malicious samples), it can be blacklisted and reclustered. This means that since then, a cluster with the same method, category, and value will not be created anymore. The sample will be clustered on the basis of another method in the hierarchy.

In continuous mode, samples are both analysed and clustered as they arrive. It is not needed to cluster samples repeatedly each day, and samples can be clustered into existing clusters instead. Another benefit is the ability to recluster samples. For example, that means that already clustered samples can be reclustered when attributes from the dynamic analysis have arrived. It can also be used on a whole cluster, for example, reclustering of tiny clusters. In that case, the method the cluster was created by will not be used during reclustering.

Shortcomings

Newer version v2 overcame several shortcomings of version v1. It supports continuous clustering now. The shortcomings of the current version concern the used clustering method and the architecture.

The clustering method uses a manually created list of criteria, which is sequentially gone through, and the first matching condition is used. The condition is often based on a presence of a specific attribute in a sample, and the value used for clustering is the attribute‘s value. Since the clusters are created based on a single criterion (attribute), the method is prone to adversarial attacks. For example, if the attribute's value is randomly generated value, all such samples will be placed into a new cluster and stay alone. On the other hand, clustering based on a value that is common for both clean and malware samples would lead to clusters having both malware and clean samples (further as mixed clusters). Nor creation of tiny clusters neither creation of mixed clusters is desirable. Using multi-criteria clustering can prevent the creation of such clusters.

The most notable shortcoming of architecture is pure vertical scalability. Clusty is not able to run in multiple instances of clustering within the same clustering result. This

[3] https://dfir.science/2017/07/How-To-Fuzzy-Hashing-with-SSDEEP-(similarity-matching).html

inability also leads to a single point of failure. Multiple instances running at the same time would cover outages and ensure high availability.

4.4 Cluster Description

Clustering results can be shown on the web. An example of such a cluster can be seen in Figure 4.1. Each cluster consists of several parts: header (top grey row), shared properties (below header), detection statistics (below shared properties), classification (below detection statistics), and additional info (bottom row).

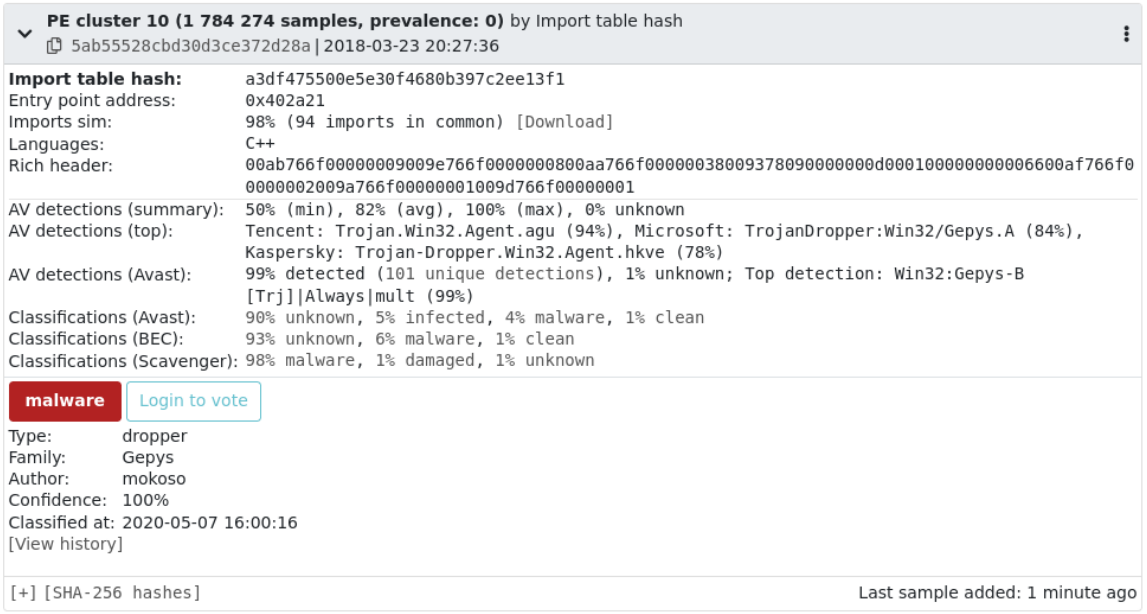


Figure 4.1: Cluster created on the basis of Import table hash manually classified as malware-dropper-Gepsys

Header

The header contains an overview of the cluster. There is a category (PE), order number (10), number of samples (1 784 274), the sum of prevalences (0), and the criterion the cluster was created by. In this case, the cluster was created on the basis of a single attribute Import table hash, which is a hash of the Import table of PE samples. Prevalence means the sum of Avast users that have seen a sample from the cluster, i.e. 0 means no user has seen any sample. The next line contains a cluster's ID and the date of creation.

Shared properties

Shared properties are properties that are shared among all samples in the cluster. The property the cluster was created by does not need to be necessary the only shared property. The more properties are common, the more similar samples should be. In this example, there are five shared properties. The lists are special parameters where they do not need to be 100% equal to be shown.

Detection statistics

Detection statistics summarise what AVs think about samples in the cluster.

Classification

The classification consists of severity (e.g., malware, clean, PUP), type (e.g., ransomware, worm), and strain (name of the malware). The rest is additional info like the author of the classification, classification's confidence (100% because of manual classification), and the date of classification. The classification can change over time. Its history of changes is also available.

4.5 Sample Description

This section will describe the process from the arrival of a sample to its clustering and introduce a sample to understand better what is being clustered concerning malware clustering.

Names *file* and *sample* are interchangeable across this work, but this section establishes a different meaning for them applying to this section only. The *file* is supposed to be an input file of any type sent for clustering, and the *sample* is referred to as a JSON document collecting information about the input file obtained from various sources.


```
{
    "category": "pe",
    "sha256_hash": "e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855",
    "yara_hits": ["virus_known_sequences"],
    "architecture": "x86",
    "entry_point_address": "0x4011cb",
    "rich_header": "000c1ac500ff090100092f690000c000000d1fe000000003",
    "sections": [".code", ".text", ".data", ".rdata", ".rsrc"],
    "import_table_hash": "87bed5a7cba00c7e1f4015f1bdae2183",
    "uncommon_mutexes": ["adxp"],
    "uncommon_touched_files": ["win32.exe"],
    "ssdeep_hash": "196608:puhNhimNau7IqKpSq8Us++lflGyUVwBifqbBysxpwoOWW:8baudKpSq8Z+
    oflGfwgfWysPDW",
    "path": "/path/to/sample.exe",
    "prevalence": 0,
    "size": 100000,
}
```


Figure 4.2: Example of reduced PE sample

After a file is sent for analysis and clustering in Clusty, it is first needed to identify its file-type. If Clusty supports the file-type, it will be stored as `category` file in a sample. The category will be set to `generic` otherwise. Alongside category, there are generic attributes that can be retrieved for any category, e.g., `path`, `ssdeep_hash`, `size`, and `sha256_hash`. SHA-256 hash is used as a file identifier. If two files have equal hashes, they are considered to be the same file. Another attribute common for all categories is `prevalence`, but it is not based on file content rather on the occurrence of the file on Avast's user devices.

If the category matches one of the categories supported by Clusty, the file is sent for static analysis and dynamic analysis (if possible). The results of both analyses are then filtered in the meaning of attributes as well as their values. Not all attributes will be stored in the sample, and not all attributes are valuable. Besides analysis, the file is also sent for YARA matching. All matched rules are then stored in `yara_hits` field.

Figure 4.2 shows a PE sample having a few attributes from each group mentioned above. We can see that the analysed file is PE x86 executable named `example.exe`. Its size is 100 kB, and it was not seen by any Avast user yet. Further, it creates a mutex named `adxp`, touches a file `win32.exe`, and matches `virus_known_sequences` YARA rule.

Observing the example, we can see attributes can have various types of values, e.g., a list of strings, strings, bytes, and integers. In the case of a string, it can contain anything from meaningful words through randomly generated strings to hashes. This representation is not usable for clustering methods requiring numeric values. Usual techniques to transform data into numbers rely on a finite number of values the attribute can have. One of the techniques to transform attributes into numeric representation regardless of their value is using LSH. It is used by Pitolli et al. in [23], for example.

Chapter 5

Design

The chapter presents a design of the clustering part of the new version of Clusty. The aim is to overcome shortcomings from the current version (e.g. scalability) while keeping its benefits (e.g. online clustering). All expectations from the new version are described in Section 5.1. The design and the expectations are split into two individual parts. The first part designs a scalable, high available, and fast tool providing an interface for clustering of various file-types using various methods for continuous clustering. It should be mostly independent of the chosen clustering method or methods. An architecture and chosen platforms are described in Section 5.2. The second part deals with a selection of usable clustering methods. Existing clustering methods are reviewed, and new methods are described in Section 5.3.

5.1 Expectations

Below is the list of expectations based on [19]. They do not consider clustering only but everything related to the clustering part.

Multi-criteria clustering. Clusty v2 uses a single criterion to cluster samples. It was shown to be very restrictive and easily susceptible to forgery. Furthermore, the attributes and their priority have to be selected manually based on experiments. It is described in Section 4.3. The new version should support multi-criteria clustering.

Homogeneous clusters. The resulting clusters should not mix clean and malware samples. It means that clusters with classification clean will never contain any sample with classification malware and vice versa. Also, types and strains should be shared among all samples in the cluster if possible. Mixed clusters cause several problems. Firstly, it can cause false positives or false negatives if it is mixed just a little. Secondly, it is harder to classify cluster properly, which leads to classification with lower confidence making the cluster unusable (they are marked with classification `unknown` in the current Clusty) in the case of a high level of the mixture.

Performance. The new version should be able to deal with millions of samples as the current one. Both program architecture and clustering method need to be considered properly.

Scalability. It should be scalable according to the number of samples arriving at the moment. For example, it should allow us to run multiple instances or scale the number of processes during run-time.

High availability. It means that there should not be a single point of failure in a whole architecture. All used tools and databases have to support HA, e.g. allow running in clusters. High availability in the new version should be achieved by running multiple instances in parallel, even on multiple servers.

Online and ad-hoc. Both modes, online and ad-hoc, should be supported as described in Section 4.1. It means to support an ability to run as a daemon in an infinite loop continuously clustering all incoming samples, besides the ability to run on a known set of samples finishing after clustering all of them.

Reclustering. It should be able to recluster samples and clusters. For example, when information about the sample has changed, the sample can be reclustered according to the new information. It can be considered as the third clustering mode, where clustered samples are moved into another cluster if any better cluster is found. If samples are not changed since the last clustering, nothing should happen.

Blacklisting. When the cluster has poor quality (e.g. it contains samples that should not be in a single cluster), there should be a possibility to blacklist this cluster. In that case, all samples from the cluster will be reclustered, and the cluster will not be created anymore. The clustering method should be able to support such behaviour.

Unlimited cluster size. The current version of Clusty has two categories of clusters: tiny and regular. Tiny clusters are clusters with one to four samples. Regular clusters contain at least five samples. Only regular clusters are classified and showed on the web. The new version should not use this separation and should treat all clusters the same.

Explainability. It should be clear to see why the cluster was created (e.g. it has `rich_header` attribute with value `00131f8e0000000a0000000000000001`) and what else samples in the cluster have in common. These values can be useful for analysts to see characteristics of malware or to find similarities among clusters.

Universality. Universality means solution independent of category and extendable to more categories (ELF, PDF, etc.). Therefore, this work will cover PE category only. It is the most prevalent category among malware. The solution, including chosen clustering method, has to be easily extensible to support clustering of other categories.

5.2 Architecture

The architecture tries to fulfil all requirements listed in the previous section 5.1, while trying newer technologies than the current version.

Database

The database used in Clusty is NoSQL MongoDB. ArangoDB[1] was selected as a new database. It is a distributed multi-model database, which means data can be stored as key-value pairs, documents, or graphs. All of this can be accessed by just one query language (AQL - ArangoDB Query Language). Its main features are multi model using a single language, multi-architecture - single instance or cluster or mix, sharding, vertical and horizontal scalability, strong data consistency, and fault tolerance [10]. Also, according to the open-source performance benchmarks [28], it performs better than MongoDB in all tests except memory consumption. It meets all the requirements.

The reason for looking after a multi-model database was to leave more options open such as using a graph database. Since the main aim of the database was to store samples as unstructured objects, either several databases or a multi-model database could be used. According to the paper [10], studying and comparing the five most popular graph databases (AllegroGraph, ArangoDB, InfiniteGraph, Neo4J, and OrientDB), ArangoDB was considered the best multi-model database regarding its features.

Message broker

RabbitMQ message broker was chosen to distribute work evenly among all running workers. As RabbitMQ's main page says "RabbitMQ is the most widely deployed open-source message broker" [1]. Besides that, it is a widely used message broker in Avast, connecting many services including Clusty.

It is an open-source message broker implementing the AMQP standard. Load balancing allowing scalability and fault-tolerance by running in HA clusters are only some of its features [15]. The mechanisms it supports allows achieving all requirements.

5.2.1 Structure

The structure of the program can be seen in Figure 5.1. It consists of (1) ArangoDB server storing all samples, clusters and other data related to clustering and synchronisation, (2) RabbitMQ server distributing hashes of samples ready to be clustered, (3) the master process spawning and managing N workers, (4) and N workers which independently consume events and cluster samples. The master process can also communicate with the message broker, depending on the current mode. Note that only the master process in the first instance can communicate with the message broker. The master process in other instances, if there are any, do not need to communicate since there is neither publishing nor consumption required. The communication between the master process and the message broker is required on the beginning only, before running of clustering.

RabbitMQ and ArangoDB are able to run in clusters to sustain *high availability*. To sustain HA of a whole application, the main role of the master process is to make sure the specified number of workers is running. There are not many opportunities for the master process to fail, but if it happens, there is a possibility to run several instances simultaneously. An ability to run in multiple instances also adds support for *scalability*. By running another instance, a number of active workers processing incoming samples can be managed.

Because of support for running multiple instances at once, communication between the master process and its workers cannot be a core for synchronisation. Two instances do

[1] https://www.arangodb.com/

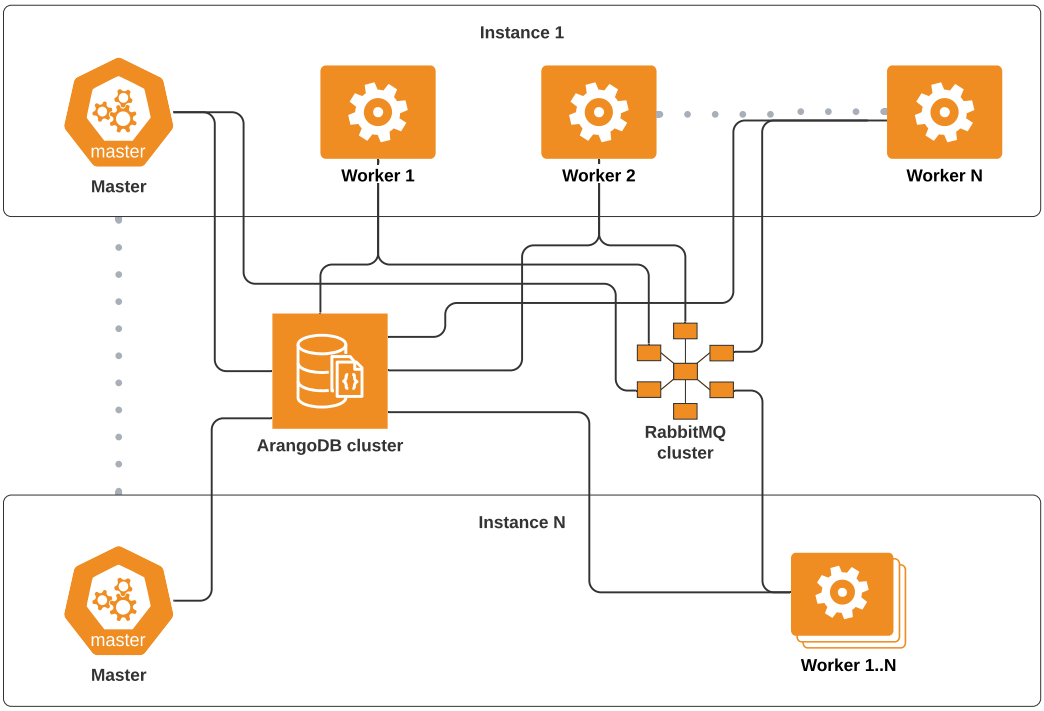


Figure 5.1: Design of architecture showing N instances of Rusty, each having several workers. The master processes and workers communicate with the database and message broker. All the master processes, except a one in the first instance, do not have to communicate with the message broker.

no have to run even on the same server. To synchronise their state (e.g. notify about the successful finishing of clustering) an independent application comes to play. Already used ArangoDB can make it using a key-value model. It supports concurrent write if the documents are different.

Analysis of files is not a part of this work, but it is a part of the clustering pipeline. It is expected that files are analysed and stored in the database before sending an event about the sample being ready to be clustered.

Instance

The instance is meant to be a single running application consisting of the main process and at least one worker. If the instance is executed as a standalone application launching clustering, the main process will work just as described down below. All other instances joining already running clustering process need to be executed with a special flag. If not, they will publish all hashes of samples again.

Master

The master process creates a specified number of independent workers and replenishes them in the case of failure. There is no need to synchronise the instances in the continuous mode because it does not make sense to show progress or finish clustering. Synchronisation in

the other modes has to be both process-independent and server-independent. ArangoDB's key-value model is used for this purpose. It checks the number of samples processed by all active workers, regardless of their master. Because the total number of samples to be clustered is well-known, it can be compared to this total count.

In ad-hoc mode, it also opens a given file and publishes all hashes to the RabbitMQ. This is basically the only difference between the continuous and ad-hoc mode.

In reclustering mode, it collects hashes of samples clustered in the given result and sends them to the RabbitMQ queue. If a sample would be placed into the same cluster, nothing will be changed. Otherwise, the sample is moved from the older into a newer cluster (attributes in the older cluster are not recomputed). In the case of moving the last sample, the cluster is removed. To avoid placing the sample into a deleting cluster, the cluster is invalidated, and the clustering method is changed to a non-existing one. Then if still empty it is removed. The rest is the same as ad-hoc mode.

Worker

Each worker is an independent RabbitMQ consumer waiting for a new hash to be clustered. It is stopped only if its master is stopped or in the case of failure. RabbitMQ cares that a single worker consumes a single hash, and in the case of connection loss (e.g. if a worker has failed), it delegates the hash to another worker. Worker acknowledges the message receiving right upon its successful clustering.

5.2.2 Database

The database has to contain several collections, some of them data-related and some communication-related. An example of the data-related collection is `sample` collection, and an example of the communication-related collection is `waiting_room` collection. A Diagram of all document, graph and key-value collections can be seen in Figure 5.2. Relationships between collections, e.g. cluster contain samples, are annotated using Crow's Foot Notation[2].

All collections except `sample` collection and `clustering_results` collection have to be unique per clustering result. It means that when a new clustering result is being created, all necessary collections are created as well. Similarly with deletion, when a clustering result is being deleted, all related collections are deleted as well.

By using unique collections for each clustering result, there is no performance penalty for having a number of different clustering results and it is easier to store and retrieve specific data.

Cluster collection

Each cluster contains, among others, a copy of the original sample it was created from, category, name of used clustering rule[3], clustering rule related data, validity flag, and a list of attributes all samples in the cluster have in common.

The original sample is used as a representative/base sample of the cluster. These values can be used for cluster usability evaluation during clustering. For example, if a method is based on `rich_header` and `pdf_path` attributes, these values are being compared with the sample's attributes.

[2] https://vertabelo.com/blog/crow-s-foot-notation/

[3] See Section 6.1.5 to learn about the meaning of rule.

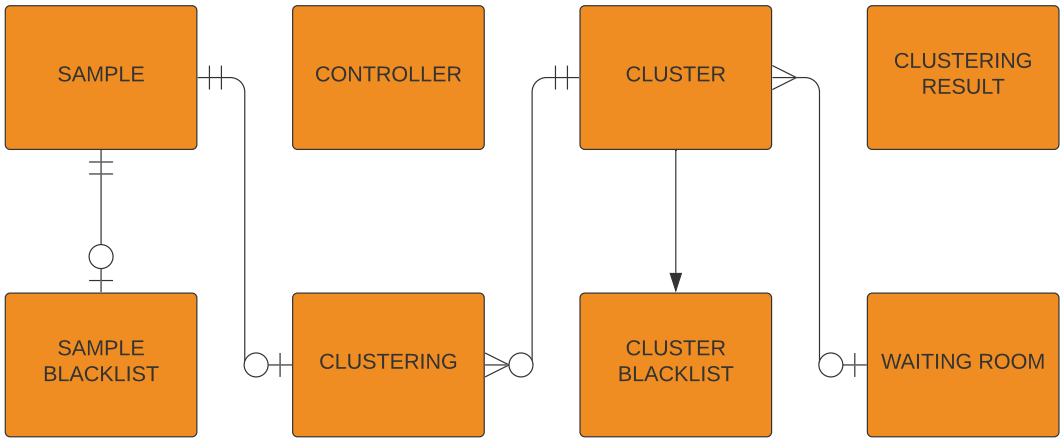


Figure 5.2: Database collections and their relationships

Because of the possibility to define any number of rules, a rule field stores the name of the rule cluster was created by. Currently, documents of all categories are stored in the same collections, this is what the category field is for. Creating a new collection for each category is considered a possible future improvement.

The validity flag determines cluster validity during its creation period. If the cluster is invalid, no sample can be clustered into. Rather they have to wait until its validation or deletion. More about this can be found in Section 5.2.2.

A list of common attributes is just a list of attribute names all samples in the cluster have in common. Values of those attributes can be found in the base sample. In the current version of Clusty, this value has to be of length at least one meaning the only attribute the cluster is based on is shared among all samples in the cluster.

Clustering result collection

This collection stores a list of names of all clustering results in the form of documents. A clustering result is a group of clusters belonging to the same-named session, i.e. clustering of a single directory in ad-hoc mode or a continuous clustering in continuous mode.

Controller collection

Synchronisation among masters is provided by watching a number of processed samples available in this collection. Each worker increments counter in its own document every time it clustered sample successfully. The sum of all counters of all workers is a total number of samples clustered yet. If this number reaches a number of samples to be clustered, it means the clustering is done. Not only are all masters are able to get the current clustering progress, they also know the time they can finish.

Cluster blacklist collection

Blacklisted clusters are fully moved into the blacklist collection (without samples). To check whether a cluster is blacklisted, clusters are fetched from blacklist collection, and the base

sample from the cluster tries to place itself into them by the given rule. If the base sample can be placed into some of the blacklisted clusters using the given rule, this cluster was blacklisted already.

Sample blacklist collection

On the opposite of global `sample` collection, this collection is result-unique. When a sample is blacklisted, it is removed from the cluster in a specified clustering result (by removing a record from `clustering` collection) and inserted into sample blacklist collection. Note the `sample` collection remain untouched.

Sample collection

After a sample is analysed, all gathered data are stored as a JSON document in this collection. It is used by all clustering results. If necessary, for example, if each ad-hoc clustering would need to have a sample analysed by the different analysers, it can be result-unique as well as almost all other collections.

Clustering collection

A relation between sample and cluster is stored in the edge collection. An edge collection is similar to relation tables in SQL. Each edge has mandatory fields `_from` and `_to` defining oriented relation between documents.

Each sample can be connected up to one cluster, and each cluster has at least one sample. The edge document contains, besides a reference to sample and cluster identifiers, also a hash of the sample. Since the hash is considered to be unique for the sample and the sample can belong to one cluster at max, the hash can be used as a unique index.

Waiting room collection

One of the worst-case scenarios is the simultaneous arrival of samples belonging to just a single non-existing cluster. In this case, each worker tries to create a new cluster, leading to several equal clusters. The waiting room is a solution to this race condition of clusters during cluster creation.

A new key-value collection is required, where each cluster (from `clusters` collection) will use its own DB key, and the value will be the key of the cluster, it is waiting for. Clusters may un/register themselves in the waiting room to tell everyone that they are waiting and whom they are waiting for. If an older created cluster knows that a cluster is waiting for it, it can be successfully validated. Otherwise, it will start to wait for the newer cluster. If the newer cluster will not wait, it will not know about the older cluster and will therefore validate itself. This will lead for the older cluster to stop waiting and proceed with using the newer one. If the newer cluster starts to wait as well, a deadlock occurs. It is detected because both participants are registered in the waiting room and waiting for themselves. In this case, the older cluster will stop waiting and validate itself. This will lead for the older cluster to be used.

5.3 Clustering

None of the standard algorithms mentioned in Section 2.3 satisfies all the criteria mentioned in Section 5.1. The same applies to the state of the art approaches presented in Section 2.4. A review of both of them can be seen in sections 5.3.1 and 5.3.2. Because of that, three custom approaches are presented and more described in Section 5.3.3.

5.3.1 Standard Methods

Standard approaches mentioned in Section 2.3 are considered below. Requiring input parameters (such as a number of clusters), supporting offline clustering only, and hard interpretability of the reason behind a cluster creation are the most common problems.

Partitioning methods An expected number of clusters is not known, and it is not a constant. That applies not only in the case of continuous clustering but also in the case of ad-hoc clustering. Also, the algorithms are mostly working iteratively and create spherically shaped clusters. Lastly, it is hard to tell what the samples in the cluster have in common.

Hierarchical methods - BIRCH BIRCH is working with numerical data only. To use the method, the conversion of string data into numbers would have to be made. See section 4.5 describing PE sample. The cluster created by BIRCH method is represented by a summary (CF) and it is hardly interpretable.

Hierarchical methods - Chameleon It is a multi-phase clustering, which is not able to process data online.

Density-based It is hard to tell why the cluster was created. It applies to all density-based methods. Moreover, DBSCAN algorithm was already used in the first version of Clusty, and its shortcomings are mentioned in Section 4.2.

Grid-based They use a finite number of cells, which is not desired. The problems are similar to the ones in Partitioning algorithms.

5.3.2 State of the Art

The problems of the state of the art methods are similar to the problems with the standard methods mentioned in Section 5.3.1 above.

The approaches in articles [5, 14, 22, 27, 30] are aimed to use a special static or dynamic feature rather than to present a novel clustering method. The analysis is not a part of this work, and the approach should be more portable to the other file categories.

Another problem is requiring a whole dataset before running clustering. In article [5], they need to discard all features that are unique among the whole dataset, and in article [14], they cluster samples in iterations to find the best prototypes.

Moreover, article in [5], they uses one of the mentioned standard methods. Similarly in articles [13,30,33], but they use a combination rather than just a single method. Article [23] uses BIRCH, and article [27] even uses DBSCAN algorithm, which was used in the first version of Clusty.

The solution in article [4] is independent of the type of data, but it creates overlapping clusters, i.e. one sample can belong to multiple clusters.

5.3.3 New Methods

Newly proposed methods based on modifying existing methods (including the current Clusty clustering method) and considering all knowledge and requirements are introduced in the following text. They are specifically designed for this work.

Experimental selection

The method is an extension of an approach used in Clusty's current version by using multiple criteria. Instead of selecting attributes and their priority, a group of attributes with a group hierarchy is selected. Attributes in a group should be something which similar files would have in common and something which does not change per sample e.g. attributes used for clustering in the current Clusty. An inter-group comparison should support similarity, e.g. four out of five attributes are in common. It will make indexing harder.

```
Format: <attr1_value>_<attr2_value>_<attr3_value>
Raw: 0xFFA98C69, 123, Corrupted header
Shortened: 0xFF, 123, Corr
Encoded: 481207070_495051_67111114117
```

Figure 5.3: Format and example of values used for clustering by Experimental selection method

Because it is not possible to index such values when using the exact match, each sample could compute a value, which will be used to fetch clusters. An example of a format of the value can be seen in Figure 5.3. Each attribute in a group can be converted into a string and shortened to a maximal defined length. This length should be defined by experiments. Then, it should be encoded as bytes. It provides a safe splitting into a list if needed. To avoid matching missing values (None), they should not be included in the value or they can be removed before or after comparison. Besides that, a condition of clustering rule can be used to prevent even creating of clusters if not all attributes in the group are present.

The similarity between two clustering values can be measured by the length of the intersection of its parts. ArangoDB has support for using intersection, split, and length in a database query.
Pros and cons of the method can be seen in Table 5.1. Clustering properties as defined in Chapter 2:

Scalability: It should be capable of processing millions of samples. Only clusters matching the value have to be fetched from the database. All necessary operations are supported by ArangoDB database.

Ability to process different types of attributes (versatility): Aimed to work with various data types such as strings or numbers.

Creating clusters of arbitrary shape: Should create clusters of arbitrary shape.

Minimum knowledge of the problem when determining parameters: It is needed to determine groups of attributes somehow.

Ability to deal with data containing noise: Noise should create its own small clusters.

Incremental (online) clustering: It is possible to cluster samples online. There is not any required knowledge related to samples or clusters. All clusters are created on-demand.

Input record order insensitivity: The method is not sensitive to input record order.

Ability to process high dimensional data: There are no restrictions about dimensionality.

Constraint-based clustering capability: Constraints can be defined in the rules.

Interpretable and usable clusters: Each cluster should have at least some equal attributes for each sample. It depends on a rule definition.

Pros	Cons
Random values of attributes used in the clustering do not have to cause generating new clusters	Attributes for each rule have to be chosen, e.g. four rules each having three to five attributes
	Manually created rules
	Harder indexing

Table 5.1: Pros and cons of Experimental selection method

Ssdeep selection

This approach is similar to Experimental selection, but the selection is automatised using ssdeep similarity hashing. It should create a few multi-criteria methods based on the result of clustering by ssdeep. The basic principle is the following: An additional script or program should take as input a great amount of samples (e.g. 100 000+) for each category. It would analyse them and cluster them based on the ssdeep similarity of the whole file. Each cluster should contain only samples with similarity above some threshold. Common attributes would be computed for each cluster and then used to generate groups of attributes that tend to be together, e.g., `rich_header` and `pdb_path` appear together in common attributes of 90% of clusters. The script should return a few such groups based on some heuristics.

Using the resulted methods would not be the same as using ssdeep itself. The resulted methods will consist of both static and dynamic attributes.

Pros and cons of the method can be seen in Table 5.2. Clustering properties as defined in Chapter 2:

Scalability: It should be capable of processing millions of samples. Only clusters matching the value have to be fetched from the database.

Ability to process different types of attributes (versatility): Aimed to work with various data types such as strings or numbers.

Creating clusters of arbitrary shape: Should create clusters of arbitrary shape.

Minimum knowledge of the problem when determining parameters: No parameters needed, the rules will be auto-generated.

Ability to deal with data containing noise: Noise should create its own small clusters.

Incremental (online) clustering: It is possible to cluster samples online. There is not any required knowledge related to samples or clusters. All clusters are created on-demand.

Input record order insensitivity: The method is not sensitive to input record order.

Ability to process high dimensional data: There are no restrictions about dimensionality.

Constraint-based clustering capability: Constraints can be defined in the rules.

Interpretable and usable clusters: Each cluster should have at least some equal attributes for each sample.

Pros	Cons
Easy generation of methods for each category	Script generating methods is required (thousands/millions of various samples are required)
Fast indexing by full match	May require "re-learning" after some period of time
	Random values of attributes used in the clustering would generate new clusters

Table 5.2: Pros and cons of Ssdeep selection method

Gradual selection

This method is closer to the standard clustering methods than the methods mentioned above. It cannot be classified into a single type of clustering method, but it shares similarities with several types. It does not belong to partitioning methods because the number of clusters is not fixed. Rather the clusters are created on-demand. Similarly with grid-based methods. The method is more like a combination of hierarchical and prototype clustering.

Samples will be clustered according to the leading sample, i.e., the first sample in the cluster, which caused its creation.

All clusters should have a placer value (list of attributes to be compared) divided into fixed and volatile parts. Contrary to the fixed part, the volatile part does not need to be

matched exactly (some attributes can have different values). The fixed part needs to be matched exactly to sustain a common part across all samples in the cluster. A sum of attributes matching both parts must be greater than the specified minimum (further as E).

There will also be a need to set a threshold (further as T), specifying how many attributes have to remain in common at least in order to place the sample into the cluster. If the sample cannot be clustered into any cluster, a new one is created. But only in the case if it has at least T values. The sample cannot be clustered by this method otherwise.

If the sample has many common attributes with the cluster (further as R), but not enough to reach the value of parameter E, clusters on the lower level, related to the cluster on the upper level, are compared. Clusters on the highest level are fetched first, then lower ones, and so on. There should be a depth limit to avoid recursion.

By using this kind of hierarchy, it is expected that at the highest level, there will be a small number of clusters with very low inter-similarity. In the lower levels, clusters with their samples should tend to be similar to samples in the clusters above them or near them (siblings). Each cluster can be considered independent (do not use relation) or dependent, i.e., clusters on the higher level will also include samples from the lower levels (may be limited to specific depth).

In the case of blacklisting or reclustering a non-leaf cluster (in any level), all direct child clusters could become clusters on the level of removing cluster — even if they have any alternative connection to another cluster — because they have a hard value that would not correspond to any other cluster. Another solution is to choose master children who will become a new high-level cluster. There is also a possibility to cluster all samples anew.

Let describe clustering in an example. Table 5.3 shows nine samples, each having six attributes. The process of clustering a single sample is described in Code 5.1. The result of three different clustering runs is shown in Figure 5.4. Three clustering runs are shown in columns, where the first column presents clustering where samples arrived in order as can be seen in Table 5.3, the second column presents clustering where samples arrived in reversed order, and the last column presents clustering where samples arrived in reversed order and odd first. The lines follow pattern `N) V1 V2 V3 V4 V5 V6:  X/Y | Z`, where `N` is a cluster number in the order clusters are created, `V1 - V6` are values of attributes `A1 - A6`, `X` is the ID of a sample (according to Table 5.3), `Y` is the order number of the sample, and `Z` is the parent cluster where `_` means that there is not a parent cluster. There is a hierarchic structure of the clustering on the bottom of each column. The first shows cluster IDs, the second samples IDs.

_	S1	S2	S3	S4	S5	S6	S7	S8	S9
A1	A	A	A	I	I	A	V	A	_
A2	B	B	B	J	J	B	W	B	_
A3	C	C	C	K	K	K	K	C	_
A4	D	D	X	L	L	L	D	L	L
A5	E	E	Y	M	Q	Q	E	M	M
A6	F	G	Z	N	R	R	F	N	N

Table 5.3: Nine samples S1–S9 with up to six attributes A1–A6. S1 and S4 are totally different, the rest are their variations and combinations.

```
def cluster_sample(sample, level=0, parent_cluster=null) -> None:
   clusters = get_clusters(level, parent_cluster)
   for cluster in clusters:
      num = match(sample, cluster)
      if num >= S:
         place_sample_in_cluster(sample, cluster)
         return
      elif num >= R:
         cluster_sample(sample, level++, cluster)
         return
   place_sample_in_new_cluster(sample)
```

Code 5.1: Pseudo code of function to cluster a sample using proposed gradual clustering method

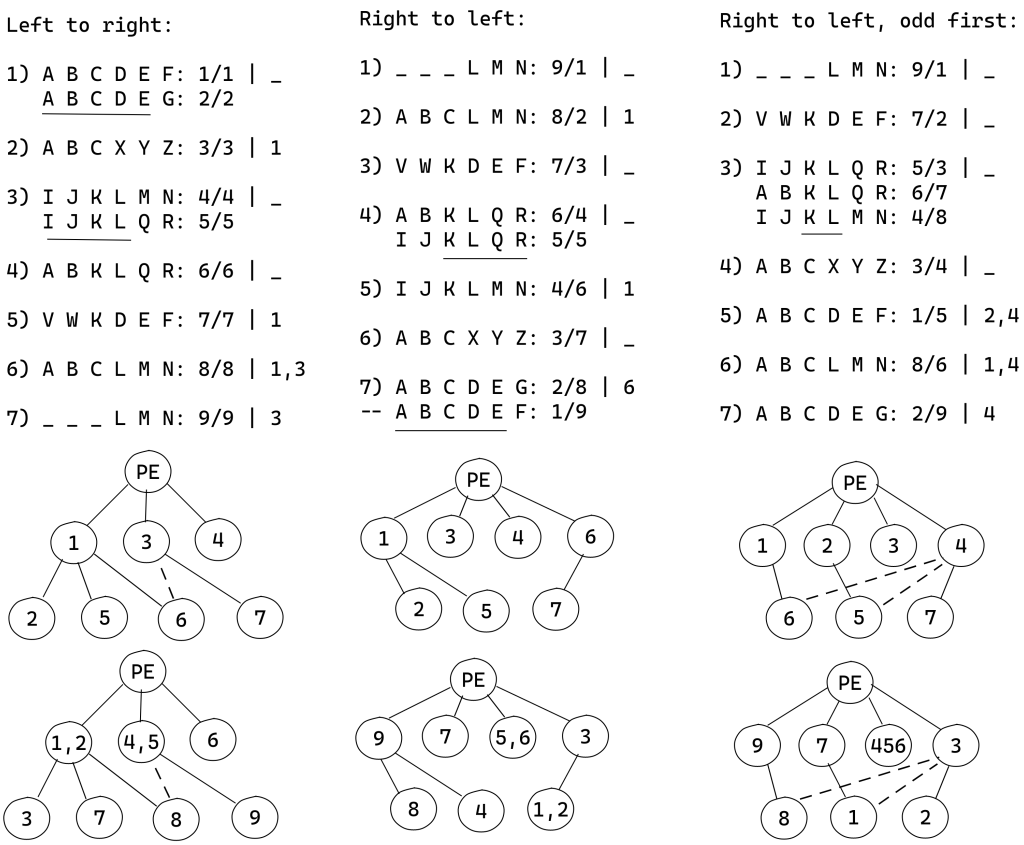


Figure 5.4: Clustering of samples from Table 5.3. Each of the columns represents different order of samples. The first diagram in column shows clusters' IDs, the second one shows samples' IDs.

Pros and cons of the method can be seen in Table 5.4. Clustering properties as defined in Chapter 2:

Scalability: It should be capable of processing millions of samples. Because it is often necessary to fetch many clusters, techniques to lower this amount are really handy. The main technique to reduce the number of samples to be fetched is using a hierarchy.

Only the clusters on the top of the hierarchy have to be fetched every time. If none of them is sufficient, either some cluster descendants are tried or a new cluster is created.

Another space reduction can be achieved using a bitmask. Each of the attributes used for clustering will have assigned a position in the mask revealing its presence in a sample or a cluster. Then, only clusters having a number of matching ones above parameter E are suitable for further investigation. If the cluster has not enough matching ones, it means that the cluster is not usable regardless of its values. The bite-wise operation has to be supported by the database to use this feature. Luckily, ArangoDB is able to do that.

Ability to process different types of attributes (versatility): Aimed to work with various data types such as strings or numbers.

Creating clusters of arbitrary shape: Should create clusters of arbitrary shape.

Minimum knowledge of the problem when determining parameters: Let's D be a dimension (number of attributes used in method). Then three parameters are required: $E; 0 < E < D$ - minimal number of equal attributes to consider cluster as correct one for sample, $R; 0 < R < D$ - minimal number of equal attributes to consider a cluster as related, $T; 0 < T < E$ - minimal size of the fixed part.

Ability to deal with data containing noise: Noise should create its own small clusters. If a sample is very different to any cluster representing a sample (base sample), a new cluster on the topmost level is created.

Incremental (online) clustering: It is possible to cluster samples online. There is not any required knowledge related to samples or clusters. All clusters are created on-demand.

Input record order insensitivity: The method is sensitive to input record order and thus can create different clustering result every time.

Ability to process high dimensional data: There are no restrictions about dimensionality. In the case of using a bitmask to reduce the number of fetched clusters, the mask is the limitation. For example, a 32b mask allows up to 32 dimensions. 32b is the current limitation of ArangoDB bit operations. It can be overcomed by using multiple masks.

Constraint-based clustering capability: We can consider parameter T to be a constraint specifying a minimum number of equal attributes among all samples in a cluster.

Interpretable and usable clusters: Each cluster should have at least T equal attributes for each sample. Also, as a side-effect of using cluster hierarchy, clusters can have relationships pointing out an inter-cluster similarity.

To sum up all ambiguities: (1) parameters E, T, R and a number of levels can be set either globally or specific for each method/category, (2) they can be set to a static value or computed at run-time, for example on the basis of cluster level, (3) each cluster can be either independent or dependent on another cluster forming a single cluster (i.e., they will be presented as a single cluster), (4) even it can be restricted to only a specific depth.

Pros	Cons
Hierarchical clustering - relations between clusters	Slower indexing
Automatic selection of relevant attributes from the set	A lot of network traffic because of fetching a number of clusters
	Sensitive to the order of samples

Table 5.4: Pros and cons of Gradual selection method

Chapter 6

Implementation

This chapter describes an implementation of the clustering tool designed in Section 5.2, clustering methods presented in Section 5.3.3, and all auxiliary scripts used during implementation and testing.

Clustering tool — further as Rusty (Rust + Clusty) — is implemented in Rust 1.50.0 edition 2018. ArangoDB version is 3.7.8, and the RabbitMQ version is 3.8.7. For easier testing and integration, Rusty is split into library and console application. They are described in sections 6.1 and 6.2, respectively.

Since Rusty provides only an interface for clustering methods and does not implement them, the implementation of clustering methods is described standalone in Section 6.3. Their implementations can also bring other features not mentioned in Section 6.1.

During the implementation of Rusty and experiments, a number of auxiliary scripts were required to automatise processes, generate statistics, test implementation, verify clustering results, etc. All of them are written in Python 3.9. According to *linguist*[1] (a tool used on GitHub.com), almost half of the source code are scripts. The most important one is *classify* package, which was used to verify clustering results and generate tables with various statistics. Auxiliary scripts are described in Section 6.5 and *classify* package in described separately in Section 6.4.

6.1 Rusty Library

Rust[2] was selected as a programming language. It is an alternative to C++ used in the current version. Rust is an open-source system programming language, designed and supported by Mozilla. It is focused on speed, memory safety, and parallelism. Rust was built from scratch and contains elements from proven system programming languages and modern programming language design. It combines the expressive and intuitive syntax of higher-level languages with a low-level language's control and performance. It also prevents segmentation failures and ensures thread safety. Even it does not support inheritance as C++ does, it supports interface inheritance using traits [21]. Also, even the language is less than six years old, it was considered the most beloved language among software developers five years in a row, including the year 2020 [20]. Last but not least, Rust has its own package manager called `Cargo`, providing libraries for working with ArangoDB and RabbitMQ applications.

[1] https://github.com/github/linguist
[2] https://research.mozilla.org/rust/

All packages in Rust consist of one or more *crates*[3]. A crate is a library or binary. A package can contain one library at max. A package contains a configuration file `Cargo.toml` describing crates and how to build them. The configuration file of Rusty can be seen in Appendix A.1.

Crates can be used as external dependencies for other crates. Crate `rustylib` uses the following third party dependencies:

- *arangors* for communication with ArangoDB database
- *chrono* to work with date-time structures
- *clap* to process console arguments needed for `get_result_name`
- *config* for parsing configuration from `INI` files
- *lapin* for communication with RabbitMQ server
- *futures_executor* for asynchronous work with *lapin* crate
- *indicatif* to display the progress bar
- *log* for logging
- *regex* to sanitise the name of directory with samples when generating result name by `get_result_name`
- *threadpool* for creating and managing a pool of threads

All dependencies with their used versions and features[4] can be seen in a snippet of the configuration file mentioned above in Appendix A.1.

Rust does not support class inheritance since it does not even have classes. Instead, it allows to define structures via `struct` keyword and implement methods for them via `impl` keyword. This allows to create objects having attributes and methods, and use `self` reference. To provide a common interface for several objects, Rust provides so-called traits. Structures implementing the sample trait have to provide all functions defined by the trait. Traits can be inherited, which causes the necessity to define functions for all deriving traits when implementing inheriting trait. All class diagrams referring to Rust in this work are Rust structures.

An entry point to Rusty library is a function `run_clustering` in `lib` module. It takes care of the creation of all necessary database collections and indexes, spawns a specified number of workers, publishes hashes of samples if needed, and replenish workers if any of them had failed. Besides that, the library provides `get_result_name` function for generating a result name from the given console arguments or from a combination of date-time and the given directory with samples.

A worker is defined in `worker` module in `Worker` structure. It does not have creator function (i.e. attributes are defined by instantiating structure) and provides only one method `run`. Worker has two attributes library configuration and clustering mode. The only parameter of `run` is the clustering result name (it is also a name of RabbitMQ query). It uses *lapin* crate to connect to the RabbitMQ server and *arangors* crate to connect to ArangoDB

[3] https://doc.rust-lang.org/book/ch07-01-packages-and-crates.html

[4] Cargo „features“ provide a mechanism to express conditional compilation and optional dependencies. See more https://doc.rust-lang.org/cargo/reference/features.html

server. Then it consumes incoming messages in a loop and clusters sample one by one by `cluster_sample` method.

All work with the database is enclosed in `db` module. This structure, alongside required collections, is described in Section 6.1.3. To cluster the sample, it first needs to fetch the sample from the database and deserialize it into `Sample` object. After that, it gets rules from a sample-specific placer by `get_rules` method (mandatory for each placer implementing `Placer` trait). Then, it goes through rules, one by one, and tries to cluster sample the current rule's clustering method. After the sample is clustered, it acknowledges the received message to the RabbitMQ server and continues with another sample. The cluster and sample structures are described in sections 6.3 and 6.1.2.

As was already mentioned above, a rule set is being run through in order defined in a specific placer. A sample placer is a place for the definition of clustering rules. It has to implement trait `Placer`. It can be defined for any type of sample, e.g. for binary.

Clustering rule is a structure which contains all necessary mechanisms to fetch usable clusters, to decide whether the rule can be used for the sample, and to decide whether a given cluster is right. It uses one of the implemented clustering methods.

The clustering method is a structure implementing `Method` trait. The trait requires implementing `place_sample_by_rule` function. A method has to define how clusters are fetched and updated based on the given rule. For example, if we take into account implementing the clustering method as defined in the current Clusty, i.e. clustering by a single attribute, a cluster created by the given rule and having a value of the given attribute is used. Updating of the cluster requires re-computation of common attributes only. Another method may require several clusters instead of one, etc.

The clustering method, clustering rule, and placer are described in sections 6.1.4, 6.1.5, and 6.1.6.

6.1.1 Cluster

A structure representing *cluster* database object is a simple structure without any methods. It uses *serde* and *serde_json* crates to serialise and de-serialise database object. Fields `id` and `key` created and required by ArangoDB are excluded from serialisation. The rest attributes are:

- `category` which cluster belongs to
- `rule` cluster was created by
- `value` cluster was fetched by
- `is_valid` referring whether cluster is valid or not
- `base_sample` which caused cluster creation
- `common` attributes of all samples in the cluster
- `created_at` date-time
- `updated_at` date-time

6.1.2 Sample

Rust does not allow structure inheritance. Each derived sample must contain a field with the deriving sample to avoid duplication definition of generic attributes. However, this makes accessing attributes convoluted. For example, to access `size` field in the grandparent, one has to use `var.sample.sample.size`, where `var` is variable with the derived sample. Using Rust macros to inject attributes is not possible.

To overcome this oddity, a trait is used to provide an interface for getting and setting values. Each derived sample structure has to implement all traits for all parents because of trait inheritance. The class diagram in Figure 6.1 shows the implementation of the generic sample, binary sample and PE sample. Each of them has a structure storing its fields and a trait providing an interface for them. It is then further abstracted behind `Sample` enumeration.

Crate *serde* allows to auto-generate serializer and deserializer for Rust structures and enumerations. The only thing needed is to call a macro. It is even able to serialise and deserialize database object into a structure as described above and shown in Figure 6.1. Code 6.1 shows Rust code where samples from the database in the form of JSON objects are automatically converted into a vector of `Sample` structures regardless of their specific type. The type is determined by the value of `_type` of each document.

```
let aql = AqlQuery::builder()
    .query("FOR␣sample␣IN␣samples␣RETURN␣sample")
    .build();
let samples: Vec<Sample> = database.aql_query(aql).unwrap();
```

Code 6.1: Rust code showing automated parsing of a list of database objects into a vector of `Sample` structures

Because Rust is very strict to types, it was better to cover up sample inheritance into enumeration called `Sample`. It enumerates all defined sample structures and implements all getters and setters for them. It means that all samples in enumeration have available getters and setters for all existing attributes regardless its own structures. It returns `None` if the current structure does not have acceded attribute.

6.1.3 Database

All database-related functions are implemented in `ArangoDB` structure. This structure holds database connection, database object, all necessary collection names, the name of the current clustering result, and `is_ad_hoc` flag.

All clustering result-specific collections use the result name as a suffix in the collection name. For example, if the result name is “rusty”, the name of clusters collection will be “clusters_rusty”, the name of clustering collection will be “clustering_rusty”, and so on. All collections are described in Table 6.1. Placeholder symbol `<rn>` stands for result name. All collections have created indexes for `_id` and `_key` by default. Edge collections have `_from` and `_to` indexes in addition. The collections purposes are described in Section 5.2.2.

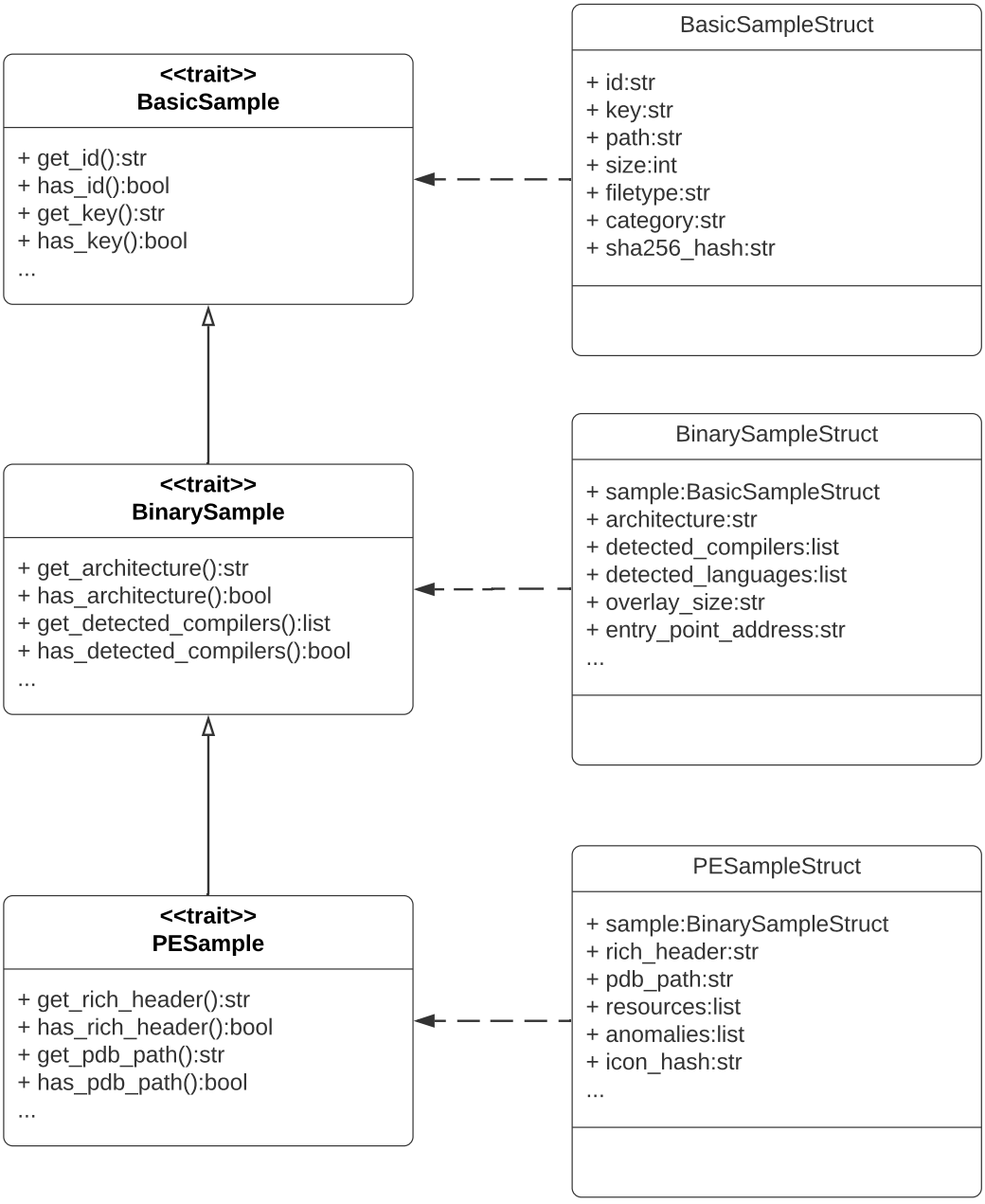


Figure 6.1: Class diagram of samples' inheritance. Each structure contains structure of inherited sample. The traits encapsulate the work with the nested structures.

Collection name	Indexes
`samples`	`sha256_hash`
`controller`	N/A
`clusters_<rn>`	`category`, `rule`, `value`
`clustering_results`	`name`
`blacklist_<rn>`	`category`, `rule`, `value`
`blacklist_samples_<rn>`	`sha256_hash`
`clustering_<rn>`	`sha256_hash`
`waiting_room_<rn>`	N/A

Table 6.1: Database collections with appropriate indexes

6.1.4 Method

Rusty supports defining multiple clustering methods. To define a new clustering method, one has to create a structure implementing `Method` trait, and register this clustering method in `MethodType` enumeration. It can be used then for creating clustering rules.

The clustering method has to implement `place_sample_by_method` function, which tries to place the given sample with the given rule into a cluster. It is up to the method to choose a proper cluster for the sample. Whether the given rule can be or cannot be used is determined by the condition in the rule.

6.1.5 Rule

Each clustering rule is implemented via `Rule` structure. It has a unique global name. It has to use one of the supported clustering methods. Supported clustering methods are stored in `MethodType` enum. If the clustering rule needs to encapsulate any related configuration, it can be stored as a key-value dictionary in `config` field.

Each rule must provide `value` according to clusters will be fetched. I.e. if a rule is based on a single attribute `rich_header`, the value would be this attribute's value. A field *value* is a closure taking sample as a parameter and returning the value extracted from the sample. Fields `cond` and `placer` are closures as well. Closure `cond` also takes a sample as a parameter and defines a condition that has to be met to use this rule for clustering. It can use the parameter to perform any logic that is required to evaluate rule usability. The last closure `placer` takes sample and cluster as parameters and determines whether the sample can be placed into the cluster or not. It does not return a boolean value but an integer. Zero means false, anything else means true.

All closures are stored in the `Rule`'s structures, but they are exposed in the form of methods. A method `get_value` calls `value`, method `can_be_used` calls `cond`, and method `place_sample` calls `placer` converting its return value to boolean. The last method `get_similarity` calls `placer` function same as `place_sample`, but it preserves its return value. This method can be used for clustering rule, which returns additional info instead of simple boolean.

```
pub fn pe_rich_header() -> Rule {
    RuleBuilder::new("pe_rich_header", MethodType::KeyFeatures)
        .with_value(Box::new(|sample| sample.get_rich_header()))
        .with_condition(Box::new(|sample| sample.has_rich_header()))
        .with_placer_function(Box::new(|sample, cluster| {
            (sample.get_rich_header() ==
                cluster.base_sample.get_rich_header()) as i32
        }))
        .build()
}
```

Code 6.2: Example of implementing `Rule` using `KeyFeatures` method with a single attribute `rich_header`. If a sample contains `rich_header`, the rule can be applied and the sample can be clustered by the value of `rich_header`.

Because of defining several closures in a row make it less readable, `RuleBuilder` can be used for `Rule` creation instead, as shown in Code 6.2. It provides `with_config`, `with_value`, `with_condition`, and `with_placer_function` building rule. Rule's `name` and `method_type` are passed in builder initialisation.

6.1.6 Placer

A placer is a collection of rules for a specific category. It has to implement `Placer` trait from `placer` module. Specific placers are implemented in specific modules, e.g. `PEPlacer` with all rules and trait implementation is located in `pe_placer` module.

In order to allow the re-usability of rules, the rules can be defined in standalone functions inside a specific placer. Any other specific placer can import these function. An example of a rule defined in a standalone function can be seen in Code 6.2.

```
pub struct PEPlacer;

impl Placer for PEPlacer {
    fn get_rules(config: Option<&config::Config>) -> Vec<Rule> {
        vec![pe_rich_header()]
    }
}
```

Code 6.3: Example of implementing `Placer` trait for `PEPlacer` with the only one rule from Code 6.2 above. All samples will be clustered based on the value of `rich_header`. If the attribute is missing, the sample can not be clustered.

To implement a placer for a new category, one has to create a new module and structure. The structure can be empty but has to implement `Placer` trait which exposes `get_rules` method. Also, he needs to register the placer in `Worker::cluster_sample` method under the appropriate sample's category. An example of the placer implementation can be seen in Code 6.3. The placer can import rules from other placers or implement new ones.

6.2 Rusty Tool

Rusty as a binary is a tool combining provided arguments and parsed configuration from a configuration file into parameters passed to Rusty library. It uses *clap* crate to parse console arguments, *simplelog* and *log* crates to set up logging, and *config* crate to parse the configuration file. All arguments have default value except `file`. Clap crate supports creating constraints between arguments. For example, `file` argument is mandatory in reclustering or ad-hoc mode only. All available arguments and the description can be seen in Figure 6.2. An example of running Rusty can be seen in Figure 6.3. It shows two progress bars. The first is a progress bar for publishing messages and the second one is for clustering of samples. It can be built by running `$ cargo build --release`. Then it can be executed by `$ ./target/release/rusty`.

Argument `-j` or `--join` can be used in any mode. Its purpose is to join already running clustering. The difference is that it will not publish any messages, only create a pool of workers.

```
$ ./target/release/rusty --help
Rusty 0.1.0
Multi-criteria clustering daemon

USAGE:
    rusty [OPTIONS]

FLAGS:
    -h, --help       Prints help information
    -j, --join       Join already running clustering
    -V, --version    Prints version information

OPTIONS:
    -c, --concurrency <N>       Maximal number of workers that will be used to analyse and
                                 cluster the samples. By default, the number from
                                configuration file is used.
    -f, --file <PATH>           Path to input file with sample hashes or cluster IDs.
    -m, --mode <MODE>           Clustering mode [default: continuous]  [possible values:
                                 continuous, ad-hoc, reclustering]
    -n, --result-name <NAME>    Result name
```

Figure 6.2: Help message of Rusty tool

Continuous mode does not expect arguments. The result name is a constant, and the samples are not loaded from any file, rather they arrive continuously by other services.

Ad-hoc mode expects a file with hashes of samples to be clustered. If the result name is not provided, a new one is generated based on filename and date-time.

Reclustering mode is the same as ad-hoc mode, except the sample is not skipped if it is clustered already.

```
$ ./target/release/rusty -c 64 -m ad-hoc -f hashes.txt
  done 100% [##############################] 100000/100000 [00:00:05<00:00:00, 16723/s]
  done 100% [##############################] 100000/100000 [00:21:04<00:00:00, 79/s]
```

Figure 6.3: Example of running clustering in ad-hoc mode, using 64 workers, and taking hashes of samples from hashes.txt file

6.3 Clustering Methods

This section describes an implementation of the clustering method described in Section 5.3.3. The Gradual selection method is implemented in `HierarchicalMethod`, and Ssdeep selection is implemented in `KeyFeaturesMethod`. Method `KeyFeaturesMethod` is not usable for Ssdeep selection only. Rather it can be used for other methods (not mentioned in the work) using a group of attributes with an exact match for clustering. If the group has a size one, it can implement a method in the current Clusty. Both methods implement `Method` trait and are registered in `MethodType` enumeration. Rules created by the different methods can be used at the same time.

6.3.1 Gradual Selection

Besides implementing `Method` trait, the method adds a new result-specific collection named `hierarchy_<rn>`, where `<rn>` is clustering result name. It stores relationships between clusters, i.e. cluster hierarchy. It also needs to extend `Cluster` structure with the following attributes:

- `mask` used to reduce the number of clusters fetched from the database
- `fixed_part` to store names of attributes common used for clustering
- `volatile_part` to stored the rest of clustering attributes
- `level` storing level of cluster (-1 for non hierarchical method)
- `value` holds the subset of attributes selected for clustering

The configuration of parameters can be stored in the configuration file or the constants. The configuration overwrite defined constants to allow specifying values in tests.

An algorithm of searching for a cluster can be seen in Code 6.4. It uses postponed decrementation of parameters E and R (see Section 7.4.3). Condition $R < 2$ is a depth limit for clusters and recursion. What clusters are fetched depends on the stage of clustering. All gathered clusters are then processed, and the most suitable or related clusters are collected and sorted. Then, the suitable clusters are processed first, and if any of them is valid, it will be used. If all of them were removed or there were not such clusters, all related clusters are processed. If any of them is valid, it will be used in a recursion call as a parent cluster.

```
def find_proper_cluster(current_cluster_parent_id, level, new_cluster):
    E = E - max(current_level - 1, 0) # Postpone decreasing by one level
    R = R - max(current_level - 1, 0)
    if R < 2:  # Do not create relationship based on a single common attribute
        return current_cluster_parent_id
    if current_cluster_parent_id: # Recursive call searching for children
        clusters = get_child_clusters(current_cluster_parent_id)
    elif new_cluster:  # Second fetch after creating new cluster
        clusters = get_recently_created_top_level_clusters()
    else:  # First fetch before creating new cluster
        clusters = get_top_level_clusters()
    for cluster in clusters:  # Iterate over all clusters
        similarity = rule.get_similarity(sample, cluster)
        if similarity >= E:
            suitable_clusters.append(cluster)
        elif abs(similarity) >= R:
            related_clusters.append(cluster)
    for cluster in sorted(suitable_clusters):
        if was_deleted(cluster):  # Wait until validity or deletion
            continue
        delete_cluster(new_cluster)  # Suitable cluster exists already
        update_cluster_and_place_sample(cluster, sample)
        return None  # Successfully clustered into existing cluster
    for cluster in sorted(related_clusters):
        if was_deleted(cluster):  # Wait until validity or deletion
            continue
        return find_proper_cluster(cluster.id, level++, new_cluster)
    # A proper cluster was not found, return parent ID if is some
    return current_cluster_parent_id
```

Code 6.4: Pseudocode of selecting a proper hierarchical cluster using `HierarchicalMethod`

6.3.2 Ssdeep and Experimental Selection

Ssdeep selection method requires creating a script that will generate groups of attributes based on the clustering by ssdeep hash similarity. Because Clusty already supports clustering by ssdeep hash, it was used for it. The script generating groups is described in Section 7.5.1.

Both approaches use the same algorithm of searching for the cluster and placement into cluster implemented in `KeyFeaturesMethod`. Searching algorithm can be seen in Code 6.5. The difference between them is that Ssdeep selection provides an exact match to get suitable clusters, while Experimental selection requires advanced operations in the database query, like intersection.

```
def find_proper_cluster(clusters, new_cluster):
    for cluster in clusters:  # Iterate over all clusters
        if rule.place_sample(sample, cluster)
            if was_deleted(cluster):  # Wait until validity or deletion
                continue
            if new_cluster:
                delete_cluster(new_cluster)  # Suitable cluster exists already
            update_cluster_and_place_sample(cluster, sample)
```

Code 6.5: Pseudocode of selecting a proper cluster using `KeyFeaturesMethod`

6.4 Classify Tool

Classify is a Python3 package used for obtaining various information, mostly based on the sample's classification. These values can be used for evaluating of quality of clustering results or for comparison. The main requirement of having a classification for each sample in the clustering result. The design of architecture and implementation of the extraction process is described in Section 6.4.1. Example usage with the description of output in input is described in Section 6.4.2.

6.4.1 Design and Implementation

The design consists of an entry point file and two classes. It uses several Python packages:

- *click* to process console arguments
- *plotly* for plotting graphs
- *tabulate* for generating tables
- *tqdm* for displaying a progress bar
- *coloredlogs* alongside *verboselogs* to provide useful messages

All console arguments are described in Section 6.4.2. A role of the entry point script, except parsing of arguments, is to gather all clusters for the given clustering result. These clusters are then processed by `ClusterListStatistics` class, which uses `ClusterStatistics` class to keep data about each cluster. After processing all clusters, it also computes final statistics for all clusters and generates output tables.

ClusterStatistics

It is a Python dataclass[5] storing information about a single cluster. It holds a list of samples, lists of all found classifications for each classification part (i.e. severity, type, strain), flags revealing whether a cluster is mixed or not according to a specific source, and all generated logging messages.

It exposes several public methods for easy addition of attributes, e.g. a new sample. A method `verify` is used to generate flags and messages upon collecting all samples in the cluster. A mixture of the samples indicated by the flags is determined by the process

[5] https://docs.python.org/3/library/dataclasses.html

described in Code 6.6. The first evaluated classification part is strain, then type, and then severity. If there is just a single value (e.g. there is only one value when considering strain), an appropriate mixture flag is set to false. If there is more than a single value, it now depends on the previous flag. If the previous flag was false, it is set to false regarding its mixture. If the previous flag is true, it will be set according to the mixture of the current part. The reason is that if the identified strain is the same among all clusters, having different types is not considered a mixture rather as a bug in the classifications. Like having several aliases for the same strain, there can also be cases of having several types for the same strain. It can be caused by older classifications for example.

```
def set_flags(classification_attr)
    len = classification_attr.len()
    if len == 1 or (len == 2 and has_none and skip_none):
        print("Single classification: {}", classification_attr[0])
    elif len > 1:
        # Set only if related attribute is mixed as well.
        # Missing classification (None) can cause mixed cluster as well.
        # If strain is not mixed, type cannot be mixed.
        # If type is not mixed, severity cannot be mixed.
        # There is possibility that all parts will be mixed in the same time.
        if related_attr_is_true or related_attr_has_none_and_another_value:
            print("Multiple classifications: {}", classification_attr)
    else:
        # No classification.
        print("No classification.")
```

Code 6.6: Pseudocode of determining mixture of cluster for a single classification part

ClusterListStatistics

Each cluster obtained from the database by the entry point script is processed by this dataclass. It creates `ClusterStatistics` dataclass for each cluster and sequentially adds information about samples. After each cluster is parsed and evaluated, it is further processed by this class. It holds a list of parsed clusters and a number of various counters used for generating statistics about the clusters. The method `compute_statistics` iterates over all parsed clusters and aggregate information about them. For example, it computes the number of clusters, the number of samples, the number of mixed clusters for each classification part, and the number of samples in the mixed clusters for each classification part.

6.4.2 Usage and Example Output

All supported arguments can be seen in Figure B.1. The tool is able to show mixed clusters, their classification, and hashes of samples inside them. The clusters can be filtered according to a specific classification part, e.g. show only clusters with mixed severity. It takes classifications from two sources. The first classification obtained from Avast internal tool Tagger, and the second one from the classification of Clusty cluster a sample belongs to. Argument `DB_NAME` is the name of ArangoDB database where the result named `RESULT_NAME` is being searched. Argument `--clusty-result-name` refers to a clustering result where the classification for samples will be gathered from. The classification from Tagger is considered as a ground truth. It is possible to filter clusters based on the specific source of classification, e.g. Clusty, or to use the better one. It also allows excluding missing classifications

(`None`) from participating in evaluating a mixture of clusters. Note that filtering clusters do not affect generated tables. It affects only listed clusters. A part of the output can be seen in Appendix B.2.

6.5 Auxiliary Scripts

The auxiliary scripts were used for various purposes. For example, automating processes like moving samples between databases or obtaining various data like the hierarchical tree of clusters. Below is a list of the scripts with a brief description of their purpose:

- Scripts `update_tags_arango.py` and `update_tags_mongo.py` were used to insert classification from Tagger into samples in the specific database. They also filter out corrupted and infected samples and rename known aliases to the canonical names of their strains. If a sample is corrupted, it is not harmful since it is not executable. The current version of Clusty places corrupted samples according to their corruption. Sample with classification infected is a clean sample infected with malware. Clusty uses advanced techniques,like YARA rules, to detect those samples. An aliases are alternative names for strains. Without renaming aliases to their canonical names, clusters having samples with several aliases would be considered mixed.
- Script `to_arango_json.py` was used to copy samples analysed in Clusty from Clusty's MongoDB to Rusty's ArangoDB.
- Script `speed_test.py` was used for testing clustering speed and generating graphs. An examples of such graphs can be seen in Figure 7.5 and Figure 7.6.
- Script `send_msg_to_rmq.py` was used in the beginning of the project. Its purpose is to send hashes of samples stored in a file to clustering running in continuous mode.
- Script `remove_empty_values.py` removes empty attributes from sample objects in the database.
- Script `get_pe_stats.py` was used to gather statistics about PE samples. An example of the output can be seen in Table 7.1 and in Figure 7.4.
- Script `get_dataset_classification_stats.py` was used to get data about classifications in the dataset. An example of the output can be seen in figures 7.1, 7.2, and 7.3.
- Script `get_cluster_tree.py` was used to generate a tree of clusters as can be seen in Appendix D.1.
- Script `get_aliases.py` was used to combine malware aliases from various sources since there was not a single database containing all of them.
- Script `generate_hierarchic_samples.py` was used for generating samples having specific attributes to test hierarchic clustering in a deterministic way.
- Script `generate_code_from_template.py` was used as a partial automation of the process of creating rules based on Ssdeep selection.

- Script `copy_arango_collection.py` was used to copy samples between databases in Rusty (e.g., test database and standard database).
- Script `clusty_clustering_results.py` was used to gather clustering statistics from the Clusty clustering result. An example of such statistics can be seen in Table 7.2.
- Script `clusty_clusters_common_attributes.py` was used for extracting common attributes of clusters created by ssdeep hash similarity in the current Clusty. The script has also a simple test script `tests_clusty_clusters_common_attributes.py`. Its behaviour is further described in Code 7.1.

Besides the scripts mentioned above, there was also an attempt to visualise clusters on the web, especially the hierarchical ones. This attempt is implemented in the script `app.py`. A preview of cluster hierarchy can be seen in Figure F.1. Each cluster will show the cluster-related information when clicked. In this case, we can see 12 clusters, four of them are top-level clusters. The displayed information concert the cluster with ID `847751502`.

It worked for several clusters but not for thousands. It can be optimised in many ways, but the app was not a part of the work target, and therefore it was only not further used in this way.

However, it can also display information about a specific cluster, e.g. what is common and what are the differences between samples, which was quite usable for manually viewing the clusters. For example, in Figure F.2, we can see a cluster having five attributes in a fixed part. They are the same for all samples in the cluster. The cluster contains five samples. The first row, below the header of the volatile part, with bold values, is a base sample of the cluster. Not all attributes are listed in the figure because of a lack of space. We can see that samples are very similar since they have five common values and the rest seem to follow the same pattern. The `version_info` attribute seems to contain auto-generated values, leading to a tiny cluster if used for clustering.

Chapter 7

Experiments

This chapter describes experiments with clustering methods presented in Section 5.3.3. All experiments were done on the dataset described in Section 7.1. The clustering results of the current version of Clusty can be seen in Section 7.3. To understand the tables and colours used in this section, see Section 7.2. Experiments with the Gradual selection method are described in Section 7.4, experiments with Ssdeep selection method in Section 7.5, and experiments with Experimental selection method in Section 7.6. All methods were designed and implemented as described in sections 5.3.3 and 6.3.

Python package *classify* (see Section 6.4) was used to verify, compare, and evaluate clustering results. The scripts described in Section 6.5 were used for generating various statistics and other related work.

All experiments and tests were executed on a server with `Linux 4.19.0-16-amd64 Debian 4.19.181-1 (2021-03-19) x86_64 GNU/Linux`. It has 2 CPUs `AMD EPYC 7502 32-Core Processor` and 252 GB RAM. All Rusty, ArangoDB, and RabbitMQ run on the same server.

7.1 Dataset

The dataset consists of more than 500 000 PE samples. They were obtained from Avast internal database by selecting samples from January and February 2021. To assure they are not from a single day or a single week, they were selected in several batches with time ranges (1–2 January, 3–4 January, etc.).

The selected samples were then analysed and clustered by Clusty. The analysed samples were then transferred from Clusty's MongoDB database into Rusty's ArangoDB database. All transferred samples were then extended with classification (i.e., severity, type, strain) obtained from another Avast internal tool, Tagger, which holds Avast classifications for samples. Samples without Avast classification, with severity `damaged` or `infected`, or with classification having the confidence below 70 were removed from the dataset. Also, all known aliases were replaced with their canonical names.

Samples without classification were removed due to the inability of determining the correctness of their placement in clusters. If a sample does not have a classification, we cannot tell whether it is malware or not and what strain exactly. It would require further investigation of each sample. Having classification for each sample allows computing the correctness of clusters without any further investigation.

Sample with classification but lower classification confidence was removed due to increased probability of the wrong classification. It could cause additional mixed clusters. Only clusters with classification created with high confidence were preserved.

Damaged samples were removed due to their special status. If a sample is corrupted, it is not harmful since it is not executable. The current version of Clusty places corrupted samples according to their corruption.

Sample with classification infected is a clean sample infected with malware. It means that the base of the sample is a clean sample, which was later infected by malware by inserting malicious code into the sample. This kind of samples is hard to detect since many extracted features belong to the clean part of the sample. Clusty uses advanced techniques, like YARA rules, to detect those samples.

After removing unwanted samples, the size of the dataset was reduced to approximately 500 000 samples, each sample having an Avast classification with confidence above 70. Due to the great number of experiments that had to be done, this subset was further reduced to 100 000 (by selecting random samples using `RAND` function in ArangoDB) in order to decrease the duration of clustering. This subset was used for experiments, and all tables and figures in this chapter refer to this subset unless explicitly stated otherwise.

All attributes obtained from analysis by Clusty can be seen in Table 7.1. The first column holds the attribute's name. The second column shows how many of the selected 100 000 samples had any value for an appropriate attribute. The greener the higher percentage, the redder the lower percentage. The third column shows how many distinct values were found. The last column shows a percentage ratio between all values and unique values. The middle value 50 % is lighter, the lesser or greater values are redder. For example, 87 % of samples have attribute `anomalies`. The distinct values made 0.13 % of all values (116 in total).

What we can see in Table 7.1 is that there are several groups of attributes. The first group are stable attributes present in almost all PE samples, e.g. `size`, `architecture`, `entry_point_address`, or `section_table_hash`. Then, there is an opposite group of very rare attributes, e.g., `uncommon_timers`, `uncommon_pipes`, `uncommon_jobs`, or `watermark`.

Last but not least, we can see is that there are attributes with only a few distinct values, e.g., `architecture` (always `x86`) or detected languages (mostly `Visual Basic`). Then there are also attributes with a great number of distinct values, e.g., `uncommon_mutexes` or `uncommon_atoms`.

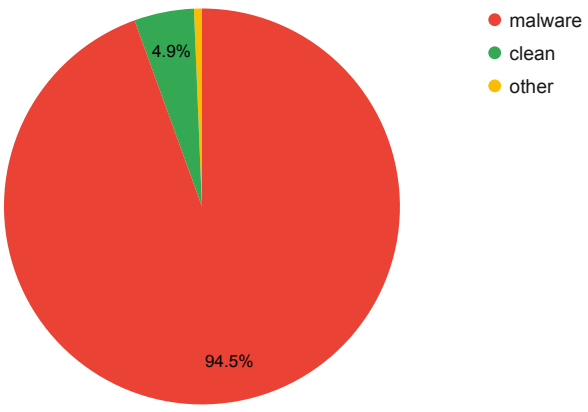


Figure 7.1: Classification severity distribution among samples

Attribute name	Has value [%]	Unique values	Unique values [%]
anomalies	87	116	0.13
api_calls	51.51	6492	12.6
architecture	100	1	0
is_corrupted_but_loadable	2.38	2	0.08
corruption	2.38	19	0.8
detected_languages	15.76	9	0.06
detected_compilers	92.99	490	0.53
entry_point_address	100	4158	4.16
entry_point_bytes	92.24	24755	26.84
exports	1.12	242	21.59
export_table_hash	1.12	242	21.59
icon_hash	41.9	3985	9.51
import_table_hash	93.74	8011	8.55
imports	94.26	7921	8.4
manifest_hash	21.62	1070	4.95
overlay_size	63.88	28341	44.37
pdb_path	3.88	676	17.42
resources	50.19	2855	5.69
rich_header	33.66	1858	5.52
section_table_hash	100	6063	6.06
sections	100	881	0.88
size	100	35696	35.7
symbols	2.06	278	13.51
version_info	34.79	3803	10.93
watermark	0.18	15	8.33
uncommon_atoms	1.18	693	58.78
uncommon_commands	35.71	26621	74.55
uncommon_cuckoo_signatures	50.74	4380	8.63
uncommon_dependencies	0.65	389	59.85
uncommon_events	1.36	543	40.04
uncommon_gvma_signatures	49.81	1359	2.73
uncommon_hosts	4.28	340	7.94
uncommon_jobs	0.01	6	60
uncommon_mailslots	0.02	7	36.84
uncommon_mutexes	26.45	1694	6.41
uncommon_named_sections	4.86	186	3.83
uncommon_pipes	0.46	189	41.09
uncommon_ports	1.01	354	35.08
uncommon_registry_keys	32.11	2872	8.95
uncommon_scheduled_tasks	1.08	100	9.29
uncommon_semaphores	5.31	1911	35.95

Table 7.1: Histogram of PE sample attributes in the dataset

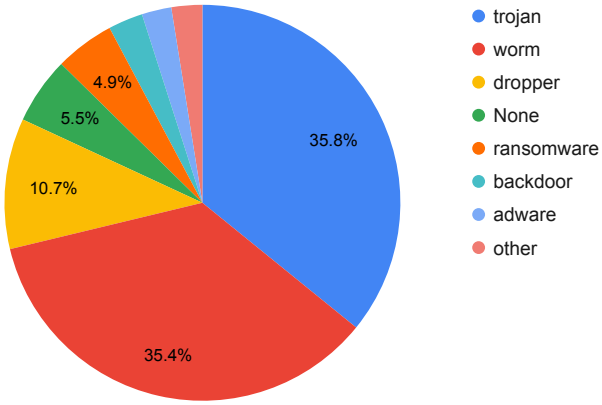


Figure 7.2: Classification type distribution among samples

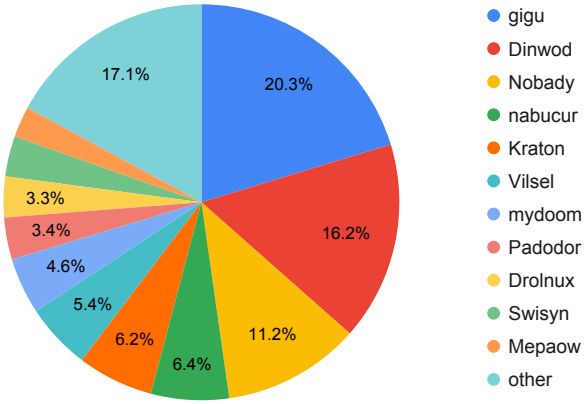
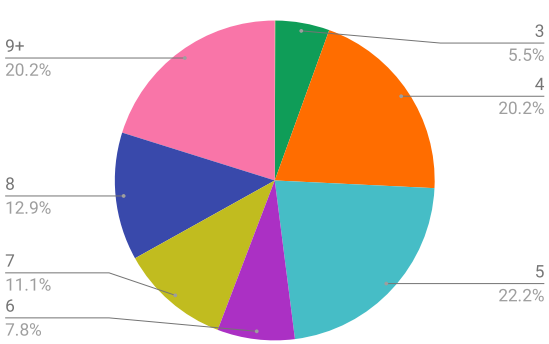


Figure 7.3: Classification strain distribution among samples

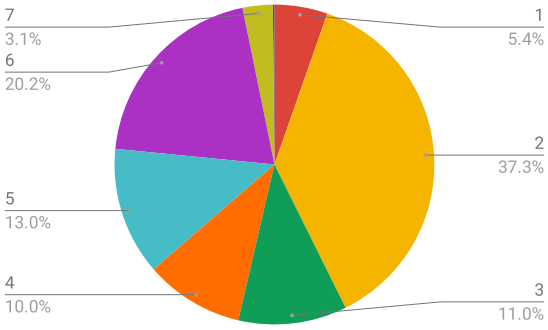
Figures 7.1, 7.2, and 7.3 show a distribution of classifications among 100 000 samples. We can see that almost all samples are malware, the most common type is trojan, and the most common strain is *gigu*. Note that the classification of clean sample is *clean* with type and strain set to *None*. Missing type (*None*) does not automatically mean the sample is clean. There can be another reason. Also, even if the type has a value, severity can be missing. The chart in Figure 7.2 shows 5453 *None* classifications, while the chart in Figure 7.1 shows only 4876 clean samples.

On the basis of Table 7.1, attributes with variability below 1 % were excluded from further consideration and usage for clustering: `is_corrupted_but_loadable`, `anomalies`, `architecture`, `corruption`, `detected_compilers`, `detected_languages`, `overlay_size`, and `size`. Attributes `exports`, `imports`, and `sections` were also excluded because of the presence of their hash variant in attributes (e.g. `sections` ≅ `section_table_hash`).

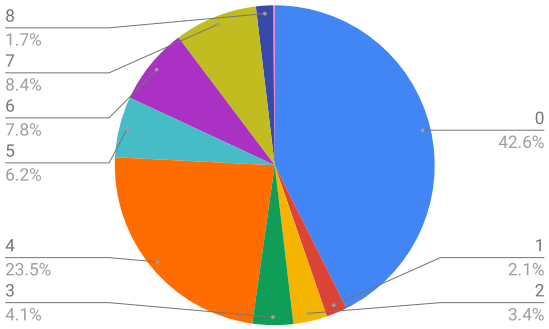
The remaining attributes can be divided into static and dynamic. This division can be seen in Appendix C.1. Figure 7.4 shows a number of attributes having value per sample. Figure 7.4a shows a chart for a subset of all static attributes. Each sample has at least two such attributes and 12 at max. The average is 6.3. Figure 7.4b shows a chart for the subset of all static attributes used for clustering in the current Clusty (a subset of all static attributes). Figure 7.4c and Figure 7.4d show similar charts but for dynamic attributes.



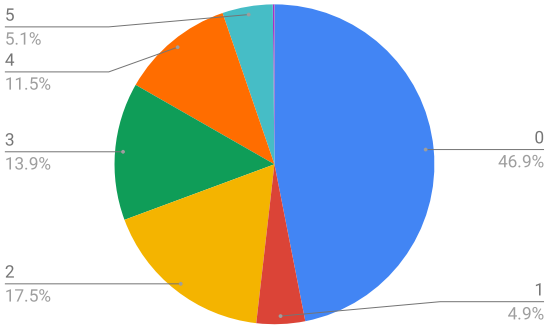
(a) All static attributes (2/12/6.3)



(b) All static attributes used for clustering in Clusty (1/8/3.6)



(c) All dynamic attributes (0/11/2.7)



(d) All dynamic attributes used for clustering in Clusty (0/7/1.5)

Figure 7.4: A number of attributes per sample (min/max/avg)

7.2 Clustering Results Overview

Clustering results in this chapter are evaluated and compared using data obtained by `classify` (see Section 6.4). It gives an aggregated statistics about the clustering result. A shortened version of the output can be seen in Appendix B.2. It returns a number of created clusters, a number of samples in clusters, and a number of samples that could not fit in any clusters. Further, it returns a table showing a level of a mixture of clusters. Signs `[s]` and `[c]` reveal whether the number is based on samples or clusters. Row `TAG` shows statistics according to classification in samples' tag attributes. Row `TAG (s N)` means that `None` classifications are not used in the measurement. Rows with `CLUSTY` are based on the classification of clusters in clustering result created by Clusty. The table on the bottom shows a histogram of sizes of created clusters. The empty cells were emptied due to a lack of space on the page. They contain values similar to the values in other cells.

The most relevant and important attributes are collected into another table for each clustering result. An example of the table can be seen in Table 7.4. All numbers are related to the tested clustering method. The table mostly consists of the following columns: (1) *Name* is a unique name of the clustering result. It often reveals a clustering configuration, (2) *o*, if present, means an ordering of rows in a group. The smaller the better, (3) *clustered* is the number of clustered samples, (4) *clusters* is a total number of clusters, (5) the first *tiny* is a percentage of the total number of tiny clusters, (6) the second `tiny` is a percentage of the total number of samples in the clusters, (7) *huge clusters* is a percentage of clusters

having more than 500 samples, (8) *severity* shows a percentage of samples clustered into mixed clusters according to severity, (9) *strain* shows the sample as a column before but according to strain, and (10) *strain (s N)* shows the same as a column before but does not count *None* classifications.

The colours used in tables have the following purpose: (1) *red* cells have value worse than Clusty, i.e. the values which are not acceptable, (2) *green* cells show a comparison between clustering results. The darker green, the better value, and (3) *yellow* colour indicates achieving a specific value unless stated otherwise.

A clustering result cannot be evaluated by a single metric (e.g. number of clusters) instead of a complex view on more data. For example, having all classifications better than Clusty cannot be considered a success if the number of clusters is so great that the average cluster size is three. The complex view means that all the columns have to be taken into account, each having its own level of importance. The most important columns are the total number of samples and all classifications. Then there is a number of clusters samples and then clusters sizes.

7.3 Current Clusty

The samples in the dataset were clustered by the current Clusty version to gain the ability to compare the results of experiments. The results from clustering using all methods in Clusty can be seen in Table 7.2. The results excluding YARA rules (which are the main method used for clustering) and digital signature-based methods can be seen in Table 7.3. The upper part of the table shows a number and a percentage of clusters mixed according to one of the classification parts, e.g. severity. TAG is the source of classification. The row with *skip None* means that *None* classifications were not considered during computation. The bottom part of the table shows the number and percentage of samples located in mixed clusters. The number in brackets in the top left corner is the total number of clusters. For example, in Table 7.2 in the upper part, we can see that there were 5 310 clusters in total, 17 clusters were mixed according to severity, 41 according to type, and 74 according to strain. The total number of mixed clusters is not a sum of those classifications since they may overlap. In this case, the total number of mixed clusters is 75. In the bottom part, we can see that there are 689 samples in mixed clusters according to severity, 151 010 according to type, and 17 232 according to strain. The total number of samples located in mixed clusters is 17 235.

We can notice that there is a huge difference in a total number of created clusters. YARA is heavily used for clustering PE samples, and there is a number of rules for known strains. These rules can be based on data not even available in Clusty, e.g. specific strings. It often creates clusters having no common attributes at all. A digital signature is another type of method grouping files that do not even have to be similar. In this case, similarity does not matter since they all have verified digital signature by a trusted company. A rule-based on corruption is another example, but this one was not excluded since all corrupted samples should be filtered out.

The next thing we can see is that clustering with all methods has better results, in a matter of the percentage of mixed clusters than the clustering with YARA and digital signature. It is caused by the difference in the total number of clusters. Looking at real numbers, we can see that clustering without YARA has more mixed clusters in total. But, considering the percentage of mixed samples, this does not apply. The total number of mixed samples is lesser when clustering without YARA. It is probably caused by not

Mixed (5 310)	total clusters	severity	type	strain
TAG	75 (1.41 %)	17 (0.32 %)	41 (0.77 %)	74 (1.39 %)
TAG (skip None)	56 (1.05 %)	17 (0.32 %)	28 (0.53 %)	40 (0.75 %)
Mixed	total samples	severity	type	strain
TAG	17 235 (17.23 %)	689 (0.69 %)	15 101 (15.1 %)	17 232 (17.23 %)
TAG (skip None)	17 095 (17.09 %)	689 (0.69 %)	14 853 (14.85 %)	16 876 (16.88 %)

Table 7.2: Result of clustering with all methods in Clusty. The upper table shows a number and percentage of clusters mixed by a specific classification, e.g. severity. The bottom table shows a number and percentage of samples placed in clusters mixed by a specific classification. TAG is the source of classifications. Skip None means that missing values were not considered.

Mixed (18 895)	total clusters	by severity	by type	by strain
TAG	134 (0.71 %)	36 (0.19 %)	93 (0.49 %)	131 (0.69 %)
TAG (skip None)	111 (0.59 %)	36 (0.19 %)	71 (0.38 %)	84 (0.44 %)
Mixed (100 000)	total samples	by severity	by type	by strain
TAG	7 943 (7.94 %)	1 296 (1.3 %)	5 784 (5.78 %)	7 916 (7.92 %)
TAG (skip None)	7 769 (7.77 %)	1 296 (1.3 %)	5 490 (5.49 %)	7 439 (7.44 %)

Table 7.3: Result of clustering with all methods in Clusty except *YARA* and *digital signature*. The upper table shows a number and percentage of clusters mixed by a specific classification, e.g. severity. The bottom table shows a number and percentage of samples placed in clusters mixed by a specific classification. TAG is the source of classifications. Skip None means that missing values were not considered.

detecting all aliases. Using the YARA rule, there are fewer clusters which increases the probability of locating samples in a mixed cluster. On the contrary, clustering without YARA rules contains much more clusters which decreases the chance of locating the sample in a mixed cluster. A mixed cluster is considered any cluster where at least one sample's classification is different from the others.

7.4 Gradual Selection

This method requires a definition of three parameters: E, R, and T. Their meaning is described in Section 5.3.3. Besides that, it is also needed to try specific subsets of attributes. For example, using dynamic attributes only can lead to much better results than using static attributes only and vice-versa. Selecting the best subset of attributes and specifying attributes values is divided into several steps. The first step is about selecting the best subset of attributes together with specifying attribute E. The reason behind this is that each subset can behave differently with the same value of E. For example, if the first subset would consist of five attributes and the second consist of ten attributes, a parameter with a value of more than five is not even acceptable for the first subset. Also, in order to decrease the influence of the rest of the attributes, they were always set to the same value as parameter E. It cancels the meaning of T and decreases the meaning of R. This step is further described in Section 7.4.1. The second step is about determining a value of attribute T and investigating its acquisition for clustering. It is described in Section 7.4.2.

Each step uses the best result from a previous step. The last step is about finding the best value of attribute R. It is described in Section 7.4.3. To better understand the table and used colours, see Section 7.2.

7.4.1 Selecting of Attributes and Parameter E

Table of aggregated clustering results of all experiments can be seen in Table 7.4. The values tried for each group of attributes are based on the data in Figure 7.4. At first, all attributes used for clustering in Clusty were used with parameter set to 3, 4, and 5. The results can be seen in the rows `clusty_3_3_3`, etc. When using $E = 3$, the number of samples in mixed clusters was greater than using Clusty (red cells). Increasing the parameter value improved the classification, but for the cost of fewer samples being clustered.

In the next results `all_5_5_5` and so on, all attributes were used. There are results with worse classification or a greater number of clusters than Clusty without YARA. The next part of the results implements an idea of putting additional logic and splitting attributes into two groups - dynamic and static. This is a logic that can be seen even in the current Clusty. The group `cls_dyn_1_1_1_sta_2_2_2`, ... shows the results for splitting attributes used for clustering in Clusty, and the group `all_dyn_2_2_2_sta_2_2_2`, ... shows the results for all attributes. The result `all_dyn_2_2_2_sta_2_2_2` is the first clustering result yet which was able to cluster all samples. Note that even the number is not equal to 100 000, it is the greatest reachable value due to 502 samples being corrupted (this error turned out later and does not impact the results except for the total number of clustered samples being different to Clusty results).

In order to be able to cluster all samples, a combination of several groups seemed to be necessary. The next group of results tries to combine using all attributes alongside attributes used for clustering in Clusty. For example, group `g1_all_5_5_5_cls_2_2_2` was selected because of the most clustered samples when used alone. The best result from this group is `g3_cls_5_5_5_cls_2_2_2`, but it is still not able to process all samples. The last result in the last group `cls_all_5_5_5_cls_2_2_2_all_2_2_2` is the previous result with additional attributes as a third rule. This result proceeded to the next step.

7.4.2 Parameter T

This step of the experiment tries to determine the importance of the parameter T and its most suitable value. The table in Appendix D.1 shows how the clustering result has changed when modifying parameter T while the other remained constant. We can see that the differences between the results are very negligible. The conclusion is that this parameter can be omitted as it does not improve clustering results in a significant way. It is reflected further by keeping parameter T equal to parameter E.

7.4.3 Parameter R

The last step is to set up parameter R, which is responsible for creating relationships between clusters. The results can be seen in Table 7.5. It does not contain *clustered* column anymore because it was the maximal value in all cases. But it contains several columns in addition. The last five columns show the number of clusters at the specific level. The lesser top-level clusters, the better. The cells coloured in red have worse values than Clusty, and the yellow ones have values worse than the result from the previous step. The same result is present here under the name `rel_5_2_2`. The results in the first group

have constant values of the parameter. It was necessary to set the maximum depth as well in order to avoid recursion error. The limit was set to 11 as a placeholder. The second group solved the problem of max depth by decreasing of parameter's value based on the cluster's depth level. The deeper, the lesser value. The second value postpones this behaviour by one level, i.e. the value at level 0 is the same as the value at level 1. The last two groups are similar to the previous group with the difference in decreasing of E parameter as well. It leads to a lower number of clusters in most cases. The reason is that there was a trend of hitting the limit of depth and stacking many clusters on the lower levels. They often differ in just a single attribute. By decreasing parameter E as well as parameter R, the clusters which were previously similar siblings were now merged into a single cluster. The depth was limited by decreasing parameter R to value two instead of implicit one due to always follow multiple criteria.

The next thing we can see in the last two groups is the difference in the number of red cells. The group without postponing of the decreasing condition has a tendency to create much more clusters than the group postponing decreasing. It is caused by two-level "sifting", where the first level find the most similar cluster among much different top-level clusters, and the second level further specifies the most similar cluster among similar clusters. The chance of finding two equally applicable clusters is postponed to lower levels.

The result `de1_rel_2_2_2` was considered the best. It uses the technique of decreasing both parameter E and R, which lead to fewer clusters and an automatic limit of depth.

Grouping similar clusters together is a very useful side-effect of this method. Appendix D.1 show a part of the cluster tree obtained by script `get_cluster_tree.py`. We can see that it tend to group clusters with the same strain. It can be probably used to detect aliases of malware strains or at least their relations. Strains *PCSpeedCat* and *SpeedCat* seem to be related at least.

Name	o	clustered [s]	clusters [c]	tiny [%c]	tiny [%s]	huge clusters [%s]	severity [%s]	strain [%s]	strain (s N) [%s]
Clusty (all)	X	100000	5310	92.67	5.87	79.19	0.69	17.23	16.88
Clusty (no Yara)	X	100000	18895	97.83	19.68	64.17	1.3	7.92	7.44
clusty_3_3_3	3	81690	15481	96.7	19.97	54.95	0.56	22.19	17.28
clusty_4_4_4	2	69555	12153	95.63	18.51	48.43	0.37	15.31	10.81
clusty_5_5_5	1	58940	7628	92.57	14.17	51.03	0.24	16.55	11.87
all_5_5_5	3	81856	7068	89.54	9.93	57.1	0.5	24.97	18.46
all_6_6_6	1	76922	9965	91.82	14.93	46.37	0.4	16.62	13.42
all_7_7_7	2	73664	14636	94.7	22.36	42.21	0.29	13.94	11.6
all_8_8_8	4	63389	18987	95.73	33.99	33.99	0.18	8.58	6.32
cls_dyn_1_1_1_sta_2_2_2	1	98806	16974	97.34	17.92	62.29	1	16.65	11.52
cls_dyn_1_1_1_sta_3_3_3	2	82082	18171	97.8	23.1	57.44	0.67	21.26	13.64
cls_dyn_2_2_2_sta_2_2_2	3	98414	20943	97.83	22.09	59.56	0.64	11.37	9.13
cls_dyn_2_2_2_sta_3_3_3	4	80679	22111	98.25	28.49	53.92	0.11	13.12	11.45
all_dyn_2_2_2_sta_2_2_2	4	99498	9042	93.12	10.19	64.05	0.86	26.01	15.43
all_dyn_2_2_2_sta_3_3_3	3	99479	9400	93.12	10.57	66.1	0.39	23.69	14.2
all_dyn_2_2_2_sta_4_4_4	1	98472	13002	94.52	14.75	60.64	0.37	15.89	11.27
all_dyn_2_2_2_sta_5_5_5	2	81755	11193	94.18	15.09	54.78	0.45	18.22	13.63
g1_all_5_5_5_cls_2_2_2	3	98864	7253	89.54	8.53	61.99	0.42	20.7	16.26
g2_all_5_5_5_all_2_2_2	6	99498	7268	89.34	8.49	60.89	0.43	22.37	19.29
g3_cls_5_5_5_cls_2_2_2	1	98806	11272	93.43	12.24	56.39	0.18	13.68	10.83
g4_cls_5_5_5_all_2_2_2	5	99498	8327	91.85	9.27	64.9	0.38	23.27	20.1
g4_cls_4_4_4_all_2_2_2	2	99498	12589	95.08	13.44	57.78	0.38	17.77	14.01
g4_cls_3_3_3_all_2_2_2	4	99498	15697	96.4	16.62	58.58	0.53	21.36	17
cls_5_5_5_cls_2_2_2_all_2_2_2	√	99498	11360	93.17	12.24	55.11	0.25	14.23	11.42

Table 7.4: Clustering results for attributes selection (see Section 7.2 to understand)

Name	X	clusters [c]	tiny [%c]	tiny [%s]	huge clusters [%s]	severity [%s]	strain [%s]	strain (s N) [%s]	L0 %	L1 %	L.. %	L11 %	L0 cnt
Clusty (all)	X	5310	92.67	5.87	79.19	0.69	17.23	16.88	XXXXX	XXXX	XXXXX	XXXXX	XXXXX
Clusty (no Yara)	X	18895	97.83	19.68	64.17	1.3	7.92	7.44	XXXXX	XXXX	XXXXX	XXXXX	XXXXX
rel_6_2_2	6	11316	93.29	12.2	57.87	0.25	14.21	11.4	99.9	0.1	0	0	11305
rel_5_2_2	5	11360	93.17	12.24	55.11	0.25	14.23	11.42	99.09	0.89	0.02	0	11257
rel_4_2_2	3	12717	94.51	13.56	56.81	0.24	13.52	10.68	64.18	5.62	8.53	21.67	8162
rel_3_2_2	1	12724	94.46	13.52	56.07	0.24	13.92	11.23	55.15	5.51	10.42	28.92	7017
rel_2_2_2	2	13753	95.22	14.45	56.77	0.25	13.46	11.23	45.47	5.11	11.28	38.14	6254
rel_1_2_2	4	14527	95.79	15.14	57.93	0.22	13.26	11.36	36.3	4.32	12.14	47.24	5273
d0_rel_5_2_2	2	11332	93.46	12.23	59.04	0.25	14.06	11.42	99.01	0.92	0.07	0	11220
d0_rel_4_2_2	1	13697	95.34	14.45	57.7	0.24	13.81	11.02	59.44	4.8	35.75	0	8142
d0_rel_3_2_2	3	20531	95.74	21.16	59.59	0.28	11.38	9.97	34.14	3.07	62.79	0	7009
d0_rel_2_2_2	4	30676	98.86	31.12	53.3	0.15	8.44	7.35	20.36	79.64	0	0	6245
d1_rel_5_2_2	3	11329	93.46	12.25	58.83	0.25	13.98	11.12	99.01	0.97	0.02	0	11217
d1_rel_4_2_2	1	13275	95.05	14.06	57.76	0.24	13.88	11.08	61.32	5.33	33.35	0	8140
d1_rel_3_2_2	2	14780	95.86	15.53	55.62	0.23	13.7	11.1	47.48	4.76	47.76	0	7018
d1_rel_2_2_2	4	18314	97.22	18.9	57.16	0.21	12.1	10.7	34.09	3.89	62.02	0	6244
de0_rel_5_2_2	2	11346	93.49	12.26	59.13	0.25	14.26	11.39	98.88	1.07	0.05	0	11219
de0_rel_4_2_2	1	9259	92.27	10.02	58.99	0.31	14.12	11.43	87.91	7.07	5.01	0	8140
de0_rel_3_2_2	3	20220	97.49	20.82	57.16	0.21	11.48	10.3	34.63	3.06	62.31	0	7002
de0_rel_2_2_2	4	28547	98.78	28.98	55.49	0.15	8.41	7.29	21.88	78.12	0	0	6246
de1_rel_5_2_2	3	11325	93.48	12.23	59.13	0.25	14.27	11.61	99.06	0.93	0.02	0	11218
de1_rel_4_2_2	1	9132	92.03	9.86	60	0.31	14.36	11.7	89.1	7.82	3.08	0	8137
de1_rel_3_2_2	2	8287	91.55	9	57.93	0.32	14.19	11.78	84.64	8.37	6.99	0	7014
de1_rel_2_2_2	4	15787	96.52	16.34	57.29	0.28	12.91	10.82	39.58	4.49	55.93	0	6249

Table 7.5: Clustering results for R parameter selection (see Section 7.2 to understand)

7.4.4 Speed Test

The speed of clustering was tested using script `speed_test.py`. It runs clustering five times for the following number of samples: 10k, 20k, 40k, 60k, 80k, 100k, 150k, 200k, 250k, 300k, 350k, 400k, 450k, and 500k. Each run generates a random set of samples from the whole dataset of more than 500 k samples. Figure 7.5 shows a relation between the number of samples and the duration of clustering. Figure 7.6 shows a relation between the number of samples and the number of clusters. We can see that that those numbers are similar. It is caused by decreasing parameters E and R.

To see an asset of using bitmask reducing the number of fetched clusters, test run with and without mask were performed. It was measured on a single instance using 64 workers in ad-hoc mode. The average duration without using a mask was 42.5 minutes, while the duration with using a mask was less than 22 minutes.

7.4.5 Evaluation

It is necessary to use several rules to be able to process all samples. The clustering results look good compared to Clusty as shown in Table 7.6. Compared to clustering with YARA rules, the total number of clusters is almost two times greater. Similarly with the percentage of tiny clusters. It is not that bad since it is almost two times better than the results without YARA. The number of huge clusters is lesser than both clustering results. It is caused probably by splitting huge clusters into several smaller ones because of using multi-criteria. The number of samples in clusters mixed by severity, e.g. mix of clean and malware, is better in both cases. The number of samples in clusters mixed by strain is between Clusty's clustering results. Since it is better than the results using YARA, we can consider it a good result. As a side-effect of using cluster hierarchy, it can reveal relations between clusters and/or detect strain aliases.

The clustering speed is pretty slow. Even the bitmask helped a lot to reduce clustering duration, it would be good to reduce the duration further.

Name	clusters [c]	tiny [%c]	tiny [%s]	huge clusters [%s]	severity [%s]	strain [%s]	strain (s N) [%s]
Clusty (all)	5310	92.67	5.87	79.19	0.69	17.23	16.88
Clusty (no Yara)	18895	97.83	19.68	64.17	1.3	7.92	7.44
Rusty	9132	92.03	9.86	60	0.31	14.36	11.7

Table 7.6: Clustering results for Gradual selection (see Section 7.2 to understand)

7.5 Ssdeep Selection

This method relies on the clustering results based on the ssdeep hash similarity. Almost one million samples were clustered by Clusty using ssdeep hash similarity method. Then, several groups of various subsets of attributes were obtained using the auxiliary script `clusty_clusters_common_attributes.py`. These groups were then written as clustering rules using `KeyFeatues` method. It allows to cluster samples based on the equality of all attributes in the group. The generation of these groups is described in Section 7.5.1. The clustering based on those methods is described in Section 7.5.2.

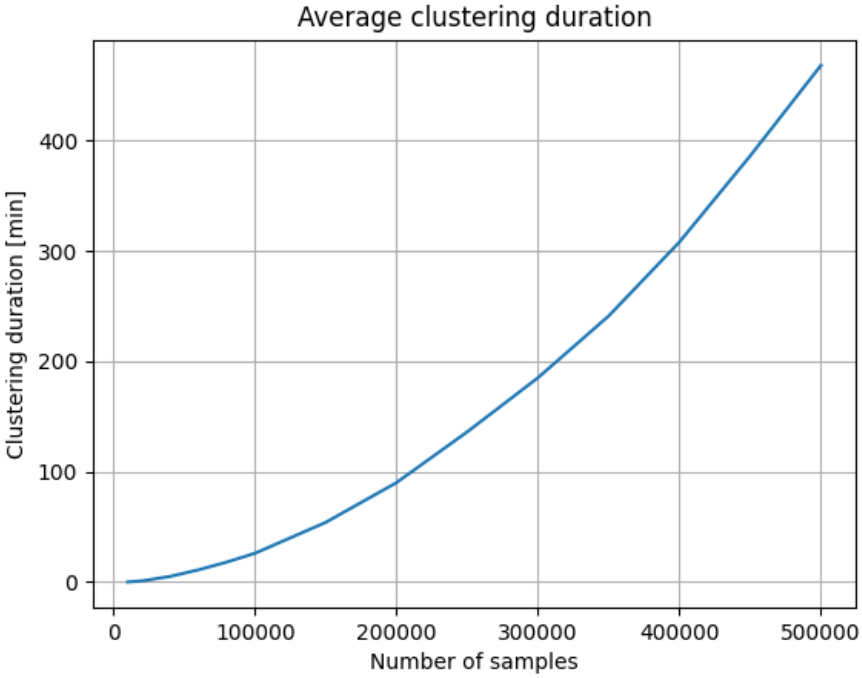


Figure 7.5: Relation between the number of samples and the duration of clustering - Gradual selection

Figure 7.6: Relation between the number of samples and the number of clusters - Gradual selection. It shows the total number of clusters as well as number of clusters on the top level.

7.5.1 Generating of Groups

The process of generating groups can be seen in Code 7.1. At first, all regular clusters (>5) are fetched from the database. Then, a group of common attributes is extracted from each cluster. Groups less than a specific value are removed. Then, all available subgroups are generated from the remaining groups and added to the groups. All these groups are then sorted by the total number of clusters matched by the group and by the total number of samples in clusters matched by the group. For example, let's have two clusters. The first one has attributes A, B, and C in the common attributes and contains three samples. The second one has attributes A, B, and D in the common attributes and contains one sample. The subgroup AB would have assigned four as a number of samples, because of the sum of samples of the clusters it was created from. After getting two sorted groups, they are passed for evaluation. They are scored according to their position in both sorted groups. The score is a sum of both positions. For example, if a group is on the top of the list sorted by clusters and on the second position in the list sorted by samples, the resulted score would be $0 + 1 = 1$. After getting the score, they are sorted again according to the score. Then, all groups are being iterated. Inside a loop, there is another loop iterating over all clusters. If the group matches a cluster not matched by any of the other group yet, it is added to the final list of groups.

```
def generate_groups():
    clusters = get_regular_clusters_from_database()
    groups, cluster_ids = get_sorted_groups_of_common_attributes(clusters)
    groups = remove_groups_of_shorter_length(groups, MIN_GROUP_LEN)
    add_subgroups_greater_than(groups, MIN_GROUP_LEN)
    sorted_cluster_groups = sort_groups_by(groups, key='clusters', NUM_OF_GROUPS_TO_USE)
    sorted_samples_groups = sort_groups_by(groups, key='samples', NUM_OF_GROUPS_TO_USE)
    score_groups = evaluate_groups(sorted_cluster_groups, sorted_samples_groups)
    sorted_score_groups = sort_groups_by(score_groups, key='score', NUM_OF_GROUPS_TO_USE)
    top_score_groups = get_best_groups(sorted_score_groups, cluster_ids)
```

Code 7.1: Pseudocode for generating groups of attributes based on the results from clustering by ssdeep similarity hash

7.5.2 Clustering

Table 7.7 shows all the clustering results using groups generated from various subsets of attributes. The first group generated groups on the basis of all attributes, the second on the basis of all dynamic attributes, the third on the basis of all static attributes, and the last on the basis of all attributes used for clustering in Clusty.

We can see that all clustering results contain at least one red cell. Also, all clustering results except the last one have the number of clusters greater than Clusty. It is even worse because the numbers of samples in clusters mixed by strain are greater as well. The best-looking result is the last one, but it is not able to cluster all samples, and the number of samples in clusters mixed by severity is worse than both Clusty's clustering results.

7.5.3 Speed Test

The speed of clustering was tested the same way as described in 7.4.4. Figure 7.7 shows a relation between the number of samples and the duration of clustering. Figure 7.8 shows a relation between the number of samples and the number of clusters.

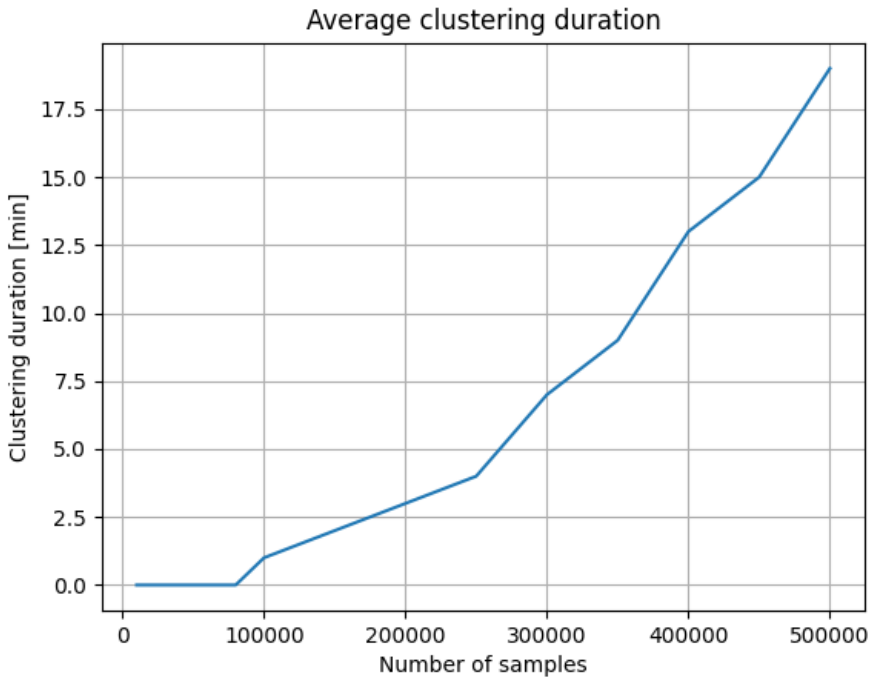


Figure 7.7: Relation between the number of samples and the duration of clustering - Ssdeep selection

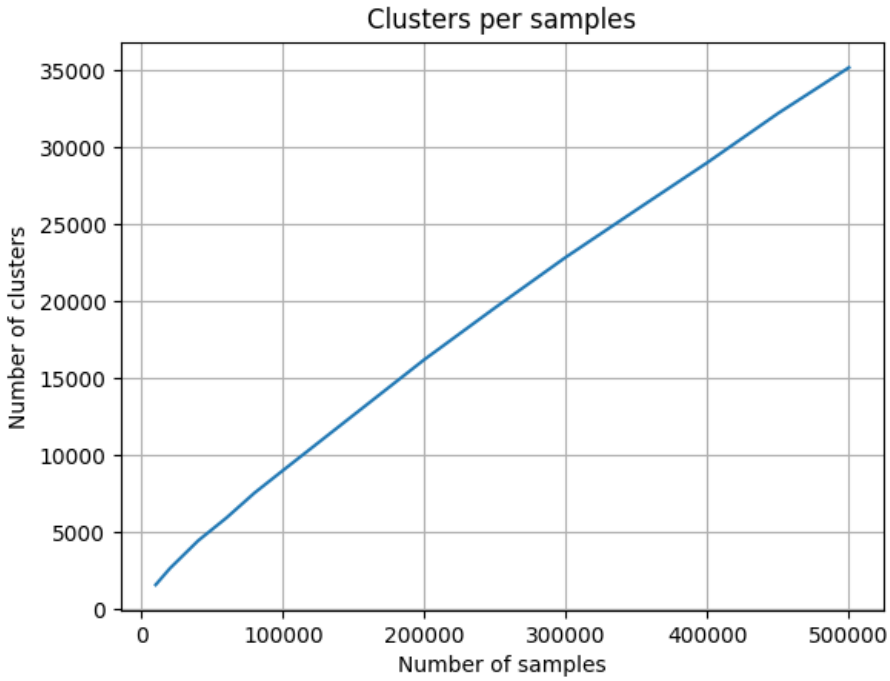


Figure 7.8: Relation between the number of samples and the number of clusters - Ssdeep selection

Name	clusters [c]	clustered [s]	tiny [c%]	tiny [s%]	huge clusters [s%]	severity [s%]	strain [s%]	strain (s N) [s%]
Clusty (all)	5310	100000	92.67	5.87	79.19	0.69	17.23	16.88
Clusty (no Yara)	18895	100000	97.83	19.68	64.17	1.3	7.92	7.44
all_min5	23846	78831	96.01	33.03	35.8	0.28	21.09	13.03
all_min4	30762	98522	97.37	33.44	46.78	0.26	17.64	11.64
all_min3	26309	99965	97.19	28.02	50.72	0.7	21.74	14.07
all_min2	25681	100000	96.95	28.16	49.8	0.13	23.89	15.69
dynamic_min3	33513	47945	99.43	70.91	12.56	0	4.96	3.18
dynamic_min2	34152	52739	99.26	66.11	13.86	0.29	5.39	3.08
static_min5	24360	74152	96.91	35.86	41.07	0.3	22.45	13.89
static_min4	27669	94363	97.29	31.35	48.85	0.27	18.41	11.62
static_min3	26212	99945	97.27	27.9	51.22	0.7	22.23	14.56
static_min2	25677	100000	97.34	27.31	54.03	0.86	28.52	25.73
clusty_min5	19144	58270	97.38	34.33	42.14	0.13	10.15	5.62
clusty_min4	23164	69350	98.36	34.13	44	0.3	10.5	6.83
clusty_min3	22713	77487	97.53	30.22	47.09	0.3	16.46	8.25
clusty_min2	12119	99229	94.89	13.99	61.98	0.27	22.22	15.76

Table 7.7: Clustering results for Ssdeep selection (see Section 7.2 to understand)

Name	clusters [c]	clustered [s]	tiny [c%]	tiny [s%]	huge clusters [s%]	severity [s%]	strain [s%]	strain (s N) [s%]
Clusty (all)	5310	100000	92.67	5.87	79.19	0.69	17.23	16.88
Clusty (no Yara)	18895	100000	97.83	19.68	64.17	1.3	7.92	7.44
ssdeep	12119	99229	94.89	13.99	61.98	0.27	22.22	15.76

Table 7.8: Clustering results for Ssdeep selection (see Section 7.2 to understand)

7.5.4 Evaluation

None of the subsets of attributes created a suitable clustering result. They were worse than Clusty in all cases. Changing the algorithm of script generating groups of attributes could improve clustering results so the method would be usable.

On the other hand, the clustering speed is very fast. It is several times faster compared to Gradual selection method.

7.6 Experimental Selection

This method can be considered as an extension of the method used in the current Clusty. It allows using multi-criteria for clustering instead of just a single one. It even allows defining fixed attributes (e.g. it has an attribute with the same value) and a group of attributes where at least some have to match (e.g. one of three attributes in a group has to match). The drawback of the method is the same as in Clusty. The groups and rules have to be selected manually based on experiments. In Clusty, there are many experiments behind finding useful attributes and their position in the final list of useful attributes. What is more, it is not a one-time activity. The experiments have to be made for each new attribute.

The selected groups and the description of clustering results is described in Section 7.6.1. The speed test results and method evaluation results can be found in sections 7.6.2 and 7.6.3.

7.6.1 Clustering

There is a great number of possible groups of attributes. Therefore, the knowledge from the Clusty was taken as a base. The rules used in experiments try to re-implement Clusty's list of attributes by extending a single attribute with others. The order of attributes remains the same. Each attribute has to contain and is clustered by one more attribute with a lower priority. The results can be seen in Table 7.9.

7.6.2 Speed Test

The speed of clustering was tested the same way as described in 7.4.4. Figure 7.9 shows a relation between the number of samples and the duration of clustering. Figure 7.10 shows a relation between the number of samples and the number of clusters.

7.6.3 Evaluation

The results can be seen in Table 7.9. The results are worse than the results by Clusty in all aspects. There is a greater number of clusters, more small clusters, more samples are located in small clusters, and there are fewer samples in huge clusters. All the classifications results are better than both Clusty's clustering results, but it can be attributed to the significantly great number of clusters. However, the result can be improved in many ways by finding better groups. The greater number of clusters is not unexpected. Since the attributes and

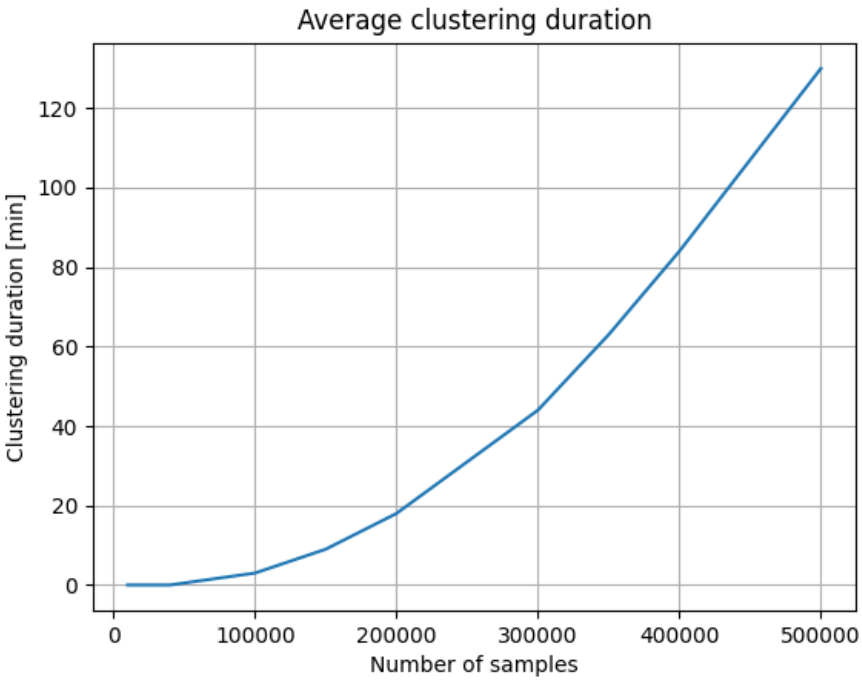


Figure 7.9: Relation between the number of samples and the duration of clustering - Experimental selection

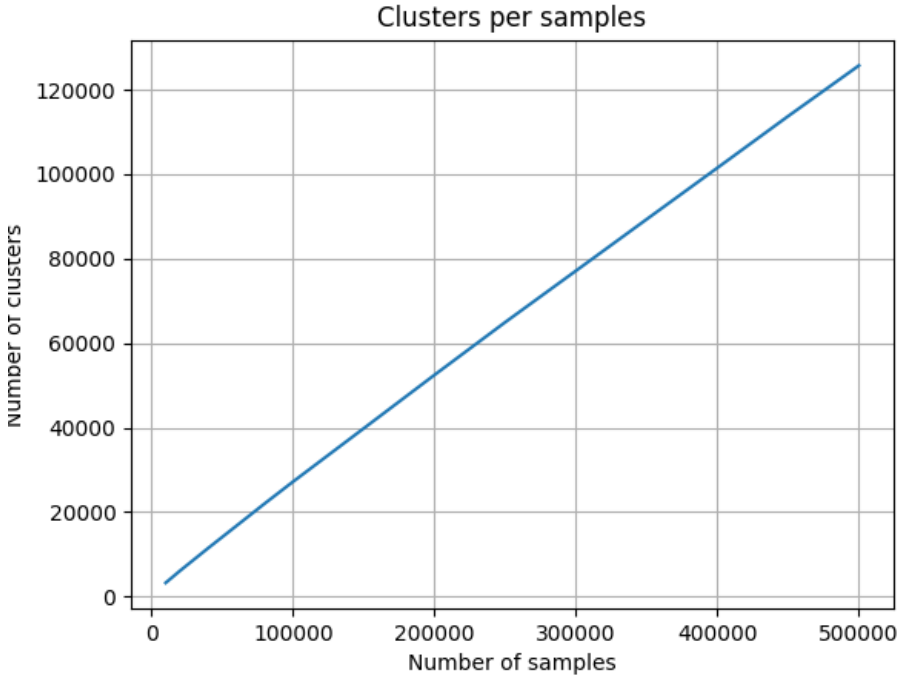


Figure 7.10: Relation between the number of samples and the number of clusters - Experimental selection

Name	clusters [c]	tiny [c]	tiny [s]	huge clusters [s]	severity [s]	strain [s]	strain (s N) [s]
Clusty (all)	5310	92.67 %	5.87 %	79.19 %	0.69 %	17.23 %	16.88 %
Clusty (no Yara)	18895	97.83 %	19.68 %	64.17 %	1.3 %	7.92 %	7.44 %
experimental	27105	98.11 %	28.09 %	49 %	0.04 %	4.28 %	3.1 %

Table 7.9: Clustering results for Experimental selection (see Section 7.2 to understand)

the order are similar to Clusty's, putting more attributes into condition should split some clusters Clusty would create, i.e. there would be more clusters than Clusty without YARA.

Chapter 8

Testing

The implemented tool is tested via a suite of unit, integration, and end-to-end tests. Besides the description of tests for the implementation, the chapter also contains tests for meeting the scalability and high availability noted in the requirements.

8.1 Rust Library

According to [21], "Rust's type system and runtime guarantee the absence of data races, buffer overflows, stack overflows, and accesses to uninitialised or deallocated memory". It provides a number of safety feature that similar languages do not have. The suite of unit and integration test is used for testing expected behaviour. The tests are organised as described in Rust documentation [18]. Total code coverage is approx. 97% according to `tarpaulin`[1].

Besides unit and integration tests, Rust also supports executing documentation examples as tests. Rusty contains just a single test like that. All tests can be run using `cargo test` command. An example of the tests output can be seen in Appendix E.1.

The library uses annotation causing to warn about missing documentation string during compilation. Based on that, there is a documentation string for each function, structure, etc. The documentation for the library and all used dependencies can be built using `cargo doc --document-private-items` and accessed in `target/doc/rustylib/index.html`.

8.1.1 Unit Tests

Unit tests are small and focused on a specific unit of a single module. A Rust convention is to create a `tests` module with unit tests for each file. It allows testing even private functions. Since the tests are in the source directory, the test module has to be annotated with `#[cfg(test)]` in order to be not included in the library. Each test inside the module is annotated with `#[test]`. Unannotated functions are not considered to be a test.

Unit tests (692 in total) in Rusty cover all structures (e.g., samples, `Cluster`, `Rule`), traits, implemented clustering rules, and utilities. They test creation of instances, serialisation and deserialisation, and all methods for structures.

[1] https://crates.io/crates/cargo-tarpaulin

8.1.2 Integration Tests

Integration tests in Rust are made to be isolated from the tested library. They should test only its public API as any other crate would do. Its purpose is to test how the code units work together. The tests are not stored in `src` directory instead of in the `tests` directory in the project's root, next to `src` directory. Each of the files in the directory is considered an individual crate. All auxiliary functions for integration tests are stored in `tests/common/mod.rs`.

The purpose of splitting Rusty into binary and library is to allow testing its functionality. Rust does not allow to test binaries. It is instead suggested to make it a library which will be then called from a binary. A library can be tested using integration tests. The Rusty binary contains only a necessary logic of console argument parsing. Everything else is a part of the Rusty library. Even it is not needed to expose functions other than the one used in the binary from the binary point of view, the library exposes a lot of code to be adequately tested in the integration tests.

The integration tests (23 in total) cover all work with the database, from basics like creating collections or indexes to moving a sample from one cluster to another and removing the first if empty.

8.1.3 End-to-End Tests

End-to-end tests (8 in total) are made as Rust integration tests. They are stored in `tests/end2end.rs` test file. They test the library as a whole by clustering a predefined set of samples using various library configurations. It tests the whole process, including spawning of workers and communicating with the RabbitMQ server. The number of clusters or a list of clusters sizes are some of the asserted statements.

8.2 Scalability and High Availability

High availability is implemented by allowing to run a number of instances in parallel. Multiple instances also allow horizontal scalability. They were tested by clustering a set of 100 000 samples multiple times, each time using a different number of instances. Table 8.1 shows the configuration and results for each run. The experiments were performed in two rounds. The first round used 16 workers per instance, and the second round used 32 workers per instance. The results can also be seen in Figure 8.1.

We can see that Rusty is able to run in multiple instances which satisfies the high availability. Further, we can see that the duration between round matches in the case of the total number of workers matches. Also, we can see that the duration radically decreased at the beginning and stabilised around 20 minutes. The results show that that vertical scalability by running multiple instances is possible.

Instances	Workers/Instances	Total Workers	Duration [min]
1st round			
1	32	32	46
2	32	64	22
3	32	96	19
2nd round			
1	16	16	95
2	16	32	47
3	16	48	29
4	16	64	22
5	16	80	20
6	16	96	19

Table 8.1: Results of testing scalability and HA by running multiple instances in parallel. The duration between round matches in the case of the total number of workers matches.

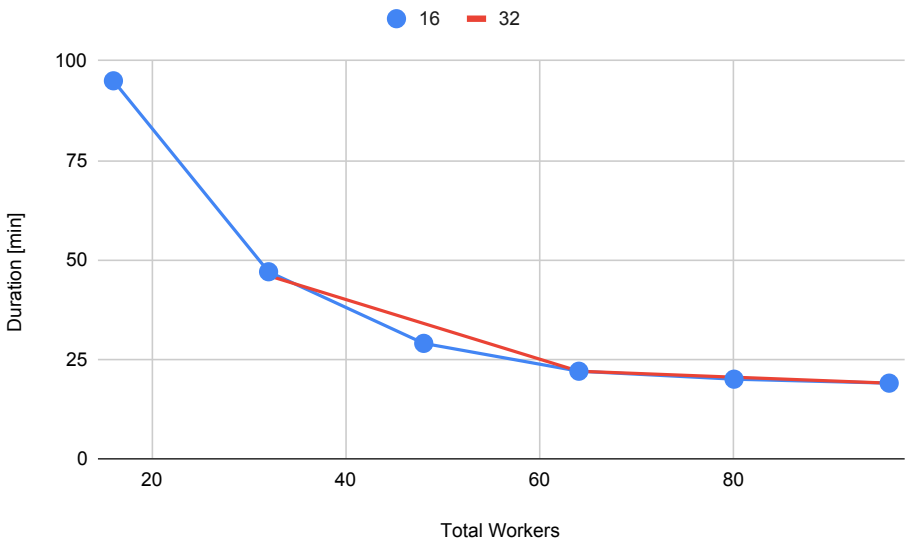


Figure 8.1: Results of testing scalability and HA by running multiple instances in parallel

Chapter 9

Evaluation

This chapter evaluates achievements and compares Rusty to Clusty. The requirements are split into design-related and clustering-related features. The chapter evaluates all considered clustering methods, including the chosen method.

9.1 Design

The design-related features can be further divided into the features Clusty supports and the features Clusty does no support.

9.1.1 Preserved Features

All design-related features of Clusty that had to be preserved are listed in this section.

Online and ad-hoc modes

Rusty can run in both modes Clusty supports. The first is continuous mode, where it accepts a stream of hashes of analysed samples and cluster them as they arrive. The next mode is ad-hoc mode, which operates over a file with hashes. It sends them for clustering by publishing hashes to the RabbitMQ server. When connected to the analysis part, which is not part of this work and does not currently exist, ad-hoc mode would not need read hashes from the files because they would be sent for clustering by analysis part.

Reclustering mode

Re-clustering mode allows to re-cluster samples and clusters. Re-clustering of samples means that already clustered sample can be sent for clustering again. It will be moved into another cluster if any better cluster is found. Nothing will change otherwise. Re-clustering of clusters mean sending all samples in the cluster for reclustering, and removing the cluster if none of the samples have left.

Blacklisting of samples and clusters

The proposed clustering methods and the system of rules also support blacklisting. A sample can be blacklisted by recording its hash in the blacklist collection. A cluster can be blacklisted as well. Based on the possibility to place a sample in the blacklisted cluster can be determined whether a cluster is blacklisted or not. If the sample trying to create a new

cluster can be placed into the blacklisted cluster using the same rule, the new cluster is considered equivalent of the blacklisted cluster and therefore will not be created.

Universality

Rusty currently supports clustering of PE samples only, but it is made to be extensible. It can support all categories Clusty supports. Adding a new category means implementing new sample and creating set of rules for the category. Each category has its own set of attributes, i.e. it can require another experiments to select the best rules or groups of attributes.

9.1.2 Design Shortcomings of Clusty

All design-related shortcoming of Clusty that were expected to be overcame are listed in this section.

Scalability

Clusty can be scaled vertically by selecting the number of workers. But it cannot be scaled horizontally since it does not support running multiple instances.

Rusty is made to be both vertically and horizontally scalable by supporting specifying number of workers and running multiple instances at once. Also, both ArangoDB and RabbitMQ supports scalability. RabbitMQ is already used by Clusty.

High availability

Because Clusty cannot run in multiple instances of clustering within the same clustering result, the program failure can lead to outage.

As mentioned in the previous section, Rusty is able to run in multiple instances. It means that if running in multiple instances, failure of a single instance do not break the whole clustering process. It will only lead to the lower throughput of clustering because of the lower number of workers available.

9.2 Multi-criteria Clustering

Rusty cluster samples based on so-called rules. They are similar to Clusty methods. The rules allow defining single-criteria methods, just like Clusty does, as well as using multi-criteria methods. The idea behind the rules is to enable using YARA rules, digital signature, and other single-criteria methods. They are special cases where the single attribute has significantly greater weight than the others. For example, a sample with a YARA rule can be clustered based on the rule regardless the rest of the attributes.

The rule is a generic structure providing an interface to define a value used for clustering, a condition used to verify the rule usability, and a condition to confirm whether a sample can fit into a cluster. The rule also needs to be assigned to one of the supported clustering methods.

Experimental selection method has many possible configurations, i.e. there are many possible groups of attributes and other constraints. The tested configuration, extending Clusty's attribute hierarchy, did not result in better quality clusters when considering all aspects. More experiments can be done in the future to find suitable combinations of

attributes resulting in better quality clusters. Nevertheless, manual selection of groups based on experiments is very time-consuming. This limitation would show up again if adding new attributes or even more when adding a new category.

A solution to avoid manual selection of groups is to automate the selection. One such automation is proposed in this work. Ssdeep selection uses LSH ssdeep to find common attributes of similar samples. The files are hashed using ssdeep hash, which is then used to cluster samples with ssdeep hash similarity above a threshold. Each cluster then computes common attributes among all its samples. These groups can serve as a base knowledge about what similar samples tend to have in common. The algorithm proposed in this work did not do well. There were much more clusters and the number of samples in the mixed clusters was worse compared to Clusty. Besides that, The groups were not able to cluster all sample in most cases. The notable advantage of having fixed groups is a very good performance. It can cluster hundreds of thousands of samples in minutes.

The last method Gradual selection moves the finding of proper groups even further. It requires two parameters (parameter T was shown to be redundant) and a set of attributes to be defined. The method is then able to find valuable attributes according to already existing clusters. The need to fetch a greater number of clusters leads to a significant deceleration compared to the previous two methods. It is the main drawback comparing to Clusty and the other methods. The bitmask is shown to be useful since it speeds up the clustering by a factor of two, but it still remains to be the slowest one. Though, there is also space for other improvements to speed up clustering. For example, reduction of cluster size can lower amount of transferred data. However, because of good clustering results, this method was chosen to be used for now.

A comparison of clustering with Rusty (using Gradual selection) and Clusty can be seen in Table 7.6. Rusty creates two times lesser clusters than Clusty while keeping a better percentage of samples in mixed clusters than Clusty with YARA. Note that clustering with YARA incorporates the knowledge of analysts. The higher number of samples in mixed clusters can be caused by malware aliases and either missing or wrong classifications. The percentage of samples in mixed clusters by severity (e.g. malware, clean) is always better. This metric should be more stable compared to strains since there are not many severities. Severity can be considered more critical than strain because severity reveals whether a sample is a malware or not.

9.2.1 Preserved Features

All clustering-related features of Clusty that had to be preserved are listed in this section. They are evaluated according to the chosen clustering method, i.e. Gradual selection.

Homogeneous clusters

The clusters are not fully homogeneous. But, there is significantly less samples in clusters mixed by severity compared to Clusty. It is more than two times less compared to Clusty using all methods, and more than four time less compared to Clusty without YARA. It does not apply to the number of sample in mixed cluster by type and strain. In those cases, it is worse than Clusty without YARA but better than Clusty using all methods. Severity can be considered as the most important part of classification, because it is used in decision if a sample is malicious or not. Because of that, Rusty can be considered to have more homogeneous clusters compared to Clusty.

Unlimited cluster size

Clusty divides cluster into two categories: tiny and regular. Tiny clusters are clusters with one to four samples. Regular clusters contain at least five samples. Only regular clusters are classified and showed on the web. Rusty does not make such division. It treats all clusters the same way.

Explainability

Each cluster has a fixed part which holds the common attributes. The attributes have to be the same in order to place a samples into cluster. Clusty uses single criterion for clustering, Rusty uses always multiple criteria.

Universality

The clustering method was tested on PE samples only. But it should also be usable for other categories. It requires several experiments to be made in order to select a proper subset of useful attributes and parameters.

Chapter 10

Conclusion

This work aimed to design and implement the clustering part of the next version of Avast's internal clustering tool, Clusty. Clusty is able to identify, analyse, and cluster all incoming samples online.

The next version was named Rusty. It overcomes shortcomings of Clusty while keeping its key features. Rusty can run in continuous mode as well as in ad-hoc mode. It provides scalability and high availability by supporting multiple instances. All external components support these features as well. Rusty offers so-called rules which allow defining a condition determining their usability for the currently clustered sample. Each rule implements one of the supported clustering methods. This approach allows using, for example, YARA rules, digital signatures, and others methods used in Clusty. They are among the most prioritised methods because of the importance of the attributes. Blacklisting samples and clusters is also possible. The implementation is tested via a suite of unit, integration, and end-to-end tests.

None of the state of the art clustering methods have satisfied the desired requirements. Therefore, three new ideas were proposed. They are based on the method in the current version of Clusty and the standard methods. The results showed that the best method according to the quality and the number of clusters is *Gradual selection*. However, it is significantly slower than the other ones. Another method *Experimental selection* did not have good results, but it still has potential since there are many possibilities to find suitable groups and conditions. Because rules allow the coexistence of several methods, rules using *Experimental selection* can be added in the future. The third method *Ssdeep selection* tried generating groups of attributes and to cluster samples based on them, but none of the experiments have shown good results.

The clusters based on *Gradual selection* were better than the Clusty in several aspects. Moreover, the hierarchy they create can be used as additional knowledge revealing relations between clusters. It can serve, for example, for discovering malware aliases or revealing similarities among strains.

Rusty is made to be extensible. It allows defining new rules, categories, and even new clustering methods. There is a possibility to improve performance of Rusty and implemented clustering methods. Rusty can be extended by new clustering methods and categories. Besides that, the remaining parts of Clusty, like identification or analysis, have to be implemented in order to replace Clusty as the main clustering programme.

Bibliography

[1] *RabbitMQ official documentation.* [cit. 2021-04-25]. Available at: https://www.rabbitmq.com/.

[2] *Welcome to YARA's documentation.* [cit. 2021-04-28]. Available at: https://yara.readthedocs.io/en/v4.1.0/.

[3] Al Shamsi, F., Woon, W. L. and Aung, Z. Discovering Similarities in Malware Behaviors by Clustering of API Call Sequences. In: Cheng, L., Leung, A. C. S. and Ozawa, S., ed. *Neural Information Processing.* Cham: Springer International Publishing, 2018, vol. 11304, p. 122–133. DOI: https://doi.org/10.1007/978-3-030-04212-7_11. ISBN 978-3-030-04212-7.

[4] Asquith, M. Extremely scalable storage and clustering of malware metadata. *Journal of Computer Virology and Hacking Techniques.* Springer. 2015, vol. 12, no. 2, p. 49–58. DOI: https://doi.org/10.1007/s11416-015-0241-3.

[5] Bayer, U., Comparetti, P. M., Hlauschek, C., Krügel, C. and Kirda, E. Scalable, Behavior-Based Malware Clustering. In: Citeseer. *NDSS.* 2009, vol. 9, p. 8–11. Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.148.7690&rep=rep1&type=pdf.

[6] Cordeiro de Amorima, R. and Ruiz, C. D. L. Identifying meaningful clusters in malware data. *ArXiv e-prints.* Elsevier. August 2020, p. 1. DOI: https://doi.org/10.1016/j.eswa.2021.114971.

[7] Dabhi1, D. P. and Pate, M. R. Extensive Survey on Hierarchical Clustering Methods in Data Mining. In: *International Research Journal of Engineering and Technology (IRJET).* November 2016, vol. 3, p. 659–665. ISSN 2395-0072. Available at: https://www.irjet.net/archives/V3/i11/IRJET-V3I11115.pdf.

[8] Egele, M., Scholte, T., Kirda, E. and Kruegel, C. A Survey on Automated Dynamic Malware-Analysis Techniques and Tools. *ACM Comput. Surv.* New York, NY, USA: Association for Computing Machinery. March 2008, vol. 44, no. 2. DOI: https://doi.org/10.1145/2089125.2089126. ISSN 0360-0300.

[9] Faridi, H., Srinivasagopalan, S. and Verma, R. Performance Evaluation of Features and Clustering Algorithms for Malware. In: *2018 IEEE International Conference on Data Mining Workshops (ICDMW).* IEEE, 2018, p. 13–22. DOI: https://doi.org/10.1109/ICDMW.2018.00010. ISBN 978-1-5386-9288-2.

[10] Fernandes, D. and Bernardino, J. Graph Databases Comparison: AllegroGraph, ArangoDB, InfiniteGraph, Neo4J, and OrientDB. In: *Proceedings of the 7th*

International Conference on Data Science, Technology and Applications (DATA 2018). SCITEPRESS, 2018, p. 373–380. DOI: https://doi.org/10.5220/0006910203730380. ISBN 978-989-758-318-6.

[11] Han, J., Kamber, M. and Pei, J. 10 - Cluster Analysis: Basic Concepts and Methods. In: Han, J., Kamber, M. and Pei, J., ed. *Data Mining (Third Edition)*. Third Editionth ed. Boston: Morgan Kaufmann, 2012, p. 443–495. The Morgan Kaufmann Series in Data Management Systems. DOI: https://doi.org/10.1016/B978-0-12-381479-1.00010-1. ISBN 978-0-12-381479-1.

[12] Hassen, M., Carvalho, M. M. and Chan, P. K. Malware classification using static analysis based features. In: *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE, 2017, p. 1–7. DOI: https://doi.org/10.1109/SSCI.2017.8285426. ISBN 978-1-5386-2726-6.

[13] Hu, X. and Shin, K. G. DUET: Integration of Dynamic and Static Analyses for Malware Clustering with Cluster Ensembles. In: *Proceedings of the 29th Annual Computer Security Applications Conference*. New York, NY, USA: Association for Computing Machinery, 2013, p. 79–88. ACSAC '13. DOI: https://doi.org/10.1145/2523649.2523677. ISBN 9781450320153.

[14] Hu, X., Shin, K. G., Bhatkar, S. and Griffin, K. MutantX-S: Scalable Malware Clustering Based on Static Features. In: *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. San Jose, CA: USENIX Association, June 2013, p. 187–198. ISBN 978-1-931971-01-0. Available at: https://www.usenix.org/conference/atc13/technical-sessions/presentation/hu.

[15] Ionescu, V. M. The analysis of the performance of RabbitMQ and ActiveMQ. In: *2015 14th RoEduNet International Conference - Networking in Education and Research (RoEduNet NER)*. 2015, p. 132–137. DOI: https://doi.org/10.1109/RoEduNet.2015.7311982. ISBN 978-1-4673-8180-2.

[16] Jamalpur, S., Navya, Y. S., Raja, P., Tagore, G. and Rao, G. R. K. Dynamic Malware Analysis Using Cuckoo Sandbox. In: *2018 Second International Conference on Inventive Communication and Computational Technologies (ICICCT)*. 2018, p. 1056–1060. DOI: https://doi.org/10.1109/ICICCT.2018.8473346. ISBN 978-1-5386-1974-2.

[17] Jovanović, B. *A Not-So-Common Cold: Malware Statistics in 2021*. March 2021. Available at: https://dataprot.net/statistics/malware-statistics/.

[18] Klabnik, S. and Nichols, C. The Rust Programming Language. In:. Chap. 11 [cit. 2021-05-07]. Available at: https://doc.rust-lang.org/book/ch11-01-writing-tests.html.

[19] Křoustek, J. and Zemek, P. *Categorization of malware samples* [Internal documentation of Avast]. [cit. 2020-12-23].

[20] Larson, Q. *The 2020 Stack Overflow Developer Survey – 65,000 Devs Share Their Salaries, Top Programming Languages, and More*. May 2020. Available at: https://www.freecodecamp.org/news/stack-overflow-developer-survey-2020-programming-language-framework-salary-data/.

[21] Matsakis, N. D. and Klock, F. S. The Rust Language. *Ada Lett.* New York, NY, USA: Association for Computing Machinery. October 2014, vol. 34, no. 3, p. 103–104. DOI: https://doi.org/10.1145/2692956.2663188. ISSN 1094-3641.

[22] Perdisci, R., Lee, W. and Feamster, N. Behavioral Clustering of HTTP-Based Malware and Signature Generation Using Malicious Network Traces. In: *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation.* USA: USENIX Association, 2010, p. 26. NSDI'10. Available at: https://dl.acm.org/doi/10.5555/1855711.1855737.

[23] Pitolli, G., Aniello, L., Laurenza, G., Querzoni, L. and Baldoni, R. Malware family identification with BIRCH clustering. In: *2017 International Carnahan Conference on Security Technology (ICCST).* IEEE, 2017, p. 1–6. DOI: https://doi.org/10.1109/CCST.2017.8167802. ISBN 978-1-5386-1585-0.

[24] Plaskoň, P. *Rozšíření systému pro shlukovou analýzu binárních souborů.* Brno, 2017. 14–15 p. Bachelor thesis. Fakulta informačních technologií VUT v Brně. Supervisor Kolář Dušan. Available at: https://www.fit.vut.cz/study/thesis/20059/.

[25] Rastin, P. and Matei, B. Prototype-based Clustering for Relational Data using Barycentric Coordinates. In: *2018 International Joint Conference on Neural Networks (IJCNN).* 2018, p. 1–8. DOI: https://doi.org/10.1109/IJCNN.2018.8489366. ISBN 978-1-5090-6014-6.

[26] Rosli, N., Mohamed, W., M.A, F. and Rahayu, S. Clustering Analysis for Malware Behavior Detection using Registry Data. *International Journal of Advanced Computer Science and Applications.* January 2019, vol. 10, no. 12. DOI: https://doi.org/10.14569/IJACSA.2019.0101213.

[27] Shuwei, W., Baosheng, W., Tang, Y. and Bo, Y. Malware Clustering Based on SNN Density Using System Calls. In: Huang, Z., Sun, X., Luo, J. and Wang, J., ed. *Cloud Computing and Security.* Cham: Springer International Publishing, 2015, p. 181–191. DOI: https://doi.org/10.1007/978-3-319-27051-7_16. ISBN 978-3-319-27051-7.

[28] Weinberger, C. *NoSQL Performance Benchmark 2018 – MongoDB, PostgreSQL, OrientDB, Neo4j and ArangoDB.* February 2018 [cit. 2021-04-25]. Available at: https://www.arangodb.com/2018/02/nosql-performance-benchmark-2018-mongodb-postgresql-orientdb-neo4j-arangodb/.

[29] Ye, Y., Li, T., Adjeroh, D. and Iyengar, S. S. A Survey on Malware Detection Using Data Mining Techniques. *ACM Comput. Surv.* New York, NY, USA: Association for Computing Machinery. June 2017, vol. 50, no. 3. DOI: https://doi.org/10.1145/3073559. ISSN 0360-0300.

[30] Ye, Y., Li, T., Chen, Y. and Jiang, Q. Automatic Malware Categorization Using Cluster Ensemble. In: *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.* New York, NY, USA: Association for Computing Machinery, 2010, p. 95–104. KDD '10. DOI: https://doi.org/10.1145/1835804.1835820. ISBN 9781450300551.

[31] Zaïane, O., Foss, A., Lee, C.-H. and Wang, W. On Data Clustering Analysis: Scalability, Constraints, and Validation. In: Chen, M.-S., Yu, P. S. and Liu, B., ed. *Advances in Knowledge Discovery and Data Mining*. Berlin, Heidelberg: Springer Berlin Heidelberg, May 2002, p. 28–39. DOI: https://doi.org/10.1007/3-540-47887-6_4. ISBN 978-3-540-43704-8.

[32] Zendulka, J., Bartík, V., Lukáš, R. and Rudolfová, I. *Získávání znalostí z databází ZZN: Studijní opora*. Brno: Fakulta informačních technologií VUT v Brně, October 2009 [cit. 2020-12-10]. 121–139 p. Private document. Available at: https://www.fit.vutbr.cz/study/courses/ZZN/private/opora/ZZN.pdf.

[33] Zhang, Y., Rong, C., Huang, Q., Wu, Y., Yang, Z. et al. Based on Multi-features and Clustering Ensemble Method for Automatic Malware Categorization. In: *2017 IEEE Trustcom/BigDataSE/ICESS*. IEEE, 2017, p. 73–82. DOI: https://doi.org/10.1109/Trustcom/BigDataSE/ICESS.2017.222. ISBN 978-1-5090-4906-6.

Appendix A

Crate Configuration

```
[package]
name = "rusty"
version = "0.1.0"
authors = ["matus"]
edition = "2018"

[dependencies]
arangors = {version = "~0.4.6", features = ["reqwest_blocking"], default-features = false}
chrono = "~0.4.19"
clap = {version = "~2.33.3", features = ["wrap_help"]}
config = "~0.10.1"
futures = "~0.3.10"
futures-executor = "~0.3.13"
futures-util = "~0.3.13"
indicatif = "~0.15.0"
lapin = "~1.6.8"
log = "~0.4.14"
regex = "~1.4.4"
serde = "~1.0.124"
serde_json = "~1.0.64"
simplelog = "~0.9.0"
threadpool = "~1.8.1"

[dev-dependencies]
function_name = "~0.2"

[lib]
name = "rustylib"
path = "src/lib.rs"

[[bin]]
name = "rusty"
path = "src/bin.rs"
```

Figure A.1: Configuration of Rusty crate (`Cargo.toml`)

Appendix B

Auxiliary Scripts

Figure B.2 shows an output of `classify` console script (see Section 6.4). It returns the number of created clusters, the number of samples in clusters, and the number of samples which were not able to fit in any cluster. Further, it returns a table showing a level of mixture of clusters. Signs `[s]` and `[c]` reveal whether the number is based on samples or clusters. Row `TAG` shows statistics according to classification in samples's tag attributes. Row `TAG (s N)` means that missing (`None`) classifications were not used in measurement. Rows with `CLUSTY` are based on the classification of clusters in clustering result created by Clusty. The table on the bottom shows a histogram of sizes of created clusters. The empty cells were emptied due to lack of space in the page. They contain values similar to the values in other cells.

`anytree`	2.8.0	tree generator
`bson`	0.5.10	BSON processor
`click`	7.1.2	argument parser
`coloredlogs`	15.0	logging
`dash`	1.19.0	web application
`dash-cytoscape`	0.2.0	web application
`exrex`	0.10.5	string generator from regex
`graphviz`	0.16	graph plotting
`matplotlib`	3.4.0	graph plotting
`pika`	1.1.0	sending AMQP messages
`plotly`	4.14.3	graph plotting
`pymongo`	3.11.0	MongoDB connection
`python-arango`	5.4.0	ArangoDB connection
`requests`	2.25.1	HTTP requests
`tabulate`	0.8.9	generating tables
`tqdm`	4.58.0	progress bar
`verboselogs`	1.7	logging

Table B.1: Used 3rd party Python packages

```
Usage: classify [OPTIONS] DB_NAME RESULT_NAME

Options:
  --show-samples / --no-samples   Whether to show or to not show links with
                                  hashes for samples.
  --show-stats / --no-stats       Whether to show or not to show aggregated statistics.
  --show-top-clusters / --no-top-clusters
                                  Whether to show or not to show 10 greatest clusters.
  --skip-none-classifications / --do-not-skip-none-classifications
                                  Whether to skip or not to skip
                                  classifications with no value (None).
  --filtering-methods / --no-filtering-methods
                                  Whether to ignore classification of samples
                                  clustered by specific method (e.g. YARA) in Clusty.
  --filtering-source [tag|clusty|better|all] Source for filtering.
  --filtering-classification [severity|type|strain|none]
                                  Show only clusters with mixed selected
                                  classification (default ‘none‘).
  --save-to-file TEXT             Store results into the file.
  --load-from-file TEXT           Load results from the file.
  --clusty-result-name TEXT       Clusty result name.
  --help                          Show this message and exit.
```

Figure B.1: Console arguments of Classify package

```
Clusters: 8287

Samples clustered: 99498

Samples unclustered: 502

| Mixed        | total [c]    | severity [c] | type [c] | strain [c] | total [s] | severity [s] | type [s]         | strain [s]       |
| ------------ | ------------ | ------------ | -------- | ---------- | --------- | ------------ | ---------------- | ---------------- |
| TAG          | 176 (2.12 %) | 23 (0.28 %)  |          |            |           | 319 (0.32 %) | 10105 (10.16 %)  | 14123 (14.19 %)  |
| TAG (s N)    | 148 (1.79 %) | 23 (0.28 %)  |          |            |           | 319 (0.32 %) | 9889 (9.94 %)    | 11723 (11.78 %)  |
| CLUSTY       | 128 (1.54 %) | 10 (0.12 %)  |          |            |           | 162 (0.16 %) | 5819 (5.85 %)    | 37100 (37.29 %)  |
| CLUSTY (s N) | 79 (0.95 %)  | 10 (0.12 %)  |          |            |           | 162 (0.16 %) | 5463 (5.49 %)    | 17050 (17.14 %)  |

| Cluster size  | 1-5            | 6-10           | 11-20 | 21-50 | 51-100 | 101-200 | 201-500          | 501+             |
| ------------- | -------------- | -------------- | ----- | ----- | ------ | ------- | ---------------- | ---------------- |
| Cluster count | 7587 (91.55 %) | 217 (2.62 %)   |       |       |        |         | 40 (0.48 %)      | 34 (0.41 %)      |
| Sample count  | 8950 (9.0 %)   | 1675 (1.68 %)  |       |       |        |         | 12992 (13.06 %)  | 57635 (57.93 %)  |
```

Figure B.2: Example of shortened output of `classify` console script

Appendix C

Division of Attributes into Groups

Attribute name	Static		Dynamic	
	all	clustering	all	clustering
api_calls	X			
entry_point_address	X			
entry_point_bytes	X			
export_table_hash	X	X		
icon_hash	X	X		
import_table_hash	X	X		
manifest_hash	X			
pdb_path	X	X		
resources	X	X		
rich_header	X	X		
section_table_hash	X	X		
symbols	X	X		
version_info	X	X		
watermark	X	X		
uncommon_atoms			X	
uncommon_commands			X	X
uncommon_cuckoo_signatures			X	
uncommon_dependencies			X	
uncommon_events			X	X
uncommon_gvma_signatures			X	
uncommon_hosts			X	
uncommon_jobs			X	X
uncommon_mailslots			X	X
uncommon_mutexes			X	X
uncommon_named_sections			X	X
uncommon_pipes			X	X
uncommon_ports			X	
uncommon_registry_keys			X	X
uncommon_scheduled_tasks			X	X
uncommon_semaphores			X	
uncommon_timers			X	X
uncommon_touched_files			X	X

Table C.1: Division of attributes into groups used in the experiments

Appendix D

Gradual Selection Experiments

```
PE
|__ 259529817: malware>dropper>None
|   |__ 260386682: malware>dropper>None
|       |__ 260497422: malware>dropper>None
...
|__ 259551502: clean>None>None
|   |__ 260540939: clean>None>None
|__ 259533675: malware>worm>None
|   |__ 259979616: malware>worm>None
|__ 259551438: malware>trojan>Swisyn
|   |__ 260099002: malware>trojan>Swisyn
|__ 259559462: pup>None>PCSpeedCat
|   |__ 259572797: malware>None>SpeedCat
|       |__ 259724713: malware>None>SpeedCat
|__ 259553704: malware>dropper>cloudeye
|   |__ 259625964: malware>dropper>cloudeye
|   |   |__ 259771662: malware>dropper>cloudeye
|   |__ 259970879: malware>dropper>cloudeye
|__ 259532701: malware>trojan>None
|   |__ 259533800: malware>trojan>None
...
```

Figure D.1: A cluster tree showing their hierarchy. It shows ID and classification for each cluster.

Name	o	clusters [c]	tiny [%c]	tiny [%s]	huge clusters [%s]	severity [%s]	strain [%s]	strain (s N) [%s]
Clusty (all)	X	5 310	92.67	5.87	79.19	0.69	17.23	16.88
Clusty (no Yara)	X	18 895	97.83	19.68	64.17	1.3	7.92	7.44
first_5_2_2	√	11 360	93.17	12.24	55.11	0.25	14.23	11.42
first_4_2_2		11 467	93.29	12.75	58.13	0.25	14.05	11.46
first_3_2_2		11 471	93.24	12.75	57.89	0.25	14.22	11.43
first_2_2_2		11 469	93.25	12.74	57.67	0.25	14.05	11.46
second_5_2_2	√	11 360	93.17	12.24	55.11	0.25	14.23	11.42
second_5_1_2		11 362	93.28	12.28	57.47	0.25	14.23	11.58
third_5_2_2	√	11 360	93.17	12.24	55.11	0.25	14.23	11.42
third_5_2_1		11 345	93.37	12.25	58.44	0.25	14.27	11.41

Table D.1: Clustering results using Gradual selection - threshold (see Section 7.2 to understand)

Appendix E

Tests

```
     Running target/debug/deps/integration-df50c264183a8943
running 23 tests
test test_create_index_for_collection ... ok
test test_add_result_to_results_if_not_exists ... ok
test test_create_collection_if_not_exists ... ok
test test_create_edge_collection_if_not_exists ... ok
test test_delete_clusters_relationship ... ok
test test_controllers ... ok
test test_cluster_blacklisting ... ok
test test_get_cluster_level ... ok
test test_get_clusters_by_rule_created_last_minute ... ok
test test_get_clusters_by_rule_created_last_minute_using_intersection ... ok
test test_get_clusters_by_value_using_intersection ... ok
test test_get_sample_by_hash ... ok
test test_get_clusters_by_value ... ok
test test_get_hierarchical_clusters_at_top_level_created_last_minute ... ok
test test_get_hashes_of_all_samples_in_result ... ok
test test_get_hierarchical_clusters_at_top_level_and_one_clusters_children ... ok
test test_is_sample_clustered ... ok
test test_sample_can_be_placed_in_cluster ... ok
test test_hierarchical_cluster_blacklisting ... ok
test test_single_controller ... ok
test test_sample_can_be_removed_from_cluster ... ok
test test_sample_blacklisting ... ok
test test_move_sample_into_new_cluster ... ok

test result: ok. 23 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in
    0.97s

   Doc-tests rustylib

running 1 test
test src/db/arangodb.rs - id_to_key (line 60) ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in
    0.87s
```

Figure E.1: Example output of tests run

Appendix F

Web Preview

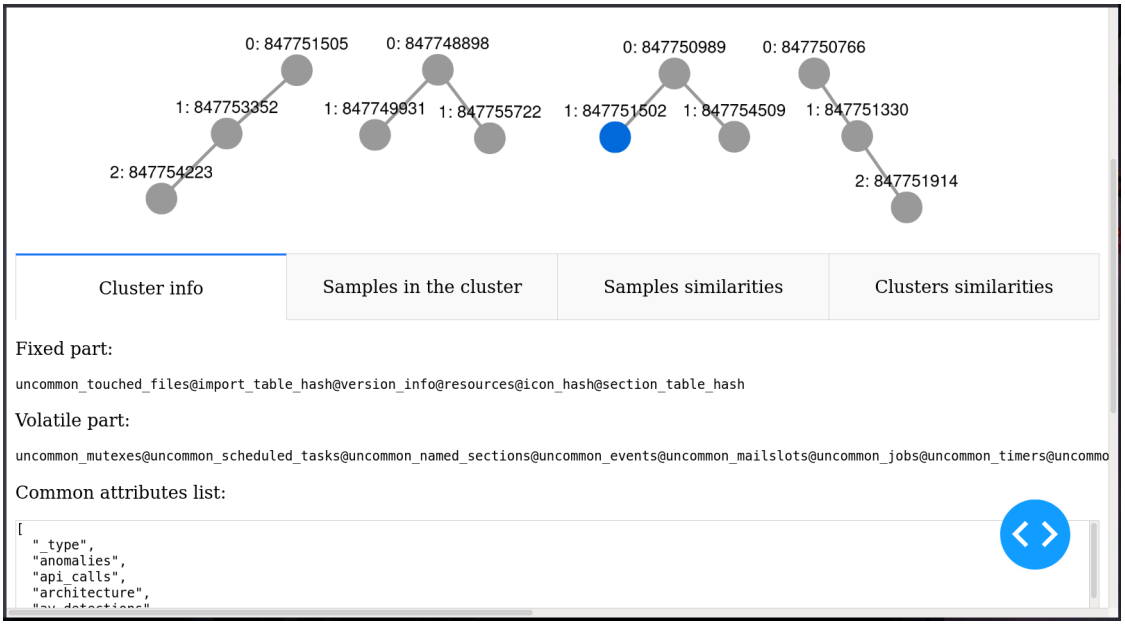
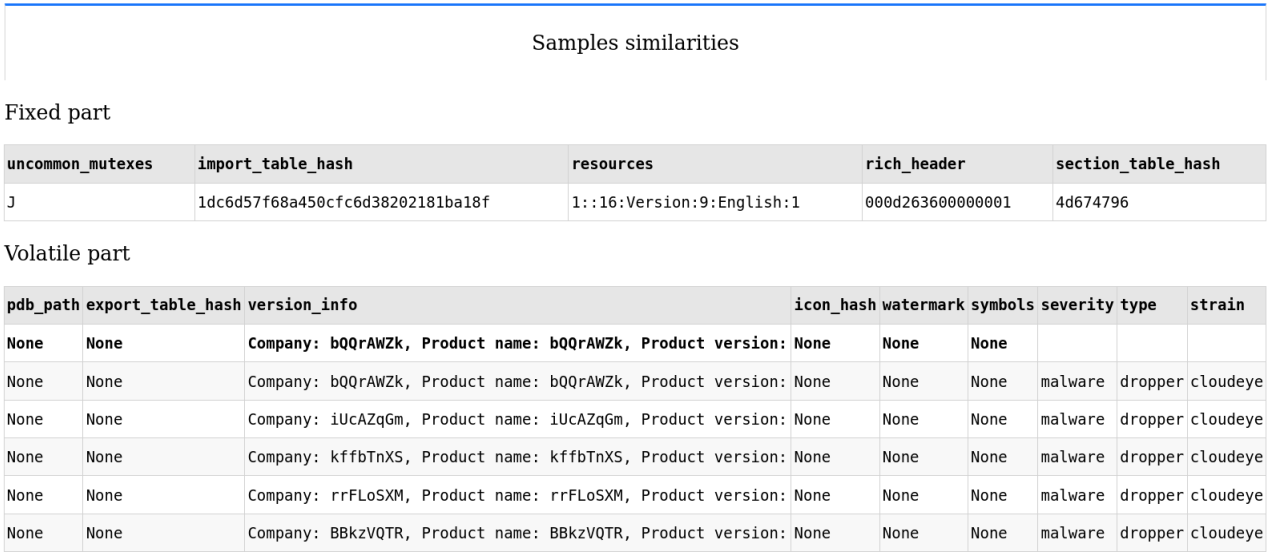


Figure F.1: Web preview of cluster hierarchy. It shows 12 clusters, four of them is top-level. The detail on the bottom of the figure shows information about cluster with ID 847751502.



Samples similarities

Fixed part

uncommon_mutexes	import_table_hash	resources	rich_header	section_table_hash
J	1dc6d57f68a450cfc6d38202181ba18f	1::16:Version:9:English:1	000d263600000001	4d674796

Volatile part

pdb_path	export_table_hash	version_info	icon_hash	watermark	symbols	severity	type	strain
None	**None**	**Company: bQQrAWZk, Product name: bQQrAWZk, Product version:**	**None**	**None**	**None**			
None	None	Company: bQQrAWZk, Product name: bQQrAWZk, Product version:	None	None	None	malware	dropper	cloudeye
None	None	Company: iUcAZqGm, Product name: iUcAZqGm, Product version:	None	None	None	malware	dropper	cloudeye
None	None	Company: kffbTnXS, Product name: kffbTnXS, Product version:	None	None	None	malware	dropper	cloudeye
None	None	Company: rrFLoSXM, Product name: rrFLoSXM, Product version:	None	None	None	malware	dropper	cloudeye
None	None	Company: BBkzVQTR, Product name: BBkzVQTR, Product version:	None	None	None	malware	dropper	cloudeye

Figure F.2: Web preview of a single cluster showing samples' similarities. It contains five samples. The fixed part contains five attributes. The volatile part contains another five attributes with either missing or different values, and classifications. The first row below the header of volatile part is a base sample of the cluster. All samples have the same classification.

Appendix G

Content of Attached Media

The attached DVD contains the following directory structure:

README.txt
: Document with a brief description of DVD content.

thesis.pdf
: PDF document with master's thesis.

thesis_print.pdf
: Printable version of PDF document with master's thesis.

thesis/
: Source code of master's thesis in LaTeX.

rusty/
: Directory with the source code of Rusty tool in Rust language.

scripts/
: Directory with all auxiliary scripts used during the development.