COUGAR: A SYSTEM FOR CLUSTERING UNKNOWN MALWARE USING GENETIC ALGORITHM ROUTINES

by

Zachary Wilkins

Submitted in partial fulfillment of the requirements
for the degree of Master of Computer Science

at

Dalhousie University
Halifax, Nova Scotia
November 2020

Table of Contents

List of Tables

List of Figures

Abstract

Malicious software is a persistent threat across our digital platforms. With unending malware growth, and increasingly higher profile attacks, organizations across the world are ramping up their cyber defence capabilities.

Cluster analysis is one such tool for understanding the threats faced. By organizing seemingly disconnected samples according to their behaviours, attack patterns can be discerned and defended against. But given the volume of malware, an automated approach is necessary to scale.

In this thesis, I design and implement a system called COUGAR which uses a multi-objective genetic algorithm to automatically optimize clustering algorithms. The clustering algorithms are applied to low-dimensional embeddings derived from high-dimensional malware behavioural data. The system employs function imports extracted from malicious binaries, but is flexible enough to accommodate many other features derived from static or dynamic malware analysis. After the optimization process completes, the system generates signatures for each cluster which prioritize usability and comprehensible signature components.

The experiments indicate that any of the chosen clustering algorithms can produce at least satisfactory results, with density-based approaches generating especially successful clusters, achieving an F-Score of 0.79 and V-Measure of 0.88. The resulting signatures are very representative of their respective clusters, with the vast majority achieving representation scores of at least 90%.

List of Abbreviations Used

ANOVA ANalysis Of VAriance

API Application programming interface

AV Anti-virus

COUGAR Clustering Of Unknown malware using Genetic Algorithm Routines

DBSCAN Density-based spatial clustering of applications with noise

DEAP Distributed Evolutionary Algorithms in Python

DoS Denial-of-Service

EMBER Endgame Malware BEnchmark for Research

EMO Evolutionary multi-objective optimization

GA Genetic algorithm

IDS Intrusion detection system

MD5 Message Digest 5

MOGA Multi-objective genetic algorithm

NSGA Non-dominated sorting genetic algorithm

OPTICS Ordering points to identify the clustering structure

PCA Principle component analysis

PE Portable executable

SCOOP Scalable COncurrent Operations in Python

SHA Secure Hash Algorithm

SSE Sum squared error

SSS Scaled summed similarity

t-SNE t-distributed stochastic neighbour embedding

TF-IDF Term frequency – Inverse document frequency

Tukey's HSD test Tukey's Honestly Significant Difference test

UMAP Uniform Manifold Approximation and Projection

Acknowledgements

Thanks to my supervisors, Dr. Nur Zincir-Heywood and Dr. Frédéric Massicotte, for their continued support and guidance, and to my readers, Dr. Malcolm Heywood and Dr. Tami Meredith.

I would like to thank the employees and friends of the Canadian Centre for Cyber Security, without whom this thesis would not be possible.

As well, my parents have always stressed the importance of education, and supported my academic ventures unreservedly. I cannot thank them enough. A special shoutout to my mom for proofreading every draft of this thesis.

Last, but certainly not least, I would like to thank my family and friends for listening to me yammer on about computers for over twenty years. It's not going to stop any time soon.

Chapter 1

Introduction

As the world has become more dependent on computers, so too has their value risen in the eyes of cyber criminals. Increasingly, malicious actors have sought to employ widely available malware to inflict damage or extract financial rewards. This scales from consumers and small businesses, which are especially vulnerable to ransomware attacks [1], to large technology companies, such as DynDNS, which suffered a major denial-of-service (DoS) attack in 2016 that crippled large parts of the global internet [2].

Part of the problem is the large numbers of consumer devices coming online as part of the Internet-of-Things, whereby household appliances (stove, kettle, doorbell, etc.) are connected to the Internet. Manufacturers, aiming to be affordable, can cut corners in regard to security, resulting in the Mirai malware employed in the aforementioned DoS attack [3]. This will be further hastened by the launch of 5G wireless [4], also enabling always-on device connectivity.

Furthermore, malware mutation engines have allowed for an unending torrent of binaries that are difficult to detect with traditional, signature-based anti-virus software [5]. Polymorphic malware, software that can change their abilities, are also a growing problem [6]. It has been estimated that in 2014, over 1,000,000 new malware samples were introduced every day, a 26% increase from the previous year [7]. This growth makes scaling manual malware analysis impossible.

As a result, it is necessary to fight computers with computers. One approach that has proven successful is clustering [5–7]. While there are innumerable ways to disguise malicious files, their attack and data exfiltration techniques are finite [7]. By clustering based on malicious behaviours, patterns, and severity, many malware may be reduced to a single grouping [8]. In doing so, newly observed malware can be compared to their already classified peers, allowing for manual analysis to be triaged to the most severe samples. Furthermore, by identifying malware clusters

with signatures, their behaviour can be concisely described and referred to [9]. These signatures often consist of descriptive information pertaining to the sample [10, 11].

The research objective of this thesis is then to identify and describe commonalities in malware samples through an automated clustering and signature generation process. This is achieved through the design and development of a system called COUGAR (Clustering Of Unknown malware using Genetic Algorithm Routines). The novel contribution of COUGAR is a tunable process that reduces high-dimensional behavioural data from malware to two-dimensions, and optimizes clustering behaviour using a multi-objective genetic algorithm; this combination of techniques is unseen in the literature. After initial experiments on a small 3,000 sample dataset, a larger 10,000 sample dataset is clustered, and signatures generated for each cluster.

The results indicate that each of the tested algorithms succeed to varying degrees on the smaller dataset, while a clear advantage is attained by one algorithm on the larger dataset. The generated signatures are highly representative of the discovered clusters, but their comprehensibility can be an issue in certain clusters.

In the following sections, I will provide necessary background information for contextualizing the remainder of the thesis. The fundamental motivations of cyber security are discussed in Section 1.1. The types of analysis enabling the decomposition of malware for clustering are given in Section 1.2. Identification of malware and their families is covered in Section 1.3. Finally, evolutionary computation and its prominent genetic operators are presented in Section 1.4.

1.1 Cyber Security Fundamentals

It should be of no surprise that when one has something of value, opponents may seek to take or destroy it. In the digital world, cyber security is concerned with the task of protecting digital resources and information.

Those that seek to undertake such malicious actions can come from a variety of backgrounds [12]:

- Large nation-states desire disruption of their adversaries.
- Cyber criminals engage in tasks for financial reward, contracting their abilities to the highest bidder.

- Hacktivists have a political or otherwise ideological message to deliver.
- Terrorist groups aim to sow discontent or provoke violence.
- Disgruntled or compromised employees work against their own organization from within.
- Others do it simply for amusement, or as a show of strength.

All of these people can be referred to as malicious or cyber-threat actors.

Malicious actors have a variety of pathways with which to sabotage their enemies [13]. Some abuse bugs in code to access forbidden resources. Others employ social engineering to trick their targets into giving up their passwords or other identifying information. In this thesis, my focus is on those that write or otherwise engage malicious software, called malware, that runs on the target's computers to achieve the desired effect.

Malware may be designed to destroy digital information, or hold it hostage. Surreptitious malware may not even make its presence known to the user, instead siphoning valuable files or computing resources. The latter case is especially true for embedded devices like home cameras. These 'zombie' devices sit dormant, functioning as normal [14], until they are activated to take part in some distributed attack, where the victim is unable to effectively block traffic originating from a variety of locations.

Furthermore, the nature of software is such that deploying a modified version of neutralized malware can be only a few lines of code away. This cat-and-mouse game between defenders and attackers naturally lends itself to further study on the defensive side. Some cyber professionals specialize in malware analysis, which is discussed further in Section 1.2.

While there may be many different variants of these malware, the individuals behind them are often working to accomplish some specific goal, and collaborating with their own colleagues. Malware variants can be broadly grouped under different families, where a family of malware may originate from the same group of individuals, or be forked from a common codebase. To enable defensive professionals to talk about these groupings in a concise way, they may label notable malware families. Discussion of this topic is elaborated on in Section 1.3.

1.2 Malware Analysis

To understand more about the malware deployed against friendly targets, one might analyze the malicious binary and extract information from it. Malware authors know that they may be under observation, and therefore seek to make this process more difficult for the defenders. Generally speaking, analysis techniques can be classified as either static or dynamic. The following subsections elaborate on each. Neither is without flaws, and drawbacks are noted.

1.2.1 Static Analysis

Static analysis consists of analysis types that do not run the executable. As a result, static analysis is typically safer, since the malware cannot inadvertently cause damage to the machine upon which it is running.

Static methods derive information from data that is already contained within the file. This might include the extraction of strings, possibly indicating IP addresses or domains that the malware calls to. As the malware will be designed to run on some target platform, the executable structure can be exploited to extract information. For example, the import table in Windows Portable Executables (PEs) contains the libraries and functions that the binary will require to execute properly. These pieces of information can be taken together and analyzed to understand the possible behaviour of the malware.

To evade static analysis, malware authors may encrypt portions of the binary or employ packers, tools that modify binaries to avoid such observation [15]. These practices can limit the effectiveness of static analysis.

This thesis engages features derived from static analysis to estimate malicious behaviours.

1.2.2 Dynamic Analysis

In contrast, dynamic analysis detonates (runs) the executable in a controlled environment to glean valuable information. Since the executable in question is known to be malicious, care must be taken to ensure that the detonation machine does not work against the defender. To that end, virtualization is an effective tool in dynamic

analysis.

Dynamic methods often include detonation in specific environments that are known or suspected to be vulnerable to the binary under observation. For example, older versions of Windows that have reached their end-of-life no longer receive security updates, but may still be used by those unaware of the security implications. These machines are especially appealing targets, given their inability to be patched. By employing different virtualized environments, defenders can discover how the malware may behave differently or exploit different vulnerabilities depending on the platform.

When the malware runs on the machine, various behaviours may be logged by the observers, using tools like Cuckoo sandbox[1]. This could include the network requests sent from the machine, the function calls made by the executable, and even new files (drops) that are downloaded or created as part of the execution process. Since malware must be allowed time to execute, and then the machine restored to a clean state, dynamic analysis is often much slower than static analysis [16], which can work as fast as the hardware allows.

Since threat actors are well aware of sandbox analysis techniques, they may include mitigation tactics within the malware to avoid discovery [17]. Checks for known sandbox characteristics or observational processes can enable the malware to avoid exposing itself, which is especially an issue with virtualized sandboxes [16].

While dynamic analysis features are not employed in this thesis, their potential should not be overlooked, and is an exploration area for future work.

1.3 Malware Identification

Given the volume of malware produced and distributed, it is important to be able to effectively identify and communicate the nature of these threats. Specific malware are usually addressed by their hash sum, while groupings of malware may have a family label assigned. These processes are detailed in the following subsections.

[1] `https://cuckoosandbox.org/`

1.3.1 Malware File Hashes

Malware are commonly identified by their hash sum, using such hashing algorithms as Message Digest 5 (MD5) or the Secure Hash Algorithm (SHA) family. Hash functions possess a number of properties that make them well suited for individual identification.

Hash functions are, in mathematical terms, surjective one-way functions. Given some input, the hash function will always produce a corresponding output. Computing the inverse, on the other hand, is prohibitively difficult. This allows hashes to be freely shared without concern for sharing the underlying malware. They are also highly efficient to compute, with some processors even having specific instructions to accelerate hash computation. In addition, hash functions use a fixed output size that is not tied to the size of the input, which allows for compact representation of very large files.

Hash sums are often represented using hexadecimal notation. This is the single largest drawback to using hashes. Without additional information, the hash is only useful for identification, and provides no context for the underlying file.

1.3.2 Malware Family Labels

Groupings of notable malware with a common authorship or codebase may be assigned a family name. Employing a family name allows for many individual samples to be broadly grouped, and discussed, as it is understood that malware within the same family will exhibit similar properties and behaviours. These names can even be given by their authors in some cases [3]. Unfortunately, the process by which family labels are computed is not standardized.

Anti-virus companies will often have their own internal naming scheme which are computed as part of their identification systems. For example, the malware with the MD5 hash of `1978c37af0e28d417198d55e8a970c8c` is identified by many vendors as similar but varied names:

- Ad-Aware, ALYac, Arcabit et al.: `Trojan.Ransom.Cerber.1`
- AhnLab-V3: `Trojan/Win32.Cerber.R184564`
- Fortinet: `W32/Cerber.4C18!tr.ransom`

- TrendMicro: `Mal_Cerber-NS1`

While a human can quickly ascertain the common token as `Cerber`, and refer to the malware as such[2], a machine approach may consider each as entirely separate labels. To address that, some researchers have created solutions to consolidate AV families to a common label. One such tool, AVClass is pre-computed as part of the dataset employed in this thesis.

AVClass is freely available software[3] that allows labels to be decided in an automatic process agnostic to the platform or antivirus vendor [18]. Notably, AVClass can be computed without the malicious executable by hooking into online APIs, such as VirusTotal[4].

AVClass begins by taking as input the assigned AV labels, as well as manually created lists of generic terms and label aliases. After removing duplicates and differing suffixes to common prefixes, AV labels are tokenized and filtered. The filtering process ensures tokens are lowercase, do not contain extraneous digits, and are of sufficient length (≥ 4). Employing the alias and generic lists to replace and remove tokens, respectively, a final list of minimal tokens is achieved. They are ranked according to their frequency, with the top token assigned as the final label.

This concludes the presentation of background information pertaining to the field of cyber security, the domain within which this research problem is framed. The following section introduces evolutionary computation, which enables optimization in the COUGAR system.

1.4 Evolutionary Computation

Evolutionary computation is a subset of computer science focused on emulating observed natural processes to solve problems. This is often implemented as a stochastic optimization process in the style of biological evolution.

The typical evolutionary process begins with the creation of an initial population of genotypic individuals. These individual chromosomes usually consist of a sequence of bits or numbers, called genes, where the genes are taken together as the input

[2] `https://blog.malwarebytes.com/threat-analysis/2016/03/cerber-ransomware-new-but-mature/`

[3] `https://github.com/malicialab/avclass`

[4] `https://www.virustotal.com`

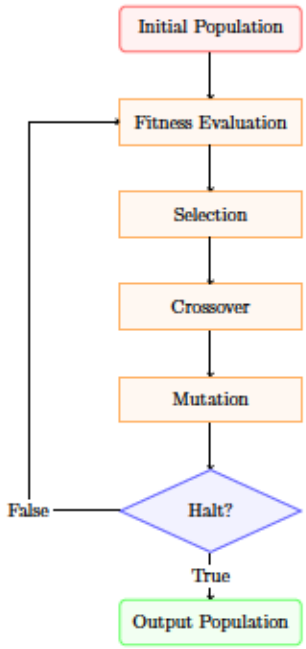

Figure 1.1: Evolutionary process in the context of computing

(phenotypic expression of the chromosome) for some problem to be solved. The algorithm runs until some halting criteria is satisifed, often a specified number of generations, wherein each individual is scored according to some fitness function. The fitness of an individual is how well each solves the given task, and may consist of multiple objectives. Individuals that do not solve the task may be culled from the population, while the most successful may be selected for reproduction. In the reproductive process, offspring of the parent individuals are created through a variety of genetic operators and incorporated into the population. When the evolutionary process completes, the output population consists of elite individuals most capable of solving the given task. This process is visualized in Figure 1.1.

1.4.1 Genetic Operators

As mentioned, genetic operators are used to create or modify offspring from a parent population. The primary operator classes are selection, crossover, and mutation.

To illustrate the usage of these operators, consider a trivial example task called the OneMax problem [19]: given a string of bits, achieve the maximum possible sum of bits. For a bitstring of length n, the maximum score will be n, and the minimum score will be 0.

Selection

Selection operators decide which individuals are to be employed. Not all individuals will successfully solve the task, and those that do solve the task may have varying levels of achievement. Selection operators allow the algorithm to be guided toward the optimal solution(s) by selecting high performing or otherwise desirable individuals.

In the OneMax example, the selection process may simply be to choose some m individuals that achieve the highest sum of the bits in their string.

Crossover

Crossover is the process by which two parent individuals produce two children. The crossover operator takes genes from each of the parents and recombines them to produce new individuals. This allows for individuals to be produced which share desirable properties of both parents. These crossover actions can be messy, where the resulting individuals may have more or less genes than their parents.

There are many ways with which to crossover individuals. The simplest example is one-point crossover. For the OneMax problem, given two binary strings of length n, a random integer p is chosen such that $0 \leq p \leq n$. This index p is the location where the genes of the parents will be joined. Genes 0 to p from the first parent, and genes p to n from the second parent, are joined to create the child individuals. This process is visually depicted in Figure 1.2, with $p = 2$.

Mutation

Mutation is the process by which one parent individual may produce a child. The mutation operator may select a gene of an individual and change the value of that gene spontaneously. This stochastic process encourages diversity within the population and can help solutions converge to the global optimum by avoiding local optima. Oftentimes, there are two parameters by which this process will be decided: the probability of any mutation happening, and the probability of any individual gene mutating. As a result, it is possible for the mutation operator to select an individual for mutation, without actually mutating any of its genes. Conversely, it is also possible for every gene of an individual to be changed by the mutation operator in one mutation

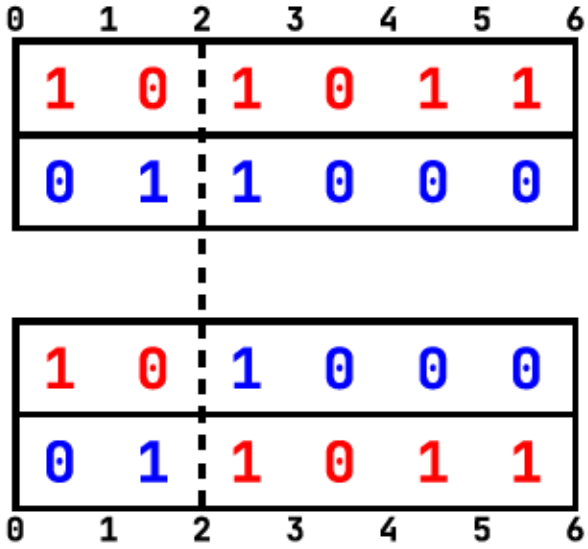

Figure 1.2: One-point crossover example for the OneMax problem

action.

In the OneMax example, a mutation will simply result in a bit being swapped from 0 to 1, or vice versa.

This concludes the presentation of background information for the thesis.

In this chapter, I introduced the COUGAR system and associated background. This system enables the research objective of automatically clustering malware behaviours extracted through static analysis processes. COUGAR does so by creating a high dimensional vectorization, reducing it to a low dimensional embedding, and optimizing parameters of clustering algorithms applied to that embedding via a multi-objective genetic algorithm.

The rest of this thesis is organized as follows. Related works in the literature are summarized in Chapter 2. The methodology followed in this research is explored in Chapter 3, including evaluation metrics, the datasets, feature processing techniques, clustering algorithms, evolutionary multi-objective optimization procedures, distributed computation, and signature generation. Evaluations and results are presented in Chapter 4. Finally, in Chapter 5, conclusions are drawn and directions for future work are suggested.

Chapter 2

Literature Review

In pursuit of an automated malware clustering system, it was necessary to consult contemporary works in the literature relevant to research areas in this thesis. Aligning fruitful approaches from these differing areas enables the methodology underlying the COUGAR system. This chapter summarizes related works and introduces motivation for design decisions in Chapter 3.

Since the primary goal of this thesis work was malware clustering, it is presented first in Section 2.1, which covers successful approaches to this task. In addition, the intention was for the clustering process to be optimized by a genetic algorithm. Hence, Section 2.2 discusses works that have engaged such techniques. Finally, the extraction of malware signatures was a secondary goal of this work, and so this area of research is presented in Section 2.3. A brief summary closes this chapter in Section 2.4.

2.1 Malware Clustering

Malware clustering is not an unknown topic, and a number of approaches have been explored to address this issue.

Beginning with a survey of the field, Faridi et al. [20] conduct an extensive study to determine optimal clustering algorithms, parameters, and features for malware clustering. They run 5,673 malware from an industry partner in a Cuckoo sandbox[1], and collect network communications and system calls. They construct ground truth reference clusters using triggered alerts from Suricata[2], a network threat detection engine. Alerts that belong to one family uniquely are taken as the label for that sample. This process yields 94 clusters. After extracting a variety of features from the collected Cuckoo information, they group the features into distinct categories. They

[1] `https://cuckoosandbox.org/`
[2] `https://suricata-ids.org/`

reduce these features to the most important strings, and vectorize the corpus using the Term Frequency – Inverse Document Frequency (TF-IDF) technique. They employ a number of distance metrics as well as constructing dissimilarity matrices directly from the vectorization. The authors then engage a variety of clustering algorithms, and present an assortment of clustering metrics. They indicate that network features were the most successful, while including all features lead to the worst performance. As far as clustering algorithms were concerned, DBSCAN with NCD, spectral with cosine similarity, and hierarchical with Bray-Curtis and average linkage provided the best overall performance.

Employing Cuckoo sandboxes for extracting behavioural information is a popular technique, also used by Duarte-Garcia et al. [6], Lee et al. [7], and Pirscoveanu et al. [21]. Each of these works use the extracted API call information as input for feature generation, but their output differs given the different approaches they use. In the first work, Duarte-Garcia et al. [6] construct a system to characterize malware behaviour, with the intent of assigning labels to unseen samples. To filter out noise, API calls common to at least 90% of malware are removed. They employ TF-IDF, akin to Faridi et al. [20], conceptualizing each malware as a document, and each call as a term in that document. k-means is employed, varying the number of clusters and evaluating cluster quality using silhouette coefficient. They then engage a number of classifiers. Their results show that 60 clusters was the most successful configuration, and gradient-boosting decision trees were the most successful classifier.

Lee et al. [7] instead concoct an "intelligent analysis" system to guide malware analysts in their triage efforts against malware mutants. API calls from Cuckoo are used to create bigrams representing call sequences. These bigrams are then compared using cosine similarity to facilitate grouping. Mutants that are above a given threshold t are joined together in a graph. This process results in 210 groups of malware. The top ten groups, however, contain 43% of the samples, which can be automatically categorized as mutants. These results depend on the sensitivity of the threshold. A larger t leads to more specificity in clusters, whereas a lowered threshold leads to a more general picture, with more samples being included.

Self-organizing maps (SOMs) are the tool of choice by Pirscoveanu et al. [21] to cluster malware behaviours. From the aforementioned API calls, they extract features

representing successful/unsuccessful calls, and the return codes from failed calls. They then apply principal component analysis (PCA), a form of dimension reduction, to reduce the feature set. After using gap statistics to choose the number of clusters, they employ SOMs to project each sample onto a two-dimensional map, where the number of clusters is equal to the number of nodes in the map. As ground truth, they label each malware by collecting their anti-virus labels from VirusTotal, and remove noise by faceting to the top five Microsoft AV labels. Their SOMs achieve accuracies of approximately 80% when using a 2x2 map, and 70% with a 4x4 map.

Alternative clustering approaches include the use of distributed computation and fuzzy clustering [5, 22]. Jang et al. [5] create an automated, large scale malware triage system called BitShred to target malware mutants. Employing feature hashing, they create a fingerprint for each malicious sample and calculate approximate Jaccard similarities. Clusters are created hierarchically based on a variable threshold. Co-clustering can then occur to discover correlated features in sub-groups of malware. Their results indicate that their approach is much faster than contemporary techniques, with similar accuracy. The second work focuses on mobile threats instead of the Windows platform, with Acampora et al. [22] creating a system to discover Android malware phylogeny. Using established Android malware datasets, they run 3,000 samples on a smartphone and collect system call traces, which are representative of the malicious behaviours. They transform these calls using a process modelling language called Declare, and create system call execution fingerprints (SEFs). From these SEFs, a dissimilarity matrix is calculated and passed to a fuzzy clustering algorithm, FANNY. This allows the researchers to determine the membership degrees of each malware family with respect to the determined clusters. They validate their results by establishing a reference clustering from previous literature, achieving precision and recall scores of 0.8 and 0.73, respectively.

Given the success of natural language approaches, I focus on similar techniques, especially taking inspiration from the intelligent triage intentions of Lee et al. [7] and the dimensionality reduction of Pirscoveanu et al. [21].

An issue all of these works have in common is their approach to feature extraction. Namely, each require the original malicious binaries to extract features that are appropriate for other systems. This makes comparisons on the same dataset very

difficult. As well, distributing malicious files is fraught with peril, as their dangerous nature can cause discomfort to system administrators, and the responsible use of these dangerous files is difficult to verify when distributed openly. A solution to this issue is the use of an open and reputable, third-party feature dataset.

2.2 Evolutionary Cluster Optimization

Optimization of parameters is a common use case for genetic algorithms (GAs), and so too have they been applied to clustering.

Three recent works try to ascertain the optimal choice of k when applying k-means clustering to datasets [23–25]. Kurinjivendhan and Thangadurai [23], as well as Anusha and Sathiaseelan [24], modify k-means to incorporate the evolutionary process, with the latter showing improved silhouette scores relative to the compared Grouping Genetic Algorithm. Irfan et al. [25] instead demonstrate improved performance with respect to the number of iterations of the algorithm. Unfortunately, the triviality of the employed datasets (≤ 3 classes, < 200 data samples) limits the applicability of the results.

Employing a more complex dataset, Al-Malak and Hosny [26] explore a combination of clustering and the evolutionary process to group the multimodal CoPhIR dataset, containing photography metadata from Flickr. Specifically, they incorporate an adaptive weighting system, that changes the relevance of the dataset features as part of the mutation process. They evaluate using the Davies-Bouldin index, a metric based on Euclidean distance to the cluster centres. Solutions developed using the adaptive weighting score are, on average, 10% better than the non-adaptive algorithm.

From a real-world application perspective, multi-objective genetic clustering is applied to MRI brain scans by Mukhopadhyay et al. [27]. They engage NSGA-II to determine optimal cluster centres and the partitioning of the space. The objectives to be optimized are three clustering metrics, namely, the Xie-Beni index, the PBM index, and the fuzzy C-means measure. Generally speaking, the GA is optimizing the separation between clusters, as well as the compactness of those clusters. As NSGA-II outputs a front of non-dominated solutions, rather than a single 'best' solution, the researchers use three distinct clustering ensembles to determine final cluster

membership. Ranking the solutions against other algorithms such as k-means and expectation maximization, the best GA-backed techniques show a 5 percentage point increase in classification accuracy and, at minimum, 20% increase in the Adjusted Rand Index. The hypergraph partitioning algorithm clustering ensemble is the most successful across their experiments.

Another area of application, this time in the realm of computer security, is the identification of encrypted traffic, as explored by Bacquet et al. [28]. The researchers seek to classify encrypted flows as SSH or not without looking at such common features as port numbers. They use a variant of the Pareto Converging Genetic Algorithm, a so-called steady state GA. This is because only two population members are replaced at each generation, resulting in a gradual change over epochs. Both feature selection and cluster count are evolved as part of the individual representation. The objectives to be optimized include maximizing distance between clusters, as well as minimizing all of: the distance within clusters, the number of clusters, and the number of features selected. After evolution has completed, k-means is run for each individual in the Pareto front and a label is assigned for each cluster based on the majority vote. The best individual is then selected based on the detection rate and false-positive rate. The researchers modify their previous work to incorporate a divisive hierarchical approach to clustering, where clusters are partitioned if they do not attain a specified "purity threshold", corresponding to the cluster label. In the experiments, the system is trained and tested on web traffic collected from a large university, and benchmarked against k-means, DBSCAN, expectation maximization, as well as their previous non-hierarchical GA. The non-hierarchical GA demonstrates performance within 5 percentage points of the otherwise best technique (k-means) while also achieving much lower false positives (decrease of 10 percentage points). The hierarchical variant further increases the performance to be within 3 percentage points of k-means, while also maintaining low false-positive rates around 1%.

Clustering optimization via genetic algorithm has also been applied to documents. Jian-Xiang et al. [29] begin by creating a document similarity matrix, based on the Euclidean distance between keyword similarity vectors. In this work, keyword similarity is based on the length of the common substring between terms. The objective for the GA is then to minimize distance between the points assigned to each class and

the class centre, thereby finding optimal cluster centres. They apply their technique to 600 documents from various disciplines in the Chinese Social Sciences Citation Index, and compare the results to those achieved by k-means. The results indicate a modest increase in classification performance (2–6 percentage points), and overall objective performance, at the cost of increased convergence time for the GA. Another important drawback is the high-dimensional dense keyword matrix, which does not scale well with an increase in document terms.

In these works, k-means is often the favoured tool, and augmented by or incorporated into the genetic algorithm. While the results do not show a dramatic performance increase, they do consistently demonstrate the well-rounded nature of solutions produced using evolutionary optimization. This is probably due in part to the Pareto-optimal sets of solutions produced by multi-objective GAs. Accordingly, further investigation of clustering algorithms tuned by multi-objective genetic algorithms is warranted.

2.3 Malware Signature Extraction

Creating signatures for malware is another popular topic in the literature. As discussed in Section 1.2, static and dynamic procedures can be applied to extract features for signature generation. These signatures are used for identification and analysis purposes.

Newsome et al. [9] create a system called Polygraph to improve the effectiveness of intrusion detection systems (IDSs) against polymorphic worms, capable of altering their malicious payload. While the payload is inconsistent, in order for the exploits to remain effective, there are "invariant bytes" that must exist for any given exploit. The authors use this to their advantage, extracting byte groups from malicious and benign network flows as tokens, and using these tokens to construct three kinds of signatures:

1. Conjunction signatures simply contain every token extracted from the sample. Another sample matches if its tokens are exactly the same as the signature.

2. Token-subsequence signatures are created by finding an ordered sequence of tokens present in every sample. Another sample matches if it contains the

chosen tokens.

3. Bayesian signatures are probabilistic models based on a naïve Bayes classifier, rather than exact sets of tokens to match. The probability of a term indicating maliciousness is calculated independently based on its presence in malicious or benign traffic flows. Another sample matches if its summed token probabilities exceed a threshold.

They evaluate each signature type against three scenarios: one worm, benign traffic and one worm, and benign traffic and multiple worms. No clear winner emerges, with each having advantages and disadvantages depending on the worm and the problem scenario. A specific issue is the rigidity of the signatures created. The authors recommend a combination of the three for maximum effect.

Three other surveyed works also employ network traffic for signature generation [30–32]. The former two employ Snort IDS[3]rules and hierarchical clustering to generate signatures, while the latter operates on the raw network packets. Zurutuza et al. [30] capture internet traffic to be passed through Snort by mocking a number of popular web services. Traffic that triggers alerts are clustered using a hierarchical agglomerative clustering method to model known attacks, while unknown traffic is separately clustered to model new attacks. These clusters are then transformed into Snort rules, and fed back into the capturing system, where the cycle continues. They are eventually able to classify 99% of traffic, with the small number remaining attributed to misconfigurations.

Choi et al. [31] also aim to rapidly generate malicious signatures by constructing a hierarchical cluster tree of network signatures. Snort IDS rules are clustered by computing similarity via their longest common substrings (LCSs). A binary tree is produced, where the leaves of the tree are signatures, while the nodes are similarity scores. When a new malware is seen, a Snort signature is produced and compared through the tree, where it is placed near its most similar signatures, rebalancing as appropriate. Using the LCS allows for generalization of the signatures, and can be computed in linear time using a suffix tree. This work only describes the technique, and would need to be assessed separately to ascertain its effectiveness.

[3] `https://www.snort.org/`

In the final network-based work, Wang et al. [32] employ feature hashing on network flow bytes to scale their system. After manually assembling clusters of similar malware, they hash n-grams extracted from the flows to generate a binary matrix. The most significant features from each cluster are extracted based on their "cluster coverage", indicating a significant presence within the cluster. They union the features, and engage a greedy Bayesian selection function to maximize the probability of malicious features. By increasing the number of selected features to 5, they reduce false positives to less than 1%, even when noise in the flow pool is increased to 10%. The downside is that the signatures are black-box models due to the feature hashing, offering no indication as to the malware they identify.

Facing an entirely different threat, Zhang et al. [10] use metadata from Android applications to identify malware families and generate signatures. The information extracted includes the app name, permissions employed, as well as activity name. After removing common terms to reduce noise, the terms are grouped by their similarity score, as calculated using TF-IDF and k-means for clustering. They extract the similarities into patterns consisting of a particular field, an operator (exact, prefix, or suffix match), and a value. These signatures are very interpretable for humans, while also being simple to implement in a classifier. They validate the effectiveness of the signatures using 1.75 million malware, and extract over 12,500 unique family signatures, which are said to achieve "no false positives in production".

Han and Olivier [11] also emphasize interpretability in their malware signature generation work. They run malware in a sandbox to capture system calls which are generalized into categories, and converted to events consisting of type, operation, and arguments. This sequence of events is taken as a malicious behaviour. For each malicious behaviour, n-grams are extracted to form signatures. The researchers find that setting $n = 5$ provides the best balance of generic versus specific. To automate this process, the events are vectorized, allowing for clustering with k-means. In their evaluation process, they achieve F-Scores as high as 0.95.

Many of these signatures have to make a trade-off between interpretability and usability. Feature hashing enables high comparison performance, for example, but preempts the possibility of inspecting the resulting model. A key concern for any future work is to ensure that the resulting signatures maintain some balance between

these conflicting objectives. Selecting features for the signature is also an area of interest, as a probabilistic approach based on labels could be faulty due to behavioural overlap between malware families. An alternative approach is to take inspiration from anomaly detection by selecting less seen features to create the final signature.

2.4 Summary

In this chapter, I discussed the three research areas of key importance to the chosen thesis topic. This includes malware clustering, evolutionary cluster optimization, and malware signature extraction.

The surveyed works suggest areas for exploration as part of this thesis. My novel contribution is enabled by the extension and conjunction of disparate techniques discussed. These approaches taken together will allow for a fully automated malware clustering system unseen in the literature. Namely, I will employ natural language processing techniques to cluster malware as one might cluster documents, and optimize the clustering parameters through the use of a multi-objective genetic algorithm. Following this, signatures for each cluster will be generated in a process that targets both usability and interpretability, and selects features for inclusion according to their global usage.

Chapter 3

Methodology

In this chapter, I will detail the techniques with which the COUGAR system was implemented and thesis results achieved. It should be noted that in the penultimate section (3.7), an analogous scenario is introduced to demonstrate the metrics and methodology.

The overall architecture of COUGAR is presented in Figure 3.1. This figure indicates the transitions between components in the system, and direction in which data is read or written. To begin, the Parquetizer converts the appropriate data from the EMBER dataset to Parquet tables, before notifying the vectorizer. The vectorizer processes the Parquet tables, saving the vectorization to a datalake, and notifies the reducer. The reducer uses a dimension reduction algorithm to transform the vectorization to a low-dimensional embedding. The Spark leader is notified that the new embedding is ready, and it creates Spark workers, which compute clusterings in parallel and return results to the leader. When the job completes, the data is saved for an analyst to inspect. On their instruction, the job parameters can be tweaked, and the Spark leader notified to recompute the clusters. This process can be iterated upon until the desired results are achieved.

The following sections further describe this processing methodology. Specifically, I discuss the chosen dataset in Section 3.1, and feature processing in Section 3.2, including feature vectorization and dimension reduction. Section 3.3 discusses the utilized clustering algorithms. Section 3.4 details the evolutionary multi-objective methodology and associated algorithms, while Section 3.5 describes the distributed computational approach. Section 3.6 features descriptions of the signature generation process, as well as pseudocode for the algorithms. Section 3.7 enumerates the metrics employed for evaluation, along with examples of each. This chapter concludes with a short summary in Section 3.8.

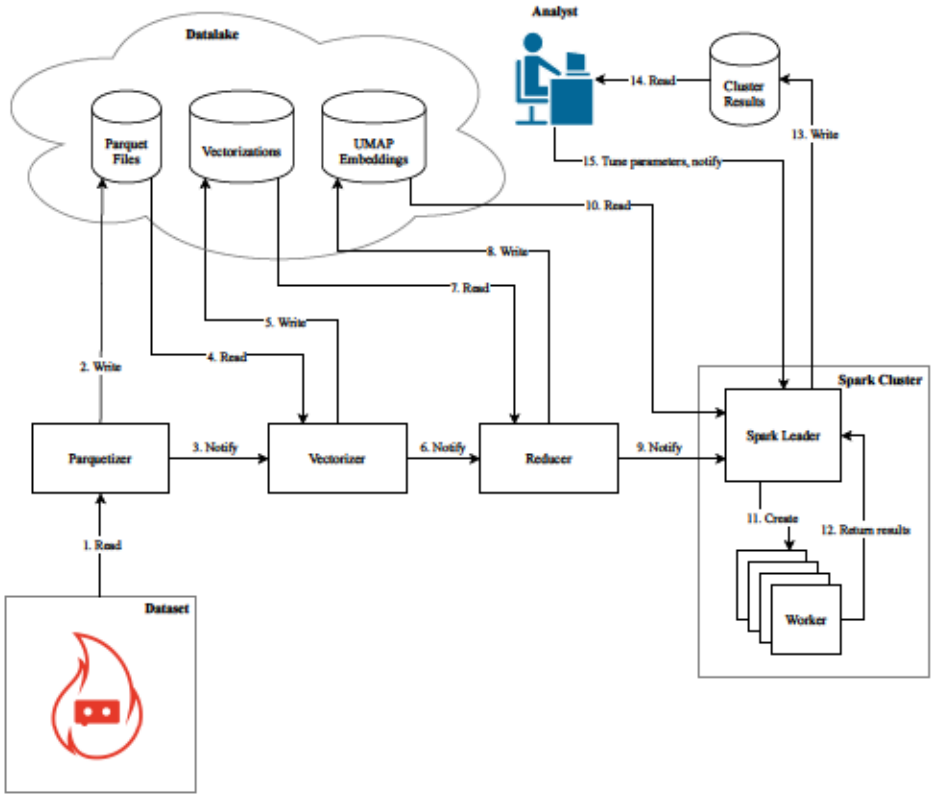

Figure 3.1: High-level COUGAR architecture

3.1 Dataset

A project involving malicious files presents challenges that typically do not exist when handling other datasets. Malware is, by its very nature, untrusted, and so must be sequestered to prevent accidental execution. Furthermore, it is usually undesirable to acquire or retain malware on a computer system, so the availability of malware is generally limited. While there are some websites [33–35] that allow users to share malicious files, community-driven aggregation does not lend itself to stable and established datasets.

As this thesis does not offer contributions in the areas of malware storage or feature derivation, a known dataset with pre-extracted features was desirable. This allowed for the work to focus on the exploitation of those features, rather than their acquisition. In addition, using a publicly available dataset facilitates reproducible research and makes for more meaningful comparisons of performance against existing and future works. Thankfully, one such dataset was exactly suited to this task.

3.1.1 EMBER

For this work, I chose to use the relatively novel Endgame Malware BEnchmark for Research (EMBER) dataset [36] from Endgame (now a part of Elastic). EMBER was designed as a "benchmark dataset for researchers" [37], and includes features extracted using LIEF [38] from over 2,000,000 Windows PEs in 2017 and 2018.

The features in EMBER are numerous. For samples from 2018 alone, the uncompressed JSON files are in excess of 9GB. These features include general file characteristics, as well as file header information, imported and exported functions, byte and byte-entropy histograms, and string statistics. I utilize the AVClass feature, described in Section 1.3, as the ground truth malware family label.

With such a variety of features, it was necessary to concentrate on a distinct subset for this work. It has been shown [39–41] that import statements are an effective tool to extract malicious behaviours, as it is difficult to obfuscate third-party imports without breaking compatibility with the library. To that end, import terms and their corresponding library were extracted from EMBER for each malicious sample. A shortened example of the extracted information is shown in Table 3.1 for the malware sample with MD5 hash `0a7a72a5853f3740ea76f9934764bd44`.

In the interest of speed, the container format was also modified during extraction. Representing imports as a table allows for compression advantages in the form of repeated values. For this task, Apache Parquet[1] was chosen. Parquet is a column-oriented data format optimized for the Hadoop ecosystem, where in-memory data access is paramount, an attribute shared with COUGAR. The MD5, SHA256, assigned AVClass, and total number of imports were also included in a separate table for reference.

It should be noted that while this thesis focuses on function imports, the COUGAR framework was designed to be feature agnostic, and employ any data that can feasibly be vectorized. There are known issues to using imported functions [42], such as packers and obfuscators, which seek to hide the malware from such static analysis. In the future work, I describe features that may be more successful when dealing with malware that act in such ways.

[1] `https://parquet.apache.org`

Library	Imported Function
kernel32.dll	LoadLibraryW
kernel32.dll	DeleteFileA
crypt32.dll	CertDuplicateStore
crypt32.dll	CertSaveStore
wtsapi32.dll	WTSFreeMemory
wtsapi32.dll	WTSEnumerateServersA
...	...

Table 3.1: Sample of import terms from an EMBER file

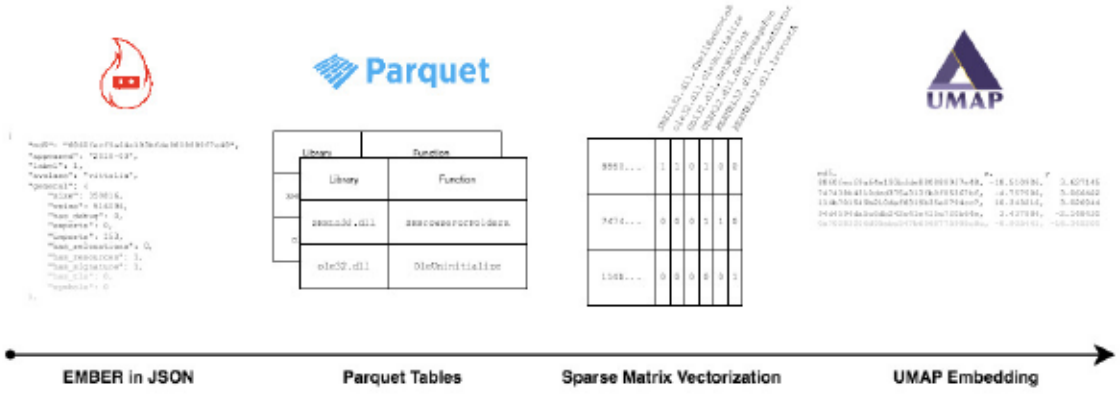

Figure 3.2: Transformation from EMBER to embedding

3.2 Feature Processing

The transformation from EMBER to embedding is described in this section. A high-level visual representation can be observed in Figure 3.2. In this figure, EMBER data is transformed from the JSON source files, to Parquet tables, to a sparse matrix vectorization of the function imports, and finally to the UMAP embedding in two-dimensional space.

3.2.1 Vectorization

Imports from each sample are converted to lowercase, joined with a comma to their parent library, and added to the corpus. A Bag-of-Words model is employed to vectorize the dataset, using the `CountVectorizer` from Scikit-Learn [43]. In the resulting matrix, each row represents a single malicious sample, and each column records the number of occurrences for each library/function in the corpus. Since

EMBER only contains a list of function imports, rather than frequency of function calls, the resulting sample representation is a bit vector, where the presence of a term is indicated by a set bit. An example vectorization is given in Table 3.4. This vectorization is then held for further computation in the form of dimension reduction.

3.2.2 Dimension Reduction

While clustering on higher dimensions is possible, the nature of a production malware triage system is continuous data ingestion. To this end, updated cluster models can be produced significantly faster on low-dimensional data.

The algorithm I employ for this task is the Uniform Manifold Approximation and Projection (UMAP) technique [44]. This technique works by modelling the data with a fuzzy topological structure, and is notable for attempting to preserve both local and global distances between data points.

UMAP was chosen for a number of reasons. UMAP demonstrates certain qualities that make it better suited as a cluster pre-processing step than other dimension reduction techniques like t-SNE [45]. Namely, it more effectively maintains the "global structure" of the data which leads to more "meaningful separation between connected components of the manifold on which the data lies" [46]. In turn, the resulting clusters in the embedding are more relevant. As well, it has been demonstrated to be much faster than similar techniques such as PCA and t-SNE in benchmarks [47], which is an important future consideration to scale the system to handle the large influx of malware every day. Finally, UMAP supports the transformation of new data into existing models [48]. While this feature is not unique to UMAP, it is essential for my use case in predicting the malware family of unseen samples based upon some training dataset.

While UMAP can be parameterized to reduce to any number of dimensions, I elected to reduce to a two-dimensional embedding for the sake of simplicity, and ease of visualization. While I had originally intended to include the UMAP parameters as part of the optimization search process, rebuilding many UMAP embeddings in each generation dramatically slowed the system. To that end, UMAP is employed with out-of-the-box defaults for almost every parameter. The notable exception is the number of neighbours, which is set to 1% of the input dataset. Neighbourhood

size is an important consideration, as it determines the degree to which the algorithm will focus on local or global structure. This is an important balancing point, as the resulting embedding should ideally illustrate the distance between both similar and different samples, which will be close or far away, respectively, in the embedding space. Having such separation is also desirable for the next step of the system: the clustering process.

3.3 Clustering Algorithms

I employ three clustering algorithms in this thesis: DBSCAN [49], OPTICS [50], and k-means [51]. The former two techniques are closely related, both working to cluster based on the density of data. In addition, neither DBSCAN nor OPTICS requires parameterization with the number of clusters in the dataset. This mitigates a 'chicken and egg' scenario, where the number of clusters is required to cluster the data, but a reasonable estimate of cluster count is difficult without first clustering the data. k-means, on the other hand, is not density based and requires the number of clusters to be chosen in advance. It does, however, possess such desirable qualities as being scalable, simple to implement, and guaranteed to converge. Accordingly, it stands as a comparative baseline to DBSCAN and OPTICS.

For each algorithm, two or three parameters were selected for optimization. DBSCAN uses `eps` and `min_samples`, while OPTICS uses `max_eps` and `min_samples`. k-means uses `n_clusters`, `n_init`, and `max_iter`. Each parameter will be discussed in the following subsections, and the default values along with parameter bounds (for the evolutionary process) are shown in Table 3.2. These were chosen in concert with the documentation for Scikit-Learn, as they identify the most consequential parameters[2].

Each algorithm is initialized using reasonable parameters in the vicinity of their defaults in the Scikit-Learn implementation. The notable exception to this is `min_samples` in DBSCAN and OPTICS, which is set higher (40), to avoid forming needlessly small clusters.

[2] `https://scikit-learn.org/stable/modules/generated/sklearn.cluster.DBSCAN.html`
`https://scikit-learn.org/stable/modules/generated/sklearn.cluster.OPTICS.html`
`https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html`

Algorithm	Parameter	Default	Lower Bound	Upper Bound
DBSCAN	`eps`	0.5	0.0	100.0
	`min_samples`	40	5	250
OPTICS	`max_eps`	50.0	0.0	100.0
	`min_samples`	40	5	250
k-means	`n_clusters`	8	2	100
	`n_init`	10	1	100
	`max_iter`	300	10	1000

Table 3.2: Default values and bounds for clustering algorithm parameters

3.3.1 DBSCAN

Density-based spatial clustering of applications with noise (DBSCAN) [49] is a clustering algorithm that operates on the premise that clusters are groups of densely packed points separated by less dense space. It considers a specified number of nearby points (`MinPts`, or `min_samples` in Scikit-Learn) as a cluster core if they fall within a specified distance (ε, or `eps` in Scikit-Learn). Clusters are expanded to border points accessible from at least one of the core points. Points that are at least ε away from a core point are not added to the cluster, and instead marked as noise. These values must be carefully chosen so as not to broadly create one large cluster, or be so specific as to create few or no clusters.

DBSCAN was selected as it supports non-convex cluster shapes, as well as *not* clustering data that it perceives to be noise. The former consideration is especially important when working with UMAP, as it is not guaranteed to produce "clean, spherical clusters" [46]. The latter is also a major benefit in this research problem, as unrelated samples with close proximity to legitimate clusters can be discarded, rather than forcing their inclusion.

3.3.2 OPTICS

Ordering points to identify the clustering structure (OPTICS) [50] is also engaged for comparison, as it is considered to be better suited[3]for large datasets than Scikit-Learn's DBSCAN implementation. OPTICS is a technique that can be extended to a clustering algorithm in the same density-based vein as DBSCAN. It too employs

`MinPts` and ε parameters, but the key difference is that ε is given as the maximum value to a range, instead of a definite value. In this way, OPTICS is capable of finding clusters even when the density of the data is not consistent, by varying ε up to the maximum value. Being a variant of DBSCAN, OPTICS also achieves the aforementioned desirable properties of DBSCAN.

OPTICS constructs an ordering of the points by expanding the ε-neighbourhoods of core points (as in DBSCAN) according to the reachability distance of the points in that neighbourhood. In this context, the reachability distance of a point p is the distance from p to the core point under evaluation, such that p is within the core point's ε-neighbourhood. Accordingly, denser neighbourhoods are evaluated first, which allows for dense clusters within less dense clusters to be identified. Combining the point ordering with the reachability distances enables clusters to be extracted according to some density-differentiating heuristic.

3.3.3 *k*-means

k-means [51] is also considered as one of the most popular iterative clustering algorithms. This technique is computationally efficient and easy to implement, hence its wide applications.

The algorithm begins by randomly initializing k cluster centroids. Data points are then assigned to the cluster with the smallest Euclidean distance between the point and the cluster centroid. Centroids are then recomputed according to the mean position of all points in that cluster. The algorithm converges when the centroids stop changing, or at least fall below a given movement threshold.

The most important parameter to choose is k, or `n_clusters`, the number of clusters. This is difficult to choose without at least observing the data. The other parameters to be chosen for the Scikit-Learn implementation control the number of initializations of the algorithm (`n_init`) and the maximum iterations of the algorithm (`max_iter`) which affects the stopping criteria for the runs.

While k-means is known to demonstrate weakness on data exhibiting characteristics similar to UMAP output [52], the popularity [53] of the technique still gives it comparative value as a baseline clustering algorithm.

[3] `https://scikit-learn.org/stable/modules/generated/sklearn.cluster.OPTICS.html`

3.4 Evolutionary Multi-Objective Optimization

In this thesis, evolutionary multi-objective optimization (EMO) is employed for the dual problems of parameter selection and cluster count optimization. This unsupervised machine learning process results in a sequence of parameters for the chosen clustering algorithms: DBSCAN, OPTICS, and k-means.

3.4.1 Objectives

There are three objectives guiding parameter tuning:

1. Maximize the count of highly homogeneous clusters. The similarity of a cluster is determined by computing the cosine similarities over all elements in the cluster, using the vectorization described in Subsection 3.2.1.

   Given a cluster of m data samples, and a matrix (A) of binary vector imports for that cluster of shape $m \times n$ with rows $\mathbf{A}_1$, $\mathbf{A}_2$, $\mathbf{A}_3$, ..., $\mathbf{A}_m$, we can calculate cosine similarity for some x^{th} and y^{th} rows as in Eq. 3.4.1.

   Equation 3.4.1 (Cosine Similarity).

   $$\cos(\theta) = \frac{\mathbf{A}_x \cdot \mathbf{A}_y}{\|\mathbf{A}_x\|\|\mathbf{A}_y\|} = \frac{\sum_{i=0}^{n} A_{x,i} A_{y,i}}{\sqrt{\sum_{i=0}^{n} A_{x,i}^2}\sqrt{\sum_{i=0}^{n} A_{y,i}^2}}$$

   These calculations are summed and scaled to the range $[0.0, 1.0]$, as shown in Eq. 3.4.2, which I call the scaled summed similarity (SSS).

   Equation 3.4.2 (Scaled Summed Similarity).

   $$SSS = \frac{1}{m} \sum_{x=1}^{m-1} \sum_{y=x+1}^{m} \cos(\mathbf{A}_x, \mathbf{A}_y)$$

   A highly homogeneous cluster is achieved if the SSS is at least 0.8.

2. Minimize the sum of the sum squared error (SSE) for each cluster. The SSE is calculated as in Eq. 3.4.3.

   Equation 3.4.3 (Sum Squared Error).

   $$SSE = (1 - SSS)^2$$

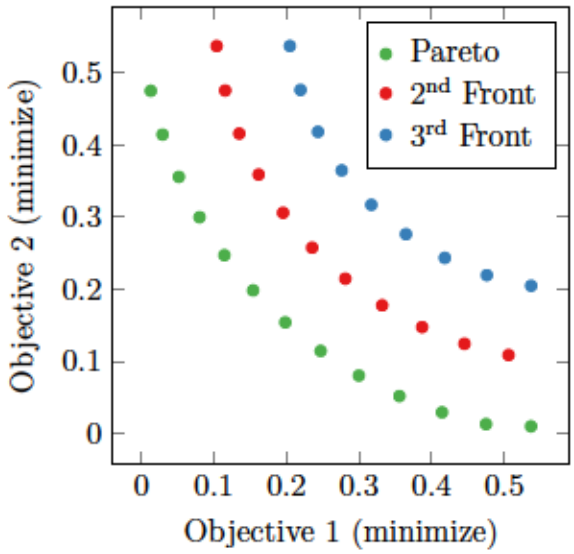

Figure 3.3: Example of three non-dominated fronts in two-objective space

3. Maximize the median cluster size. Median was chosen over mean to mitigate clustering parameters that would create a small number of unreasonably large clusters, thereby skewing the metric and incorrectly indicating success. This situation is less likely to occur when considering the median cluster size.

3.4.2 NSGA-III

I employ the Non-dominated Sorting Genetic Algorithm III (NSGA-III) [54] to optimize the aforementioned objectives. NSGA-III is an extension of NSGA-II [55] to many-objective problems. NSGA-III was chosen as it maintains diversity within the population of candidate solutions, while simultaneously preserving elitist solutions.

Evolutionary algorithms preserving elitism are proven [56] to converge to the global optimum solution for some function. Since the parameters to be optimized are primarily real-valued numbers, effectively an infinite search space, random search is infeasible. This is because it is less efficient than genetic algorithms [57], while also offering no guarantees of optimality.

I will describe NSGA-II before explaining the changes made to it to achieve NSGA-III.

Algorithm Descriptions

NSGA-II works by sorting individuals into fronts within the objective space, where each front dominates all lesser fronts. In this context, an individual dominates another if:

1. the dominating individual achieves a better fitness score in at least one objective, and

2. no fitness score for any objective is less than the dominated individual.

As such, each individual in a front is considered as good as any other individual in that front, with respect to at least one objective. The leading front is called the Pareto-optimal front, and it dominates all other fronts. The 2^{nd} front dominates all individuals except those in the Pareto front, the 3^{rd} front dominates all individuals except those in the Pareto or 2^{nd} front, and so on. A visualization of example fronts in two-objective space is shown in Figure 3.3.

In NSGA-II, ascending fronts are selected as long as space remains in the population. In the event that an entire front cannot be selected for inclusion, a crowded-comparison operator is applied to that front, which selects individuals such that the final front is evenly distributed, ensuring genetic diversity. At this point, crossover/ mutation can proceed to breed the next generation.

NSGA-III modifies its predecessor by employing a set of reference points to achieve a uniform diversity across the population. A normalized hyper-plane is computed over the objective axes, and points are evenly distributed across the plane. In each generation, the final selected front has its population members normalized in the objective space, and associated with a point on the reference plane. A niching operator is applied to choose points that align, as closely as possible, with the reference points. As before, usual genetic operators can be applied to breed the next generation.

Implementation Details

The implementation of NSGA-III was handled by the Distributed Evolutionary Algorithms in Python (DEAP) library [58], which notably supports out-of-the-box parallelization through Scalable COncurrent Operations in Python (SCOOP) [59]. It

can also be expanded to other distributed computation frameworks, such as Apache Spark (see Subsection 3.5.1 for more details).

Parameterization

To parameterize the algorithm, one must consider the partitions (p) of the reference plane, and the number of objectives ($M = 3$). Jain and Deb [54] give the total number of reference points as Eq. 3.4.4.

Equation 3.4.4 (Number of NSGA-III Reference Points).

$$H = \binom{M + p - 1}{p}$$

The same work also suggests $p = 12$ for the number of partitions in the reference plane, but this leads to a reference points size of $H = 91$. The resulting population size, which is calculated to be the smallest multiple of four greater than H, per Jain and Deb [54], is 92. This dramatically slows the evolutionary process due to the large size and slow fitness evaluation, courtesy of cosine similarity's $O(n^2)$ time complexity. This is a side effect of any pairwise distance comparison, and using an approximation for this calculation is left to future work. Choosing $p = 4$ is much more reasonable, leading to $H = 15$ and a final population size of 16.

The algorithm is run for 100 generations, with a crossover probability of 1.0, an individual mutation probability of 1.0, and a gene mutation probability of 0.25. These values are taken from the NSGA-III example in the DEAP documentation[4], though the generation count and gene mutation were modified. The generation count is lowered to 100 from 400, as the algorithm converges sufficiently within that time, as shown in Figure 4.1. The gene mutation probability is suggested as being $\frac{1.0}{n}$, where n is the number of genes. Given the small size of the chromosome (two or three genes), the resulting probabilities are too high and result in significant population fluctuation when the values are changing with such high probability.

Individual Representation

Individuals are represented with two or three floating points numbers, according to the parameters of the given clustering algorithm. To account for the variable scales upon

[4] `https://deap.readthedocs.io/en/master/examples/nsga3.html`

which the parameters can be specified, I scale the floating point gene, and optionally convert it to an integer, before passing it to the clustering algorithm. For example, to evaluate the default DBSCAN individual $\langle 0.005, 0.04 \rangle$, the first gene would be multiplied by 100, and the second would be multiplied by 1000 and converted to an integer. This would then be interpreted as `eps`= 0.5 and `min_samples`= 40.

3.5 Distributed Computation

As previously mentioned, the intended use-case for the COUGAR system is one of ingesting and clustering many new malware every day. To that end, scalability is an important consideration. One approach for scaling is to distribute the work across many computers, and aggregate their results. While DEAP supports SCOOP by default (as mentioned in Subsection 3.4.2), running on a well established and popular distributed computing platform is advantageous for the sake of compatibility with existing infrastructure in an organization. Enter: Apache Spark[5].

3.5.1 Apache Spark

Spark is a popular solution for distributed computing tasks. It works by splitting data processing pipelines into basic tasks that can be computed across many nodes in a cluster of computers, physical or virtual. After tasks are completed on worker nodes, the results are returned to the driver program, typically running on a node within the cluster, which can delegate further tasks as necessary.

A spiritual sequel to the Apache Hadoop and MapReduce approach, Spark mitigates a serious handicap present in the MapReduce framework. Namely, after every transformation, data is written to disk, and must be read again to continue [60]. In a pipeline with many transformations, these glacial I/O tasks can significantly slow the computation process. Spark nullifies these problems by employing lazy evaluation, delaying computation until strictly necessary, and performing the transformations entirely in-memory.

Furthermore, while driven by the Java Virtual Machine, a Spark job can be implemented in Java, Scala, or Python, unlike the Python ecosystem lock-in with SCOOP.

[5] `https://spark.apache.org/`

With the introduction of lambda expressions in Java 8, any of these languages can articulate complicated MapReduce operations in much less code than previously required.

3.5.2 COUGAR on Spark

To assist in scaling COUGAR to handle 10,000 samples and beyond in the second round of evaluations, COUGAR was re-implemented with Spark to parallelize certain calculations. This includes the evaluation of COUGAR individuals in each generation, as well as signature generation and merging for the final, surviving population. Since DEAP does not support Spark by default, it was necessary to fork my own copy[6] of the project to incorporate the Spark-supporting changes[7] that are inexplicably unmerged to the upstream repository. In this way, COUGAR on Spark can produce results much faster than sequential execution.

To further the effectiveness and expediency of the system, some other optimizations were introduced during this reimplementation. A cache was added to track the parameter settings that have already been evaluated, and thereby avoid the costly recalculation process. In addition, at the conclusion of the evolutionary process, a pruning function is applied to remove individuals of identical fitness, or individuals that cannot form at least one highly homogeneous cluster. This serves to discard trivial clusterings, such as all samples grouped together, or no clusters formed at all.

3.6 Signature Generation

When we consider clustering malware by behaviour, we can blur the lines between various malicious families, to the detriment of conventional malware family labels.

For example, consider two families of ransomware which depend on the same underlying cryptographic library for file encryption. Malware samples from both of these families could very well be clustered together because of their shared methodology. Describing the cluster as a mixture of malware families is problematic, however, as it does not scale with many families, nor does it convey the reason why these samples are considered similar to each other.

[6] `https://github.com/znwilkins/deap`
[7] `https://github.com/DEAP/deap/pull/76`

For that reason, signatures are a popular technique for identifying malware groupings [11, 61, 62]. COUGAR generates a signature for each cluster to summarize the cluster elements. In doing so, the system provides an 'at a glance' indication of the prominent behaviours that are shared by malware within the cluster.

3.6.1 Signature Considerations

There are many different ways to construct a cluster signature, depending on the desired properties.

A signature used only for identification purposes may well employ a hash of certain properties, and compare hashes at rapid speed [61]. Unfortunately, the one-way nature of hashing functions does not allow for signature inspection, nor for human modification. To address that limitation, a signature could consist of those important properties serialized, but without some bounds, the signatures may become hyper-specific, and difficult to generalize [62]. In addition, choosing which properties are most informative is a task unto itself.

Thus, COUGAR on Spark takes a simple approach to signature generation that places emphasis on ease of computation and interpretability.

3.6.2 Signature Generation Algorithms

Signature generation is divided into two separate steps: generation and merging of maximal signatures, followed by minimalization to fixed-length signatures.

The first step of this process is depicted in an algorithmic fashion in Algorithm 1. Consider a matrix of binary vectors for some cluster i, where a set bit indicates that the function in some column j has been imported. This is the `clusterVectorization` in line 2 of the algorithm. If we sum the column vectors (line 3), and then select the indices with the maximum values (line 5), we have a list of indices to columns most frequently imported by samples in cluster i. After translating the indices to the actual imports (line 6), we have calculated the maximal signature for the cluster (line 9). This is similar to the token-subsequence signatures from Newsome et al. [9] and significant feature extraction process from Wang et al. [32], but removes ordering from the former and the selection threshold from the latter. By hashing the imports, we can quickly merge clusters with identical maximal signatures (line 11).

Algorithm 1: Generation of maximal signatures

Input : allClusterVectorizations: list of Tuples with sparse matrix and cluster info

Output: maximalSignatures: list of Tuples with hash, imports, cluster info

```
1  maximalSignatures ← List()
2  for clusterVectorization ∈ allClusterVectorizations do
3      vecSummed ← SumColumnVectors(clusterVectorization)
4      maxTermCount ← Max(vecSummed)
5      maxIndices ← SelectIndicesByValue(maxTermCount, vecSummed)
6      imports ← ConvertIndicesToImports(maxIndices)
7      hash ← ComputeHash(imports)
8      clusterInfo ← clusterVectorization.info
9      maximalSignatures.Append(Tuple(hash, imports, clusterInfo))
10 end
   /* Signatures are generated in parallel:  need to merge        */
11 return MergeByHash(maximalSignatures)
```

These maximal signatures are highly specific, but too lengthy to be useful. Minimizing the maximal signatures to fixed-length signatures remedies this issue. As with the maximal signatures, the generation of fixed-length signatures is given in Algorithm 2.

My solution is to obtain the global count of function imports from the complete vectorization, which is passed as `importToCount` in the algorithm input. By selecting the n least frequently imported functions over the entire dataset from the maximal signature (lines 4–7), a fixed-length representation of the cluster can be achieved. The rationale is that if every single function in the maximal signature is equally representative of a given cluster, then the least frequently imported functions overall will be most representative of a specific behaviour in that cluster.

This approach generates predictable and interpretable signatures in an efficient manner. Of course, it is possible that two maximal signatures could be minimized to the same fixed-length signature. In the event that a unique fixed-length signature cannot be computed from the maximal signature, the smaller cluster is discarded (line

12). This happens rarely in practice, and the choice of n was made to minimize such collisions (detailed in Subsection 4.2.2).

Algorithm 2: Generation of fixed-length signatures

Input : maximalSignatures: list of Tuples with hash, imports, cluster info
importToCount: dictionary mapping imports to usage count
n: the desired length of the signature

Output: fixedLengthSignatures: list of Tuples with hash, imports, and cluster info

```
1  fixedLengthSignatures ← List()
2  for maximalSignature ∈ maximalSignatures do
3      imports ← List()
4      for term ∈ maximalSignature.imports do
5          imports.Append(Tuple(importToCount.Get(term), term))
6      end
7      imports ← SubList(Sort(imports), n)
8      hash ← ComputeHash(imports)
9      signature ← Tuple(hash, imports, maximalSignature.info)
10     fixedLengthSignatures.Append(signature)
11 end
   /* Signatures are generated in parallel:  need to reduce      */
12 return ReduceByHash(fixedLengthSignatures)
```

3.7 Metrics & Examples

In this section, I explain the metrics with which the final results will be evaluated, as well as demonstrate their usage on a sample problem analogous to the thesis research area.

The metrics employed are frequently seen in the context of classification and clustering, and engage the predicted and actual labels. For classification/prediction purposes, I employ F-Score [63], weighted by support. For clustering purposes, homogeneity, completeness, and V-Measure [64] are calculated. In addition, I introduce

Song No.	Pred. Genre	True Genre	Cluster No.	Cluster Genre
0	Rock	Rock	0	Rock
1	Rock	Rock	1	Rock
2	Alt.	Alt.	2	Alt.
3	Rock	Alt.	0	Rock
4	Rap	Rap	3	Rap
5	Pop	Rap	3	Rap
6	Jazz	Jazz	0	Rock
7	Jazz	Jazz	4	Jazz
8	Pop	Pop	4	Jazz
9	Rap	Pop	3	Rap

Table 3.3: Prediction and truth labels for the musical dataset

a simple metric, signature representation, which is derived from the generated signatures for each cluster.

For the sake of clarity, I now define a trivial example problem to assist with the explanations.

3.7.1 A Musical Example Problem

Consider a number of songs labelled by one[8]of five genres: rock, alternative, rap, jazz, or pop. Each song also includes the types of instruments employed. One might seek to build a classifier and train a clustering algorithm using this data. This is conceptually similar to the malware clustering problem: the songs can be likened to malware samples, while the instruments are the imported functions.

After a theoretical training process, predicted and actual genres, as well as generated clusters, are given in Table 3.3.

3.7.2 Classification

To understand F-Score, one must first understand precision and recall. Precision is a measure of the relevancy of the results, while recall is a measure of the quantity of relevant results returned. Given counts of true positives (T_p), false positives (F_p), true negatives (T_n), and false negatives (F_n), precision and recall are defined in Eqs. 3.7.1 and 3.7.2.

[8]While multi-genre compositions are a celebrated (and sometimes contentious) occurrence, their consideration is well beyond the scope of this example.

Equation 3.7.1 (Precision).

$$P = \frac{T_p}{T_p + F_p}$$

Equation 3.7.2 (Recall).

$$R = \frac{T_p}{T_p + F_n}$$

For example, calculating the precision and recall of rock songs:

$$P_{rock} = \frac{2}{2+1} = 0.67 \qquad R_{rock} = \frac{2}{2+0} = 1.0$$

F-Score is then the harmonic mean of precision and recall, as defined in Eq. 3.7.3 [63].

Equation 3.7.3 (F-Score).

$$F = 2 \cdot \frac{P \cdot R}{P + R}$$

Following that, the F-Score for rock songs is:

$$F_{rock} = 2 \cdot \frac{0.67 \cdot 1.0}{0.67 + 1.0} = 0.8$$

Since there are multiple classes, reported F-Scores are the averaged F-Scores for each class, weighted by support (S, number of occurrences for each class in true labels).

Given a set of F-Scores for all n classes (F_C), from N data samples, and a set of support values (S) for each class, the formula is given in Eq. 3.7.4.

Equation 3.7.4 (Weighted F-Score).

$$F_w = \frac{1}{N} \sum_{i=0}^{n} F_{C_i} \cdot S_i$$

Weighting the F-Score by support for each label is necessary, as not every potential AVClass label is represented in the training sample subsets. Otherwise, this serves to falsely distort the F-Score, as achieving many F-Scores of 0.0 on labels without any samples is not helpful when assessing performance.

For example, consider a binary classification scenario where there are nine true instances of class A and one true instance of class B. If the classifier predicts all ten as class A, F-Scores of 0.95 and 0.0 are achieved for classes A and B, respectively.

Their macro average, where each F-Score is not weighted for label imbalance, is 0.47, despite 90% of the data being correctly classified. Using a weighted F-Score instead returns 0.85, which is more in line with the actual performance of the classifier.

Finally, the weighted F-Score for the predicted labels is:

$$\begin{aligned} F_w &= \frac{1}{10} \cdot [(F_{rock} \cdot S_{rock}) + (F_{alt} \cdot S_{alt}) + (F_{rap} \cdot S_{rap}) + (F_{jazz} \cdot S_{jazz}) + (F_{pop} \cdot S_{pop})] \\ &= \frac{1}{10} \cdot [(0.8 \cdot 2) + (0.67 \cdot 2) + (0.5 \cdot 2) + (1.0 \cdot 2) + (0.5 \cdot 2)] \\ &= 0.69 \end{aligned}$$

As can be seen from the F-Scores for each class in the previous equation, this weighted F-Score of 0.69 shows that the classifier is very successful on some classes (rock and jazz), but is held back by poor performance on the remaining genres (alt., rap, pop). In this example, each genre had equal support. Had their labels been biased toward rock and jazz, and the classifier predicted with equivalent precision and recall, then the weighted F-Score would have been higher.

3.7.3 Clustering

A clustering is considered perfectly homogeneous if each of its clusters only contain data points from a single class. It is considered perfectly complete if all data points from a given class are contained in the same cluster. Given a set of classes (C), and a set of clusters (K) from N data samples, as well as a matrix mapping true classes to cluster classes (a), we can define entropy (H, a mutual dependency), homogeneity (h), and completeness (c), in Eqs. 3.7.5, 3.7.6, and 3.7.7 [64, 65].

Equation 3.7.5 (Entropy).

$$\begin{aligned} H(C,K) &= -\sum_{c=1}^{|C|}\sum_{k=1}^{|K|}\frac{a_{ck}}{N}\ln\frac{a_{ck}}{N} \\ H(C|K) &= -\sum_{k=1}^{|K|}\sum_{c=1}^{|C|}\frac{a_{ck}}{N}\ln\frac{a_{ck}}{\sum_{c=1}^{|C|}a_{ck}} \\ H(C) &= -\sum_{c=1}^{|C|}\frac{\sum_{k=1}^{|K|}a_{ck}}{N}\ln\frac{\sum_{k=1}^{|K|}a_{ck}}{N} \end{aligned}$$

Equation 3.7.6 (Homogeneity).

$$h = \begin{cases} 1 & \text{if } H(C,K) = 0 \\ 1 - \frac{H(C|K)}{H(C)} & \text{else} \end{cases}$$

Equation 3.7.7 (Completeness).

$$c = \begin{cases} 1 & \text{if } H(K,C) = 0 \\ 1 - \frac{H(K|C)}{H(K)} & \text{else} \end{cases}$$

V-Measure is then the harmonic mean of homogeneity and completeness, defined in Eq. 3.7.8 [64].

Equation 3.7.8 (V-Measure).

$$v = 2 \cdot \frac{h \cdot c}{h + c}$$

For the musical problem in Subsection 3.7.1, we can calculate $H(C,K)$, $H(C|K)$, $H(K|C)$, $H(C)$, and $H(K)$:

$$\begin{aligned}
H(C,K) = & -[(\frac{2}{10}\ln\frac{2}{10}) + (\frac{1}{10}\ln\frac{1}{10} + \frac{1}{10}\ln\frac{1}{10}) + (\frac{2}{10}\ln\frac{2}{10}) + \\
& (\frac{1}{10}\ln\frac{1}{10} + \frac{1}{10}\ln\frac{1}{10}) + (\frac{1}{10}\ln\frac{1}{10} + \frac{1}{10}\ln\frac{1}{10})] = 2.025 \\
H(C|K) = & -[(\frac{1}{10}\ln 1) + (\frac{1}{10}\ln\frac{1}{2} + \frac{1}{10}\ln\frac{1}{2}) + (\frac{1}{10}\ln\frac{1}{3} + \frac{2}{10}\ln\frac{2}{3}) + \\
& (\frac{1}{10}\ln\frac{1}{4} + \frac{1}{10}\ln\frac{1}{4} + \frac{2}{10}\ln\frac{2}{4})] = 0.745 \\
H(K|C) = & -[(\frac{1}{10}\ln\frac{1}{2} + \frac{1}{10}\ln\frac{1}{2}) + (\frac{1}{10}\ln\frac{1}{2} + \frac{1}{10}\ln\frac{1}{2}) + (\frac{1}{10}\ln\frac{1}{2} + \frac{1}{10}\ln\frac{1}{2}) + \\
& (\frac{2}{10}\ln\frac{2}{2}) + (\frac{2}{10}\ln\frac{2}{2})] = 0.416 \\
H(C) = & -[5 \cdot (\frac{2}{10}\ln\frac{2}{10})] = 1.61 \\
H(K) = & -(\frac{4}{10}\ln\frac{4}{10} + \frac{1}{10}\ln\frac{1}{10} + \frac{3}{10}\ln\frac{3}{10} + \frac{2}{10}\ln\frac{2}{10}) = 1.28
\end{aligned}$$

which gives us:

$$h = 1 - \frac{0.745}{1.61} = 0.537$$

$$c = 1 - \frac{0.416}{1.28} = 0.675$$

with which we can calculate V-Measure:

$$v = 2 \cdot \frac{0.537 \cdot 0.675}{0.537 + 0.675} = 0.598$$

While these metrics are well suited for tasks with cleanly labelled data, the reality of a clustering system based on behaviour is that malware with different labels may very well exhibit similar behaviours. Since these metrics are not capable of compensating for this situation, it is necessary to engage another metric to ascertain the success of the system.

In Section 3.6, I discussed the signature generation algorithm employed to label each cluster with a signature for the behaviours represented. The final metric, signature representation, is a natural measurement of the accuracy of those signatures with respect to the underlying cluster.

Given a cluster of m data samples, and a matrix (A) of binary vector imports for that cluster of shape $m \times n$ with columns $\mathbf{A}_1$, $\mathbf{A}_2$, $\mathbf{A}_3$, ..., $\mathbf{A}_n$, we can calculate the representation of the cluster, as a percentage, by first computing a row vector of the summed column vectors, as in Eqs. 3.7.9 and 3.7.10, respectively.

Equation 3.7.9 (Signature Representation).

$$r = (\frac{\max_i \boldsymbol{v}_i}{m}) \cdot 100$$

Equation 3.7.10 (Row Vector of Summed Column Vector).

$$\boldsymbol{v} = \sum_{i=1}^{n} \boldsymbol{A}_i$$

Returning to the musical example for the final time, we might compute a vectorization of the 'imported' instruments as in Table 3.4. After summing the column vectors for each cluster genre, Table 3.5 is created, with which we can compute the signature representation for each cluster:

- Alt. signature representation: $\frac{1}{1} \cdot 100 = 100\%$
- Jazz signature representation: $\frac{2}{2} \cdot 100 = 100\%$
- Rap signature representation: $\frac{3}{3} \cdot 100 = 100\%$

Song No.	Cluster Genre	Bass	Drums	Guitar	Piano	Saxophone	Synthesizer	Voice
0	Rock	1	1	1	0	0	0	1
1	Rock	0	1	1	0	0	0	1
2	Alt.	0	0	1	0	0	0	1
3	Rock	1	1	1	0	0	0	1
4	Rap	0	0	0	0	0	1	1
5	Rap	0	0	0	0	0	1	1
6	Rock	1	1	0	1	1	0	0
7	Jazz	1	1	0	1	0	0	0
8	Jazz	0	1	0	0	0	1	1
9	Rap	0	0	0	0	0	1	1

Table 3.4: Vectorization of instrument use in the musical dataset

Cluster Genre	Bass	Drums	Guitar	Piano	Saxophone	Synthesizer	Voice
Alt.	0	0	**1**	0	0	0	**1**
Jazz	1	**2**	0	1	0	1	1
Rap	0	0	0	0	0	**3**	**3**
Rock	3	**4**	3	1	1	0	3

Table 3.5: Summed instrument use by genre in the musical dataset

- Rock signature representation: $\frac{3}{3} \cdot 100 = 100\%$

In this toy example, every cluster had at least one instrument used in every song in that cluster genre. Hence, each cluster has a signature representation of 100%. That is, the signature for each cluster represents 100% of the samples in that cluster.

3.8 Summary

In this chapter, I discussed the methodology of the COUGAR system.

Using features from the open-source EMBER dataset, a high-dimensional vectorization was calculated using a Bag-of-Words model. This vectorization is representative of all imported functions by samples in the dataset, where a set bit indicates that the function was imported. These imported functions are indicative of the behaviour of the malware, in that the functions describe potential actions taken by the malware.

After being reduced to two dimensions using the UMAP technique, three clustering algorithms were presented for optimization. Two of the algorithms, DBSCAN and OPTICS, are closely related, and work to cluster data based on density. The third, k-means, is a more naïve but well known approach that does not consider density.

Values for the most important parameters in each of these clustering algorithms were chosen by an evolutionary multi-objective algorithm called NSGA-III. This algorithm creates a diverse population of individuals according to a set of reference points.

After implementing the system on Apache Spark, a distributed computation platform, algorithms to compute fixed-length signatures for each of the clusters of malware were presented.

Finally, I defined and demonstrated all of the classification and clustering metrics by which the final results would be evaluated.

Chapter 4

Evaluations

This chapter discusses the results and observations of experiments conducted for this thesis. In each evolutionary process, the optimal clustering parameters are determined for the underlying dataset. In this way, a malware analyst does not need to directly apply the clustering algorithms, but can instead tune the COUGAR system parameters based upon their desired search criteria.

An overview of the evaluations is given in Table 4.1, which delineates the high-level details of all experiments run and analyses conducted for this chapter, and the subsection in which they are discussed.

For the sake of clarity, in this chapter, I refer to the three objectives defined in Subsection 3.4.1 as 'the objectives', and refer them numerically for brevity. Objective 1 is then the maximized count of highly homogeneous clusters, Objective 2 is the minimized sum of sum squared error across all clusters, and Objective 3 is the maximized median cluster size.

In Section 4.1, I detail the preliminary experiments on a small dataset of 3,000 samples. Section 4.2 scales the experiments to a larger dataset of 10,000 samples, and incorporates distributed computation and signature generation. The chapter concludes with a short summary in Section 4.3.

4.1 Preliminary Experiments on 3,000 Samples

To obtain optimal parameters for the clustering algorithms, they were each trained on 2,000 samples randomly selected from EMBER. Accounting for the stochasticity of genetic algorithms, this process was repeated ten times. I refer to each of these training processes as a 'run'.

After completing these runs, statistical analysis was performed via analysis of variance (ANOVA) tests [66], to determine significant differences from the mean for each objective. For those that reject the null hypothesis, additional post hoc testing

Samples	Runs per Algo.	Clus. Algo.	Eval. Type	Subsec.
2,000	10	D, O, k	Metrics on train data	4.1.1
-	-	-	ANOVA/Tukey	4.1.2
3,000	10	D	Metrics on train/test data	4.1.3
2,000	1	D, O, k	Sign. length	4.2.2
10,000	10	D, O, k	Metrics on data	4.2.1
-	-	-	Sign. rep. analysis	4.2.2
-	-	-	Sign. discussion	4.2.2
-	-	-	Wall-clock runtime analysis	4.2.3

Table 4.1: All experiments/analyses conducted

was conducted. This yielded a leading algorithm, which was then subject to additional training, before introduction of a testing dataset of 1,000 samples for labelling. From this labelled data, final metrics were calculated.

Each individual produced over all training runs was ranked according to their ascending objective scores. The ordering was determined by the count of highly homogeneous clusters, deferring to summed SSE, and then median size, in case of ties. In addition, a number of metrics were calculated to determine how well labelled these clusters are.

The labelling process proceeded as follows. For each output cluster, all sample record information was gathered, including the original AVClass. Then, each sample in the cluster had a predicted label assigned according to the majority vote of original AVClass labels in the cluster.

With these predicted and actual AVClass labels, weighted F-Score, homogeneity, completeness, and V-Measure were calculated. The top three clusterings for each algorithm, with these statistics, can be seen in Table 4.2. This table is discussed in the following subsection.

In addition, for a more general view of the population, Figures 4.2, 4.3, and 4.4 depict the average score for each objective across the final population in each of the ten runs. Linear regressions are calculated across these values. A shallower slope for a trendline indicates more consistency for the achieved results across runs. These figures are also discussed in the following subsection.

One characteristic associated with all three algorithms is a quick convergence to a local optimum, followed by regular scrubbing until the evolutionary process completes.

Algo.	Obj. 1	Obj. 2	Obj. 3	F-Score	Homo.	Comp.	V-Meas.
DBSCAN 1	**65**	**3.88**	**8**	**0.668**	**0.725**	**0.862**	**0.788**
DBSCAN 2	62	3.87	9	0.678	0.748	0.872	0.805
DBSCAN 3	62	4.42	8.5	0.673	0.733	0.864	0.793
OPTICS 1	**84**	**10.83**	**8**	**0.549**	**0.675**	**0.792**	**0.729**
OPTICS 2	83	10.86	8	0.554	0.677	0.795	0.731
OPTICS 3	83	13.35	7	0.539	0.671	0.780	0.722
***k*-means 1**	**52**	**13.62**	**15**	**0.520**	**0.593**	**0.774**	**0.671**
k-means 2	50	13.52	15	0.528	0.588	0.773	0.668
k-means 3	49	12.78	16	0.526	0.588	0.767	0.666

Table 4.2: Top three unique clusterings per algorithm on 2,000 samples, ranked by ascending objectives

This can be observed in Figure 4.1, where the yellow markers represent the start and end scores of each clustering algorithm. The marker shape is related to the algorithm through the plot legend.

4.1.1 Training Results

In this subsection, I will describe the results achieved by evolving parameters for each clustering algorithm.

DBSCAN

Looking first at trendlines for each objective in Figures 4.2, 4.3, and 4.4, we can see that DBSCAN is relatively flat and averaging 30 homogeneous clusters by the conclusion of each evolutionary process. Its summed error is much the same, sitting just above 2.0. Median cluster size is the more variable objective, with lows of 200 and highs near 400, the trend somewhere in the middle around 325.

As far as its top performers are concerned, DBSCAN achieves over 60 highly homogeneous clusters with each. More notably, it still attains low errors averaging around 4.06. The average median cluster size is 8.5 across the best performers, significantly lower than the average. These scores suggest that those homogeneous clusters are achieved by grouping smaller clusters of data points.

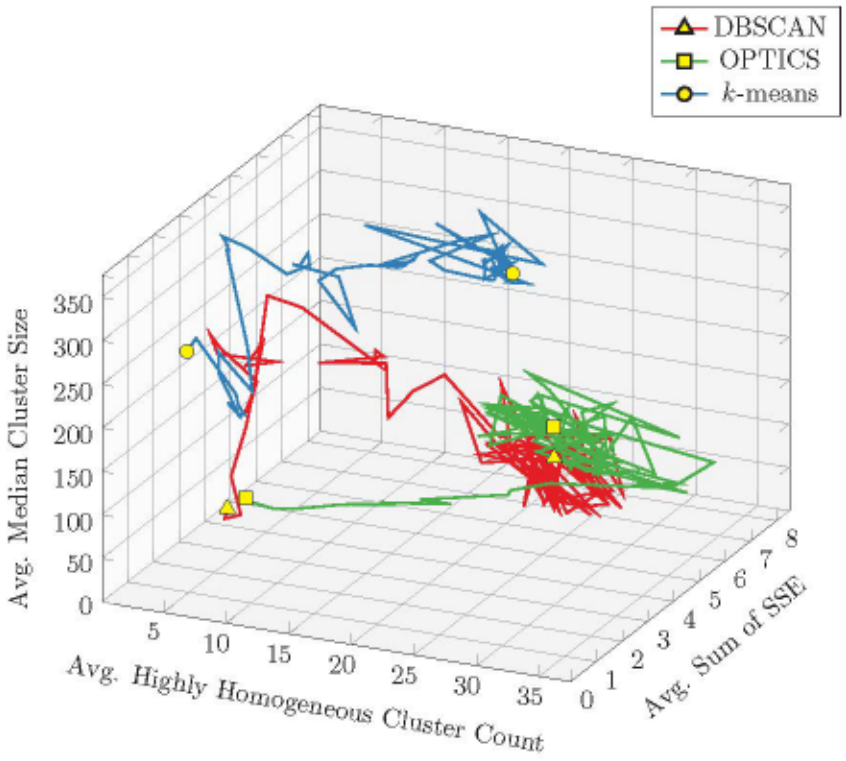

Figure 4.1: Avg. objective performance over one run of each clustering algorithm on 2,000 samples

Looking at the metrics for the labelled data in Table 4.2, we see consistent weighted F-Scores around 0.673. Homogeneity is higher, averaging at 0.735, but completeness is higher still, at 0.866. This seems to indicate that, while having some mislabeled data, the predicted groupings contain most of the instances for their given label. This naturally leads to a V-Measure in the middle, averaging 0.795.

OPTICS

In the trendlines of Figures 4.2, 4.3, and 4.4, OPTICS achieves a marginally higher homogeneous cluster count than DBSCAN, sitting at approximately 34. The variability of this objective is noticeable, however, with lows of 25 and highs approaching 40. In fact, this variability extends to the other objectives. The summed error ranges from below 3.0 to nearly 5.0, with the trend at just under 4.0. The spread on median cluster size is not as dramatic as DBSCAN. It ranges from below 100 to above 200, with the trend around 140.

Top performing clusterings are a mixed bag. The homogeneous cluster counts are

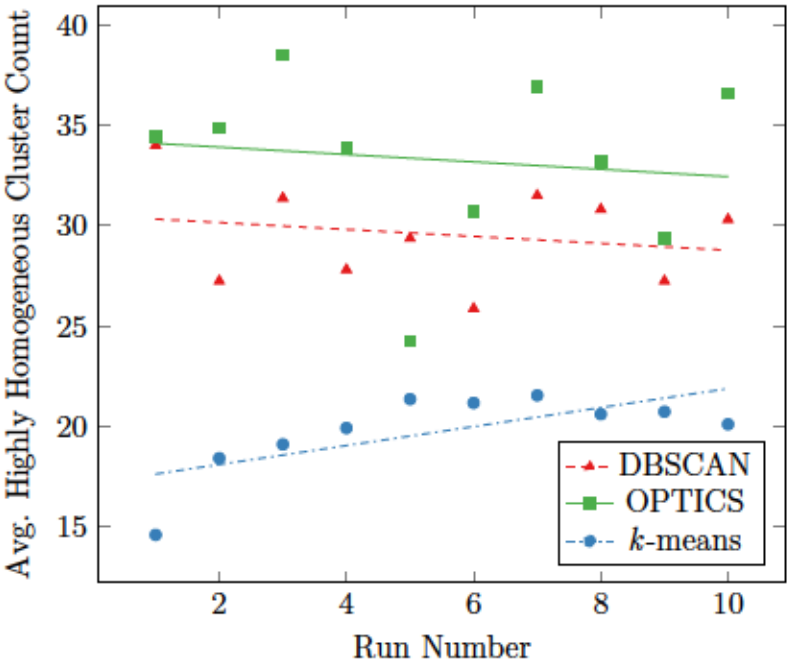

Figure 4.2: Avg. count of highly homogeneous clusters (obj. 1) over 10 runs on 2,000 samples

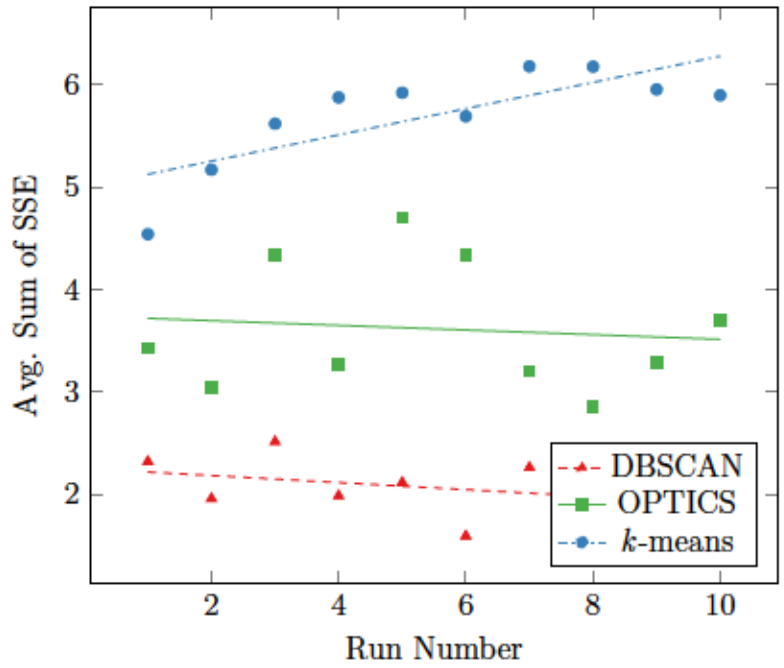

Figure 4.3: Avg. sum of SSE (obj. 2) over 10 runs on 2,000 samples

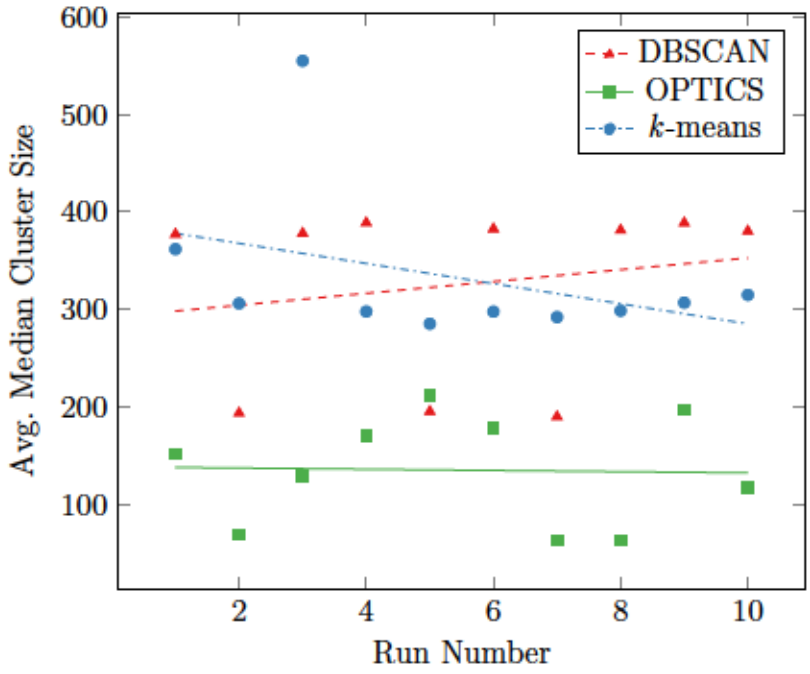

Figure 4.4: Avg. median cluster size (obj. 3) over 10 runs on 2,000 samples

excellent, at 83 and 84, 20 clusters more than DBSCAN. Unfortunately, the summed SSE for those clusterings are nearly triple their DBSCAN peers, averaging 11.68. Median size is much the same as DBSCAN, if slightly lower, with an average of 7.6. All-in-all, these clusterings manage to find more highly homogeneous clusters of similar size, with the consequence of higher noise.

The metrics in Table 4.2 are lower, across the board, relative to DBSCAN. The previously mentioned noise drags the average weighted F-Score down to 0.547. Homogeneity and completeness are more successful, though, averaging 0.674 and 0.789, respectively. Their average V-Measure comes out to 0.727.

k-means

A final look at the trendlines in Figures 4.2, 4.3, and 4.4 show that *k*-means actually has the most consistent homogeneous cluster count, where nine of the ten runs are very close to the trend, at or around 20. This stability is lacklustre, however, as it is also the lowest average cluster count of the three algorithms. This is also true of the summed SSE, the highest of the three algorithms with the average at 5.7. The average median cluster size, on the other hand, is actually competitive with

Obj.	Algo. 1	Algo. 2	$\Delta\mu$	P-Adj.	Reject?
1. Homo.	DBSCAN	k-means	-9.7812	0.001	Yes
1. Homo.	DBSCAN	OPTICS	3.7125	0.030	Yes
1. Homo.	k-means	OPTICS	13.4938	0.001	Yes
2. Sum of SSE	DBSCAN	k-means	3.6401	0.001	Yes
2. Sum of SSE	DBSCAN	OPTICS	1.5532	0.001	Yes
2. Sum of SSE	k-means	OPTICS	-2.0869	0.001	Yes
3. Med. Size	DBSCAN	k-means	6.0812	0.9	No
3. Med. Size	DBSCAN	OPTICS	-190.4063	0.001	Yes
3. Med. Size	k-means	OPTICS	-196.4875	0.001	Yes

Table 4.3: Tukey's HSD test on each of the three objectives

DBSCAN, at an average of 331, and fairly stable across eight of the ten runs. This could be an effect of every k-means parameter needing to be truncated to an integer, so minor mutations of the floating point genes do not result in dramatic changes to the clustering performance.

Given that k-means cannot label data as noise, and must include every point in some cluster, its best clusterings are impressively close to DBSCAN on the homogeneous cluster front, with an average of 50. In fact, the gap between DBSCAN and k-means is smaller than the gap between DBSCAN and OPTICS, at approximately 13 clusters difference. Naturally, from the necessarily clustered data, the median cluster size is also the largest of the three algorithms, at 15.3. Unfortunately, the summed SSE is again the highest of all three algorithms, at 13.31 on average.

With such error, it comes as no surprise that the metrics on predicted labels are the worst of all three algorithms. The weighted F-Score is just under OPTICS, averaging at 0.525. Homogeneity is not much better, at 0.560. Completeness is more successful, though, just under OPTICS at 0.771. Their harmonic mean then averages to 0.668.

4.1.2 Statistical Analysis of Training Results

To determine which algorithm would be subject to further testing, it was necessary to apply statistical analysis tests to the final average of each objective over all ten runs of each clustering algorithm. These averages are given in Tables A.1, A.2, and

A.3. The test of choice was a one-way analysis of variance (ANOVA), which tests the null hypothesis that the averages for each clustering algorithm share a mean. That is, it is assumed that the objective averages for each clustering algorithm are similar. ANOVA then calculates P-Values that indicate whether or not this hypothesis is likely. P-Values less than α, where $\alpha = 0.05$, indicate that the null hypothesis should be rejected, and that the population means differ.

The following P-Values were calculated:

- Obj. 1 — Homogeneous cluster count: $5.83 \cdot 10^{-10}$
- Obj. 2 — Sum of SSE: $5.52 \cdot 10^{-15}$
- Obj. 3 — Median cluster size: $3.61 \cdot 10^{-6}$

With such small P-Values ($P \ll 0.05$), the null hypothesis is rejected for each objective, indicating statistically significant differences from the mean.

ANOVA does not tell us which algorithms are different from each other, however. To determine the difference between means for each algorithm in each objective, I ran Tukey's honestly significant difference (HSD) test [67] on each objective. The results are shown in Table 4.3. For each objective, each pair of clustering algorithms is compared, and the difference between their means and Tukey-adjusted P-Values given. If that P-Value is less than α, we reject the null hypothesis that their means are similar.

These tests support much of what I have discussed in Subsection 4.1.1. Namely, for the number of homogeneous clusters, OPTICS narrowly edges out DBSCAN. For error, the reverse is true. For median cluster size, DBSCAN comes out on top, with a significantly larger median cluster size. In fact, the only comparison that does not show a statistically significant difference is the average median cluster size difference in Table 4.3 between DBSCAN and k-means which, as previously mentioned, is very similar.

Considering these realities, as well as the weighted F-Score, homogeneity, completeness, and V-Measure for each of the algorithms, it becomes clear that DBSCAN is the top performer on this data subset. As a result, it was the natural choice for further evaluations.

Data	Obj. 1	Obj. 2	Obj. 3	F-Score	Homo.	Comp.	V-Meas.
Train Only	60	**3.21**	**9.5**	**0.680**	0.734	0.870	0.796
Test Only	–	–	–	0.671	**0.821**	**0.922**	**0.868**
Combined	**74**	5.42	9.0	0.673	0.727	0.870	0.792

Table 4.4: Objective scores and metrics for the final DBSCAN training/testing run

4.1.3 Testing Results

I ran another ten runs of COUGAR using DBSCAN on the 2,000 EMBER samples, but this time preserving an additional 1,000 samples in the vectorization as testing data. All 160 individuals were scored, and one of the top 16 individuals was chosen, with similar objective scores to the DBSCAN clusterings in Table 4.2. This individual achieved scores of 60, 3.212, and 9.50 over the three objectives, with clustering parameters of `eps`= 0.034971304580205594 and `min_samples`= 5.

At this point, I reloaded the embedding, and fit the new data into it. After re-clustering using the same settings as the chosen individual, I labelled the data as in Section 4.1, and calculated metrics on those predictions.

It is important to note that only approximately 30% of testing data gets clustered, as DBSCAN labels much of it as noise. As well, since clusters which only contain testing data do not get a label assigned, they must be discarded. Accordingly, only about 20% of the testing data is assigned a label. It is for this reason that labels are given decreased prominence in subsequent experiments, and used more for informational purposes than evaluation.

With that said, the results can be observed in Table 4.4. We can see that the testing and training F-Scores are very similar, around 0.67, and this aligns with what we saw for DBSCAN in Table 4.2. Homogeneity and completeness are also nearly identical, with this training data ever so slightly higher than before, with an accompanying V-Measure.

When looking only at the testing data, we can see a slight drop in weighted F-Score, but the highest homogeneity, completeness, and V-Measure across any of our reported values, at 0.821, 0.922, and 0.868, respectively. The implication is that, looking only at unseen data, the resulting groupings are fairly homogeneous, and highly complete, with the unseen data for their respective labels being grouped

together.

Putting it all together, we see that the final count of homogeneous clusters has improved, with median cluster size nearly unchanged, and summed SSE marginally increasing. There is a negligible decrease ($\leq 1\%$) across the label metrics, relative to training data only.

The takeaway here is that, while leaving room for improvement, the testing data predictions are as good on seen as on unseen data.

That said, there were instances of clusters that highlighted the weakness of categorical labels in this research problem. For instance, some cluster labels were decided with a slim majority (e.g.: ten samples of AVClass A versus nine samples of B), while others were decided with a plurality (e.g.: four samples of AVClass A, three samples of B, three of C). In these examples, the precision with respect to AVClass A would be 0.53 and 0.4, respectively, despite the underlying cluster similarity in imports.

For this reason, in subsequent experiments, I decided to include a metric based on the generated signatures, and how well that signature represents the cluster, as described in Subsection 3.7.3. In addition, this decreased emphasis on label performance obviates the need for a split train/test dataset, as the new metric directly evaluates the composition of the clusters, rather than the assigned labels.

4.2 Final Experiments on 10,000 Samples

This second round of experiments was conducted after the COUGAR system was reimplemented in a distributed fashion to operate on an Apache Spark cluster, along with signature generation capabilities, as discussed in Subsections 3.5 and 3.6.

As before, 10,000 samples were randomly selected, and the evolutionary process was conducted ten times per clustering algorithm. Individual ranking and the labelling process was also as discussed in Section 4.1.

In the following subsections, I will discuss the results as calculated from the objectives, label metrics, and the generated signatures, as well as the computation times that result from scaling to a larger dataset.

Algo.	Obj. 1	Obj. 2	Obj. 3	F-Score	Homo.	Comp.	V-Meas.
DBSCAN 1	**225**	**17.253**	**11**	**0.660**	**0.706**	**0.821**	**0.759**
DBSCAN 2	223	19.095	11	0.649	0.692	0.817	0.749
DBSCAN 3	222	19.088	12	0.651	0.695	0.818	0.751
OPTICS 1	**406**	**34.950**	**8**	**0.644**	**0.690**	**0.806**	**0.743**
OPTICS 2	405	34.341	8	0.646	0.695	0.806	0.746
OPTICS 3	405	34.369	8	0.646	0.694	0.806	0.746
***k*-means 1**	**56**	**13.759**	**59**	**0.427**	**0.484**	**0.712**	**0.577**
k-means 2	56	14.109	53	0.434	0.496	0.702	0.581
k-means 3	55	14.246	59	0.430	0.487	0.713	0.579

Table 4.5: Top three unique clusterings per algorithm on 10,000 samples, ranked by ascending objectives

4.2.1 Objective & Labelling Results

This subsection details the objective and labelling results on the clusterings produced from the 10,000 sample dataset with respect to each clustering algorithm.

Table 4.5 shows the top unique clusterings per algorithm, as determined by the objective scores. What is especially striking about these objectives, across all of the algorithms, is the uniformity of the results achieved. To obtain the top three *unique* clusterings, a number of individuals from each algorithm needed to be disregarded, as they were scoring identical to others. This speaks to the consistency of solutions produced in the evolutionary process.

DBSCAN

I begin by looking at the trendlines in Figures 4.5, 4.6, and 4.7. We can see that the ordering of the results for the highly homogeneous count is identical to that shown in Figure 4.2. That is, DBSCAN is the middle performer, averaging just under 100 clusters across all runs. The slope of this line is smaller than previously seen, suggesting that the clustering process is much more stable when provided with the additional data. The summed error trendline is also similar, if higher than before around 6.5. Median cluster size is, in contrast to the previous experiments, the most stable objective, with only a single run failing to achieve a value in close proximity

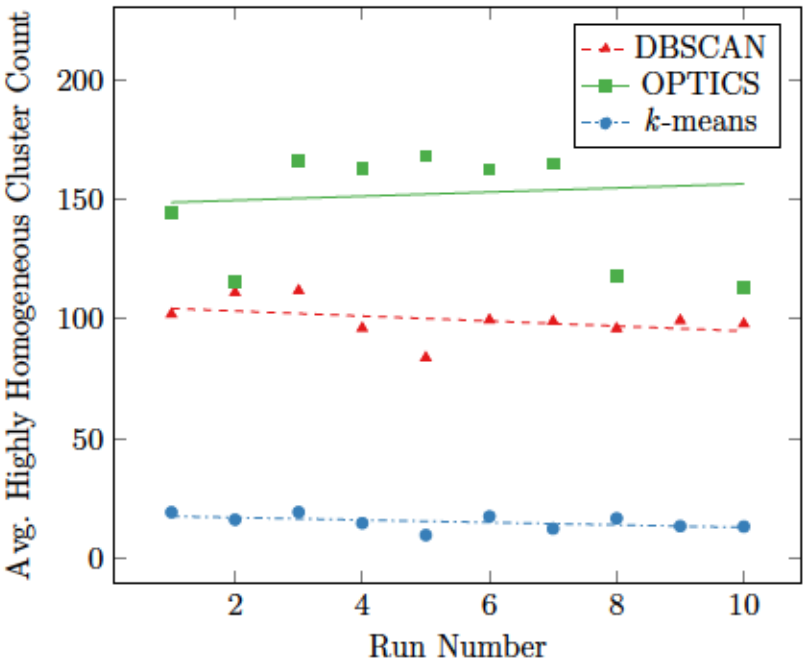

Figure 4.5: Avg. count of highly homogeneous clusters (obj. 1) over ten runs on 10,000 samples

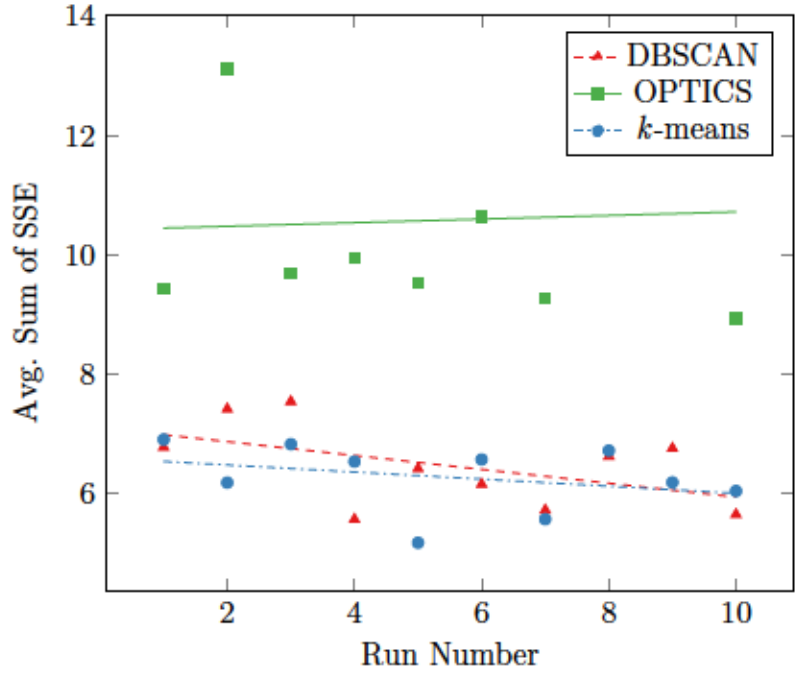

Figure 4.6: Avg. sum of SSE (obj. 2) over ten runs on 10,000 samples

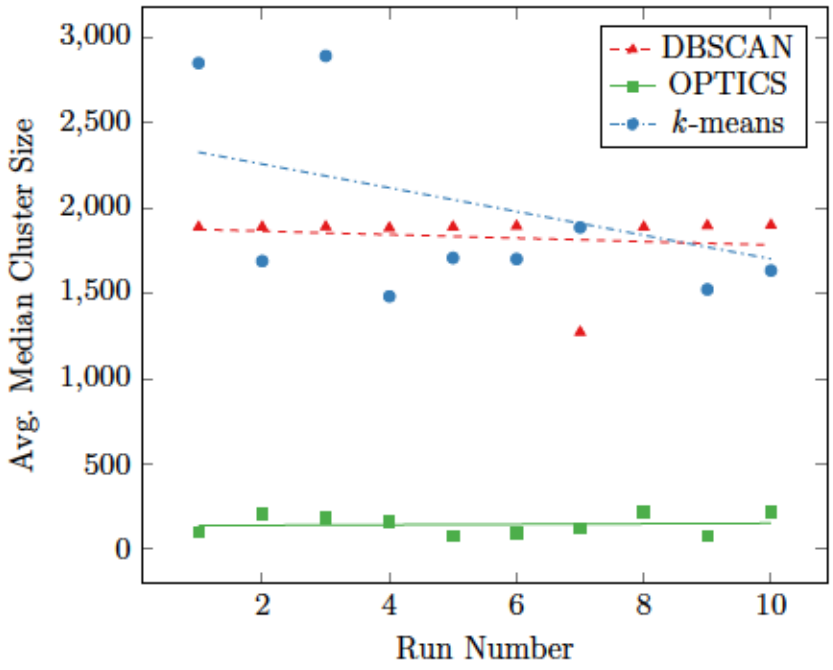

Figure 4.7: Avg. median cluster size (obj. 3) over ten runs on 10,000 samples

to 1,900. This is likely a result of some individuals favouring this objective in the Pareto front, and generating a small number of large clusters at the expense of other objectives.

Pivoting to the top performers in Table 4.5, DBSCAN nearly quadruples the number of highly homogeneous clusters to an average of 223, which is not far from the quintupling volume of the underlying data to be clustered. The error follows this, more than quadrupling to 18. This objective score sounds worse than it actually is though, as it is not scaled by the number of clusters, so is expected to increase as the number of clusters does. A 33% increase in the median cluster size to 11 shows that these clusters are still relatively small groupings.

Compared to the metrics in Table 4.2 for DBSCAN, these top performers are a touch lower across the board. F-Score averages around 0.653, a drop of 0.02, while homogeneity is more affected, dropping by 0.04 to an average of 0.698. Completeness continues the downward trend to end up around 0.819, a decrease of 0.05, and V-Measure is accordingly 0.753, down 0.04. It is encouraging to see that, while the amount of data has increased dramatically, the actual scores are almost identical to the results achieved by DBSCAN on the smaller dataset. Suffice it to say, scaling

the system was highly successful when using DBSCAN, at least according to these metrics.

OPTICS

Starting again with Figures 4.5, 4.6, and 4.7, OPTICS is once again the top performing algorithm when looking at counts of highly homogeneous clusters, at an average of 153. This is more than a 50% increase relative to DBSCAN, in stark contrast to the five cluster difference between these algorithms on the smaller dataset. As well, this average is quintuple that of the average for 2,000 samples, exactly in line with the increase in dataset size. The only caveat is that these results are more variable than for DBSCAN, with the extreme low and high at 113 and 210, respectively. This variability also extends to the summed error, which has tripled to 10.6, the highest of the three algorithms. The upside here is that the error is growing at a rate much lower than the cluster count. Finally, the average median cluster size has grown at a moderate pace, from 135 to 146. This is much more reasonable than the 1,900 average seen from DBSCAN and k-means, emphasizing the applicability of OPTICS to larger datasets.

The objective scores for OPTICS in Table 4.5 continue to deliver good news. The count of highly homogeneous clusters has nearly quintupled to an average of 405, demolishing the competition with a score almost double that of DBSCAN and octuple that of k-means. The error has also increased in kind, to an average of 34.5. Interestingly, the median cluster size for top performers was identical to that of the previous dataset.

The narrative for the labelling metrics is opposite that of DBSCAN: scores have actually increased, relative to Table 4.2. F-Score makes a moderate jump of 0.1 to an average of 0.645, while homogeneity, completeness, and therefore V-Measure, are much more tempered with an increase of approximately 0.02 each. These metrics for OPTICS are then within striking distance of DBSCAN, making it much more competitive than on the smaller dataset.

k-means

The trendlines do not paint a flattering picture of k-means on this dataset. For the count of highly homogeneous clusters, the average across all runs has dropped from 20 in Figure 4.2 to 15 in Figure 4.5. There is respite in Figure 4.6, as the summed error has increased by only 0.6, and is in line, quite literally, with the results achieved by DBSCAN. As in Figure 4.4, the results in Figure 4.7 show k-means tangling with DBSCAN at an average median cluster size of 2,000. Unfortunately, this objective is much less meaningful for k-means, as all of the data is clustered anyway. In addition, the stability shown on the smaller dataset is no longer present, with values ranging from 1,524 all the way up to nearly 3,000.

The picture is not any rosier when one considers the objective scores for the top three individuals in Table 4.5. The averaged count of highly homogeneous clusters increased by a measly 10%, to 55, despite a 500% increase in data to be clustered. The error was nearly unchanged, at 14, while the median cluster size quadrupled to 57. Once again, this performance jump is undercut by the inclusive nature of the algorithm.

A final look at the labelling scores for the best individuals shows that every metric except for completeness drops by approximately 0.1 compared to the results shown in Table 4.2. Completeness is still lower by 0.06. All-in-all, the performance difference between DBSCAN, the algorithm with the best scores on these metrics, and k-means is substantial. The gap between average F-Scores is 0.223, and the corresponding V-Measure gap is 0.174. These results seal k-means fate as the lowest performing clustering algorithm on this problem.

4.2.2 Signature Results

In the previous subsections, I judged the success of the clustering algorithm based on the assigned labels, using weighted F-scores, homogeneity, completeness, and V-Measures. This worked well enough from a classification perspective, but this task is not truly one of classification, but clustering. We want to know what is similar about each sample. In classification, mixed labels are unwelcome. Here, they are potentially *desired*. As a result, the dominating metric in these evaluations is the representation for each cluster. That is, the percentage of samples in each cluster that have the same

Length	DBSCAN	OPTICS	k-means	Average
3	42.93	35.64	45.375	42.93
5	40.11	37	39.11	38.74
7	40.44	40.86	43.91	41.74

Table 4.6: Avg. homogeneous cluster count signature length comparison

imports.

In this subsection, I will detail the process by which the signature length was determined, before discussing the results of the signature generation process. This includes the signature representation metric, as well as a survey of the produced signatures, highlighting strengths and weaknesses of the approach.

Fixed-Length Parameter Selection

As previously discussed, it was important to choose a reasonable length for the signature to balance interpretability and specificity. Since the signature generation process will drop clusters that cannot produce a unique signature, this parameter is essential to preserving as much data as possible. Han and Olivier [11] suggest $n = 5$ as providing the best balance between generic and specific. Accordingly, it was a natural place to start.

Using 2,000 randomly selected malware from the EMBER dataset, the COUGAR system is run three times for each clustering algorithm, with the signature length set to 3, 5, and 7. The resulting counts of highly homogeneous clusters were collected, along with the number of dropped clusters, and divided by the number of individuals contained in the final population of each run. This yields comparable values of average cluster counts and average dropped clusters. The length should then be chosen such that the former objective is maximized, while the latter is minimized. This emperical process is not guaranteed to give consistent results, as it is dependant on the underlying malware, but should suffice as a reasonable estimate. Logically, a longer signature is less likely to be duplicated, so this process was more about finding an acceptable threshold than an absolute best possible value.

The averaged counts for each clustering algorithm, as well as their overall average,

Length	DBSCAN	OPTICS	k-means	Average
3	0.44	0.81	0.38	0.55
5	0.22	0.33	0.22	0.26
7	0.0	0.14	0.0	0.05

Table 4.7: Avg. dropped cluster count signature length comparison

is given in Tables 4.6 and 4.7. Surprisingly, a length of 5 leads to the worst performance with respect to the count of highly homogeneous clusters, while 3 marginally beats 7, achieving a single cluster more on average.

On the dropped cluster front, however, it is as one would expect: a longer signature is more likely to be unique. The count of dropped clusters is at least cut in half by increasing the signature size from 3 to 5, and again from 5 to 7. In fact, when the signature size is set to 7, DBSCAN and k-means do not drop any clusters at all, and OPTICS only drops a single cluster across all individuals. As a result, the signature size was set to 7 for this second round of experiments.

Signature Metrics

To begin, the surviving individuals of the evolutionary processes produce clusterings that routinely exhibit excellent results. When one considers the number of individuals that score at least 90% across their averaged signature representation, the numbers speak for themselves:

- $^{89}/_{91}$ DBSCAN individuals $\geq 90\%$
- $^{115}/_{115}$ OPTICS individuals $\geq 90\%$
- $^{39}/_{81}$ k-means individuals $\geq 90\%$

DBSCAN has only two individuals below 90% (80% and 72%), while OPTICS has none. Even k-means, which has been demonstrated to break down on this larger dataset, manages a median representation of 88.8%.

Since so many solutions are highly ranked by this metric, I have chosen three differing individuals that each score $\geq 95\%$ to illustrate the diversity of solutions obtained through the fronts, and therefore the tunability of the system. These can be seen in Table 4.8.

Algo.	Rep.	Obj. 1	Obj. 2	Obj. 3	F-Score	Homo.	Comp.	V-Meas.
DBSCAN 1	100	187	8.487	9	0.735	0.787	0.879	0.830
DBSCAN 2	100	118	0.825	9	0.793	0.845	0.924	0.882
DBSCAN 3	98.84	41	6.172	65	0.515	0.555	0.789	0.651
OPTICS 1	100	223	3.672	7	0.760	0.815	0.905	0.858
OPTICS 2	100	156	1.654	8	0.794	0.836	0.918	0.875
OPTICS 3	98.15	90	16.732	33	0.556	0.619	0.768	0.686
k-means 1	97.32	56	13.759	59	0.427	0.484	0.712	0.577
k-means 2	96.19	33	9.957	69	0.402	0.458	0.701	0.554
k-means 3	95.73	31	11.157	87.5	0.406	0.460	0.704	0.556

Table 4.8: Selected elite clusterings per algorithm on 10,000 samples, ranked by cluster representation

The first DBSCAN individual achieves a cluster representation of 100%, with objective scores better or nearly as good as the best given in Table 4.5. This individual falls within the part of the non-dominated front that prioritizes counts of highly homogeneous clusters, finding 187. The second individual also manages a representation of 100%, but instead promotes error minimalization, with a minuscule 0.825. While they do not compete for the top raw objective scores, their 'jack-of-all-trades' versatility results in heretofore unseen F-Scores and V-Measures, with the second individual achieving 0.793 and 0.882, respectively. The last individual, on the other hand, has a slightly lower representation, but maximizes the median cluster size to 65, significantly higher than seen from other DBSCAN individuals. An individual like this can be seen as 'casting a wider net' on the data, by sacrificing top performance to cluster more data. This represents a key strength of the COUGAR system, in that its output can be filtered to the desires of the searcher.

The narrative for chosen OPTICS individuals follows the same trends, with slight differences in objective scores and metrics. Counts of highly homogeneous individuals and error are up, while median cluster sizes are down, relative to DBSCAN. Labelling metrics are almost universally higher, with the exception of the second individual, which is slightly lower than its DBSCAN equivalent on V-Measure. The upside is that it narrowly edges out that same DBSCAN individual for the highest reported F-Score across all highlighted individuals, at 0.794. With all of this said, OPTICS

demonstrates that it can produce highly representative clusters and outpace DBSCAN cluster counts, while also keeping the error at acceptable levels.

These trends apply less so to the selected k-means individuals. The first individual is the same highest ranked solution as in Table 4.5, with a representation of 97.3%. The second and third individuals reduce their homogeneous cluster count in the interest of lowered error and higher median cluster size, respectively. They also achieve respectable representations of 96.2% and 95.7% across their clusters. That said, the errors are higher than most of the selected DBSCAN/OPTICS individuals with much lower counts of homogeneous clusters. As a result of the algorithm's propensity for large clusters, the median cluster size is always relatively high, an upside that is undercut by labelling metric scores as low as half that of DBSCAN or OPTICS.

Discussion of Produced Signatures

Since an explicit goal of the signatures generation methodology was interpretability, one has to ask the question: can we understand them? The answer depends on the cluster. Some clusters have indeed been assigned signatures suggestive of the underlying behaviour, while others are more cryptic. For example, in one run of DBSCAN, out of 175 clusters, 150 of them are considered 'legible'. That is, their cluster signature consists of at least four functions, and not more than three of those functions are ordinal functions (discussed below). This heuristic is sufficient to determine whether the functions are suggestive or cryptic, for the purposes of exploration. I will explore three of the former, and two of the latter. It should be noted, however, that every one of the 175 cluster signatures in this example run is 100% representative of its respective cluster, regardless of legibility. The MD5s and AVClass labels for each sample in these clusters can be found in Appendix B.

The first cluster under observation consists of 93 samples, with the majority (89) having the AVClass label `flystudio`. The chosen signature consists of the following seven functions imported from corresponding libraries:

- `comctl32.dll,imagelist_duplicate`
- `oleaut32.dll,ordinal165`

- `winmm.dll,midioutunprepareheader`
- `winmm.dll,midistreamclose`
- `winmm.dll,midistreamopen`
- `winmm.dll,midistreamproperty`
- `winmm.dll,midistreamstop`

The first two functions are unrelated to the rest, but the final five are all from the same library, enabling access to multimedia functionality in Windows. As can be seen from the function names, these functions allow for MIDI (Musical Instrument Digital Interface) streams to be controlled which implies that these malware may play some sort of music or sound. While this behaviour is not particularly malicious, it is still a distinct behaviour by which these malware have been grouped.

Moving to a behaviour that is decidedly more nefarious, the second cluster is composed of ten samples labelled as `emotet`. This is a well-known persistent threat that is especially destructive [68]. The signature for this cluster is as follows:

- `crypt32.dll,certcreatectlcontext`
- `crypt32.dll,certduplicatestore`
- `crypt32.dll,certfindctlinstore`
- `crypt32.dll,cryptmemrealloc`
- `crypt32.dll,cryptmsgduplicate`
- `crypt32.dll,cryptmsgupdate`
- `crypt32.dll,cryptsignmessage`

These functions are all from the Windows library enabling access to the CryptoAPI. This could be suggestive of ransomware behaviour, and these functions could be used as part of the user file encryption process. Another possibility could be that these malware encrypt some of their own binaries, and use these functions to deobfuscate themselves. Regardless, the behaviour here is still coherent.

In the final interpretable cluster, all but one of the 25 have been assigned the label `ramnit`. This family is known to hook into the Windows service host process, according to Microsoft [69]. With that in mind, the following signature functions seem appropriate:

- `advapi32.dll,changeserviceconfiga`
- `advapi32.dll,createservicea`
- `advapi32.dll,queryserviceconfiga`
- `advapi32.dll,registerservicectrlhandlera`
- `advapi32.dll,startservicea`
- `advapi32.dll,startservicectrldispatchera`
- `kernel32.dll,flushviewoffile`

The first six are all imported from the same library, and related to the creation or management of Windows services, just as previously mentioned. Once again, this signature provides an 'at a glance' snippet of the underlying malware similarity.

Looking instead at clusters which do not have a discernible behaviour, we have a cluster of 95 samples, 94 of which are labelled as `zbot`. While all of the signature functions are from the same library, their function 'names' leave much to be desired:

- `mfc42.dll,ordinal1849`
- `mfc42.dll,ordinal1942`
- `mfc42.dll,ordinal2583`
- `mfc42.dll,ordinal303`
- `mfc42.dll,ordinal3371`
- `mfc42.dll,ordinal3399`
- `mfc42.dll,ordinal3641`

These functions are not imported by name, but instead their ordinal number, identifying their position within the library's export table [70]. These are impossible to decipher without expert knowledge or disassembly software which is capable of translating the ordinal numbers back to the appropriate symbol, such as IDA Pro[1]. This is not a failing of the clustering methodology so much as the underlying features. For these malware, additional static or dynamic analysis feature might produce more interpretable results.

In the final cluster, we have an example of the least informative signature pattern: a small number of generic imports. In this case, it is only a single function. Of the 63 samples in this cluster, 61 of them are labelled as `wannacry`, a ransomware family. The function is:

- `kernel32.dll,closehandle`

This function simply closes an open handle to an object, a trivial similarity between samples. Unfortunately, this is a weakness of the signature generation methodology. Functions that are present in, for example, 99% of samples might have been more informative than this single function that is present in 100%.

Even in these two examples with less informative signatures, the clusters themselves are composed of many malware that share the same label. Accordingly, a refinement in signature generation that can identify and recalculate trivial signatures might mitigate this sort of issue in future work.

4.2.3 Computation Time

In this second round of experiments, parallel and distributed computation was stated as an important part of scaling the system. Parallel computing can allow for tasks to complete in the fraction of the time they would take if computed in sequence. With that said, distributed resiliency is a major benefit of Spark, but not strictly necessary. To that end, these experiments were conducted on a powerful server[2]running Spark in a pseudo-distributed configuration. Given the population size of 16, up to 16 executors would be deployed during periods of parallel computation.

[1] `https://www.hex-rays.com/products/ida/`

[2] Four 6-core dual-threaded Intel Xeon E7350 CPUs @ 1.87GHz, for a total of 48 logical processors, and 1TB of RAM.

Clustering Algorithm	Mean	Median	Standard Deviation
DBSCAN	435.61	437.78	42.49
OPTICS	**42.02**	**41.38**	**2.66**
k-means	393.73	355.58	73.52

Table 4.9: Elapsed time statistics, in minutes, on 10,000 samples

For each of DBSCAN, OPTICS, and k-means, all execution times are listed in Tables A.4, A.5, and A.6, respectively. An overview of the results is found in Table 4.9.

DBSCAN takes the longest on average, at 435 minutes (7.25 hours), with only a two minute difference between mean and median run times. The standard deviation is 43 minutes, equivalent to 9.75% of the mean time. k-means is similar, but slightly faster at 393 minutes (6.6 hours), with a significantly wider gap between the mean and median times, at 38 minutes. This corresponds to a standard deviation of 73 minutes, nearly double that of DBSCAN, and 18.67% of the mean time. This is almost certainly due to the changing number of initializations and iterations specified as part of the k-means evolutionary process.

OPTICS truly shines in this evaluation, however, as it completes its runs in 42 minutes on average, a speed tenfold faster than the other algorithms. The difference between the mean and median speed is less than a minute, and the standard deviation is only 2.7 minutes. The latter measurement is a small fraction of the others, and a much lower percentage of the mean (6.3%). As a result, OPTICS is far and away the fastest clustering algorithm on the larger dataset, and most stable with respect to runtimes. This supports the assertion of the Scikit-Learn developers that their OPTICS implementation is more suitable to large datasets.

It is also important to consider the objective performance of the algorithms in the context of their differing runtimes. As shown in Tables 4.5 and 4.8, OPTICS competes with or exceeds DBSCAN across every metric, while k-means is not competitive. In conclusion, OPTICS allows for very good results to be computed in one-tenth the time of the other compared algorithms.

4.3 Summary

In this chapter, I discussed the experiments performed and the results of those experiments.

In a round of preliminary experiments, the COUGAR system was trained on 2,000 samples, with 1,000 held back for testing. Three clustering algorithms were compared across the objective scores for the multi-objective genetic algorithm, as well as the aforementioned labelling metrics. After evaluating the training results with statistical analysis tests, DBSCAN was selected for further testing, and showed stable performance on the testing dataset.

After reimplementing the system using Apache Spark, the second and final round of experiments replicated the first but on a larger dataset of 10,000 samples. To account for labelling issues in the first round, a new metric was introduced based on the signatures being generated. The evaluations showed that the system scales to the larger dataset. DBSCAN and k-means operate at similar speeds, but the former is significantly more successful than the latter across the objectives and metrics. The performance of OPTICS exceeds that of DBSCAN in many cases, while it runs in much less time. The most successful clusterings achieve F-Scores of 0.79 and V-Measures of 0.88.

The signatures generated are tuned to the ideal length by repeating the first round experiments with an eye for the number of cluster signatures that collide. With the length chosen, COUGAR generates cluster signatures that represent the vast majority of samples contained within. This applies across all three algorithms, with 243 of the 287 (85%) output individuals capable of creating clusterings with averaged signature representations of at least 90%.

Chapter 5

Conclusion

In this thesis, I studied the problem of clustering malicious software and generating signatures to describe those clusters. This contribution was enabled by the implementation of a system called COUGAR, which marries natural language processing methodology and genetic optimization.

COUGAR reduces high-dimensional behavioural data from malware to two-dimensions, and optimizes the clustering process for those embeddings using a multi-objective genetic algorithm. This behavioural data is provided by imported functions extracted from the malicious binaries, a form of static analysis. The imports imply potential behaviours by considering the use of third-party libraries that are difficult to obfuscate without breaking compatibility with the library. The objectives to be optimized include such desirable cluster properties as homogeneity, low error, and cluster size. Signatures are then assigned based on the commonalities in the underlying cluster features, emphasizing computational speed and intelligibility.

The data for these experiments was provided by EMBER, an open-source dataset containing pre-extracted malicious features, such as byte histograms, function imports, exports, AV labels, and other identifying metadata. While this information was pre-extracted, and the system tested on only one type of feature, the methodology is flexible enough to accommodate other features with little adaptation.

Two rounds of experiments were performed, on small and large datasets, where the system was reimplemented for the second set of experiments to support parallel, distributed computation on scalable compute clusters. In these experiments, three clustering algorithms, DBSCAN, OPTICS, and k-means, were directly compared to ascertain the performance differences across objective scores, labelling metrics, cluster signature representation, and computation time.

The results indicate that DBSCAN and OPTICS both produce respectable clusterings that are supported by the detailed metrics. k-means manages some success on

the smaller dataset, but has more difficulty finding acceptable clusters on the larger dataset. The most successful clusterings achieve F-Scores of 0.79 and V-Measures of 0.88. The signatures produced during these experiments are highly representative of the underlying clusters, and can be interpreted to discover shared behaviours of the malware. 85% of the individuals in the final populations were capable of creating clusterings with averaged signature representations of at least 90%. If k-means is discarded, 99% of individuals achieve that distinction.

5.1 Future Work

The scope of this thesis was to survey and join successful techniques observed in the literature for the purposes of building and demonstrating a proof-of-concept automated malware clustering and signature generation system. As such, there are a number of areas for growth present in the system that encourage further research. I will enumerate ideas that would contribute from an operational standpoint, as well as areas for exploration.

There are weaknesses to the signature generation approach. As mentioned, not all signatures are of the desired length (7), owing to the term selection process. This could be alleviated by considering the size of the maximal signature in the generation process, and ensuring that the length of the maximal signature is equal to or greater than the desired length. This could be enabled by selecting functions with a range of term frequencies, rather than the single most occurring frequency in the cluster. As well, the underlying functions can be difficult to understand. This could be addressed by employing a variety of malware behaviour features, rather than just function imports, and using those most informative.

The similarity comparison process itself has scaling issues, owing to the quadratic time complexity of cosine similarity, and the vectorization process. The former is still a bottleneck of the current implementation of the system, as these comparisons are made only on a single core. The latter is not presently a concern, but the growing dictionary will almost certainly prove problematic when sample counts reach higher orders of magnitude.

To address the scaling problem of dictionary sizes when vectorizing samples, the hashing trick could be employed. By hashing the features into a matrix of finite size,

a dictionary would no longer be required. Unfortunately, this would undercut the signature generation capabilities of the system, so is not an end-all solution.

As far as cosine similarity is concerned, a more powerful processing core would suffice for some time, but would eventually need to give way to an alternative methodology or implementation. The latter could be done using some sort of hardware acceleration, such as computing on a graphics processing unit (GPU), or could itself be distributed across a compute cluster.

In contrast, rather than speeding up the comparison process, the number of comparisons could be reduced by defining a representative sample for each cluster, as suggested by Lee et al. [7]. In doing so, clusters that are obviously different and do not merit further comparison could be disregarded during the evaluation process by first comparing to the representative sample, and returning if below a particular threshold.

Finally, the clustering algorithm itself could be reconsidered. HDBSCAN, a hierarchical variant of DBSCAN, was not explored in this work, and could give performance benefits by negating the ε determination process, similar to OPTICS. In addition, HDBSCAN supports soft clustering, where degrees of membership for each sample are assigned. This may provide another analysis angle in the structure of the cluster.

Notwithstanding these areas for improvement, the experiments in this thesis demonstrate the potential that exists in employing multi-objective genetic algorithms for tuning clustering algorithms in the domain of malware identification. The COUGAR system stands as proof of this concept.

Bibliography

[1] T. Daigle, "'Definite uptick': Global wave of ransomware attacks hitting canadian organizations," *CBC News*, Oct 2019. [Online]. Available: https://www.cbc.ca/news/technology/more-ransomware-canada-1.5317871

[2] S. Gallagher, "DoS attack on major DNS provider brings Internet to morning crawl," *Ars Technica*, Oct. 2016. [Online]. Available: https://arstechnica.com/information-technology/2016/10/dos-attack-on-major-dns-provider-brings-internet-to-morning-crawl/

[3] B. Krebs, "Who is anna-senpai, the mirai worm author?" *Krebs on Security*, Jan. 2018. [Online]. Available: https://krebsonsecurity.com/2017/01/who-is-anna-senpai-the-mirai-worm-author/

[4] J. O'Malley. (2018, Jul.) The Internet of Things will thrive on 5G technology. [Online]. Available: https://www.verizon.com/about/our-company/5g/internet-things-will-thrive-5g-technology

[5] J. Jang, D. Brumley, and S. Venkataraman, "Bitshred: Feature hashing malware for scalable triage and semantic analysis," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. New York, NY, USA: ACM, 2011, pp. 309–320. [Online]. Available: http://doi.acm.org/10.1145/2046707.2046742

[6] H. L. Duarte-Garcia, A. Cortez-Marquez, G. Sanchez-Perez, H. Perez-Meana, K. Toscano-Medina, and A. Hernandez-Suarez, "Automatic malware clustering using word embeddings and unsupervised learning," in *2019 7th International Workshop on Biometrics and Forensics (IWBF)*, May 2019, pp. 1–6.

[7] T. Lee, B. Choi, Y. Shin, and J. Kwak, "Automatic malware mutant detection and group classification based on the n-gram and clustering coefficient," *The Journal of Supercomputing*, vol. 74, no. 8, pp. 3489–3503, Aug. 2018. [Online]. Available: https://doi.org/10.1007/s11227-015-1594-6

[8] Z. Wilkins and N. Zincir-Heywood, "COUGAR: Clustering Of Unknown malware using Genetic Algorithm Routines," in *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, ser. GECCO '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1195–1203. [Online]. Available: https://doi.org/10.1145/3377930.3390151

[9] J. Newsome, B. Karp, and D. Song, "Polygraph: automatically generating signatures for polymorphic worms," in *2005 IEEE Symposium on Security and Privacy (S P'05)*, 2005, pp. 226–241.

[10] X. Zhang and Z. Xu, "On the feasibility of automatic malware family signature generation," in *Proceedings of the First Workshop on Radical and Experiential Security*, ser. RESEC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 6972. [Online]. Available: https://doi.org/10.1145/3203422.3203430

[11] X. Han and B. Olivier, "Interpretable and adversarially-resistant behavioral malware signatures," in *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, ser. SAC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 16681677. [Online]. Available: https://doi.org/10.1145/3341105.3373854

[12] Canadian Centre for Cyber Security. (2018, Dec.) Cyber Threat and Cyber Threat Actors. [Online]. Available: https://cyber.gc.ca/en/guidance/cyber-threat-and-cyber-threat-actors

[13] ——. (2018, Dec.) Cyber Threat Activities. [Online]. Available: https://cyber.gc.ca/en/guidance/cyber-threat-activities

[14] T. Moffitt. (2016, Oct.) Source Code for Mirai IoT Malware Released. [Online]. Available: https://www.webroot.com/blog/2016/10/10/source-code-mirai-iot-malware-released/

[15] P. Arntz. (2017, Mar.) Explained: Packer, Crypter, and Protector. [Online]. Available: https://blog.malwarebytes.com/cybercrime/malware/2017/03/explained-packer-crypter-and-protector

[16] D. Kirat, G. Vigna, and C. Kruegel, "Barebox: Efficient malware analysis on bare-metal," 12 2011, pp. 403–412.

[17] VIPRE Labs. (2020) Analysis of Ursnif. [Online]. Available: https://labs.vipre.com/analysis-of-ursnif/

[18] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero, "AVClass: A Tool for Massive Malware Labeling," in *Research in Attacks, Intrusions, and Defenses*, F. Monrose, M. Dacier, G. Blanc, and J. Garcia-Alfaro, Eds. Cham: Springer International Publishing, 2016, pp. 230–253.

[19] J. Schaffer and L. Eshelman, "On crossover as an evolutionary viable strategy," 01 1991.

[20] H. Faridi, S. Srinivasagopalan, and R. Verma, "Performance evaluation of features and clustering algorithms for malware," 11 2018, pp. 13–22.

[21] R. Pirscoveanu, M. Stevanovic, and J. M. Pedersen, "Clustering analysis of malware behavior using self organizing map," in *2016 International Conference On Cyber Situational Awareness, Data Analytics And Assessment (CyberSA)*, June 2016, pp. 1–6.

[22] G. Acampora, M. L. Bernardi, M. Cimitile, G. Tortora, and A. Vitiello, "A fuzzy clustering-based approach to study malware phylogeny," in *2018 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*, July 2018, pp. 1–8.

[23] N. Kurinjivendhan and K. Thangadurai, "Modified k-means algorithm and genetic approach for cluster optimization," in *2016 International Conference on Data Mining and Advanced Computing (SAPIENCE)*, 2016, pp. 53–56.

[24] M. Anusha and J. G. R. Sathiaseelan, "An enhanced k-means genetic algorithms for optimal clustering," in *2014 IEEE International Conference on Computational Intelligence and Computing Research*, 2014, pp. 1–5.

[25] S. Irfan, G. Dwivedi, and S. Ghosh, "Optimization of k-means clustering using genetic algorithm," in *2017 International Conference on Computing and Communication Technologies for Smart Nation (IC3TSN)*, 2017, pp. 156–161.

[26] S. Al-Malak and M. Hosny, "A multimodal adaptive genetic clustering algorithm," in *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion*, ser. GECCO '16 Companion. New York, NY, USA: Association for Computing Machinery, 2016, p. 14531454. [Online]. Available: https://doi.org/10.1145/2908961.2931633

[27] A. Mukhopadhyay, U. Maulik, and S. Bandyopadhyay, "Multiobjective genetic clustering with ensemble among pareto front solutions: Application to mri brain image segmentation," in *2009 Seventh International Conference on Advances in Pattern Recognition*, 2009, pp. 236–239.

[28] C. Bacquet, A. N. Zincir-Heywood, and M. I. Heywood, "Genetic optimization and hierarchical clustering applied to encrypted traffic identification," in *2011 IEEE Symposium on Computational Intelligence in Cyber Security (CICS)*, 2011, pp. 194–201.

[29] W. Jian-Xiang, L. Huai, S. Yue-hong, and S. Xin-Ning, "Application of genetic algorithm in document clustering," in *2009 International Conference on Information Technology and Computer Science*, vol. 1, 2009, pp. 145–148.

[30] U. Zurutuza, R. Uribeetxeberria, and D. Zamboni, "A data mining approach for analysis of worm activity through automatic signature generation," in *Proceedings of the 1st ACM Workshop on Workshop on AISec*, ser. AISec '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 6170. [Online]. Available: https://doi.org/10.1145/1456377.1456394

[31] S. Choi, J. Lee, Y. Choi, J. Kim, and I. Kim, "Hierarchical network signature clustering and generation," in *2016 International Conference on Information and Communication Technology Convergence (ICTC)*, 2016, pp. 1191–1193.

[32] W. Wang, X. Wang, H. Lu, and J. Su, "Automatic signature analysis and generation for large-scale network malware," in *IET International Conference on Information Science and Control Engineering 2012 (ICISCE 2012)*, 2012, pp. 1–5.

[33] J.-M. Roberts. VirusShare. [Online]. Available: https://virusshare.com

[34] Y. Nativ, L. Ludar, and 5fingers. theZoo. [Online]. Available: https://thezoo.morirt.com/

[35] IdoNaor1 and D. Goland. VirusBay. [Online]. Available: https://beta.virusbay.io/

[36] H. S. Anderson and P. Roth, "EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models," *ArXiv e-prints*, Apr. 2018.

[37] P. Roth. (2019, Sep.) Endgame Malware BEnchmark for Research Readme. [Online]. Available: https://github.com/endgameinc/ember/blob/master/README.md

[38] R. Thomas. (2019, Nov.) LIEF - Library to Instrument Executable Formats. [Online]. Available: https://github.com/lief-project/LIEF

[39] A. Martín, H. Menéndez, and D. Camacho, "MOCDroid: Multi-objective evolutionary classifier for Android malware detection," *Soft Computing*, vol. 21, no. 24, pp. 7405–7415, 2017.

[40] Z. Wilkins and N. Zincir-Heywood, "Darwinian Malware Detectors: A Comparison of Evolutionary Solutions to Android Malware," in *Proceedings of the 2019 Genetic and Evolutionary Computation Conference Companion*, ser. GECCO '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 1651–1658. [Online]. Available: http://doi.acm.org/10.1145/3319619.3326818

[41] Z. Wilkins, I. Zincir, and N. Zincir-Heywood, "Exploring an artificial arms race for malware detection," in *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, ser. GECCO '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 15371545. [Online]. Available: https://doi.org/10.1145/3377929.3398090

[42] B. Abrath, B. Coppens, S. Volckaert, and B. De Sutter, "Obfuscating Windows DLLs," in *2015 IEEE/ACM 1st International Workshop on Software Protection (SPRO)*. IEEE, 2015, pp. 24–30. [Online]. Available: http://dx.doi.org/10.1109/SPRO.2015.13

[43] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos,

D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[44] L. McInnes, J. Healy, and J. Melville, "UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction," *ArXiv e-prints*, Feb. 2018.

[45] L. van der Maaten and G. Hinton, "Visualizing data using t-sne," *Journal of Machine Learning Research*, vol. 9, no. 86, pp. 2579–2605, 2008. [Online]. Available: http://jmlr.org/papers/v9/vandermaaten08a.html

[46] L. McInnes. (2018, Jul.) Frequently asked questions. [Online]. Available: https://umap-learn.readthedocs.io/en/latest/faq.html

[47] ——. (2018, Jun.) Performance comparison of dimension reduction implementations. [Online]. Available: https://umap-learn.readthedocs.io/en/latest/benchmarking.html

[48] ——. (2018, Jul.) Transforming new data with umap. [Online]. Available: https://umap-learn.readthedocs.io/en/latest/transform.html

[49] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise." AAAI Press, 1996, pp. 226–231.

[50] M. Ankerst, M. M. Breunig, H. peter Kriegel, and J. Sander, "Optics: Ordering points to identify the clustering structure." ACM Press, 1999, pp. 49–60.

[51] J. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*. Berkeley, Calif.: University of California Press, 1967, pp. 281–297. [Online]. Available: https://projecteuclid.org/euclid.bsmsp/1200512992

[52] J. Sukup. (2018, Feb.) When K-Means Clustering Fails: Alternatives for Segmenting Noisy Data. [Online]. Available: https://blogs.oracle.com/datascience/when-k-means-clustering-fails%3a-alternatives-for-segmenting-noisy-data

[53] I. Dabbura. (2018) K-means Clustering: Algorithm, Applications, Evaluation Methods, and Drawbacks. [Online]. Available: https://towardsdatascience.com/k-means-clustering-algorithm-applications-evaluation-methods-and-drawbacks-aa03e644b48a

[54] H. Jain and K. Deb, "An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point Based Nondominated Sorting Approach, Part II: Handling Constraints and Extending to an Adaptive Approach," *IEEE Transactions on Evolutionary Computation*, vol. 18, no. 4, pp. 602–622, Aug. 2014.

[55] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.

[56] G. Rudolph, "Convergence of evolutionary algorithms in general search spaces," in *Proceedings of IEEE International Conference on Evolutionary Computation*, 1996, pp. 50–54.

[57] B. Doerr, E. Happ, and C. Klein, "Crossover can provably be useful in evolutionary computation," *Theor. Comput. Sci.*, vol. 425, pp. 17–33, Mar. 2012. [Online]. Available: https://doi.org/10.1016/j.tcs.2010.10.035

[58] F.-A. Fortin, F.-M. De Rainville, M.-A. Gardner, M. Parizeau, and C. Gagné, "DEAP: Evolutionary algorithms made easy," *Journal of Machine Learning Research*, vol. 13, pp. 2171–2175, Jul. 2012.

[59] Y. Hold-Geoffroy, O. Gagnon, and M. Parizeau, "Once you scoop, no need to fork," in *Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment*. ACM, 2014, p. 60.

[60] S. Neumann, "Spark vs. hadoop mapreduce," November 2014. [Online]. Available: https://www.xplenty.com/blog/2014/11/apache-spark-vs-hadoop-mapreduce/

[61] G. Wicherski, "peHash: a novel approach to fast malware clustering," in *Proceedings of the 2nd USENIX conference on Large-scale exploits and emergent threats: botnets, spyware, worms, and more*, 2009, pp. 1–1.

[62] X. Han and B. Olivier, "Interpretable and adversarially-resistant behavioral malware signatures," in *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, ser. SAC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 16681677. [Online]. Available: https://doi.org/10.1145/3341105.3373854

[63] C. Van Rijsbergen, *Information Retrieval*. Butterworths, 1979. [Online]. Available: http://www.dcs.gla.ac.uk/Keith/Preface.html

[64] A. Rosenberg and J. Hirschberg, "V-measure: A conditional entropy-based external cluster evaluation measure," in *Proceedings of the 2007 joint conference on empirical methods in natural language processing and computational natural language learning (EMNLP-CoNLL)*, 2007, pp. 410–420.

[65] T. Cover and J. Thomas, *Elements of Information Theory*. Wiley, 2012. [Online]. Available: https://books.google.ca/books?id=VWq5GG6ycxMC

[66] R. Bedre. (2018, Oct.) ANOVA using Python. [Online]. Available: https://reneshbedre.github.io/blog/anova.html

[67] K. Beck. (2018, Nov.) What Is the Tukey HSD Test? [Online]. Available: https://sciencing.com/r2-linear-regression-8712606.html

[68] Cybersecurity & Infrastructure Security Agency. (2020, Jan.) Emotet Malware. [Online]. Available: https://us-cert.cisa.gov/ncas/alerts/TA18-201A

[69] Microsoft Security Intelligence. (2017, Mar.) Win32/Ramnit. [Online]. Available: https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Win32%2fRamnit

[70] Microsoft. (2020, Aug.) PE Format. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/debug/pe-format

Appendix A

Supplementary Tables

Homogeneous Cluster Count	Sum of SSE	Median Cluster Size
34.00	2.32	377.22
27.25	1.96	193.75
31.38	2.51	378.16
27.81	1.98	388.94
29.38	2.11	195.38
25.88	1.59	382.69
31.50	2.26	190.00
30.81	2.05	381.34
27.25	1.81	388.78
30.31	2.02	380.38

Table A.1: Final average objective score for each DBSCAN run on 3,000 samples

Homogeneous Cluster Count	Sum of SSE	Median Cluster Size
34.44	3.43	151.69
34.88	3.05	69.31
38.50	4.33	129.56
33.88	3.27	170.88
24.25	4.70	211.75
30.69	4.33	178.75
36.94	3.20	63.28
33.19	2.86	62.75
29.38	3.29	196.88
36.56	3.70	117.72

Table A.2: Final average objective score for each OPTICS run on 3,000 samples

Homogeneous Cluster Count	Sum of SSE	Median Cluster Size
14.63	4.54	361.94
18.44	5.17	306.22
19.13	5.62	555.09
19.94	5.88	297.97
21.38	5.92	285.34
21.19	5.69	297.84
21.56	6.18	292.25
20.63	6.18	298.75
20.75	5.95	307.06
20.13	5.90	314.97

Table A.3: Final average objective score for each k-means run on 3,000 samples

Clustering Algorithm	Elapsed Seconds
DBSCAN	24398
DBSCAN	24650
DBSCAN	23175
DBSCAN	21783
DBSCAN	26714
DBSCAN	28467
DBSCAN	27019
DBSCAN	29202
DBSCAN	30136
DBSCAN	25819

Table A.4: Execution times of COUGAR on Spark using DBSCAN on 10,000 samples

Clustering Algorithm	Elapsed Seconds
OPTICS	2363
OPTICS	2731
OPTICS	2464
OPTICS	2352
OPTICS	2365
OPTICS	2502
OPTICS	2621
OPTICS	2635
OPTICS	2369
OPTICS	2810

Table A.5: Execution times of COUGAR on Spark using OPTICS on 10,000 samples

Clustering Algorithm	Elapsed Seconds
k-means	31071
k-means	20572
k-means	28948
k-means	21398
k-means	19986
k-means	19536
k-means	20812
k-means	30714
k-means	21926
k-means	21272

Table A.6: Execution times of COUGAR on Spark using k-means on 10,000 samples

Appendix B

Malware for Cluster Signature Discussion

Flystudio Cluster

```
d6fe74d3ded8f7139e5acd611228f4be flystudio
e6643dc44c046a01a7fe19c2d5d8dd43 flystudio
f6b291abdf8d15d2581d1eadf3d23df9 flystudio
feb47513ea207955106545cc9f72d8c0 flystudio
6bf1c31c8d280c65c6ed47153c2575d7 flystudio
e2ac60f8d3087e83da38f878ee4fd6ef flystudio
a4c64b5be474ea4d95f49543d67dbe5b flystudio
ab3afbd1419383492abddca1d4a10a5e flystudio
d308edad6b8ae7af7eb93bd5dc632e8e flystudio
fe7f40bd757aca5f5609a29bcd2ef5de flystudio
a1a2ce1a96d76047c6499fd303175ba1 flystudio
af03e2250ff35aa0504fc753b06c1af8 flystudio
dd52f33cd9d5296a7d10d7535ebb728f flystudio
d750316d7a8cb8eb5ba115585f7af1ae flystudio
c5f62b7183311b600aec2bb43be3f6f0 flystudio
c006ce36fc4255c2f28b38eda47838a1 flystudio
b37a259aa4a68c4a424863aeef1cdeea flystudio
ec2535df30d20fca3c86725244351190 flystudio
fe462ecd84b555d3b3c54755f5df18a6 flystudio
3e1b20eeda470be3e01508cd4e862050 flystudio
e13b26933adada18a375bd8b4454c8be flystudio
db26608c1381a270c52473f2b9e30bd3 flystudio
a25fe4029d5ef3058f4acddd90744e92 flystudio
b06cb582cf2e7d453a93b15ecb8e64c7 flystudio
3e259ebc5cf14b9c4b2fb037359ace33 flystudio
```

```
b20df23e0e6e4af8e0d4385fc0a45ca6 flystudio

b86ce6c836f56bd7f55731192b6224f6 flystudio
d814faff9bc05674f86a767ae6a71b24 flystudio
f212e8bc202ba1ad708eace794aa07c2 flystudio
d64cbf1d7c836eaba9e7b5bd433264d8 flystudio
d938b823b71b9d9c32a418cca5383126 flystudio
cf545d68d39dfdae00c0b15d923714b3 flystudio
ebd066edecbb41ba2b1c21c0ad9a9ef2 flystudio
d832daa7406fd951bfb05075860b7a1d flystudio
a104ae32ef38243e3078d45ee09e88b7 flystudio
c3ffc9493b9bbdb5c44ea7c487190795 flystudio
d9d1a0f9c091d6277db4088bde655d42 flystudio
ccfe8868571e2a29528b87b762532850 flystudio
e3ed556d781e1f82eab808423c1d675a flystudio
b5c457f4ad8516bbc7fbdebe34777dcf flystudio
a2e502d82e3ae6fbf7885b2d8fb5cd24 flystudio
b9e360dbbfd109334934288e66879159 flystudio
b3e896da3e4cf8792fd0df7c35b5ebe1 flystudio
dfef020e3ede8abe405f7bde2e9a7fbc flystudio
df9813202bd1165f243b8fbb38fd8c5d flystudio
b53e56d20366d814ec093e5b8e2490bc flystudio
a1becbf701fa82c3467c897a8f349050 flystudio
febd3879547418ed6367b3d079d3389c flystudio
c1ce1d857ab0f13b3b66bd6236683944 flystudio
caa531bf1d23cd0d913700b087b94f60 flystudio
b07cbf8a2f04e9612fcd3564f7bf54f7 flystudio
bb3dcd9f1271d7348b0a42dc39778b38 flystudio
38cc8e32054bb05d7b261d0e13b7b22d flystudio
eb68497891bb9bbd4e4879cbf636f6b0 flystudio
```

```
b0957ff3bdc1b5f9ced5401784282290 flystudio
c3439aa61fae8b1eb1e8cffcb6edc5cd flystudio
e700e79849327ed624add2b7f6848bcd flystudio
db78718147545b5b1774209692b479d5 tescrypt
16195521d7f568df6db43b9b00f8d350 flystudio
cdc08134bdcc825a35b298729f75d364 flystudio
ae59f77f3bf40562e366259df93903f4 flystudio
c8acc23060bbbb56f0b5f9762e10bcb0 flystudio
de8480ee1dbbeb93d57757a3ee5118b5 flystudio
cf78b85a00c761a6751a26e04d0b2668 flystudio
c25253847b88d7da33c90c3518a19cf3 flystudio
cebd9de242385072b85468d5e719900d flystudio
f5452f166f5f281438b376b9c6122a39 flystudio
ad358de505039a623755832465b65ede flystudio
c63b1a9af254e6fb4be903b13de8a650 skeeyah
a642ac62490cc19e37e996cd85959842 flystudio
e624bd8675afcffa69f9431c3bad94af flystudio
ac634a6241ff486b717bc5803b117459 flystudio
aa463ad61d7bcd872004de2405998697 flystudio
b4538e8f2a8f6943b17d5513f6f1ba4a flystudio
c1e159087e5b2049727394c2d56f0d5c flystudio
befca464fa4a10bb4125272166276e6c flystudio
c1ca6cf846b7cb831b4c8c6ed15de26f flystudio
a7546ef3a6cd44e08239e631c00b3daf flystudio
ba8fac5d6ae721de010e9679d719006f flystudio
b2f5111a43d9d8a18c6e23d3a1e9c266 flystudio
e21db1af523a4a40db7289b4e4c42043 flystudio
1c79f3c9c35fbf3f4e1db43c82778240 flystudio
c222fabec91b74dfe476cd27ee13d61d flystudio
e3fa207271f124672f6a933df934ae4c flystudio
d6daa98b836e1fb4dffc12aebcb38ad1 tescrypt
1f0f7900cfba911682a58d792c9c2e83 flystudio
```

```
bb4bf49e1d8abb21c56e09c47a62f567 flystudio
47ea70b8ee98a23dba2f3e2dddce9a50 flystudio
a4c0f5d59bf739060ab64343aaa0b837 tescrypt
d86a96b4afcfa144f52cefbe415653a0 flystudio
a220704358e0f8d23e98844f1fb37180 flystudio
dc4e1bc89ba14e6e7d87ed4af8c38519 flystudio
b9aaa38f7e29e4433c5e386360c1e722 flystudio
```

Emotet Cluster

```
0a7a72a5853f3740ea76f9934764bd44 emotet
b1fa339a197acac27eef6520bfe5b84e emotet
23f8eec4c3b31bad4dddf80784365b7d emotet
1708d489301fdcc0427382d0a738442b emotet
2266fdee5307b9e7a5e61ed39ed05cb9 emotet
1d13b92e9fa263416759f8577c73a73a emotet
e547dfc077329146bbcadd3be7c3b4dd emotet
f55c58b1d8ade1bf02646daf45efa7b8 emotet
2a8445ecea1aaa52d1e4c0fb7bff9bb8 emotet
c60287a31f0c02a49229a01480520e9f emotet
```

Ramnit Cluster

```
77df831182da5d1870e8856d6e0c3e77 ramnit
fa53de356be8a08c209a6ec2d6544666 ramnit
03b4a13300c6546a5df492331a159845 zbot
0dd1d6bc6c0d1205104eeb46b792baf4 ramnit
2373fcbe0fd0135f4d90fb6efc06594b ramnit
54eaf0fac033c6c9c9c56fa9a01c1ecc ramnit
2e761806cb47fe0f051743b44dd191f8 ramnit
6c728118aae5d502bc397e706ac5d0b0 ramnit
d095cb0f2d9ec64da8243c795cad5f8f ramnit
```

```
417829796fc6d4d347eefa7288258ea2 ramnit
b416724c7f39349c53ca91b838caa141 ramnit
ef4777d55cfe87302de001d0cda72e64 ramnit
dd8852137f6ff82e30fcb2e3a8ed2497 ramnit
599e9f38051ae95d7f56fb13b5941f7e ramnit
401060b5991f47afed4e24e60f646215 ramnit
c2d250b86f774971cf0b1ac2835bbda1 ramnit
69eff1e3da36a3c37a59f42a14ee5c44 ramnit
413af0e78cb8812c578a00a86fde8e9f ramnit
e06191c650b9bca9cdd4835a99428432 ramnit
162f053d845244ec77d60efed413c378 ramnit
2dc709bbb11af6a768d6e8f17594fc64 ramnit
41052d3bb50fc313e67f804136e61d19 ramnit
40b705020ab8486496221a89dd90ad90 ramnit
79ef60598aa752ff5eadffb8eb28faa0 ramnit
```

Zbot Cluster

```
11a9e236a596da2741587663e3fb1849 zbot
c31cb2c4abd19ab6fb413276e5d2bf41 zbot
d1a2a9ff3667f2ea799db44c4febe65d zbot
ff35b4cb9ee15524e99ae97bbd92cdaa zbot
62e1a96c1317e92628eb765bb5631752 zbot
d2d64e27903c3ab2d79779b520cd5acd zbot
01a540e09c495d56530d1d1fc807897e zbot
16d264f393ecdc008a053f1c328b600c zbot
eec992d6c259af40459d4ad68b3be0d4 zbot
f82e35ef978df782613a077610548268 zbot
cbf4076f5e028b3d6c71ee1256466915 zbot
a2edbf237d55306f8c203330e51bc71d zbot
1734c9459631969754fdc4b907c584d3 zbot
16f0519aaa846e62411be6b2dd8239cf zbot
```

f54c593d9107739dfc60c08376ad624b zbot
f9db29266a2e23eacb2f66472aa765eb zbot
afc5514b3971fd8f99bb55c251c1b2c9 zbot
bff02293887627c319f1d21dfc18d285 zbot
b75f3d6b4b75b064152c30b430416324 zbot
469f1dfd7f0637f46f4eb68690888058 zbot
c8f9d7d31bbf9a60db49c95878971575 zbot
a4267859082fa2eebe7b648fea8b8003 zbot
0f605a347527142b59766ecfa16393f6 zbot
e71165594e9ab33ff801bb01a7303f91 zbot
b24aa28d357284cc4eb09dd03db482ed zbot
eb7b16051956cdd9cb48bd9ec21e28d6 zbot
b494e47a6e418a2d9f01dc5d0b5279fb zbot
52c9fe4a4aae8e5dd92b87732dea6066 zbot
99fd1eba65a6db79a94727db5b359d36 zbot
fc072d09dd607e1f88e6fe6556114fb7 zbot
cd24cbec992667f70ddbc51ed23f5334 zbot
d6cc45f00928550e70e5276d54e1ef67 zbot
e91e6267070649610bc09b591457c3b0 zbot
2941cf7072418d1daf08381aeb9cff3e zbot
25630ad24bb27837ee045c3a96bd806f zbot
eb3536d10a4e839abf01cf465afba219 zbot
9fde5383dbb583d6c335c3d2ad444828 zbot
fa2d59a2a8834e4288c8d439b88dee24 zbot
c836372d6b1f686bbc8fe7862226959d zbot
e5217bcd2fa6b4f9f30891106263cc7d zbot
fe60c59f7250f76a0a78ade8123a7017 zbot
f3ca2dfa18f79693cb9113103b879e18 zbot
fad4c963cfe420a7e6fd044d5051548f zbot
a90575f9e584e5c1d5f42d0032f9edd6 zbot
e933e9aa413ca0548f286274335b8888 zbot
a15a02544355ac8c424b9778337b2ce2 zbot

1a0488da205a0e435065a5aa054e4de9 zbot

cafb9b8deda87a24858603871211245d zbot

d9929b7279e0ec80ae14f9e7e537da5c zbot

3eeeecd10c9baa5133191b040d7b1e42 zbot

bf971cf841c3f50cee2d9634c0923b98 zbot

8e0a6a5a7f77c53ebebd7e0370606d44 zbot

e0ea582cf301e1ae095e174efa319d1d zbot

b73363c0d26ba537059516b55e21ef3a zbot

a7975a8461eb38f12a6f01c0b4621a10 zbot

a565c4e39b427baf389d498c03a80278 zbot

0f4014b932b9f11e2851a79c966e3560 zbot

da1ca2770f4163c3c08d76b41b8d9dc6 zbot

600d638eebd2510f2ed3fdc163032cfe zbot

a8e572365b2062005324ca4ba3857e62 zbot

d54dea07baef2aaa01acf0fd0010fcfe zbot

b1d84805224e2084bca7fab865d6eafd zbot

cb10b731a338652155f7a32e8329c611 zbot

03998066aec71a8d5b075a479b1b70ef zbot

5b3ad6090bcb5b24bcc6b55084e1d763 zbot

65df040268bac4fa532e21ab031d5751 zbot

a3eada1072c6493f4e6b1a2ea82bc2c5 zbot

916bffd24180bc1446896bd4447925f6 zbot

c96dc320910b0607eedbcc3bac693b47 zbot

e74b137fbb65fb7705ae6e47ecd8285b zbot

66f1e8aee900acc6d3360443f54f02c3 zbot

683be24a8ff8ed1598abb8db0c45a437 zbot

e9c1e13b0b8825d00f69a895c8543e5f zbot

2925ae9e10e1ef92bb595377fb7be431 zbot

8b171ad0a8372b96219082c652c4733b zbot

6fcc9583f21d2a4caad6410c365f55cb zbot

e38228b9c00e5397327d66845835ebd3 zbot

fa8f03b5073f205d7dc6c435a8ad6c5b zbot

53cbc64a8e4a6504370cd58919a0d4b8 zbot
bf992b3a77f471294a09860bae9f4b4e zbot
fe9263c2787588d26e101ced2d26787f zbot
cdaffc5e638f9987a455eae01068009a zbot
a0e114f172dc69b8895382a346cff572 zbot
7b0b852c0197eb075f8059c1b2d9186f zbot
1ea8b944c8fc9bd729b43ce1c5a5c3b8 zbot
c27e50eb3ec6fa108b942dc750eb03c2 zbot
ab4e08d7e2e61f9ad034ab55dd96fcee zbot
75287df03dfb57242e45450876b8457b zusy
a78f893c9eb1d0d23cf1f7d6363eaaa7 zbot
75f8f22b01f64ec83f7cf699d3f30df3 zbot
a0c6cb7abacca86a79a86c8a3d484228 zbot
5bb01fa794f3016a0124725e34d1b772 zbot
e0714cb8cfd37b7dcfb203d68911d144 zbot
a9baf67a13b37b2d5773af49e13ebd7a zbot
80a0624c8428e2c8c892954609a45128 zbot

WannaCry Cluster

20193d9b262e6a2f296073ea0855fed4 wannacry
7d51fa4aff9ef41229353c46f3b4c20c wannacry
e6bfac31c1536ae1a49b5cb67d606805 wannacry
8b31308f2bf97e940dab49334d2d2011 wannacry
d30aa7aad0beb177a00f6af0946239b6 wannacry
0068a31f704348396fa213561734210b wannacry
d2df22eef4af4d5e75e6000e4d149c21 wannacry
3e8e1dcedcfb86a6318019660908ef42 wannacry
ab06e75f40fbe63b000790dadc1729c4 wannacry
d42d50380526d334be6f2bd7124be8a6 wannacry
d5ad19af05312fa9508cde8dcf87ac67 wannacry
53bf1b10050deecbbd45b2b7b3fb5015 blackmoon

```
3d2e12ddbd216f49904148bd24ac800e wannacry
bcd30b55110588d474e8826fa5290c4c wannacry
4b38eb3132fe8a0d1ed57bbbd48f2264 wannacry
7e3ffa2ea1f01fad04f9645840d45959 wannacry
6ecc0dbf8e043d359df6a510b4961e43 wannacry
67db7cea0cd227285e8bb72b71cee34e wannacry
5111cf0f281f11152075025958e828a5 wannacry
230cef07feab6bae5cce0fa22068647c wannacry
a34d8bd7493c5f8c2bf381a0267de463 wannacry
70f47070e87bfaa3fcd66b98f2d1fe71 wannacry
082153c1540b51dd2b899dd5dceea7bf wannacry
fe086079c8724b7c3bba62d4991f4b57 wannacry
63d99eb9dbb58306cbb3a9b7e2ecfb27 wannacry
2f80c6592fdbc97515790cb36b5ffa39 wannacry
de4a5b1c9b4c2d69da573c3dfbfc6f5f kelihos
9054bcff637f9b30dda45ac50b5c444e wannacry
53df35ec321c54b0a0f58ef7c66d5236 wannacry
edb425707fcced793626cda84f14310a wannacry
64d545341c725af087f5ad65c2f5211a wannacry
212b7b8a02e6657c11668f1b58a171ba wannacry
1543169fd15084ba7a5a9388fce77dea wannacry
ffafe05991d988d69f994b0b863b32b9 wannacry
3a50f312bfe14f92ae1d56c0c2c2cb4c wannacry
3da3494d7c4205034648150d3ccc58a8 wannacry
f470d27f84d065da769fdae3b34f39e5 wannacry
93f38909d92c11ac7a710b9397de6d86 wannacry
ee64d96b3078b1e12e75d2ba32f93b7c wannacry
47765f90c0fb0320afb71996d787206e wannacry
bcca7ff04667645d98e20ab6458f69bf wannacry
eefb5217f9fb344b506279d6caf1e0dd wannacry
2c49423b8e81de5400078544ebc7ca3f wannacry
e6a999cd5df18b0962b89e1a9c1ebfaf wannacry
```

```
d57959ccb426b310dc80552f97932f43 wannacry
f4f96192ab5cfa56507a73a12e946cfc wannacry
f6740d11f8d6aa4c836fb9d7efa34fd8 wannacry
5d805e2873275cdd033813f1a1e1fd77 wannacry
9ed65061f9ee339845748fa81f09fa53 wannacry
3a79fd05da85431c82669c3ba2c9c799 wannacry
e86fef56e79f9b0def210299ff73142d wannacry
568c0dba89d5fd2f5169c235d7b93873 wannacry
a63840fe82d9f879e792b9ba36906cdd wannacry
b1f531440cb8b9f0de48a60cff4e5317 wannacry
8292d3b39f6b4de384d1b95c36a16311 wannacry
87eec0f1c5ec57fca9af922ac0086b98 wannacry
d04c1b80557a7793b38a7bf876ef61c7 wannacry
3e99a142993ca19c97e08ea23d06b7fe wannacry
ee6e3c33fc545f85db7424ce15677994 wannacry
47b4d4a68f2f151d79c3e43c963ecf06 wannacry
834b4e12b8eaf4c7258f0d4fc8b6b37c wannacry
c688aaf68c68b2570d10258d7e435de4 wannacry
de9bfd4f33a95213423b723aa07e75fb wannacry
```