Thesis Dissertation

CLUSTERING CRASH REPORTS FROM FUZZING SESSIONS USING VECTOR EMBEDDINGS

Emmanouil Theofilou

UNIVERSITY OF CYPRUS

COMPUTER SCIENCE DEPARTMENT

May 2023

UNIVERSITY OF CYPRUS

COMPUTER SCIENCE DEPARTMENT

Clustering Crash Reports from Fuzzing Sessions using Vector Embeddings

Emmanouil Theofilou

Supervisor
Dr. Elias Athanasopoulos

Thesis submitted in partial fulfilment of the requirements for the award of degree of Bachelor in Computer Science at University of Cyprus

May 2023

Acknowledgments

First and foremost, I would like to thank my supervisor, Dr. Elias Athanasopoulos, for his uninterrupted guidance and advice throughout the entirety of this year. The consistent and valuable feedback and communication has been crucial for the completion of this thesis.

I would also like to extend my thanks to the SREC (Security Research) group, and especially to Antreas Dionysiou, for his willingness to help me in all stages of this thesis, with valuable knowledge and advice.

I would like to express my gratitude to my family and friends, who have provided me with continuous support, love and motivation throughout my whole life. I consider myself very fortunate to have surrounded myself with people who I most importantly consider excellent characters. The effect you have had in my life is undeniable, and the role you have played in this achievement is massive. I am sincerely grateful for the experiences and memories I have made and shared with you, and I hope that there are many more to be made.

Summary

Fuzzing or *fuzz testing* is an automated software testing technique often used in software development where the tested program is fed various inputs, with the goal of discovering irregular behaviour such as crashes and vulnerabilities. Fuzzers have continuously evolved to become more effective in identifying faults in software with smarter techniques that replaced random inputs with mutation-based or generation-based inputs, and better code coverage, allowing for the testing of obscure parts of a program.

A direct result of the evolution of fuzzers is a significant increase in the number of faults that software developers and programmers must triage after a fuzzing session. The overwhelming amount of crash reports that fuzzers generate presents a difficult challenge in the prioritization of vulnerabilities or faults that need to be resolved first.

In this thesis, we explore the potential of vector embeddings, a popular technique used in the field of Natural Language Processing, as a means of organizing the crash reports returned by a fuzzing session into groups using unsupervised learning algorithms. The end goal is to provide software developers and programmers with a framework that performs grouping of crash reports into clusters, where each individual cluster contains reports caused by the same bug in the program. By achieving this, the large number of faults reported by fuzzers is reduced into a much smaller set of bugs, which as a result, simplifies the investigation of each bug's severity and ultimately, facilitates the prioritization of bugs and their timely repair.

By training clustering algorithms on numerous datasets of stacktraces extracted from crash reports, we produced clusters of stacktraces, and evaluated their quality. We determine that there is promise in the use of vector embeddings and clustering algorithms for crash report grouping and prioritization. We also hypothesize that the existence of an extensive ground-truth dataset that could be used for training can further enhance the quality of the resulting clusters.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software development is a large field of computer science that includes designing, implementing, and maintaining software systems. Over the years, software has been tasked with performing an increasing number of tasks, ranging from rather straightforward tasks like calculators, to more elaborate and complex tasks like managing a device and its applications using operating systems. Regardless of the complexity of the task that is to be performed, the software development process usually includes a common set of steps that need to be followed. Although the order in which these steps are followed can vary depending on the chosen development methodology for each development team, the steps typically include planning, designing, implementing, testing, deploying, and maintaining the software.

The content of this thesis pertains to the testing step of the software development process, and more specifically, focuses on an automated software testing technique known as *fuzzing*. Fuzzing works by repeatedly providing the program or software under test with inputs, in a random or systematic number, in an attempt to uncover vulnerabilities or bugs present in the code. As software has evolved to become more advanced, accommodating for the increased complexity of the tasks that need to be performed, a mirroring effect can be observed on fuzzers. New fuzzing techniques have replaced the previous standard of providing random inputs to the program, with strategically mutating inputs, in order to uncover a larger number of bugs with greater efficiency. Nowadays, state-of-the-art fuzzers are able to reach and test even the most obscure code present in a codebase, and as a result, can report more bugs and vulnerabilities in the tested software than ever before.

The wide variety of bugs that are uncovered by fuzzers can range from high-severity bugs to low-severity bugs, or even false positives. High severity bugs pose a significant security concern and could very likely be exploited by malicious actors to steal sensitive data or gain authorized access to a system, giving them the opportunity to cause irreparable damage to individuals and organizations. Therefore, it is imperative that software developers and programmers are able to correctly identify severe bugs and deal with them in a timely manner.

This thesis aims to explore a possible solution to the problem of prioritizing bugs from an ever growing list of bugs returned by a given fuzzing session. As described above, fuzzers have continuously evolved over the years to the point of uncovering an overwhelming number of bugs on a given fuzzing session, especially when the program under testing contains multiple lines of code and performs complex operations. While the great number of vulnerabilities that fuzzers report to software developers is indicative of the quality of the program, it also can be a double-edged sword, since it hampers the programmers' ability to sift through them, organize them and address the most dangerous ones promptly. The software development process is cyclical, and software is tested not only before release, but also after is has been deployed. In these cases, resolving severe security vulnerabilities as fast as possible is incredibly important, as these same vulnerabilities can be uncovered and exploited by malicious attackers.

The problem of grouping bugs into severe and non-severe, given a large collection of crash reports returned by a fuzzing session is not trivial. The first reason is the lack of a pre-existing publicly released dataset of crash reports that can be used by researchers in order to perform bug prioritization or clustering. The release of such a dataset on behalf of a software development team has several risks, particularly regarding the security of the software itself, since crash reports often include important information about the program, that can be used to understand its structure (e.g. functions/method stack traces), or even understand the specific events or sequences that led to each crash in the first place. With that in mind, making a dataset of crash reports publicly available can directly empower malicious individuals to exploit the software and develop more effective attacks against the software. Another reason why the problem of grouping bugs is quite difficult to solve, is because there is no way to tangibly measure the severity of a bug, or a unanimously used metric to compare two bugs with each other.

Various approaches have been followed in attempts to provide a solution to the problem of bug prioritization, which make use of various techniques. Dongsun Kim et al. [1] used supervised machine learning to classify bugs as frequent or not frequent by leveraging a dataset of past crash reports. This way, they can predict whether or not a new crash report deserves immediate attention, if it is classified as frequent. Jiang et al. [2] developed Evocatio, a capability-guided fuzzer that explores the code around a crash point in order to assess the severity of a bug. Spanos et al. [3] make use of text mining techniques and supervised learning algorithms to predict the Common Vulnerability Scoring System (CVSS) [4] and Weighted Impact Vulnerability Scoring System (WVISS) [5] severity scores for vulnerabilities based on their text description in the National Vulnerability Database (NVD), and achieved high prediction accuracy.

In this thesis, we leverage vector embeddings, a popular technique used in natural language processing, with the goal of creating clusters of crash reports, where each cluster represents vulnerabilities caused by a single bug. Our approach aims to give a solution to the phenomenon where fuzzers generate multiple crash reports that in reality are caused by a single bug in the source code of the program under test. Therefore, the goal is to reduce the overwhelming number of crash reports returned by fuzzers into a much smaller, manageable number of bugs that need to be addressed. Our approach is different in that it does not require a training dataset, as we make use of unsupervised learning algorithms for clustering. The only labelled dataset that is needed is a ground-truth dataset that can be compared against the labels produced by the algorithm, in order to evaluate the algorithms ability to clearly divide a large number of crash reports into a much smaller number of bugs. More specifically, we extract the stack traces from each crash report produced by a fuzzing session, and transform them into their numerical representation in the vector space, creating a dataset of vector embeddings. Then, with the use of unsupervised learning algorithms, we create clusters of vector embeddings and evaluate their quality by comparing the clusters against a manually produced labelled dataset.

Our results show that representing stack traces as vector embeddings is a promising direction that can provide solutions to the problem of clustering and prioritizing bug reports, as the calculated quality scores for the resulting clusters are quite high. (add the scores here?)

Our contributions include a framework for extracting the corresponding stack traces from a collection of crash reports reported by fuzzers, a simple parser for representing stack traces originating from programs written in the C language with Python objects, therefore facilitating further work into studying and devising comparison metrics between stack traces.

In addition, we contribute an implementation of the full working process of training a word embedding model with a collection of stack traces, retrieving their vector representation, training and optimizing clustering algorithms to maximize the quality of the produced clusters, and then evaluating the optimized clustering algorithms on various datasets of stack traces.

Lastly, we contribute a sizeable dataset of stack traces originating from crash reports generated by the mutation-based fuzzer AFL++ [6], a state-of-the-art fuzzer based on American Fuzzy Lop (AFL) [7], performed on altered versions of Linux libraries, programs and utilities, injected with bugs with the use of the LAVA Vulnerability Injection Framework [8].

1.1 Thesis Structure

In the Background section we introduce basic concepts and terms that are necessary to understand the research we present in this thesis.

The Architecture section follows, in which we propose an architecture for collecting crash reports, training word embedding models to produce vector embeddings for stack traces and using clustering algorithms to produce label assignments for each dataset of crash reports.

In the Implementation section, we present our implementation of the described architecture, with technical details.

We evaluate our work in the Evaluation section, where we examine the quality of our produced vector embeddings as well as the quality of the label assignments that were computed by the clustering algorithms.

In the Future Work section, we identify and explain weaknesses with the work presented in this thesis, and propose ways in which these deficiencies can be overcome. We also propose some future research directions that could give a better insight into the potential of vector embeddings in crash report prioritization.

Chapter 2

Background

In this section, background information necessary for the in-depth comprehension of our work is explained. Key concepts and terms are defined, as well as brief overviews of the clustering algorithms and metrics used. The goal of this section is to provide the necessary knowledge for understanding the research we present in this thesis.

2.1 Crash Reports and Stack Traces

A crash occurs when a program or software stops operating properly and forcibly stops execution. Most crashes can be attributed to bugs in the software source code, which differ depending on the programming language used. Some crashes, especially memory-related crashes such as buffer overflows indicate the existence of bugs that can be exploited by malicious attackers to harm a software system and its users and jeopardize a system's confidentiality, integrity or availability.

Crash reports from a program can contain various information about the program, such as the information about the system the program is running on, the stack trace of the program at the time of the crash and a snapshot of the memory at the time of the crash. The purpose of crash reports is to supply the programmers with as much information as possible about the crash, in order to facilitate a timely response to bugs existent in the program's code.

In this thesis, our work is focused on using stack traces extracted from crash reports in order to create groups of crashes, based on datasets of stack traces that have been encoded into vector embeddings using word embedding models. The stack traces were collected from crash reports that were generated by testing a collection of C programs using fuzzers; a software testing technique we explain below.

2.1.1 Stack Traces

A stack trace is a report that illustrates a snapshot of the call stack of a program during execution. It provides a list of the functions or methods that were called before the crash, along with additional information about memory addresses, line numbers (in the source code) and parameters that each function was called with.

In a stack trace, function calls are sorted in decreasing order based on how recently they were called, meaning that at the top of the stack trace sits the function most recently called, and at the bottom the one least recently called. The full list provided by the stack trace shows the sequence of nested functions that were called up until the crash occurred.

2.2 Software Testing

In software engineering, software testing is defined as the process of thoroughly examining and evaluating the behavior and functionality of a software system. This is performed with the use of manual or automated techniques, which aim to ensure that the application meets the required expectations and is not prone to erroneous or undefined behavior in any conditions.

Software testing techniques, as described above, can be divided in two categories, automated and manual, depending on whether a specialized software tool is used to perform testing, or the testing is performed by humans that perform manual steps.

2.3 Fuzzing

Fuzzing is an automated software testing technique that is used to discover bugs, errors and security vulnerabilities in software. It involves a program, called a fuzzer, that repeatedly provides input to the program under test, and monitoring the program for unexpected behavior, such as crashes, memory leaks, hangs, buffer overflows and more. Depending on the type of fuzzer used, the inputs provided to a program under test can be random, or generated in a more structured, smarter way, by mutating existing inputs, or generating data in order to test a larger portion of the program.

Software that is typically tested by fuzzers shares the characteristic that it expects some form of structured input on behalf of the user (e.g. images, videos, audio files). Fuzzers can be categorized in different ways, either based on the way that they generate and feed the inputs to the target program, based on their general awareness of the structure of the program, or based on their awareness of the intended input structure. [9]

2.3.1 Input generation strategy

Fuzzers typically fall into one of two categories, depending on the way they generate the data that is fed to the program under test.

Mutation-based Fuzzers

A mutation-based fuzzer makes use of a pre-existing corpus of test cases during the fuzzing process. The way these fuzzers generate inputs to test the program is by mutating the pre-existing inputs, to create modified versions that are considered semi-valid. The term semi-valid input refers to inputs that are not meaningful semantically (e.g. an image that doesn't necessarily depict anything to the human eye or that is different to a valid comprehensible image in a very subtle way), but generally maintain the required format and structure of the input expected by the target program. These semi-valid inputs are used to explore edge cases of the program, that are usually not thoroughly tested through any other software testing techniques, and uncover unexpected behaviors, the likes of which are mentioned above.

Generation-based Fuzzers

The main difference between mutation-based fuzzers and generation-based fuzzers is that generation-based fuzzers generate inputs from scratch, without the need of a pre-existing corpus of test cases. In cases where a program only expects structured input, the use of such fuzzers can be quite inefficient, so what can be done is to break a valid input into pieces (e.g. take apart an image into segments), and fuzz each piece randomly, so that the generated inputs are accepted by the program under test.

2.3.2 Awareness of target program structure

A very common measure of effectiveness in fuzzers is code coverage, a metric that illustrates the percentage of the source code that has been executed. A fuzzing session that achieved high code coverage tested a bigger portion of its source code, therefore there is a smaller number of bugs that can remain undetected in the non-executed parts of the code. Fuzzers that take into account the structure of the target program are able to achieve higher code coverage, since they can test and uncover bugs that are dependent on this structure.

Black-box Fuzzers

Black-box fuzzers, as the name suggests, are completely unaware of the target program's structure, and generate input as random. Due to this, black-box fuzzers are limited to discovering surface-level bugs, however, because of their simpler structure, they can execute a larger number of inputs per second.

White-box Fuzzers

White-box fuzzers are aware of the structure of the program under test, and leverage this knowledge in order to generate inputs that will test deeper in the source code of the program, thus achieving higher code coverage than black-box fuzzers. This ability of white-box fuzzers to test and uncover bugs that are hidden deeper in the program comes at the cost of time, as they typically take a longer amount of time to perform program analysis and generate adequate inputs.

Gray-box Fuzzers

Gray-box fuzzers are an attempt to combine elements from both approaches, in order to achieve the efficiency of black-box fuzzers in addition to the effectiveness and code coverage of white-box fuzzers. This in-between approach replaces program analysis, the technique used with white-box fuzzers to retrieve information about the target program's structure, with *instrumentation*, which is defined as the addition of small pieces of code in the source code or the binary file that can be used to keep track of events such as function calls and memory usage. Of course, instrumentation adds some performance overhead, but it comes with the added benefit of actively informing the fuzzer about the code coverage it has achieved at any time.

2.3.3 Awareness of intended input structure

As previously explained, fuzzers are typically used to test programs that expects input that abides by some sort of format. An effective fuzzer can generate semi-valid inputs that abide by the same format so that they are accepted by the program under test, so that they can investigate the behavior of a program and test edge cases that would otherwise be difficult to test.

Smart Fuzzers

Smart fuzzers make use of the input structure the program requests, also called the input model, to more efficiently create semi-valid inputs that can be accepted by the program's parser. In many cases however, the input model is difficult to be provided, in cases where the software that is being tested is proprietary or complex.

Dumb Fuzzers

Dumb fuzzers, on the other hand, do not explicitly require to know the input model of the target program. Because of this, dumb fuzzers can be used on a wider variety of programs, however, since they are not aware of the input model, they generate less semi-valid inputs. As a result, dumb fuzzers implicitly perform testing on the parser of the target program; if it exists. An example operation that can be performed by a dumb fuzzer to generate more inputs, are random bit flips, a technique which is employed by AFL.

2.3.4 AFL

American fuzzy lop (AFL) [7] is a publicly available gray-box fuzzer that tests programs by employing genetic algorithms. As a gray-box fuzzer, AFL works by injecting the target programs with instrumentation in order to quantify the code coverage it has achieved and to generate subsequent inputs, based on a collection of test cases that is given to the fuzzer before execution. Because of the above, AFL is considered a mutation-based fuzzer.

The algorithm that AFL implements in its operation is described below [10]

1. Load initial test cases into the queue,
2. Take next test case from the queue,
3. Minimize the test case,
4. Mutate the test case, and add the new version to the queue if it leads to greater code coverage,
5. Repeat from step 2.

For clarity, small explanations about how minimization and mutation is performed are given.

Minimization

AFL removes blocks from each test case in the queue, and evaluates the code coverage achieved with the truncated test case. If coverage is not affected, the original test case is then replaced with its minimized alternative.

Mutation

The mutations performed by AFL do not take into account the input model of the target program - hence why AFL can be categorized as a dumb fuzzer. There is a series of mutations applied to each input by AFL. Mutations applied by AFL include bit flipping, deleting and duplicating blocks from the test case and splitting and concatenating test cases from the input queue into new test cases. [11]

2.3.5 AFL++

AFL++ is a community-driven open source tool that started as a fork of AFL, and it incorporates various features unseen in AFL to facilitate state-of-the-art fuzzing. Amongst other additional features, AFL++ incorporates various new mutators than the ones present in the original version of AFL. More details about the implementation and new features that AFL++ adds can be found in the original paper, which is cited in this thesis. [6]

2.4 Vector Embeddings

In Machine Learning, the task of applying algorithms to a dataset of records is not always a straightforward procedure. A main reason for this is the fact that the data that is to be processed often is complex, making it inefficient and difficult for an algorithm to extract information about the records and achieve satisfactory performance.

The term *vector embeddings* refers to the transformation of such complex data (images, words, strings, or in our case, stacktraces) into a numerical representation where each record of the dataset corresponds to a vector. This transformation is used in order to reduce the amount of processing power and memory needed for machine learning models to be trained and used. Another advantage of transforming complex data into vector embeddings is that the resulting vectors capture semantic value or hidden relationships between data points that would otherwise be very difficult to capture using traditional feature extraction techniques. A very popular example of vector embeddings being used to capture underlying semantics and relationships in data are *word embeddings*, which are widely used in Natural Language Processing (NLP) to perform a variety of tasks, such as sentiment analysis, translation, grammar correction and more.

The transformation of complex data to vector embeddings is performed with the use of an embedding model. There are numerous publicly available implementations for creating such models, which leverage artificial neural networks to automatically perform feature extraction on the input data, something which previously could only have been done manually.

2.4.1 Stack Traces as Vector Embeddings

Our dataset consists of stack traces, which can be considered as complex data structures. This is because, despite following a specific format (or formats, depending on the language), there are textual features, such as function and parameter names, combined with numerical features, such as addresses or parameter values. Another important property of stack traces that constitute them as a complex data structure, is the relationship between stack records in a stack trace. By this, we mean that each stack record in a stack trace is directly related to its neighbouring stack records, as they indicate which function invoked the current function, and which function the current function invokes.

Of course, in a machine learning task, capturing this relationship becomes very important, as stack traces with mostly different stack records may be caused by the same bug in the source code. Two examples of such cases include: a) Stack traces with mostly different stack records but similar exit records, which indicate that a bug in the source code can be reached by different function, but it remains a single bug, and b) Stack traces with mostly different stack records but very similar entry records, which indicate that a bug is caused from a specific location in the code but materializes much later.

Capturing this relationship between stack records is crucial in order to improve triaging of bugs, and in order to help software engineers focus their bug-fixing efforts on specific locations in the code.

2.5 Unsupervised Learning

In the field of Machine Learning, unsupervised learning refers to a category of algorithms that are used to analyze and learn patterns from unlabelled data. This means that the input data fed into the algorithm is not tagged, therefore in training, the algorithm cannot compare its computed result with the real result i.e. label and adjust its parameters to achieve better performance. An unsupervised learning algorithm of this category attempts to find underlying structure in the data, and often tries to organize the input data in clusters or groups, where items in a cluster present some similar underlying features.

Unsupervised learning algorithms are used in a variety of applications, most notably, natural language processing, recommendation systems and anomaly detection. The relevant to us use of unsupervised learning, *clustering* is described in detail below.

2.6 Clustering

Clustering is defined as the task of grouping a set of items (called a dataset) into clusters, such that items in the same cluster are more similar to one another than they are to items that belong to other clusters. This grouping of items is performed based on their characteristics or properties, and allows for the identification of latent patterns in the input data that would otherwise be difficult to distinguish.

In this thesis, our main task is to leverage a dataset of vectors, that correspond to a collection of stack traces collected from buggy programs, to perform clustering and examine the members of each stack trace to evaluate the quality of the resulting clusters.

The task of clustering can be performed by various clustering algorithms, with each one processing and organizing the items in clusters based on different mathematical formulas. In our work, we used the implementations of the below clustering algorithms from Python's *scikit-learn* library. [12]

2.6.1 K-Means Clustering

K-Means is an unsupervised learning algorithm widely used for partitioning a set of items into clusters. The itemset is partitioned into a total of **k** clusters, where **k** is a parameter given to the algorithm before execution. The algorithm works by trying to create **k** groups of equal variance, by minimizing the within-cluster sum of squared Euclidean distances, a term also called *inertia*. The K-means algorithm is also known as *Lloyd's algorithm*.

Before describing the algorithmic steps of the algorithm, some terminology should be defined.

Centroids

In mathematics and physics, the centroid (also known as geometric center of a figure) is the arithmetic mean position of all points that belong to a figure.

The centroid of a figure with a finite number of points is

$$C = \frac{x_1 + x_2 + ... + x_n}{n}.$$

Inertia

As explained above, the K-means algorithm aims to create clusters in a way that minimized the inertia, or within-cluster sum of squared Euclidean distances. The inertia of a cluster is computed using the below formula:

$$\Sigma_{i=0}^{n} \min_{\mu_j \in C}(||x_i - \mu_j||^2)$$

Algorithmic Steps

The K-means algorithm is split into 4 steps.

1. Choose the initial centroids by selecting k samples from the dataset.
2. Assign each sample of the dataset to its nearest centroid, thus populating the cluster.
3. Create new centroids by calculating the mean position of all samples assigned to a centroid.
4. Loop between step 2 and 3 until centroids move less than a threshold.

Properties of K-Means

K-means is an algorithm that works best when the size of the resulting clusters is even, and generally does not deal all that well with outliers.

K-means is characterized as a general-purpose clustering algorithm, and can be used in both small and large datasets. However, K-means suffers from a phenomenon known as the *curse of dimensionality*.

The Curse of Dimensionality

Because *inertia*, the metric that K-means aims to optimize by choosing new centroids at each iterations, is not a normalized metric, meaning that it is not contained within a lower and upper bound, its interpretation is entirely based on the use case. Generally, lower inertia values are better and an inertia value of zero is optimal.

However, when dealing with data that is depicted with many features, which means that K-means has to optimize the clusters in a high-dimensional space, Euclidean distances are inflated by the existence of so many features. In addition, in the domain of machine learning, a high-dimensional dataset needs to contain a sufficiently large number of records, such that there is enough training data for the algorithm to be able to generalize for any unseen inputs.

Thus, a technique known as *dimensionality reduction* is very commonly applied before the clustering takes place to mitigate the impact of this problem. Dimensionality reduction will be explained below, as we make use of it in our thesis for various purposes.

2.6.2 Hierarchical Clustering

Hierarchical Clustering is a clustering method that works by creating a "hierarchy" of clusters. This means that clusters are created by the consecutive merging or splitting of clusters, depending on the method used. There are two approaches to building clusters using this method.

Divisive Clustering

Divisive clustering, as the name implies, is a *top-down* approach that involves clustering the data points in a collection by initially placing them all in the same cluster, then recursively splitting the cluster into two, until the number of clusters, which is given as a parameter to the algorithm, has been reached.

Agglomerative Clustering

Agglomerative clustering, is a *bottom-up* approach where initially, each data point is considered its own cluster, and clusters are merged successively until the desired number of clusters is met.
Our work makes use of the *agglomerative clustering* approach for the grouping of the stack traces.

Linkage Criteria

From the above definitions of divisive and agglomerative clustering, it is clear that clusters are split or merged repeatedly, using some linkage criterion. There are numerous linkage criteria, the most notable of which are briefly explained below. [13]

- **Ward Linkage**: Minimizes the sum of squared differences within all clusters. Aims to reduce the within-cluster variance.

$$\frac{|A| \cdot |B|}{|A \cup B|} ||\mu_A - \mu_B||^2 = \sum_{x \in A \cup B} ||x - \mu_{A \cup B}||^2 - \sum_{x \in A} ||x - \mu_A||^2 - \sum_{x \in B} ||x - \mu_B||^2$$

- **Complete Linkage**: Minimizes the maximum distance between data points in two clusters. In agglomerative clustering, this means that the pair clusters with the furthest observations that have the minimum distance are combined.

$$\max_{a \in A,\, b \in B} d(a,b)$$

- **Average Linkage**: Minimizes the average distance between data points in two clusters. In agglomerative clustering, this means that the pair clusters with the minimum average distance between any two observations are combined.

$$\frac{1}{|A| \cdot |B|} \sum_{a \in A} \sum_{b \in B} d(a,b)$$

- **Single Linkage**: Minimizes the minimum distance between data points in two clusters. In agglomerative clustering, this means that the pair clusters with the closest observations that have the minimum distance are combined.

$$\min_{a \in A,\, b \in B} d(a,b)$$

Each cluster linkage criterion has its advantages and disadvantages, but in our case, experimenting with optimization will determine the the best criterion.

Properties of Hierarchical Clustering

Hierarchical clustering algorithms are scalable, meaning that they could be used for large datasets as well as when there are a large number of clusters to be produced. Agglomerative clustering can also be used with non-Euclidean distance metrics, which makes the algorithm useful in a wider variety of cases.

2.6.3 DBSCAN

Density-Based Spatial Clustering of Applications with Noise (DBSCAN) [14] is a clustering algorithm that works by separating areas of high density using areas of lower density. As it groups together observations that are close together in space, it is able to label distant observations as *outliers*.

DBSCAN Algorithm

There are parameters that control how the algorithm perceives dense and sparse areas, namely *epsilon* and *minimum samples*. Epsilon controls the maximum distance between two observations that can be considered neighbors, and minimum samples controls the number of observations that need to be in the neighborhood of an observation for the observation to be considered a *core point*.

Starting with an arbitrary observation from the collection, the neighborhood of the the observation is computed using a *distance metric*, which is also a parameter given to the algorithm. If the neighborhood contains enough observations to be considered dense, it is considered a cluster, else the selected observation is labelled as noise. An observation that is sufficiently close to enough neighbors is considered a *core sample*.

A step-by-step explanation of the algorithm is explained below.

1. Find all core points and their neighbors in a distance dictated by *epsilon*.
2. Assign unassigned core points to new clusters.
3. For each core point, recursively find all its neighboring points and add them to the same cluster.
4. Iterate through the remaining points in the dataset, and add them to clusters, if possible. The remaining points are considered noise.

Properties of DBSCAN

Because of the way DBSCAN produces clusters by separating high-density areas, it has the ability to produce clusters of any shape, which in our case, is really important, since the shape that our clusters of stack trace embeddings will resemble is unknown.

The additional ability of DBSCAN to categorize points as outliers is also quite promising, because when working with stack traces, it is entirely possible that a bug is triggered only once, therefore, there is only one stack trace that originated from that bug, therefore, we can avoid placing that observation in a cluster where its distance with other members would be too large. As we will see in the Architecture and Implementation chapters, we performed hyper-parameter optimization on the parameters of DBSCAN to produce the best possible clustering results.

2.6.4 OPTICS

Ordering Points To Identify the Clustering Structure (OPTICS) [15], like DBSCAN, is a clustering algorithm that works using density. It addresses one of the issues that DBSCAN suffers from, which is its difficulty in finding clusters in areas where the density of the points varies. It is considered a generalization of the DBSCAN algorithm, and replaces the *epsilon* parameter with a range of values. The algorithm makes use of a *reachability graph*, to which the ability to create clusters in areas of varying density is attributed.

2.6.5 Evaluating Clustering Algorithms

The evaluation of clustering algorithms can be harder than evaluating the performance of a supervised learning algorithm, since there is not always a ground truth dataset that can be compared to the computed clusters. Another obstacle that makes evaluation difficult is the fact that even with a ground truth dataset that contains the optimal labels, an evaluation metric has to somehow avoid comparing the numerical values of the labels assigned by the clustering algorithm, as this will lead to an evaluation that is not representative of the ability of the algorithm to separate the data into clusters. [16] [17]

There are two categories of clustering evaluation metrics: those that compare the resulting labels with a collection of ground truth labels, called *external evaluation* metrics, and those that evaluate the quality of the clusters using internal measures, called *internal evaluation* metrics.

External Evaluation Metrics

In external evaluation, the resulting clusters are compared to pre-existing data that was not used for clustering. The labels assigned to this data were assigned manually, or through

some valid classification/labelling technique that is correct, and serve as ground-truth labels that accurately reflect how the produced clusters should look.
Several such metrics exist, some of which are discussed below:

- **Rand index**: The *Rand index* [18] of a label assignment computes how similar the label assignment is to a ground truth label assignment.

  If C is a ground truth label assignment, and K the produced clustering, with **a** and **b** defined as:

  - **a**: The number of pairs of observations that are in the same cluster in C and in the same cluster in K
  - **b**: The number of pairs of observations that are in different sets in C and in different sets in K

  The rand index (RI) can then be calculated using the below formula:

  $$RI = \frac{a+b}{C_2^{n_{samples}}}$$

  Where $C_2^{n_{samples}}$ is the total number of possible pairs in the dataset.

  Rand index ranges from 0, which indicates random clusters, to 1, which indicates identical clusterings to the ground truth data.

- **Adjusted Rand Index**: Rand index does not guarantee that a random assignment of labels will get a score close to 0, a deficiency which Adjusted Rand Index (ARI) aims to fix. [19]

  The adjusted rand index of a clustering can be computed as follows:

  $$ARI = \frac{RI - E[RI]}{\max(RI) - E[RI]}$$

  Adjusted rand index ranges from -1 to 1, where 0 indicates random clustering, 1 indicates perfect clustering, and -1 indicates clustering performance worse than random.

Internal Evaluation Metrics

In internal evaluation, the produced clustering is evaluated based on the data that was clustered itself, without the need for comparison against a ground-truth labelling. Such measures typically include scoring clusters based on a low inter-class similarity and high intra-class similarity. A weakness of such metrics is that they don't exactly return a representative score for how accurately data is labelled using a clustering algorithm.

As in our work we make use of a ground-truth assignment to evaluate our clusters, we only make use of one internal evaluation metric, *silhouette coefficient*. [20]

The silhouette coefficient for a labelling is a result of two scores[1]:

- **a**: The mean distance between an observation and all other observation in the same cluster.
- **b**: The mean distance between an observation and all other observation in the nearest cluster.

The *silhouette coefficient* for an observation can be calculated using the below formula:

$$s = \frac{b - a}{max(a, b)}$$

The silhouette coefficient of a whole labelling is the mean of the silhouette coefficient for each sample.

2.7 Dimensionality Reduction

Dimensionality reduction is defined as the transformation of data from a high-dimensional to a low-dimensional space, in a way that the low-dimensional product retains some of the information present in the original data. It is often perform to combat the difficulties of working with high-dimensional data, a phenomenon known as the *curse of dimensionality*. [21] [22]

Two applications of dimensionality reduction, which we also utilize in our work are *data visualization* and *cluster analysis* or *clustering*.

2.7.1 Principal Component Analysis

Principal component analysis [23], or PCA in short, is a dimensionality reduction technique used to reduce the dimensionality of a given dataset. PCA aims to retain as many properties about the dataset as possible. PCA works by transforming the data into a coordinate system of the desired dimensions, which are a parameter.

PCA is commonly used to enable the visualization of a dataset by reducing its dimensions to 2 or 3, making the plotting of the data points in the space possible. By visualizing the data in the space, it is possible to identify possible clusters that are formed in the data.

As a dimensionality reduction technique, PCA can also be used as a preprocessing step before using a clustering algorithm on the dataset. PCA works by identifying a set

[1]The silhouette coefficient can be calculated using any distance metric.

of orthogonal vectors, which are called *principal components*, that explain the maximum possible amount of variance in the dataset. Each principal component is successively selected in the direction of the data with the largest variance.

2.8 Similarity Metrics

Similarity metrics in this thesis were used primarily for purposes of evaluating the produced vector embeddings. As stack traces before their encoding into vector embeddings were in plain text format, we made use of metrics for both *string* similarity, as well as *vector* similarity.

2.8.1 Levenshtein Distance

Levenshtein distance [24], also referred to as *edit distance*, is a metric used for measuring the difference between two strings. The Levenshtein distance between two strings is the number of single-character edits that are needed to transform one string into the other.

In our work, we use the Levenshtein *ratio* [25], which is the normalized Levenshtein distance of two strings, to make comparisons between pairs of stack traces in their text format, before they are encoded into vector embeddings using our trained embedding model. Levenshtein ratio is bounded between [0,1], where 0 indicates a pair of two completely different strings and 1 indicates a pair of identical strings.

2.8.2 Cosine Similarity

Cosine similarity is a similarity measure for comparing pairs of vectors. It can be computed using the following formula:

$$cos\theta = \frac{\mathbf{A} \cdot \mathbf{B}}{||\mathbf{A}||\,||\mathbf{B}||}$$

Where $||\mathbf{A}||$, $||\mathbf{B}||$ is the *magnitude* of vector **A**, **B** respectively, and $\mathbf{A} \cdot \mathbf{B}$ is the *dot product* of vectors **A** and **B**.

2.9 Error Metrics

In our work, error metrics were used to evaluate the quality of the produced vector embeddings. Specifically, for each pair of stack traces, various error metrics were computed to compare the cosine similarity of the pair to the Levenshtein distance of the pair.

2.9.1 Mean Absolute Error

Mean Absolute Error (MAE) is calculated as the sum of absolute errors divided by the size of the sample.

$$\text{MAE} = \frac{\sum_{i=1}^{n} |y_i - x_i|}{n} = \frac{\sum_{i=1}^{n} |e_i|}{n}.$$

2.9.2 Mean Square Error

Mean Squared Error (MSE) is the average squared difference between the predicted values and the real values.

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} \left(Y_i - \hat{Y}_i\right)^2.$$

2.9.3 Root Mean Squared Error

Root Mean Squared Error (RMSE) is defined as the square root of the mean square error.

$$\text{RMSE} = \sqrt{\frac{\sum_{i=1}^{n} (Y_i - \hat{Y}_i)^2}{n}}.$$

2.9.4 Mean Absolute Percentage Error

The Mean Absolute Percentage Error (MAPE), also known as Mean Absolute Percentage Deviation (MAPD), is used to express the accuracy of a prediction using a ratio. It is calculated using the following formula:

$$\text{MAPE} = \frac{100\%}{n} \sum_{i=1}^{n} \left| \frac{A_i - F_i}{A_i} \right|$$

Chapter 3

Architecture

The Architecture section first lists the basic tools and technologies used for our work, specifically regarding our fuzzer of choice and the word embedding library used, and the reasoning behind our choices. After that, a high-level overview of our working pipeline is discussed, where we explain our working process through all stages, by including as few implementation-specific details as possible.

3.1 Basic Tools

3.1.1 The use of AFL++ for fuzzing

In this thesis, AFL++ was our fuzzer of choice for the process of collecting the crash reports from a set of target programs. The reason for this choice is that AFL++ integrates features that stem from recent notable fuzzing research.

In addition to the implementation of some of the latest research, AFL++ was also chosen because of its ease of use, and it allowed us to fuzz our target programs with relative ease with regards to set-up and preparation. As a mutation-based fuzzer however, AFL++ does require a pre-existing set of test cases, a requirement that we make sure to satisfy before all fuzzing sessions.

3.1.2 The use of the FastText library for vector embeddings

FastText is a Python library, developed by Facebook, that can be utilized for creating word embeddings. The main idea behind fastText is to represent words as bags of *character n-grams*[1], which allows fastText word embedding models to capture subword information. FastText vector embedding models have the additional advantage of being able to produce

[1]character n-gram: a series of characters of length n. [26]

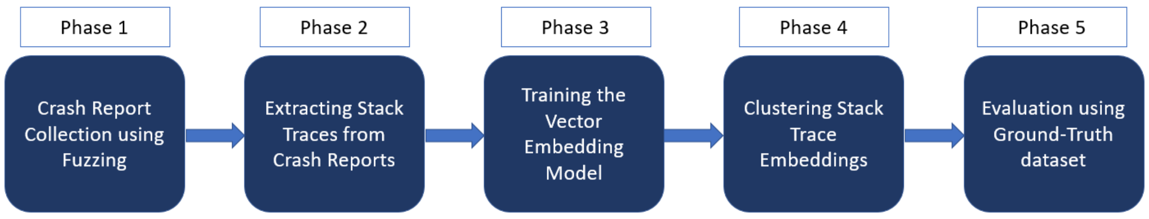
vector embeddings for *unseen* words, as they don't produce vector embeddings for each specific word, but rather for n-grams, as described above. [27] [28] [29]

Why fastText?

Although stack traces are represented by strings in our approach, they are not words in their entirety; They contain information such as address of the invoked function and source code locations which are hexadecimal and decimal numbers respectively. However, because of fastText's ability to capture subword information and produce vector embeddings for *out-of-vocabulary* data, we deemed fastText as an adequate way to train our vector embedding models to produce vectors even for unseen stack traces.

Taking into account the ability of fastText models to create embeddings for unseen stack traces, we performed a *train-test split* on our stack traces before training a fastText skipgram model. More details will be given on how our model was specifically trained in the Implementation section.

3.2 High-level Overview of Pipeline



Phase 1
Crash Report Collection using Fuzzing
Phase 2
Extracting Stack Traces from Crash Reports
Phase 3
Training the Vector Embedding Model
Phase 4
Clustering Stack Trace Embeddings
Phase 5
Evaluation using Ground-Truth dataset

Figure 3.1: *Schematic overview of the 5 pipeline stages*

The work we performed can be separated into 5 distinct phases.

First, we had to create a collection of crash reports, by fuzzing a set of programs to discover vulnerabilities.

After the dataset of crash reports is collected, we implemented tools that can retrieve the stack trace that corresponds to each crash report, thus creating a dataset of stack traces, which is the dataset actually used for the rest of our work.

Then, we perform preprocessing on the stack trace dataset, by formatting the stack traces in such a way that a corpus appropriate for training our vector embedding model is created. In parallel, we parse the stack traces into object representations, and implement methods for stack trace object comparison.

Following that, we use our trained vector embedding model to produce vector embedding representations for each stack trace in the dataset, and evaluate the quality of the embeddings by comparing stack trace similarity metrics between object representations and vector representations, and perform clustering on the dataset of stack trace vector embeddings using a variety of clustering algorithms, and optimize the algorithms using a ground-truth assignment dataset.

In the last phase, we use other ground-truth assignment datasets in order to evaluate the performance of the clustering algorithms, as well as the quality of the resulting clusters and to which extent they are able to provide a solution to the problem of grouping the stack traces based on the bug that caused them.

3.2.1 Creating a collection of crash reports

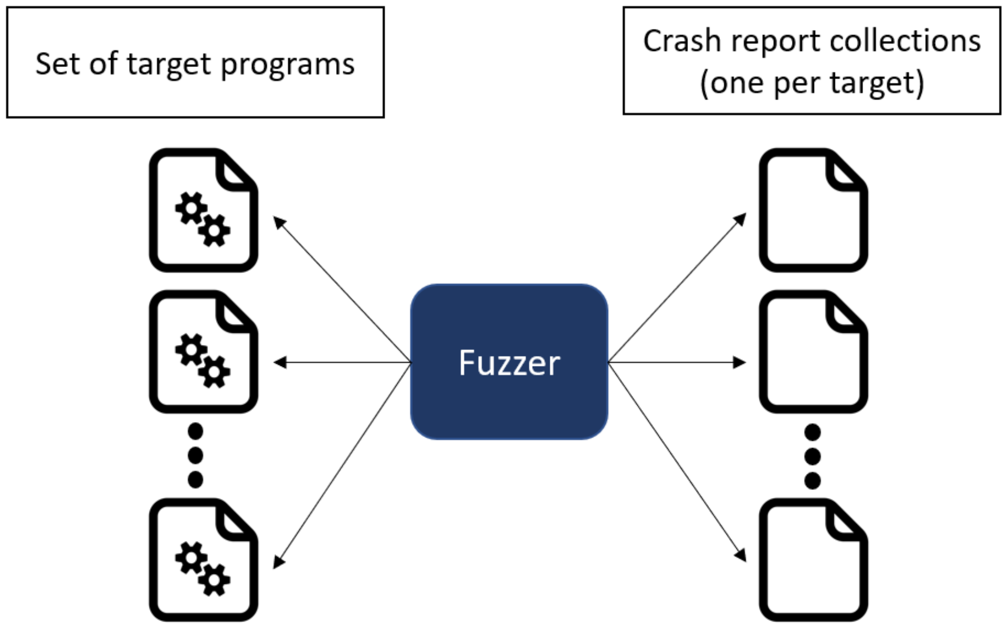


Figure 3.2: *Schematic overview of the collection phase*

In this first phase of our work, the goal is to produce a sizeable dataset of crash reports that can be later leveraged to produce stack traces. The first step in this phase was to identify a set of programs that are going to be the fuzzing targets, as well as choosing a fuzzer.

Although most programs would be satisfactory as fuzzing targets, for the specific purposes of maximizing the size of the dataset of produced crash reports, it is preferable to choose target programs that are known to contain vulnerabilities, which is also the approach we followed in our work.

As with target programs, there are no restrictions regarding the fuzzer that is selected for fuzzing the targets, however, considering our choice of vulnerable target programs, mutation-based fuzzers can be very useful in speeding up the bug finding process, especially if a collection of inputs that trigger the bugs is available. This way, these test inputs can be supplied to the fuzzer, and the fuzzer can generate mutations that could test the buggy code faster. In our work, we selected the mutation-based fuzzer AFL++ for fuzzing, to take advantage of the way mutation-based fuzzers work.

3.2.2 Extracting stack traces from crash reports

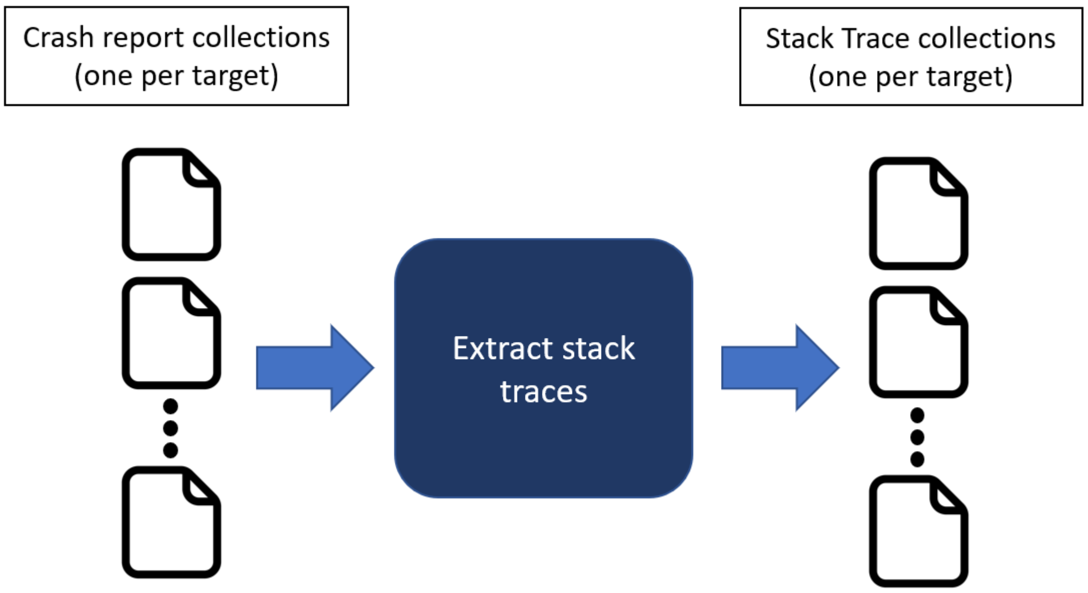


Figure 3.3: *Schematic overview of the stack trace extraction phase*

In order to create a dataset of stack traces, we first needed to leverage the generated crash reports in order to retrieve stack traces for each crash. This can be done in various ways, including running the target programs against the crashing inputs that are recorded by fuzzers, or retrieving the stack traces directly from the fuzzer-generated core dumps.

In our specific case, AFL++ required a redirection of core dumps in order to function, therefore we were not able to directly generate a collection of core dumps through fuzzing. For this reason, we opted to make use of the generated crashing inputs that AFL++ stores in the `crashes/` directory, feeding them into the target program and retrieving each stack trace individually.

It is important to store stack traces from each source in different files, or use a labelling scheme to clearly separate stack traces based on the programs they were extracted from, because during clustering, no stack traces from two different sources should be present in the same dataset fed into the clustering algorithm. Doing so defeats the purpose of grouping the stack traces, as in a real-life scenario, programmers take on the task of prioritization for a single program at a time.

3.2.3 Pre-processing stack traces and creating the train corpus

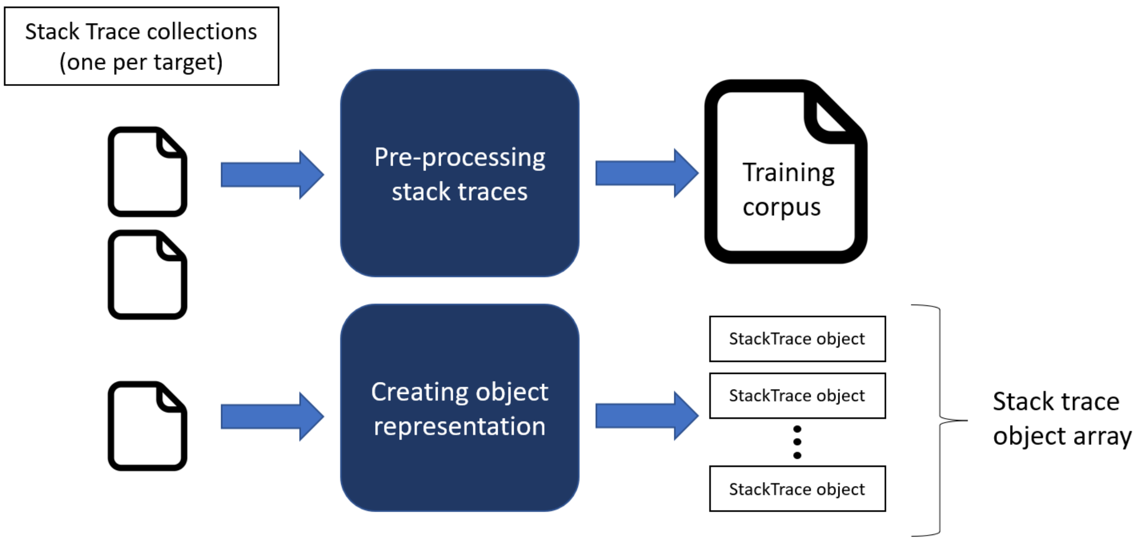

Stack Trace collections (one per target)
Pre-processing stack traces
Training corpus
Creating object representation
StackTrace object
StackTrace object
StackTrace object
Stack trace object array

Figure 3.4: *Schematic overview of the third phase*

At the end of the previous phase, we were left with a collection of datasets of stack traces, each one from a different fuzzed program. In order to produce vector embeddings for each stack trace individually, we first need to create a training corpus for the vector embedding model, in the appropriate format. For fastText, the requested format for train corpora is a `.txt` file containing all information in plain text, therefore we need to pre-process the collection of stack traces.

For pre-processing, we take all datasets of stack traces, rather than one dataset at a time, and perform a *stratified train-test split*. It is recommended to perform a stratified train-test split since it is very likely that the number of stack traces collected per program is not equal. With the use of a stratified train-test split, the distribution of stack traces per source program in the original dataset is retained in the resulting train and test datasets, thus avoiding the phenomenon of unbalanced datasets and any other complications that may arise due to randomized selection. We then *flatten* the train dataset, to create a large single string that contains all stack traces for training, to abide by fastText's requested corpus format, and save it in a `.txt` file.

In parallel, we parse stack traces to create their object representations. This is important, as we will implement stack trace object comparison methods to later evaluate the quality of the embeddings that the embedding model will produce. It should be noted that we also create an adjacent train-test split for these objects, that perfectly matches the above train-test split to facilitate the evaluation of the embeddings.

3.2.4 Embedding production and cluster optimization

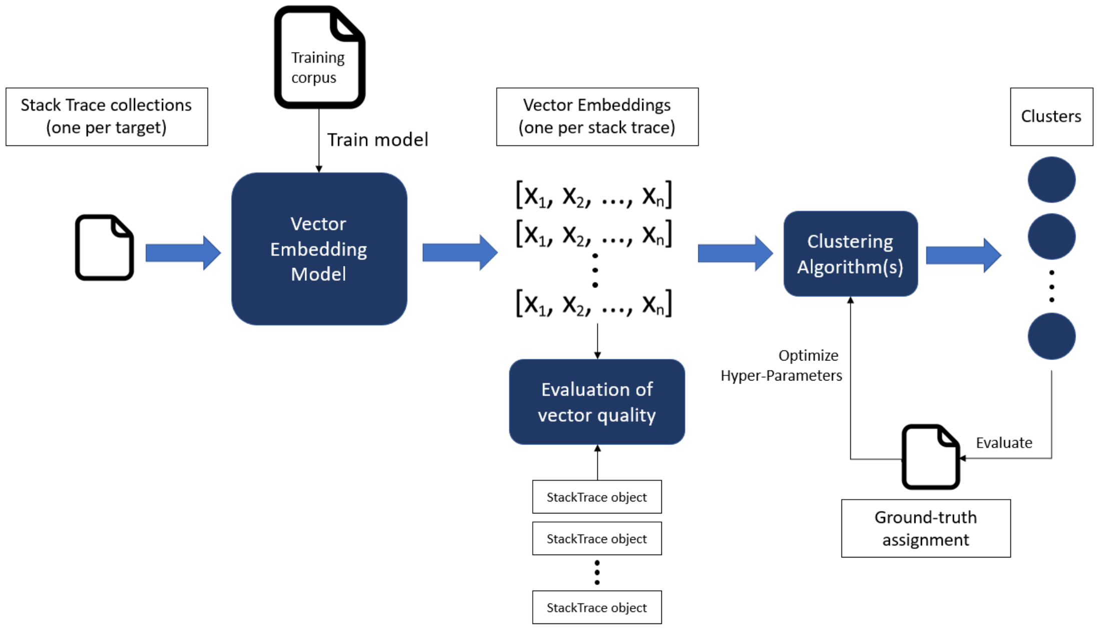


Figure 3.5: *Schematic overview of the clustering and optimization phase*

This phase starts with the training of the vector embedding model using the previously created training corpus of stack traces. The trained model is then used to compute stack trace vector embeddings for each stack trace in our dataset, thus creating a dataset of embeddings that belonged to the training set, and a dataset of embeddings that belonged to the test dataset.

The produced embeddings are then evaluated against a reliable stack trace comparison metric. Specifically, we compare the performance of the *cosine similarity* for a pair of stack trace embeddings to the *Levenshtein ratio* for the same pair of stack trace objects, for all pairs. This way, we are able to illustrate the difference between the two metrics using a variety of error metrics, such as *Mean Absolute Error* and *Root Mean Square Error*. Any metric can be used for comparison to the cosine similarity of the stack trace embeddings, as long as it can be applied to stack traces.

Upon successful evaluation of the produced vector embeddings, we proceeded to clustering. In our work, for optimizing various clustering algorithms, we make use of a specific dataset of stack traces for which the ground-truth label assignment is available, in order to use external cluster evaluation metrics, such as the *adjusted rand score*. We perform hyper-parameter optimization for all selected clustering algorithms in order to maximize the label assignment's adjusted rand score. This phase concludes with the identification of the best hyper-parameters for each clustering algorithm.

3.2.5 Evaluation of produced clusters

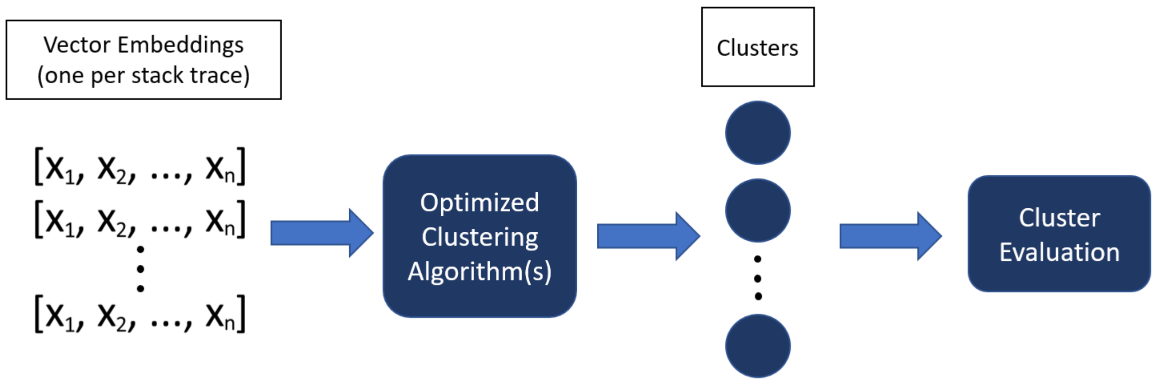


Figure 3.6: *Schematic overview of the evaluation phase*

In this last stage of our pipeline, we aim to use the optimized clustering algorithms to perform grouping of stack traces from all target programs and evaluate the quality of the produced clusters.

For each target program, we feed the stack trace vector embeddings produced by our model into an optimized clustering algorithm and produce a label assignment. In the case that for this target program there is no ground-truth label assignment available, we are limited to evaluating the quality of the clusters using internal cluster evaluation metrics, like the silhouette score. On the other hand, if the ground-truth label assignment is available, we are able to evaluate the clusters using a more reliable metric that better explains how well the stack traces were separated.

In our case, ground-truth label assignments were not available for any target programs, therefore, we manually created our own ground-truth label assignments. Limitations such as the size of the program (in lines of code), as well as the number of stack traces, rendered it difficult for us to manually create ground-truth assignments for all target programs, so in our work we provide a limited number of ground-truth assignments for evaluation.

Chapter 4

Implementation

In the Implementation section, we present the implementation details of our work from start to finish. We go in depth about implementation choices we made regarding the collection of the stack traces, the fuzzing sessions, the pre-processing of the stack traces and training of the model, as well as the clustering algorithms used.

Our goal in this section is to provide a greater understanding about the specifics of our working process, by explaining the detailed steps we took, and certain observations that impacted our choices.

4.1 Implementation of Pipeline

As described in the Architecture section, our work can be separated into 5 phases. First comes the phase of data collection, where we aim to collect crash reports by fuzzing a set of programs to uncover vulnerabilities. Then, we implement and use tools that retrieve the stack traces from each of these crashes and store them in separate text files depending on the target program they originated from. The next phase includes the pre-processing of the stack trace dataset, and the creation of the training corpus for the vector embedding model. After that, we compute vector embeddings from the stack traces, and training and optimize various clustering algorithms. Lastly, we evaluate the performance of the clustering algorithms using some manually labelled ground-truth datasets.

The evaluation of the performance of the clustering algorithms is discussed in detail in the Evaluation section.

4.2 Creating a collection of crash reports

In this first phase, we aimed to collect as many crash reports as possible by fuzzing various programs whose source code was accessible. This task was not trivial, since our goal was to create a sizeable dataset of crash reports for each program, therefore, fuzzing any ordinary program could have not sufficed, as publicly released versions of open-source programs are typically well-maintained and collecting a large number of crash reports would have been a very time-consuming effort.

For this reason, we carefully selected target programs that were either: a) intentionally buggy, or b) injected with bugs. This choice was made to ensure that by fuzzing these targets we would generate a sufficient number of crash reports in a considerably smaller amount of time, and, provided that the programs were injected with multiple bugs, we would discover crash reports stemming from a variety of bugs rather than a collection of crash reports that all are caused by a very small number of bugs in the code.

4.2.1 Fuzzgoat

`Fuzzgoat` [30] is a *buggy* C implementation of a JSON parser adapted from `udp/json-parser`, that has been deliberately written with 4 memory corruption bugs in order to serve as a benchmark for fuzzers and other software testing tools. Fuzzgoat also supplies a directory that contains input files for triggering each bug, which is very useful for fuzzing, as they can be used to generate even more test cases that will result in crashes.

The 4 memory corruption bugs are located in the `fuzzgoat.c` source file, and are commented in the source code for clarity. The comments and few surrounding lines of code for each bug are shown below. The comments for each bug explain what type of bug it is, as well as the input file that should be supplied to the program on execution in order to trigger the bug.

```
         if (value->u.array.length == 0)
            {

/*****************************************************************
  WARNING: Fuzzgoat Vulnerability

  The line of code below frees the memory block referenced by *top
    if
  the length of a JSON array is 0. The program attempts to use
   that memory
    block later in the program.

  Diff       - Added: free(*top);
  Payload    - An empty JSON array: []
  Input File - emptyArray
  Triggers   - Use after free in json_value_free()
*****************************************************************/
```

```

                free(*top);
/****** END vulnerable code ************************************/

                break;
            }
```

Listing 4.1: Bug 1 - *Use-after-free* bug triggered by test case *emptyArray*

```
case json_object:

    if (!value->u.object.length)
    {
        settings->mem_free (value->u.object.values, settings->
    user_data);
        break;
    }

/****************************************************************
  WARNING: Fuzzgoat Vulnerability

  The line of code below incorrectly decrements the value of
  value->u.object.length, causing an invalid read when attempting
   to free the
  memory space in the if-statement above.

  Diff       - [--value->u.object.length] --> [value->u.object.
   length--]
  Payload    - Any valid JSON object : {"":0}
  Input File - validObject
  Triggers   - Invalid free in the above if-statement
****************************************************************/

    value = value->u.object.values [value->u.object.length--].
    value;
/****** END vulnerable code ************************************/

    continue;
```

Listing 4.2: Bug 2 - *Invalid free* bug triggered by test case *validObject*

```
case json_string:

/****************************************************************
```

```
  WARNING: Fuzzgoat Vulnerability

  The code below decrements the pointer to the JSON string if the
   string
  is empty. After decrementing, the program tries to call mem_free
    on the
  pointer, which no longer references the JSON string.

  Diff       - Added: if (!value->u.string.length) value->u.string
   .ptr--;
  Payload    - An empty JSON string : ""
  Input File - emptyString
  Triggers   - Invalid free on decremented value->u.string.ptr
****************************************************************/

    if (!value->u.string.length){
    value->u.string.ptr--;
    }
/****** END vulnerable code ************************************/


/****************************************************************
  WARNING: Fuzzgoat Vulnerability

  The code below creates and dereferences a NULL pointer if the
   string
  is of length one.

  Diff       - Check for one byte string - create and dereference
   a NULL pointer
  Payload    - A JSON string of length one : "A"
  Input File - oneByteString
  Triggers   - NULL pointer dereference
****************************************************************
   */

    if (value->u.string.length == 1) {
        char *null_pointer = NULL;
        printf ("%d", *null_pointer);
    }
/****** END vulnerable code ************************************
   */

    settings->mem_free (value->u.string.ptr, settings->user_data);
    break;
```

Listing 4.3: Bugs 3 and 4 - *Invalid free* bugs triggered by test cases *emptyString* and *oneByteString* respectively

The full source code of `fuzzgoat` is available on GitHub, and is referenced by our work.

Because of `fuzzgoat`'s simplicity, both in how it is smaller than the other fuzzed targets and in how the bugs are clearly commented and distinguishable, the crash reports that were produced from this fuzzing session were manually assigned labels by us and served the purpose of a *ground-truth assignment*, primarily for clustering optimization purposes.

4.2.2 The Rode0day competition

The aforementioned `fuzzgoat` program, although a good starting point for finding possible fuzzing targets that would result in the generation of crash reports, was not enough, as we wanted to explore the quality of the clusters using stack traces from many different target programs.

This prompted us to look into programs that, rather than being written with bugs intentionally, were injected with bugs, thus counting a larger number of bugs present in the source code, that would in return lead to more successful fuzzing sessions.

Rode0day [31] is a competition ran in collaboration between MIT, the MIT Lincoln Laboratory, and NYU. Its purpose is to assist in the evaluation of different bug finding techniques, including fuzzing, by regularly supplying the competitors with a dataset of programs injected with bugs using the LAVA *automated vulnerability injection* framework [32].

This competition, which ran about once per month beginning from July 2018 to May 2020, worked by releasing a set of programs, either in binary form or with a given source code, which had been injected with bugs using LAVA. Teams could then perform program analysis to discover bugs using any bug finding technique they preferred, and at the end of the competition, results would be released that showed the score of each team, as well as the percentage of the discovered bugs and which bugs they found first, if any.

In our work, we downloaded an archive of the all rode0day competitions that had released the relevant dataset, other than the *Rode0day-Beta* competition. In detail, the downloaded competition datasets were the following:

- 18.07 (July 2018)
- 18.09 (September 2018)
- 18.10 (October 2018)

- 18.11 (November 2018)
- 19.01 (January 2019)
- 19.02 (February 2019)
- 19.03 (March 2019)
- 19.05 (May 2019)
- 19.06 (June 2019)
- 19.07 (July 2019)
- 19.09 (September 2019)
- 19.10 (October 2019)

We then investigated the programs that were released with each dataset, and decided to select only the programs whose source code was provided. This choice was made because we want to take advantage of AFL++'s instrumentation capabilities to optimize our fuzzing sessions, maximizing the code coverage achieved with each session.

Not all programs whose source code was available were selected, as there were some programs that presented some difficulties in compilation due to a mismatch in libraries, or other problems that rose during the fuzzing process. Although we attempted to resolve all issues with libraries, there were some target programs whose issues could not be resolved.

We present a list of all fuzzed programs that were sourced from rode0day competitions, as well as the competition they were sourced from.

- **audiofileS**, from competition 19.06
- **fileS3**, from competition 19.09
- **fileS4**, from competition 19.10
- **jpegS2**, from competition 19.05

Since these datasets were released after the end of the competitions, they were also supplied with inputs that directly triggered the injected bugs. These inputs were placed in the directory `solutions/` contained in each competition archive.

In general, the format of each competition dataset is as follows:

The main directory for each competition generally contains 4 common directories: `download/`, `solutions/`, `source/`, and `uploads/`. For our work, the two relevant directories are `download/`, and `source/`.

The `download/` directory contains one subdirectory for each program included in the competition dataset. If the name of the directory includes the letter '*S*', then the source code is included. For example, the directory `fileB2/` does not include the source code, whereas the directory `fileS2/` does. This is important as we only selected programs whose source code was available, so we could leverage the instrumentation abilities of AFL++.

Inside directories that contain the source code, there are 3 common subdirectories: `built/`, `inputs/` and `src/`. The `built/` directory contains a compiled version of the source code. The `inputs/` directory contains a small collection of inputs, that are used as the test corpus for the first of two fuzzing sessions for each target program. Lastly, the `src/` directory contains the source code of the program, and a `Makefile` to compile it.

Regarding fuzzing, we ran two fuzzing sessions on each of the above programs. The first fuzzing session was performed using test inputs that we produced, or that were present in the original release of the competition, and the second fuzzing session was performed using the *solution* inputs. This was done to ensure that even if our first fuzzing session was unsuccessful and produced no crash reports, the second fuzzing session would guarantee the generation of some crash reports, thus populating our dataset.

4.2.3 The LAVA-1 and LAVA-M datasets

In the original publication of LAVA, the authors create two corpora of programs injected with bugs, in order to evaluate the ability of the fuzzers to uncover these bugs. These two corpora are called LAVA-1 and LAVA-M, and are available on Dolan-Gavitt's website. [33]

LAVA-1 contains 69 versions of the `file(1)` Linux utility injected with one bug each, whereas LAVA-M contains 4 programs from the Linux `coreutils` suite: `base64`, `uniq`, `md5sum` and `who`. There exists a `backtrace/` subdirectory for each of these programs that contains the stack traces that correspond to each bug, extracted from a core dump with the use of `gdb`'s [34] `backtrace` command.

As these datasets already contain the stack traces for each bug, we simply parsed these stack traces and added them to our dataset. This was performed using a simple tool we implemented in Python, `parse_backtraces.py`. This short program receives the directory where the stack traces are located and the output JSON file name as command line parameters, and parses the stack trace files in the specified directory to retrieve the stack traces, saving them in a JSON file with the specified name.

A usage example of `parse_backtraces.py` is provided below.

```
  $ python3 parse_backtraces.py <path_to_backtraces> <
output_file_name>
```

4.2.4 Preparing and starting a fuzzing session

For each fuzzing target, we create a directory `inputs/` and `outputs/`. The input directory is used for storing the initial test cases from which the fuzzer will generate new test cases, and the output directory stores information generated by the fuzzing session, such as crash reports and hangs. Hangs are defined as states where the program becomes unresponsive, and do not concern our work in this thesis.

Since we chose fuzzing targets whose source code is available, we leverage compile-time instrumentation to the binary that will be fuzzed by compiling the source code using an alternative version of the `gcc` compiler provided by AFL++, called `afl-gcc`. In the case the source code is compiled with the use of a `Makefile`, we set the default C compiler to `afl-gcc` by changing the value of the environment variable `CC`, and then executing `make`.

In case the target program is compiled using a `Makefile`, instrumentation can be added using the following command, in the same directory as the `Makefile`:

```
  $ CC=afl-gcc make
```

If the target program is compiled without a `Makefile`, simply compile the program using the `afl-gcc` compiler rather than the `gcc` compiler, as shown below:

```
  $ afl-gcc [compiler_flags] <source_file>
```

Once the target program has been compiled into an executable, we started the fuzzing session using the `afl-fuzz` command. The `-i` flag is used for specifying the test input corpus directory, and the `-o` flag is used for specifying the output directory. In case the program takes a file as input, the `-f` flag is used for specifying the filename that test cases will assume upon starting the fuzzing session.

An example instruction for starting a fuzzing session on a binary named `jpeg` that takes a JPEG image as a command line argument is shown below. The relative location of the test corpus is `inputs/` and the relative location of the output directory is `outputs/`. The instrumented binary is located in the relative directory `src/src/jpeg`:

```
$ afl-fuzz -i inputs/ -o outputs/ -f image.jpeg src/src/jpeg
image.jpeg
```

For the fuzzgoat program, the initial test corpus is comprised of the 4 JSON files provided in the GitHub repository, that trigger each of the 4 bugs present in the code.

For each of the programs found in the rode0day competition, we performed two fuzzing sessions. In the first fuzzing session, we used the simple test cases that are located in the <competition_name>/download/<program_name>/inputs/ directory, whereas for the second fuzzing session we use the solution test inputs, provided in the <competition_name>/solutions/<program_name>/ directory. As explained previously, fuzzing with the solution test corpus enables us to ensure crash reports will be generated for each of the selected programs.

4.3 Extracting stack traces from crash reports

Now that we have collected a dataset of crash reports for each fuzzed program, we need to somehow extract the information that is useful to us; the stack traces. This step does not refer to the stack traces extracted from the LAVA-1 and LAVA-M corpora. We implemented a tool for extraction of stack traces given a crashing input, called gdbscript.py.

4.3.1 gdbscript

gdbscript.py is the tool we implemented for extracting the stack trace from a crashing input, as collected in our crash reports. What gdbscript.py does in short, is that it is executed through the gdb debugger, and iterates over the directory that contains the crash reports, executing the program with each crashing input that it finds. When the program crashes, gdbscript.py extracts the stack trace using the backtrace command, and stores them in .txt and .json files for later processing.

In more detail, gdbscript.py takes 2 parameters. The first parameter is a relative location of the directory where the crashing inputs, generated by the fuzzer are located, and the second parameter is the prefix of each output file that will be generated by gdbscript.py. The script then retrieves a list of all crashing input files located in the directory, and removes the README.txt file that AFL++ adds.

The program then uses the function create_directories() to create the directories where the generated datasets will be placed. The function log_crash_filenames() creates a text file and an adjacent JSON file that contain the filenames of the crashing inputs. This function is primarily used for making sure that all crashing inputs were ran against the target program. The bulk of the work is performed through the function `execute_gdb_stacktraces()`, which creates the command that will be executed through gdb by concatenating `gdb`'s `r` command with each crashing input filename. The commands are executed, and appended into a list for logging purposes. After the program crashes, the script executes the `bt` command, short for `backtrace`, which returns the stack trace of the crashed program. The returned stack trace is appended to a list of stack traces.

The generated outputs of the `gdbscript.py` Python program are as follows:
Consider `output_prefix` to be the name of the second parameter the program takes, that specifies the prefix added to each output. The main output directory of the program is `<output_prefix>_information/`, which contains two subdirectories: `json/`, and `txt`. The two directories contain the exact same information, just in different file formats, so we will explain only the `json/` directory.

Inside the `json/` directory 4 files will be generated:

- Prefix `commands.json`: Contains the `gdb` commands that the script executed for running the target program against the crashing inputs.
- Prefix `filenames.json`: Contains the filenames of the crashing inputs, as found in the `outputs/default/crashes` directory created by AFL++.
- Prefix `stacktraces.json`: Contains a dump of all retrieved stack traces in no specific format. The contents of this file are not organized.
- Prefix `org_stacktraces.json`: Contains an organized JSON array of all stack traces retrieved in a 2D array format. The file contains an array whose elements are retrieved stack traces, where each retrieved stack trace is an array of stack records, thus creating this 2-dimensional array. This file is used in the following stages of our implementation.

How to use `gdbscript.py`

Usage of `gdbscript.py` is described in this small section. First, the script needs to be placed in the same directory as the `outputs/` directory generated by AFL++, as it makes use of relative locations to access the crashing inputs. In addition to that, since the program is executed through `gdb`, it cannot take command line parameters. To specify the relative address of the crashing input directory, as well as the prefix added to all outputs generated by the script, we needed to manually change the following lines[1], located in the `main()` function:

[1]The above lines were changed to fit each fuzzed program's individual case.

```python
    args = list()
    args.append("outputs/default/crashes/")
    args.append("<prefix>")
```

Listing 4.4: *Setting the relative location of the crashing inputs directory and the output prefix*

To execute the script, we first needed to run gdb with the target program's compiled and instrumented executable, as shown below:

```
    $ gdb ./<path_to_executable>
```

Then, in gdb's interface, execute the following command:

```
    (gdb) source gdbscript.py
```

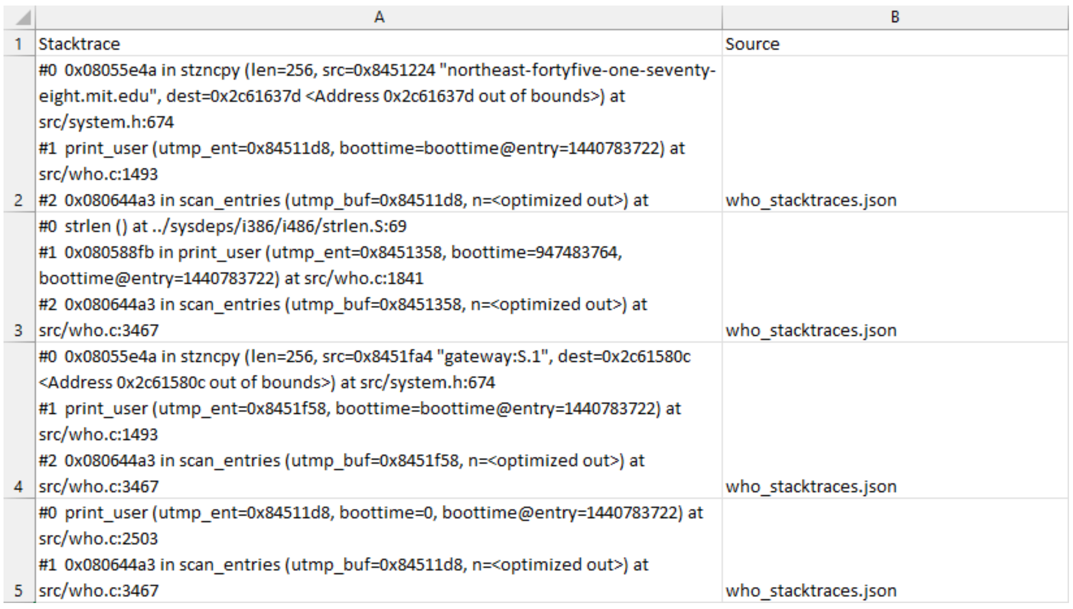
4.4 Preprocessing stack traces and creating the train corpus

With the completion of the previous phase, we now have a collection of files that contain the retrieved stack traces for the fuzzed programs - one for each. In order to train our vector embedding model, and produce embeddings, we first need to prepare and preprocess the stack trace datasets.

4.4.1 Preprocessing of stack traces

We start by reading each stack trace dataset, and creating object representations of each stack trace, using our implemented custom Python objects `StackTrace` and `StackRecord`, which represent complete stack traces and singular stack records respectively. We go into more detail about these in the Stack trace object representation subsection. These object representations are used to evaluate the quality of the produced vector embeddings later on.

Simultaneously to the above procedure, we "flatten" stack traces, meaning that we concatenate all stack records of a stack trace to create a single continuous string that represents the stack trace, and store them back into other files - one for each source program. A file containing the total collection of the preprocessed stack traces is also created. These files follow the `.csv` format, and each record in the file consists of its index, the flattened stack trace, and the source program from which the stack trace was extracted. The latter is done so that we ensure a balanced train-test split before training our vector embedding model, using a *stratified* split.



	A	B
1	Stacktrace	Source
2	#0 0x08055e4a in stzncpy (len=256, src=0x8451224 "northeast-fortyfive-one-seventy-eight.mit.edu", dest=0x2c61637d <Address 0x2c61637d out of bounds>) at src/system.h:674 #1 print_user (utmp_ent=0x84511d8, boottime=boottime@entry=1440783722) at src/who.c:1493 #2 0x080644a3 in scan_entries (utmp_buf=0x84511d8, n=<optimized out>) at	who_stacktraces.json
3	#0 strlen () at ../sysdeps/i386/i486/strlen.S:69 #1 0x080588fb in print_user (utmp_ent=0x8451358, boottime=947483764, boottime@entry=1440783722) at src/who.c:1841 #2 0x080644a3 in scan_entries (utmp_buf=0x8451358, n=<optimized out>) at src/who.c:3467	who_stacktraces.json
4	#0 0x08055e4a in stzncpy (len=256, src=0x8451fa4 "gateway:S.1", dest=0x2c61580c <Address 0x2c61580c out of bounds>) at src/system.h:674 #1 print_user (utmp_ent=0x8451f58, boottime=boottime@entry=1440783722) at src/who.c:1493 #2 0x080644a3 in scan_entries (utmp_buf=0x8451f58, n=<optimized out>) at src/who.c:3467	who_stacktraces.json
5	#0 print_user (utmp_ent=0x84511d8, boottime=0, boottime@entry=1440783722) at src/who.c:2503 #1 0x080644a3 in scan_entries (utmp_buf=0x84511d8, n=<optimized out>) at src/who.c:3467	who_stacktraces.json

Figure 4.1: *Small sample of records from the full dataset CSV file*

The total number of collected stack traces for each program is shown in the table below.

Stack trace dataset	
Source Program	**# of collected stack traces**
LAVA-1 and LAVA-M datasets	
who	2136
md5sum	57
uniq	28
file	69
Fuzzed programs	
audiofileS	104
jpegS2	95
fileS3	63
fileS4	165
fuzzgoat	37
Total	**2754**

Table 4.1: *Collected stack traces per program*

4.4.2 Stack trace object representation

We implement two Python objects `StackTrace` and `StackRecord`, which represent stack traces and their individual stack records respectively.

The `StackRecord` object contains the following fields:

- `function_index`: The index of the stack record in the stack trace.
- `function_address`: The address of the invoked function in memory at the time of the crash.
- `function_name`: The name of the invoked function.
- `function_params`: Parameters passed to the function upon invocation.
- `function_location`: The location of the invoked function in the source code.

The `StackTrace` object contains the following fields:

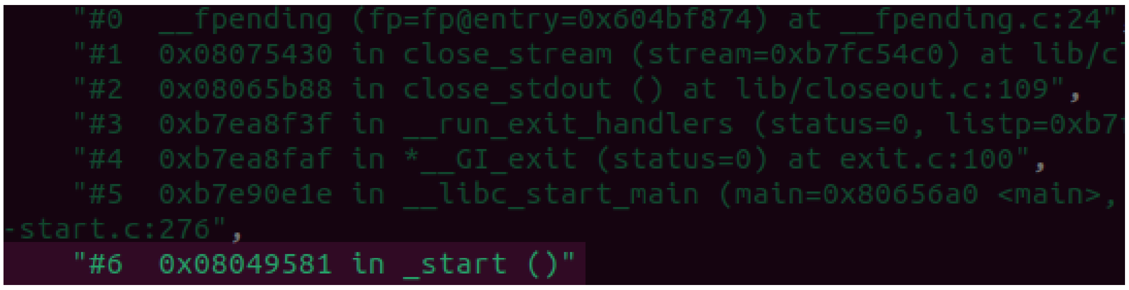
- `stack_records`: A list of `StackRecord` objects, that constitute a complete instance of a stack trace.

The `StackTrace` object is implemented in the source file `stacktrace_object.py`. In the same source file, the function `stacktraces_array_to_objects()` is implemented, which is used for parsing stack traces in the string format into their object representation. This is performed through *regular expression* (RegEx) matching, which is implemented in the function `get_stackrecord_info_format()`. Regular expression matching was used for parsing because we observed that in our dataset of stack traces, there were 4 prominent formats for stack records, therefore, we could not parse the stack trace without recognizing which format each stack record followed.

The 4 identified unique stack trace formats, along with *highlighted* examples from our collected stack trace dataset, are shown in the next page:

Stack trace format 1

```
#[function_index] [function_address] in [function_name]
```

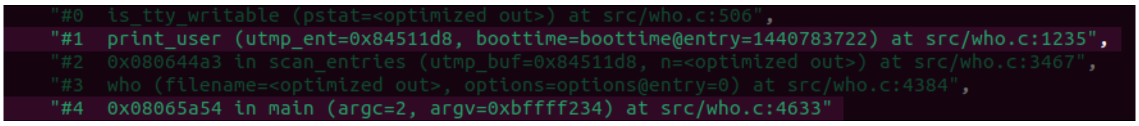


```
    "#0  __fpending (fp=fp@entry=0x604bf874) at __fpending.c:24"
    "#1  0x08075430 in close_stream (stream=0xb7fc54c0) at lib/c
    "#2  0x08065b88 in close_stdout () at lib/closeout.c:109",
    "#3  0xb7ea8f3f in __run_exit_handlers (status=0, listp=0xb7
    "#4  0xb7ea8faf in *__GI_exit (status=0) at exit.c:100",
    "#5  0xb7e90e1e in __libc_start_main (main=0x80656a0 <main>,
-start.c:276",
    "#6  0x08049581 in _start ()"
```

Figure 4.2: *In-dataset example of stack record(s) that follows format 1*

Stack trace format 2

```
#[function_index] [function_address] in [function_name] ([
function_parameters] at [function_location]
```

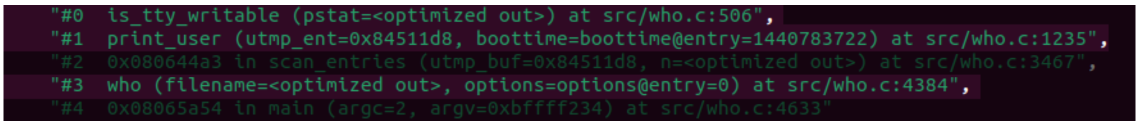


```
    "#0  is_tty_writable (pstat=<optimized out>) at src/who.c:506",
    "#1  print_user (utmp_ent=0x84511d8, boottime=boottime@entry=1440783722) at src/who.c:1235",
    "#2  0x080644a3 in scan_entries (utmp_buf=0x84511d8, n=<optimized out>) at src/who.c:3467",
    "#3  who (filename=<optimized out>, options=options@entry=0) at src/who.c:4384",
    "#4  0x08065a54 in main (argc=2, argv=0xbffff234) at src/who.c:4633"
```

Figure 4.3: *In-dataset example of stack record(s) that follows format 2*

Stack trace format 3

```
#[function_index] [function_name] ([function_parameters] at [
function_location]
```

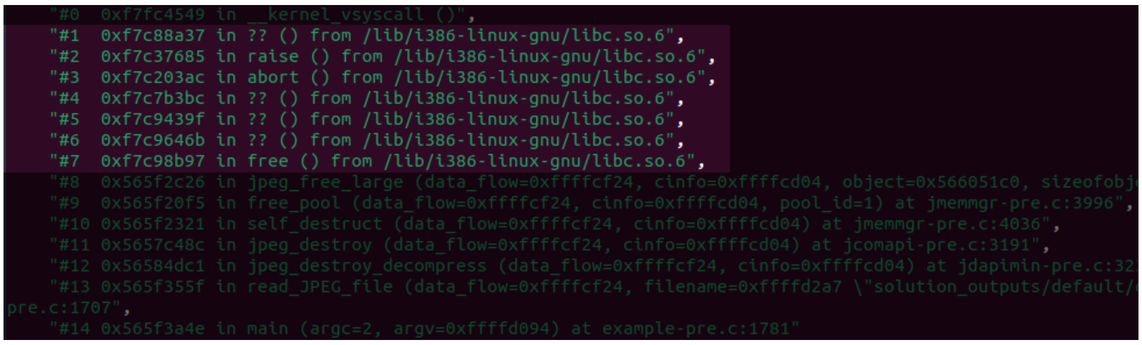


```
    "#0  is_tty_writable (pstat=<optimized out>) at src/who.c:506",
    "#1  print_user (utmp_ent=0x84511d8, boottime=boottime@entry=1440783722) at src/who.c:1235",
    "#2  0x080644a3 in scan_entries (utmp_buf=0x84511d8, n=<optimized out>) at src/who.c:3467",
    "#3  who (filename=<optimized out>, options=options@entry=0) at src/who.c:4384",
    "#4  0x08065a54 in main (argc=2, argv=0xbffff234) at src/who.c:4633"
```

Figure 4.4: *In-dataset example of stack record(s) that follows format 3*

Stack trace format 4

```
#[function_index] [function_address] in [function_name] ([
function_parameters] from [function_location]
```



```
    "#0  0xf7fc4549 in __kernel_vsyscall ()",
    "#1  0xf7c88a37 in ?? () from /lib/i386-linux-gnu/libc.so.6",
    "#2  0xf7c37685 in raise () from /lib/i386-linux-gnu/libc.so.6",
    "#3  0xf7c203ac in abort () from /lib/i386-linux-gnu/libc.so.6",
    "#4  0xf7c7b3bc in ?? () from /lib/i386-linux-gnu/libc.so.6",
    "#5  0xf7c9439f in ?? () from /lib/i386-linux-gnu/libc.so.6",
    "#6  0xf7c9646b in ?? () from /lib/i386-linux-gnu/libc.so.6",
    "#7  0xf7c98b97 in free () from /lib/i386-linux-gnu/libc.so.6",
    "#8  0x565f2c26 in jpeg_free_large (data_flow=0xffffcf24, cinfo=0xffffcd04, object=0x566051c0, sizeofobj
    "#9  0x565f20f5 in free_pool (data_flow=0xffffcf24, cinfo=0xffffcd04, pool_id=1) at jmemmgr-pre.c:3996",
    "#10 0x565f2321 in self_destruct (data_flow=0xffffcf24, cinfo=0xffffcd04) at jmemmgr-pre.c:4036",
    "#11 0x5657c48c in jpeg_destroy (data_flow=0xffffcf24, cinfo=0xffffcd04) at jcomapi-pre.c:3191",
    "#12 0x56584dc1 in jpeg_destroy_decompress (data_flow=0xffffcf24, cinfo=0xffffcd04) at jdapimin-pre.c:32
    "#13 0x565f355f in read_JPEG_file (data_flow=0xffffcf24, filename=0xffffd2a7 \"solution_outputs/default/
pre.c:1707",
    "#14 0x565f3a4e in main (argc=2, argv=0xffffd094) at example-pre.c:1781"
```

Figure 4.5: *In-dataset example of stack record(s) that follows format 4*

As can be seen in Figure 4.5, there are cases where information is not present, with question marks (??) taking its place. In such cases, the question marks are treated as the piece of information that would take their place, according to the format the stack record adheres to. For the above image, the question marks are parsed as the name of the function in each stack record. This is done in order to avoid incomplete fields in our object representation.

4.4.3 Creating the train corpus for the vector embedding model

After preprocessing the stack traces into their current flattened format, we then use the dataset that contains the full collection of stack traces (as shown in Figure 4.1), and perform a *Stratified K-fold shuffle split*, to ensure that the train-test splits retain the distribution of the labels (in this case, the source program of each stack trace) in the original dataset.

```python
stacktraces = pd.readcsv("datasets/full_dataset.csv")

X = stacktraces["Stacktrace"]
y = stacktraces["Source"]

from sklearn.model_selection import StratifiedKFold

skf = StratifiedKFold(n_splits = 5, random_state = 42, shuffle =
    True)
skf.get_n_splits(X,y)
```

Listing 4.5: *Code snippet that performs a Stratified K-fold shuffle split over the full stack trace dataset*

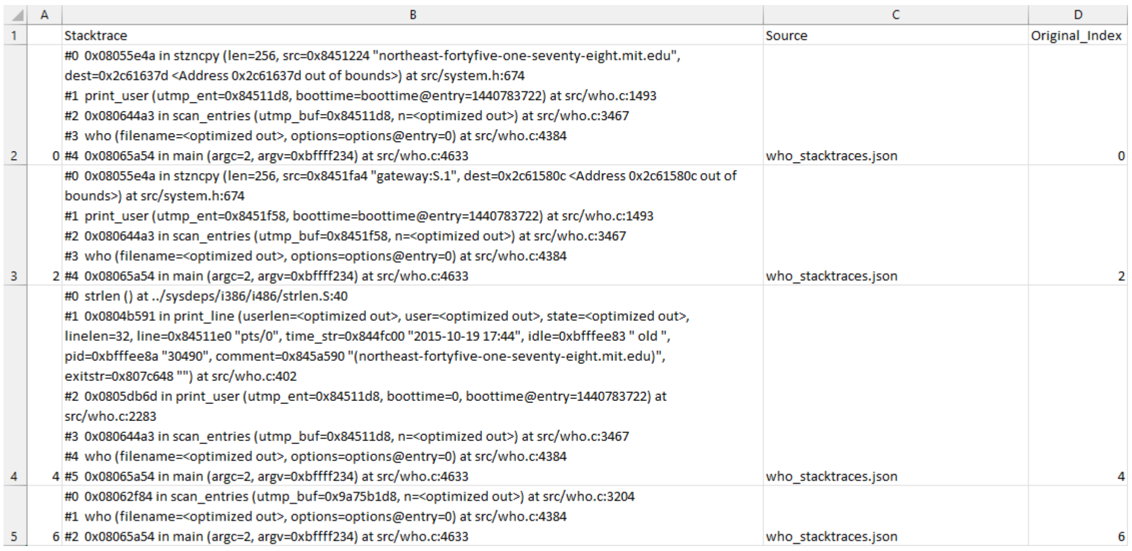
The `StratifiedKFold` Python object can be used to obtain two lists of indices that contain the indices of the records that were placed in the `train` and `test` set respectively. We use these indices to create our balanced train-test split, as shown below.

```python
for train_index, test_index in skf.split(X,y):
    train_indices.append(train_index)
    test_indices.append(test_index)

x_train = X[train_indices[0]]
y_train = y[train_indices[0]]
x_test = X[test_indices[0]]
y_test = y[test_indices[0]]
```

Listing 4.6: *Retrieving the train and test indices to create the train and test datasets*

After that, we store the `train` and `test` datasets in their respective `.csv` files. In addition to the columns found in the full dataset's `.csv`, these two new files feature the column *Original_Index*, which represents the index of each stack trace in the original dataset.



	A	B	C	D
1		Stacktrace	Source	Original_Index
2	0	#0 0x08055e4a in stzncpy (len=256, src=0x8451224 "northeast-fortyfive-one-seventy-eight.mit.edu", dest=0x2c61637d <Address 0x2c61637d out of bounds>) at src/system.h:674 #1 print_user (utmp_ent=0x84511d8, boottime=boottime@entry=1440783722) at src/who.c:1493 #2 0x080644a3 in scan_entries (utmp_buf=0x84511d8, n=<optimized out>) at src/who.c:3467 #3 who (filename=<optimized out>, options=options@entry=0) at src/who.c:4384 #4 0x08065a54 in main (argc=2, argv=0xbffff234) at src/who.c:4633	who_stacktraces.json	0
3	2	#0 0x08055e4a in stzncpy (len=256, src=0x8451fa4 "gateway:S.1", dest=0x2c61580c <Address 0x2c61580c out of bounds>) at src/system.h:674 #1 print_user (utmp_ent=0x8451f58, boottime=boottime@entry=1440783722) at src/who.c:1493 #2 0x080644a3 in scan_entries (utmp_buf=0x8451f58, n=<optimized out>) at src/who.c:3467 #3 who (filename=<optimized out>, options=options@entry=0) at src/who.c:4384 #4 0x08065a54 in main (argc=2, argv=0xbffff234) at src/who.c:4633	who_stacktraces.json	2
4	4	#0 strlen () at ../sysdeps/i386/i486/strlen.S:40 #1 0x0804b591 in print_line (userlen=<optimized out>, user=<optimized out>, state=<optimized out>, linelen=32, line=0x84511e0 "pts/0", time_str=0x844fc00 "2015-10-19 17:44", idle=0xbfffee83 " old ", pid=0xbfffee8a "30490", comment=0x845a590 "(northeast-fortyfive-one-seventy-eight.mit.edu)", exitstr=0x807c648 "") at src/who.c:402 #2 0x0805db6d in print_user (utmp_ent=0x84511d8, boottime=0, boottime@entry=1440783722) at src/who.c:2283 #3 0x080644a3 in scan_entries (utmp_buf=0x84511d8, n=<optimized out>) at src/who.c:3467 #4 who (filename=<optimized out>, options=options@entry=0) at src/who.c:4384 #5 0x08065a54 in main (argc=2, argv=0xbffff234) at src/who.c:4633	who_stacktraces.json	4
5	6	#0 0x08062f84 in scan_entries (utmp_buf=0x9a75b1d8, n=<optimized out>) at src/who.c:3204 #1 who (filename=<optimized out>, options=options@entry=0) at src/who.c:4384 #2 0x08065a54 in main (argc=2, argv=0xbffff234) at src/who.c:4633	who_stacktraces.json	6

Figure 4.6: *Small sample of records from the train dataset CSV file*

Then, the assigned train set is directly printed to a `.txt` file, in order to follow the requested format for training `fastText` vector embedding models.

4.5 Embedding production and cluster optimization

4.5.1 Training the word embedding model

We then train a `fastText` word embedding model on the train dataset. Now that we have a trained model, we produce the vector embeddings for all collected stack traces.

```python
import fasttext
```

```

model = fasttext.train_unsupervised("datasets/split/train_corpus.
    txt",
        model = "skipgram",
        dim = 100,
        epoch = 50)
```

Listing 4.7: *Code snippet - Training a fastText vector embedding model*

As shown in the above code snippet, we train our vector embedding model to produce vector embeddings of *100 dimensions*, which is the default parameter value for fastText models. This is so that our vector embeddings do not have too many dimensions, thus amplifying the effect of the curse of dimensionality, but also not too few, as we can still apply dimensionality reduction techniques to reduce our dimensions if need be.

4.5.2 Evaluating vector embeddings

To evaluate our stack trace vector embeddings, we compared the *cosine similarity* of stack trace embedding pairs to the *Levenshtein ratio* of stack trace object pairs. We aim to achieve a cosine similarity between any two stack trace embeddings that is as close as possible to the Levenshtein ratio between the corresponding stack trace objects.

We compare cosine similarity and Levenshtein ratio of all possible pairs in the train and test dataset, and compute the Mean Absolute Error, Root Mean Square Error and Mean Absolute Percentage Error between the two metrics.

We compute the *Levenshtein ratio* of stack traces by creating a string representation of the `StackTrace` object, which includes the fields `function_index`, `function_name` and `function_location` for each `StackRecord` object in the stack record array of the stack trace. Function addresses and function parameters are not included, as there were specific formats for which function addresses and/or parameters were not present, and were denoted as `null`. The string representation of `StackTrace` objects is performed with the use of the `serialize_stacktrace()` method of the `StackTrace` object.

```
def serialize_stacktrace(self):
    #Get a single-line representation of the stacktrace
    #The features kept are: index, function_name and location
    serial_stacktrace = ""
    for record in self.stack_records:
        serial_stacktrace+=' '.join([record.function_index,
  record_function_name, record_function_location])
        serial_stacktrace+=' ,'

    return serial_stacktrace
```

Listing 4.8: Method definition of the *serialize_stacktrace()* method

The comparison of two `StackTrace` objects is facilitated through the `compare()` method. The method returns a tuple consisting of two comparison metrics: Levenshtein *distance* (`distance`), and Levenshtein *ratio* (`norm_distance`), the latter of which is used in our work for the evaluation of the vector embeddings.

```
def compare(self, other_stacktrace):
    #Compare Levenshtein distance between all records in the two
    stacktraces
    serial_stacktrace1_str = self.serialize_stacktrace()
    serial_stacktrace2_str = other_stacktrace.serialize_stacktrace
    ()

    distance = Levenshtein.distance(serial_stacktrace1_str,
    serial_stacktrace2_str)
    norm_distance = Levenshtein.ratio(serial_stacktrace1_str,
    serial_stacktrace2_str)

    return distance, norm_distance
```

Listing 4.9: Method definition of the *compare()* method

The evaluation results of the vector embeddings are explained in detail in the Evaluation section.

4.5.3 Training and hyper-optimizing clustering algorithms

As described in the stack-trace collection step, we created a labelled ground-truth assignment for the stack traces extracted by fuzzing `fuzzgoat`.

It should be noted that the vector embeddings produced by fastText have 100 features, meaning that they are very high-dimensional. With the curse of dimensionality in mind, we also produced parallel datasets of embeddings of 3-dimensions using the PCA dimensionality reduction technique.

```
from sklearn.decomposition import PCA

X = df_ground_truth_embeddings.iloc[:, :-1]      #Remove the last
    column and get the features
y = df_ground_truth_embeddings.iloc[:, -1]       #Get the last
    column - ground truth label

pca_2D = PCA(n_components = 2)  #Reduce to 2 dimensions
pca_result_2D = pca_2D.fit_transform(X)
```

```

pca_3D = PCA(n_components = 3)  #Reduce to 3 dimensions
pca_result_3D = pca_3D.fit_transform(X)
```

Listing 4.10: *Code snippet - Dimensionality reduction using PCA*

As can be seen in the above code snippet, we performed dimensionality reduction *twice*. The first time we reduced the dataset to 2 dimensions, and the second time we reduced the dataset to 3 dimensions. By using dimensionality reduction, we were able to visualize our stack trace embeddings in the 2-d and 3-d space respectively, thus getting a general idea of how the produced clusters should look.

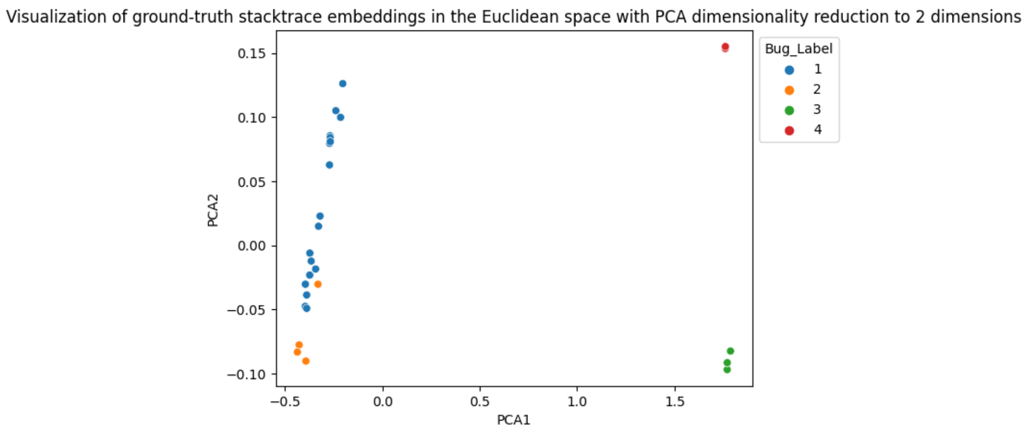


Figure 4.7: *Visualization of ground-truth embeddings reduced to 2 dimensions*

By observing Figure 4.7, we can see that there definitely are groups of stack trace embeddings, particularly in the top-right and bottom-right of the plot, however, in the left side of the plot, we can see that stack traces from 2 different labels seem to be difficult to separate, therefore, creating 2 optimal clusters for these labels would be very difficult.

With the simplicity of `fuzzgoat`'s stack traces in comparison to the stack traces of other more complex fuzzed programs, we deduced that the dimensionality reduction of vector embeddings down to 2 dimensions was too drastic, and it would negatively impact the quality of our clusters.

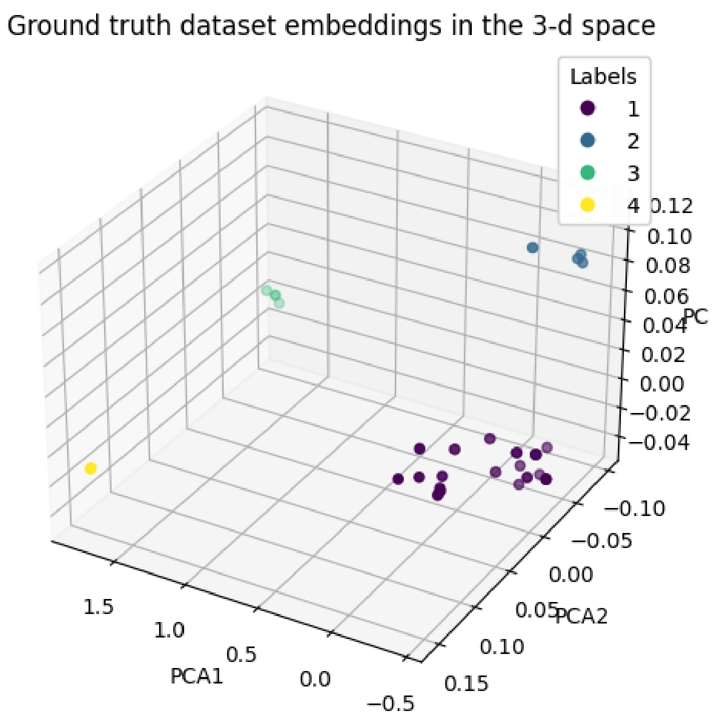


Figure 4.8: *Visualization of ground-truth embeddings reduced to 3 dimensions*

As seen above in Figure 4.8, the groups of stack traces are now much more defined and distant from one another. This observation indicates that the dimensionality reduction of stack trace embeddings down to 3 dimensions preserves enough information to allow for optimal clusters to be computed. As a result, we perform clustering for both the 100-dimensional dataset and the 3-dimensional dataset.

We train a variety of clustering algorithms on this ground-truth dataset, and attempt to optimize them in order to maximize the *adjusted rand score* of the computed labelling. Below is a list of the clustering algorithms that were optimized.

- **K-Means**: Selected because we know the number of labels/clusters for the ground-truth assignment; a general-purpose clustering algorithm.
- **Agglomerative Clustering**: Selected because we know the number of labels/clusters for ground-truth assignment.
- **DBSCAN**: Selected because DBSCAN works with clusters that are uneven or not flat in shape. Also does not require the number of labels/clusters as an input parameter.
- **OPTICS**: Like DBSCAN, works with clusters that are uneven or not flat in shape. Also does not require the number of labels/clusters as an input parameter.

For each algorithm, we explain which parameters were optimized using hyper-parameter tuning, provide the code snippet that performed the optimization, as well as the best estimator's *adjusted rand score*.

K-Means

The parameters of K-Means clustering that were subject to optimization are the following:

- `n_clusters`: The number of clusters to be formed by the algorithm.
- `init`: Centroid initialization method.
- `n_init`: Times the algorithm is ran using different starting centroids.
- `max_iter`: Maximum number of iterations for the K-Means algorithm for a run.

Agglomerative Clustering

The parameters of agglomerative clustering that were subject to optimization are the following:

- `n_clusters`: The number of clusters to be formed by the algorithm.
- `linkage`: The linkage criterion to be used. (See linkage criteria)

Since `sklearn`'s implementation of agglomerative clustering allows for the number of clusters to not be passed as a parameter, we also optimized a different set of parameters for which the number of clusters was not pre-defined. In the following tables, this optimization of agglomerative clustering is displayed as **Agglomerative Clustering (Alternative)**.

- `metric`: The metric used to compute the linkage. We used cosine similarity and euclidean distance.
- `linkage`: The linkage criterion to be used.
- `distance_threshold`: The linkage distance threshold at or above which clusters will not be merged.

DBSCAN

The parameters of the DBSCAN clustering algorithm that were subject to optimization are the following:

- `eps`: The value of epsilon - maximum distance between neighboring samples.

- `min_samples`: The number of samples required for a point to be considered a core point.
- `algorithm`: Algorithm to be used for finding nearest neighbors.

OPTICS

The parameters of the OPTICS clustering algorithm that were subject to optimization are the following:

- `max_eps`: The maximum epsilon value - maximum distance between neighboring samples.
- `min_samples`: The number of samples required for a point to be considered a core point.
- `algorithm`: Algorithm to be used for finding nearest neighbors.

Optimization results

The below table shows the best achieved *adjusted rand index* (ARI) for each of the 4 optimized algorithms, and the values of the parameters that achieved that result.

Optimization Results				
Algorithm	**Best ARI on 100 dimensions**	**Parameters**	**Best ARI on 3 dimensions**	**Parameters**
K-Means	0.6367	n_clusters=2, init='kmeans++', n_init=10, max_iter=100	0.6367	n_clusters=2, init='kmeans++', n_init=10, max_iter=100
Agglomerative Clustering	0.9909	n_clusters=5, linkage='single'	1.0	n_clusters=4, linkage='single'
Agglomerative Clustering (Alternative)	0.9849	metric='euclidean', linkage='single', distance threshold=0.1	1.0	metric='euclidean', linkage='single', distance threshold=0.1
DBSCAN	0.9849	eps=0.1, min_samples=1, algorithm='auto'	1.0	eps=0.1, min_samples=1, algorithm='auto'
OPTICS	0.3554	max_eps=0.1, min_samples=7, algorithm='auto'	0.3554	max_eps=0.1, min_samples=7, algorithm='auto'

Table 4.2: *Best achieved adjusted-rand-index (ARI) scores for each clustering algorithm*

Visualization of results

We decided to visualize the label assignment for each algorithm on the ground-truth dataset with 3 dimensions, to illustrate the difference in performance between clustering algorithms.

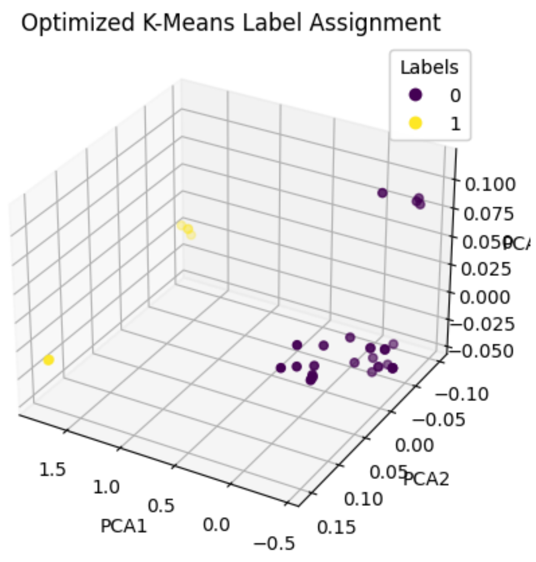


Figure 4.9: *Optimized K-Means assigned labels on ground-truth data*

From Figure 4.9, it is visible that the best performance achieved by K-Means clustering on the ground truth label assignment is not satisfactory, as it produces 2 clusters, rather than the expected 4.

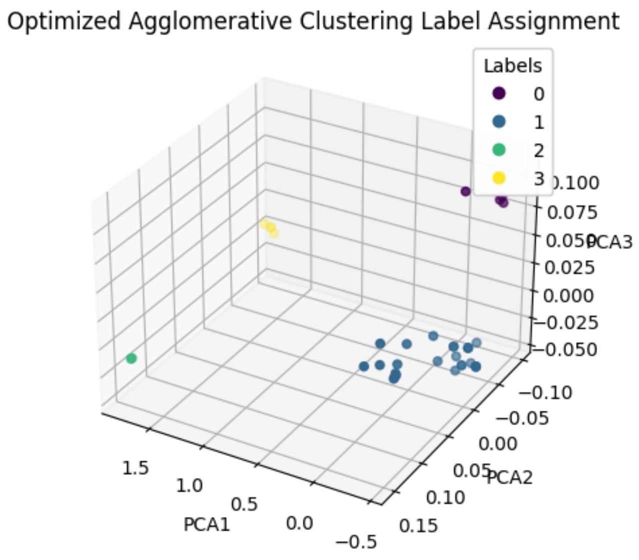


Figure 4.10: *Optimized agglomerative clustering assigned labels on ground-truth data*

On the other hand, Agglomerative Clustering (Figure 4.10) scored a perfect adjusted rand index score.

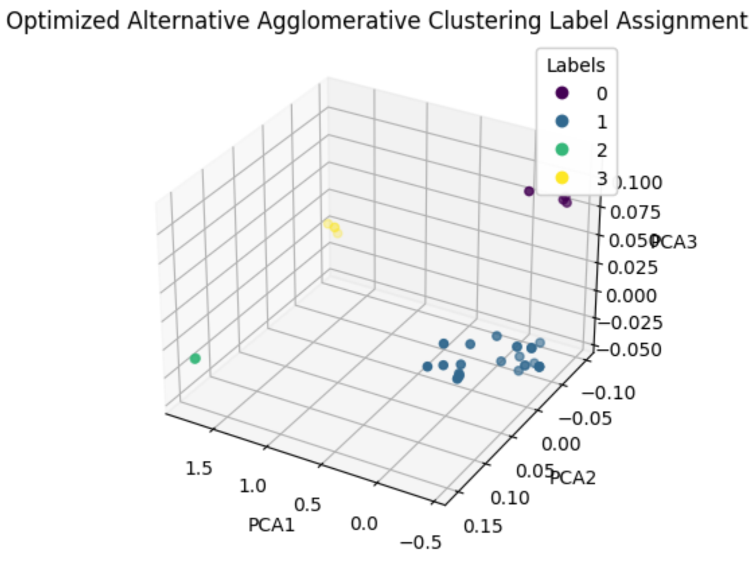


Figure 4.11: *Optimized agglomerative clustering (alternative) assigned labels on ground-truth data*

The alternative Agglomerative Clustering algorithm, where the number of clusters is not predefined also scored a perfect adjusted rand index score, managing a perfect label assignment, which can be seen in Figure 4.11.

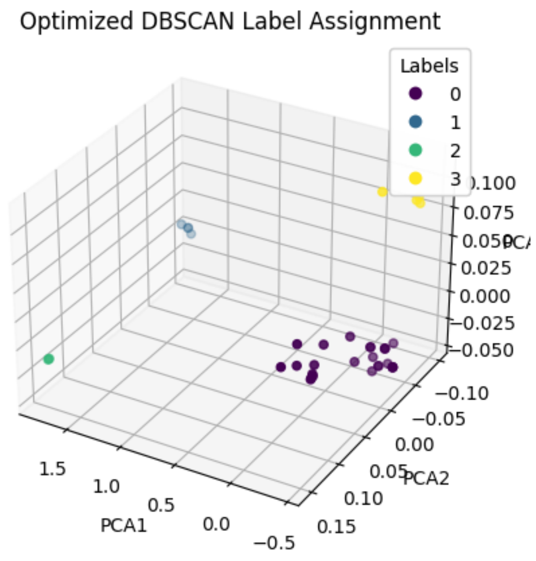


Figure 4.12: *Optimized DBSCAN assigned labels on ground-truth data*

Similar to Agglomerative Clustering, DBSCAN (Figure 4.12) also managed a perfect label assignment.

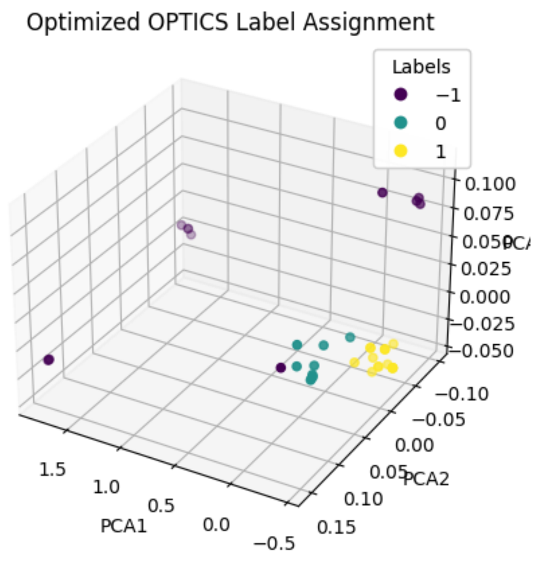


Figure 4.13: *Optimized OPTICS assigned labels on ground-truth data*

Lastly, the OPTICS clustering algorithm (Figure 4.13) did not match the performance of Agglomerative Clustering and DBSCAN, and seems to label a lot of stack trace embeddings as outliers (labelled with `-1`).

Interpretation of results

From Table 4.2 and Figures 4.9, 4.10, 4.11,4.12, 4.13 it became clear to us that the K-Means and OPTICS algorithms achieve a significantly lower adjusted rand index score compared to Agglomerative Clustering and DBSCAN, and are not suitable for use with the larger and more complex datasets of stack traces that we have collected. This prompted us to try to explain the reasons why these two algorithms performed considerably worse, while at the same time justifying our choice to not use them for the final evaluation of our work.

K-Means, although a widely used general-purpose clustering algorithm, performs best in cases where the cluster size is even, and has been proved to lean towards creating clusters of spherical shape. In our specific problem of clustering stack traces, there are no indications that the formed clusters assume a spherical shape, so this property of K-Means is detrimental to its performance.

OPTICS, despite being described as a generalization of DBSCAN, suffers from the existence of near-duplicate points, which are very common among stack traces, since a bug can be triggered multiple times, thus creating numerous very similar crash reports. OPTICS makes use of a reachability matrix and reachability distances, which for very similar points approach 0, therefore making it difficult to discern between high-density and low-density areas.

Chapter 5

Evaluation

In this chapter, we evaluate the performance of our proposed approach, both by measuring the quality of the produced stack trace vector embeddings, and by evaluating the performance of the clustering algorithms we used. More specifically, we discuss in detail how the evaluation of each approach was performed, as well as the final results of each evaluation.

As explained above, there were two milestones of our approach that were evaluated: The quality of the produced vector embeddings, and the performance of the clustering algorithms.

5.1 Evaluating the produced vector embeddings

To evaluate the produced stack trace vector embeddings, we compared the *cosine similarity* of stack trace vector embedding pairs to the *Levenshtein ratio* of the corresponding stack trace object pairs, for both the train and test dataset. In our work, we considered the Levenshtein ratio of two stack traces as the standard stack trace comparison metric, therefore, ideally, we wanted to achieve cosine similarity values that are as close as possible to the Levenshtein ratios, for each pair. In Subsection 4.5.2 of the Implementation section, we thoroughly go over the necessary work we performed to facilitate the comparison of the two metrics, such as the creation of the stack trace object representation, the `StackTrace` and `StackRecord` Python objects, as well as the implementation of the `compare()` method, that returns the Levenshtein ratio of two `StackTrace` objects.

In this section, we go over the actual procedure of evaluating the vector embeddings, which includes the computation of the cosine similarity of all stack trace pairs, and the computation of the Levenshtein ratio for all corresponding stack trace object pairs, their visualization using heatmaps, and finally, their evaluation using three error metrics: Mean Absolute Error (MAE), Root Mean Square Error (RMSE), and Mean Absolute Percentage

Error (MAPE).

5.1.1 Computing Levenshtein ratio for all pairs in each dataset

Using the indices produced by `StratifiedKFold` when we performed the train-test split of the dataset, we also performed an equivalent train-test split for the dataset containing the stack trace object representation, thus producing train-test datasets parallel to the stack trace embedding train-test datasets.

The following code snippets were used to compute the Levenshtein ratio of all pairs, placing them in a 2-dimensional Python list.

```python
#Create the 2D array of Levenshtein distances for both train and test
train_levenshtein_distance = list()
test_levenshtein_distance = list()

#TRAIN
for x in stacktraceobj_train_corpus:
    tmp = list()
    for y in stacktraceobj_train_corpus:
        tmp.append(x.compare(y)[1])

    train_levenshtein_distance.append(tmp)

#TEST
for x in stacktraceobj_test_corpus:
    tmp = list()
    for y in stacktraceobj_test_corpus:
        tmp.append(x.compare(y)[1])

    test_levenshtein_distance.append(tmp)
```

Listing 5.1: Code snippet - Computing the Levenshtein ratio of all pairs in the train and test datasets

In the above snippet of code, we make use of the implemented `StackTrace` object and its `compare()` method. As a reminder, the `compare()` method returns a 2-tuple, which contains the Levenshtein distance in index `0`, and the Levenshtein ratio in index `1`.

5.1.2 Computing cosine similarity for all pairs in each dataset

To compute the cosine similarity between a pair of stack trace vector embeddings, we first define a `cosine_similarity()` function that takes two vectors as parameters and returns their cosine similarity.

```
def cosine_similarity(vec1, vec2):
    arr1 = np.array(vec1)
    arr2 = np.array(vec2)
    cosine = np.dot(arr1, arr2) / (np.linalg.norm(arr1)*np.linalg.
   norm(arr2))
    return cosine
```

Listing 5.2: Code snippet - Cosine Similarity function

With the definition of this function, we can now compute the cosine similarity of all stack trace vector embedding pairs, using the following snippet.

```
train_cosine_similarity = list()
test_cosine_similarity = list()

#TRAIN
for i in range(df_train_embeddings.shape[0]):
    tmp = list()
    for j in range(df_train_embeddings_shape[0]):
        tmp.append(cosine_similarity(df_train_embeddings.iloc[i].
   values, df_train_embeddings.iloc[j].values))

    train_cosine_similarity.append(tmp)

#TEST
for i in range(df_test_embeddings.shape[0]):
    tmp = list()
    for j in range(df_test_embeddings_shape[0]):
        tmp.append(cosine_similarity(df_test_embeddings.iloc[i].
   values, df_test_embeddings.iloc[j].values))

    test_cosine_similarity.append(tmp)
```

Listing 5.3: Code snippet - Computing the cosine similarity of all pairs in the train and test datasets

5.1.3 Visualization of produced grids

In the above code snippets, we created 2-dimensional lists for each similarity metric, one for the train dataset and one for the test dataset. Before calculating the error between the two metrics, we visualized the produced grids using heatmaps, in order to get a visual idea of how similar the values calculated by each metric are. The heatmaps offer a colorbar on the legend that illustrates what the color gradient means, for each heatmap. Because of the size of the train and test datasets, interpreting the produced heatmaps can be quite difficult, so our provided observations help to understand what is seen in the figures.

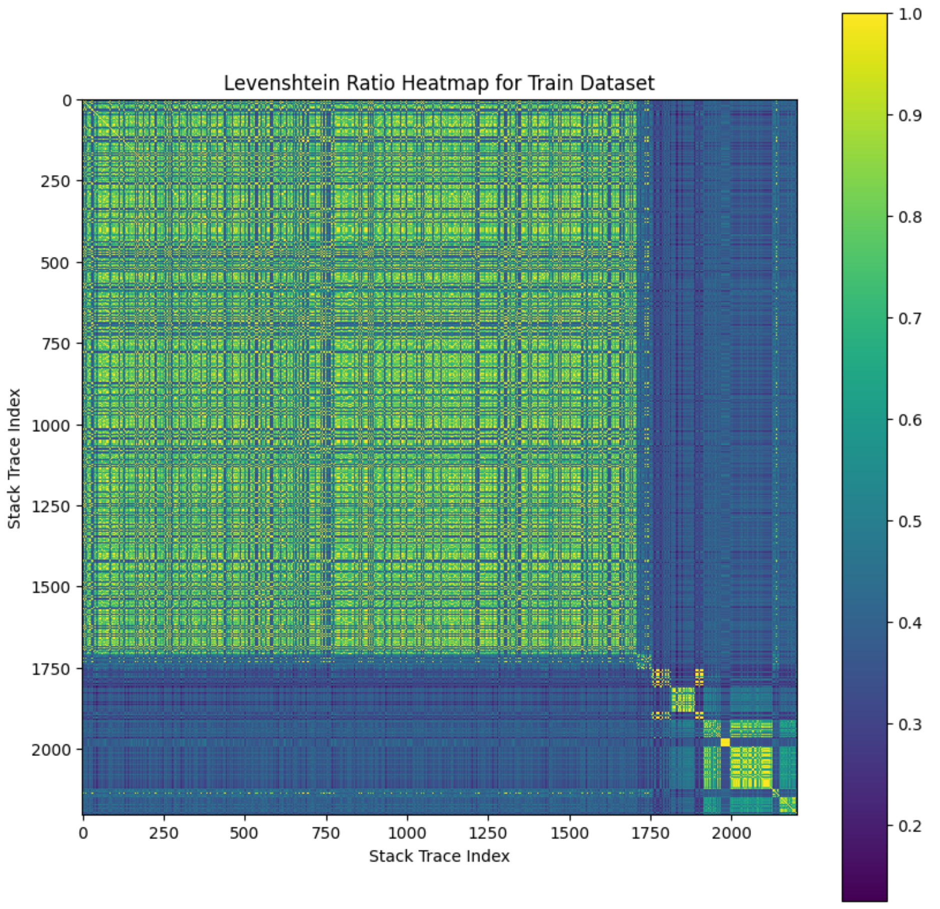


Figure 5.1: *Levenshtein ratio heatmap for the train dataset*

Firstly, the main diagonal of the grid is populated with values of 1.0, which is logical, since the comparison is between the same `StackTrace` object. Most importantly, we can see areas of higher similarity values, that are again situated on each side of the main diagonal. This is because these stack trace pairs were generated by crash reports originating from the same fuzzed program, therefore they are more similar than stack traces that were produced by crashes from two completely different programs.

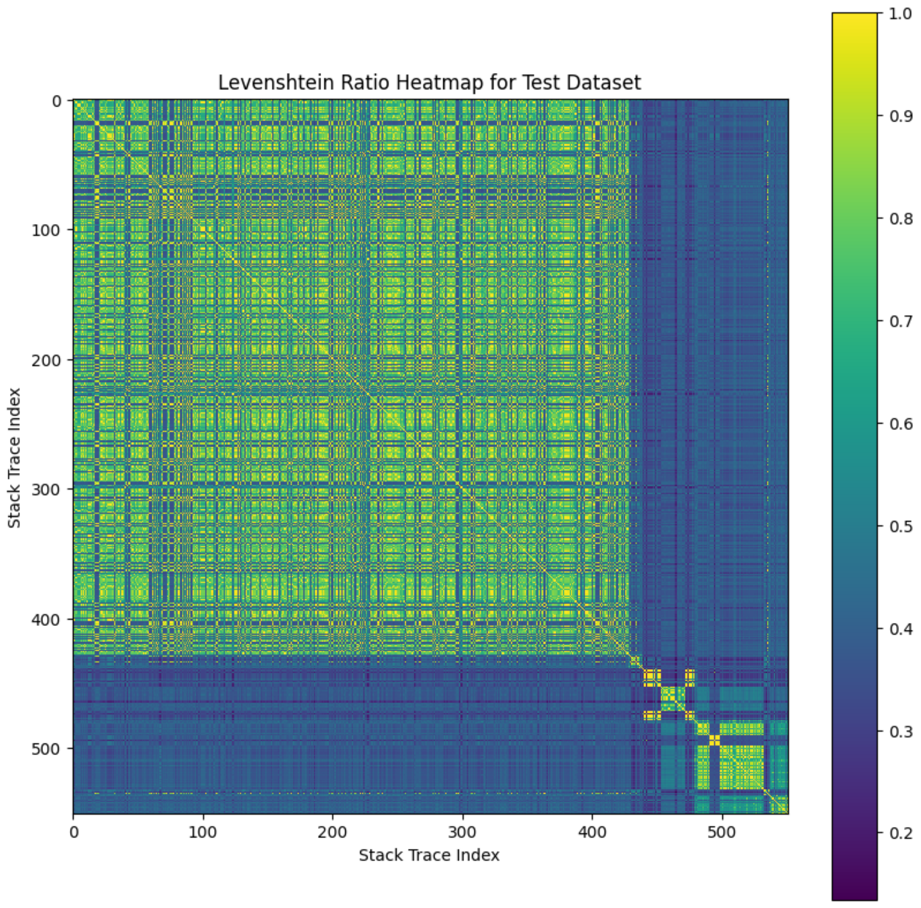


Figure 5.2: *Levenshtein ratio heatmap for the test dataset*

Similar to the Levenshtein ratio heatmap for the train dataset shown in Figure 5.1, the Levenshtein ratio heatmap for the test dataset has a main diagonal full of identical stack traces. Also, there are the areas of higher similarity values of the same ratio, which can be attributed to the StratifiedKFold split we performed to create the train-test dataset.

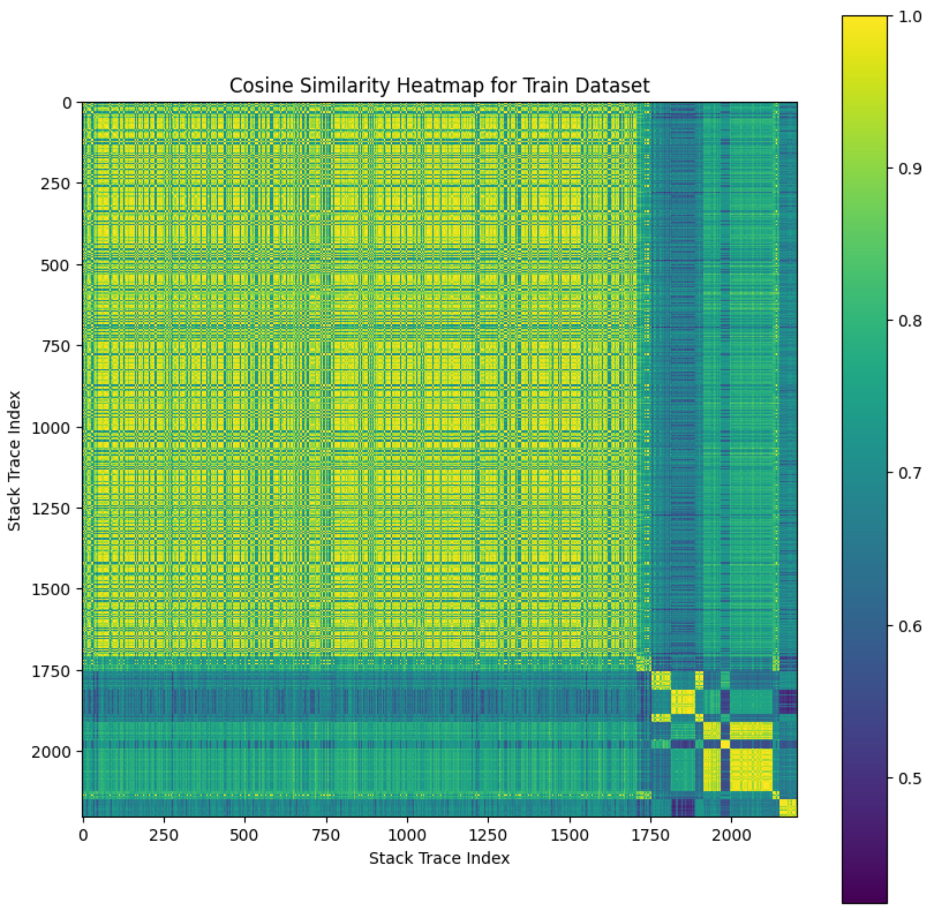


Figure 5.3: *Cosine similarity heatmap for the train dataset*

In Figure 5.3, we observe that the cosine similarity heatmap for the train dataset, although not identical to the Levenshtein ratio heatmap for the train dataset, shown in figure 5.1, is very similar. Other than the main diagonal, we can still notice the higher similarity areas around the main diagonal, which represent the stack traces originating from the same source programs.

A difference that we noticed between the two heatmaps is that the cosine similarity values of stack trace pairs are generally higher, which visually explains the lighter colors of the heatmap. This observation is further supported by the colorbar of the figure, where it can be seen that the lower range of the cosine similarity values is a little lower than 0.5, whereas for Levenshtein distance, it is lower than 0.2.

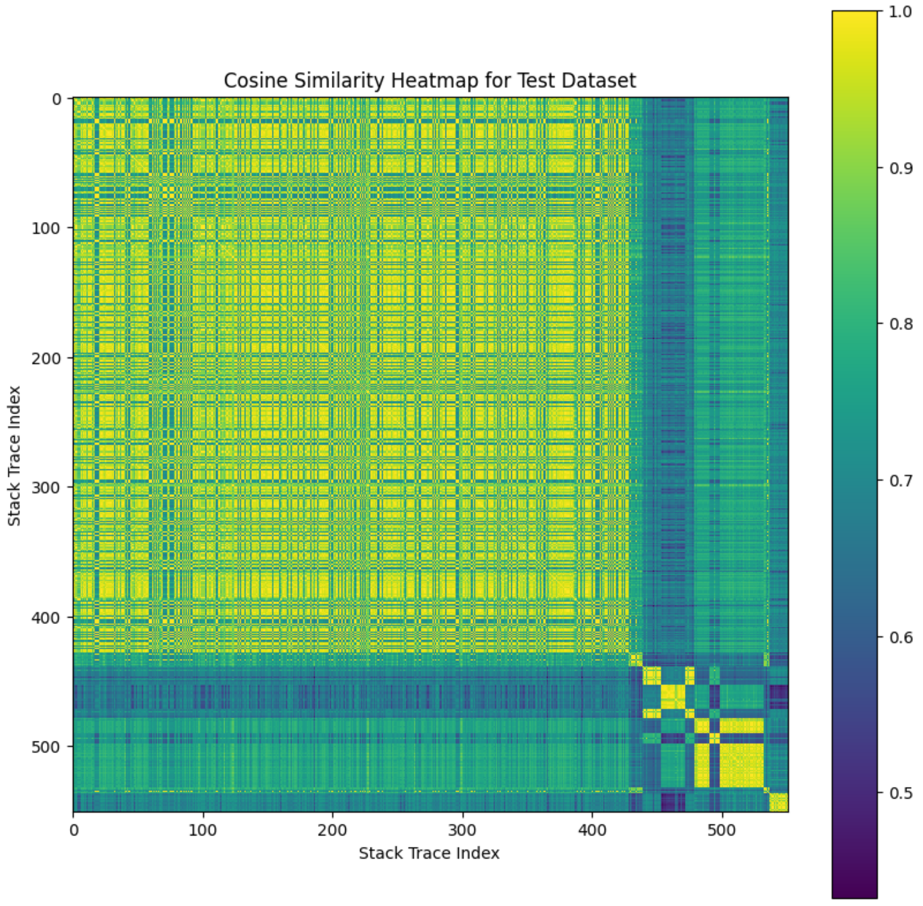


Figure 5.4: *Cosine similarity heatmap for the test dataset*

Our observations for the cosine similarity heatmap for the test dataset, compared to the Levenshtein ratio heatmap for the test dataset, shown in figure 5.2, are the same as above.

The above observations prompted us to investigate the maximum and minimum values present in each grid for both metrics, for the train and test dataset.

Optimization Results				
	Levenshtein Ratio		**Cosine Similarity**	
Dataset	Maximum observation	Minimum observation	Maximum observation	Minimum observation
Train	1.0	0.1254	1.0	0.4174
Test	1.0	0.1342	1.0	0.4324

Table 5.1: *Maximum and minimum stack trace similarity metric values for train and test dataset*

The above table clearly illustrates the prevalent difference between the lower bound of the computed cosine similarity and the Levenshtein ratio of stack traces. We hypothesize that a reason for this discrepancy is that stack trace vector embeddings always have the same number of dimensions (100), whereas `StackTrace` objects might not have the same length, when represented in string format. The `Stack Trace` objects are used for the calculation of the Levenshtein distance. For this reason, for pairs of stack traces that are very different in length, the Levenshtein ratio might be much lower than the corresponding cosine similarity.

These differences in lower bounds may impact the results of our error metrics. For this reason, we computed the errors between cosine similarity and Levenshtein ratio both with the unchanged values, as well as with values scaled using *Min-Max* scaling. [35]. The values were scaled such that the minimum and maximum bounds became 0 and 1 respectively.

5.1.4 Calculating errors between two similarity metrics

As already stated, the 3 error metrics we used to evaluate our vector embeddings were Mean Absolute Error (MAE), Root Mean Square Error (RMSE) and Mean Absolute Percentage Error (MAPE). For all 3 of these error metrics, we aim for a value as low as possible, which indicates that the cosine similarity values are as close as possible to the Levenshtein ratio values for stack trace pairs.

The following table shows the evaluated performance of the produced vector embeddings, with and without Min-Max scaling.

Vector Embedding Evaluation Results						
	No Min-Max Scaling			Min-Max Scaling		
Dataset	**MAE**	**RMSE**	**MAPE**	**MAE**	**RMSE**	**MAPE**
Train	0.272	0.299	0.356	0.184	0.213	6.222×10^{11}
Test	0.270	0.520	0.351	0.189	0.435	1.702×10^{12}

Table 5.2: *Computed error metrics for train and test stack trace datasets*

From the above table, we observed that without using min-max scaling, the discrepancy between values for cosine similarity and the values for Levenshtein ratio were a little high, but could be explained by the difference in ranges the values took. When looking at the discrepancy between values using min-max scaling, we notice that MAE and RMSE are noticeable smaller, although MAPE is extremely high. However, this can be attributed to the fact that Min-Max scaling reduces the values drastically, bringing values that are close to the lower bound of a collection near zero. Therefore, the difference between values is greatly amplified, leading to the huge increase of the MAPE when using min-max scaling.

We concluded that the quality of the stack trace vector embeddings is satisfactory, thus allowing us to move on to clustering the stack traces using their vector embeddings.

5.2 Evaluating the produced clusters

As explained in the Implementation section, we made use of a ground-truth dataset to optimize the hyper-parameters of 4 clustering algorithms: K-means, Agglomerative Clustering, DBSCAN and OPTICS. Through hyper-parameter optimization, we found out that K-means and OPTICS could not match the achieved performance of Agglomerative Clustering and DBSCAN, therefore we opted not to use them in further clustering tasks.

For the evaluation, we wanted to assess the quality of the clusters, as well as the quality of the label assignment of the clustering algorithms using their optimized hyper-parameters for the `fuzzgoat` ground-truth dataset. This is because in a real-life scenario, a pre-optimized model could be transferred from a pre-existing training set to perform clustering on new unseen stack traces.

The collection of stack traces we have is far larger than the stack traces that were collected from `fuzzgoat`. In this section, we cluster the stack traces that were collected from the rest of the fuzzed programs, and attempt to evaluate them using both internal and external cluster evaluation metrics.

5.2.1 Evaluation process details

It is understandable that evaluating a clustering using external evaluation metrics requires the existence of a ground-truth label assignment which can be directly compared to the produced label assignment. As described, our dataset did not contain these ground-truth datasets, and for hyper-parameter optimization of the adjusted-rand-index, which is an external evaluation metric, we manually created our own ground-truth dataset for `fuzzgoat` stack traces.

Our approach here is no different. For 3 target programs, `fileS3`, `uniq` and `jpegS2`, we manually created our own ground-truth label assignments, based on the criterion of stack trace similarity. While an imperfect criterion, especially for more complex programs and stack traces, this method of manual assignment was chosen for `fuzzgoat`, and returned accurate results. The choice of these 3 programs was made primarily based on the total number of collected stack traces per program, as the task of manually assigning stack traces to bugs becomes considerably harder the more stack traces and possible bugs there are.

For the rest of the fuzzed programs, the evaluation is limited to internal evaluation metrics, specifically the *silhouette score* of the resulting clusters.

We reiterate that the clustering algorithms were evaluated on the full dataset (not test or train), and that the evaluated full datasets include both the 100-dimensional embeddings as well as the 3-dimensional embeddings produced through dimensionality reduction (PCA).

For clarity we list the optimal parameters for DBSCAN and Agglomerative Clustering, as found through hyper-parameter optimization for the `fuzzgoat` ground-truth label assignment. Alternative Agglomerative Clustering refers to the Agglomerative Clustering algorithm where the number of clusters is not pre-defined and given as a parameter.

```
dbscan = DBSCAN(eps = 0.1, min_samples = 1, algorithm="auto")

agglomerative = AgglomerativeClustering(n_clusters = 4, linkage="
    single")

alt_agglomerative = AgglomerativeClustering(n_clusters=None,
    linkage="single", metric="euclidean", distance_threshold=0.1,
    compute_full_tree=True)
```

Listing 5.4: Optimal parameters for DBSCAN and Agglomerative Clustering

Before proceeding to the actual evaluation, it should be noted that we hypothesize that the optimized Agglomerative Clustering algorithm is going to achieve the worst performance, since it requires the number of clusters as a parameter. It is very clear and understandable that in our evaluation, where we use different stack trace embeddings than the ones used for optimization, the optimal number of clusters most likely is not the same as the optimal number of clusters found while optimizing the algorithms for *fuzzgoat* specifically. This is why we include the optimized Agglomerative Clustering that doesn't require the number of clusters to be predefined, and we expect that variation of the algorithm to outperform the original.

5.2.2 External cluster evaluation metrics

As explained above, for the 3 target programs for which we produced ground-truth label assignments, we were able to evaluate the quality of the resulting clusters using *adjusted-rand-score*, an external evaluation metric which compares the computed label assignment to the ground-truth label assignment.

It is important to note that adjusted rand index is interpreted as the quality of the produced assignment compared to a random assignment. An adjusted rand index of 0 indicates that the label assignment is as good as a random assignment, and a positive adjusted rand index indicates an assignment that is better than normal. The adjusted rand index implementation of sklearn is bounded between -0.5 and 1, where negative values indicate an assignment that is even worse than a random assignment.

Evaluation was performed for both the 100-dimensional stack trace embeddings, as well as the 3-dimensional vector embeddings which came as a result of dimensionality reduction. The evaluation results for both are shown in the table below.

External Cluster Evaluation Results (Adjusted Rand Index)						
	100 dimensions			3 dimensions		
Source Program	DBSCAN	Agglomerative	Alt. Agglomerative	DBSCAN	Agglomerative	Alt. Agglomerative
uniq	0.9639	0.5806	0.9639	1.0	0.4896	1.0
jpegS2	0.5224	0.1673	0.5224	0.5224	0.1527	0.5224
fileS3	0.7439	0.5009	0.7439	0.6148	0.5009	0.6148

Table 5.3: *Adjusted-rand-index of computed clusters for different optimized clustering algorithms*

From the above table, we can see that the results follow our hypothesis, as the agglomerative clustering algorithm optimized to cluster the stack traces into 4 clusters performs the worst. Another interesting observation is that DBSCAN and Agglomerative Clustering without a pre-defined number of clusters perform identically.

By comparing the performance of clustering algorithms for the 100-dimensional datasets and the 3-dimensional datasets, we can see that there is no significant performance gain, at least for the 3 datasets for which we created ground-truth assignments. On the contrary, the performance of the clustering algorithms seems to suffer for `fileS3`. This indicates that the reduction of dimensions to 3 might cause a significant loss of information.

5.2.3 Internal cluster evaluation metrics

Not all clusters could be evaluated with external cluster evaluation metric due to the lack of a ground-truth label assignment. For these cases, silhouette score, which is an internal cluster evaluation metric was used to evaluate the produced clusters. It should be noted that the silhouette score was also calculated for the datasets for which a ground-truth label assignment existed.

Similar to the external evaluation of the clusters, both 100 dimensional and 3 dimensional datasets were used for clustering evaluation. The results of the evaluation are shown in the table below.

Internal Cluster Evaluation Results (Silhouette Coefficient)						
	100 dimensions			3 dimensions		
Source Program	DBSCAN	Agglomerative	Alt. Agglomerative	DBSCAN	Agglomerative	Alt. Agglomerative
who	0.9234	0.5524	0.9234	0.7418	0.5617	0.7418
md5sum	0.7474	0.6949	0.7474	0.7794	0.7813	0.7794
uniq	0.8985	0.7149	0.8985	0.9358	0.7856	0.9358
file	0.5033	0.6070	0.5033	0.7257	0.6962	0.7257
audiofileS	0.7337	0.6461	0.7337	0.8386	0.5821	0.8386
jpegS2	0.7738	0.5838	0.7738	0.8234	0.2853	0.8234
fileS3	0.7592	0.7573	0.7592	0.8826	0.8318	0.8826
fileS4	0.8528	0.5806	0.8528	0.8245	0.6415	0.8245

Table 5.4: *Silhouette score of computed clusters for different optimized clustering algorithms*

By looking at the table, we can understand that in the general case, the silhouette score of the clustering increased when reducing the dataset from 100 dimensions to 3. Although the silhouette coefficient of a clustering does not indicate the quality of the computed clusters based on a ground-truth assignment, it does indicate how similar the items placed in each cluster are to each other, and how dissimilar they are to items placed in different clusters.

With the above in mind, we can deduce that the quality of our clusters is satisfactory, since the silhouette coefficient of the produced label assignments is quite high in most cases. This indicates that the clustering algorithms can adequately distinguish similar from dissimilar stack traces based on their produced vector embeddings, which in turn shows that representing stack traces with word embeddings is a technique that can potentially be used to solve the problem of bug prioritization.

5.3 Final Comments

In this section, we evaluated both our produced word embeddings as well as the clusters produced by the various clustering algorithms we optimized in the Implementation section.

The evaluation of the produced vector embeddings for stack traces showed that representing stack traces using word embeddings is a promising technique and is comparable to representing stack traces using strings or text. The cosine similarity of stack trace embeddings was compared to the Levenshtein ratio of the stack traces, and we found that the two techniques are comparable in terms of representing textual features of the stack traces.

We then used the produced stack trace embeddings to perform clustering using various clustering algorithms that were optimized on the `fuzzgoat` ground-truth label assignment. Since a ground-truth label assignment was not available for any of the datasets, we produced such ground-truth assignments for 3 of the datasets we collected. For these 3 datasets, we were able to evaluate the produced clusters using external evaluation metrics, which directly compare the produced labels with the ground-truth labels, however, we also performed internal cluster evaluation for all datasets, which evaluates the produced clusters based on the intra-cluster and inter-cluster similarity of data.

The external cluster evaluation results were satisfactory, by consistently achieving scores substantially better than a random assignment, even achieving a perfect label assignment for a dataset of stack traces. This indicates that there is potential in utilizing vector embeddings and unsupervised learning to solve the problem of bug prioritization, provided that there are ground-truth label assignments that can be used to train and optimize clustering algorithms.

The internal cluster evaluation results, although much harder to associate with the actual performance of a label assignment produced by a clustering algorithm, show that the produced clusters contain data points which are similar to each other, and different to data points that belong to other clusters. This shows that vector embeddings capture significant information about the stack traces which allows the clustering algorithms to produce clusters which are clearly defined, further exhibiting the promising abilities of vector embeddings in the space of bug prioritization.

What we overall notice regarding the evaluation of the produced clusters, regardless of whether the evaluation was performed using external or internal criteria, is that the performance of the clustering algorithms was inconsistent across different programs. This can be attributed to the fact that the algorithms were trained on a ground-truth dataset for a single program, and differences in the structure or complexity of the unseen stack traces could play an underlying role in these inconsistencies.

Chapter 6

Future Work

In this section, we state some deficiencies of our followed approach, possible solutions as well as future work that could be done to further research the potential of vector embeddings in bug triaging and bug report prioritization.

The first improvement that we discern as a future possibility is the creation of a more extensive crash report dataset, by collecting a larger number of programs to be fuzzed. A big obstacle that we faced in our work was discovering buggy programs that could very easily be fuzzed to produce a large number of crash reports. Limiting factors for our choices of programs to fuzz were the size and functionality of the program, as well as the number of bugs present in the source code, as programs with no known bugs could lead to very long fuzzing times for the discovery of fewer bugs.

However, by fuzzing programs with *unknown* or *"natural"*[1] bugs, we can collect stack traces that are more representative of a real-life bug collection scenario, either as a result of software testing, or crash reporting. By collecting crash reports in this way, there is the possibility that the performance of the algorithms used for clustering might improve, creating more accurate label assignments that would greatly help alleviate the problem of bug prioritization and categorization. On the other hand, there is the prospect that real-life crash reports originating from software more nuanced and complicated than the software tested in our work might lead to the exact opposite regarding the clustering performance. More work in this field is crucial in order to determine the best way to perform clustering and assist in the categorization of bugs.

The rode0day competitions provided numerous such programs and their source code, which could be fuzzed to produce an even larger dataset of crash reports. By fuzzing for a longer period of time as well as fuzzing the binary files found in the dataset using binary fuzzing techniques, we can amass more bug reports that could be used to produce stack trace embeddings.

[1]"Natural": Bugs that are present due to programming errors, not a vulnerability injection framework such as LAVA.

Another difficulty that we faced in our work is the lack of a ground-truth label assignment that maps crash reports to the bug that caused them. As a result, we resorted to creating our own ground-truth label assignments, where the labelling of stack traces to bugs was made by grouping stack traces that are similar and assigning the same label to them. Understandably, this is an imperfect approach, as it neglects one of the most important problems that bug clustering aims to solve, which is the assignment of different crash reports in the same cluster, if the bug that causes them is common. Another deficiency of this approach is the fact that the produced ground-truth datasets were small in size, which can be a contributing factor behind the inconsistent performance of the clustering algorithms on stack traces originating across different programs. A real-life ground-truth dataset that is sizeable enough to be used for cluster evaluation and optimization would allow for the training of algorithms on the ground-truth dataset, and their transfer to an unseen dataset with a similar and more consistent performance.

Lastly, the area of vector embedding production and evaluation is one where refinements could be made. Further research into stack trace comparison techniques that examine the stack traces in depth, rather than using measures based on string similarity, such as the Levenshtein ratio could be used to better evaluate the quality of the produced vector embeddings, thus leading to refinements to their production process, which could be accompanied by the production of stack trace vector embeddings that capture in-depth details of each stack trace, which are currently not captured. Attempts at creating more comprehensive stack trace comparison methods have been made, such as TraceSim by Vasiliev et al. [36], which combines a variety of techniques, namely TF-IDF, Levenshtein distance and machine learning, to improve performance compared to other baseline comparison methods. The application of these comparison methods as a means to evaluate and optimize the produced stack trace vector embeddings might give a clearer insight into the potential of using word embedding models to create stack trace representations.

Chapter 7

Related Work

The problem of bug prioritization and categorization is one that is exacerbated by advancements in the field of software testing. As software testing techniques become more nuanced and their ability to discover faults and errors in programs improves, the number of crashes and bugs that software developers and programmers are asked to triage is overwhelming. Thus, the prioritization and categorization of bugs can greatly ease the load of programmers, allowing them to identify severe bugs faster and easier, greatly reducing the time spent sifting through them and increasing the time spent actually dealing with them.

There have been various attempts at tackling the problem of bug prioritization and categorization. Dongsun Kim et al. [1] analyze a large dataset of crash reports, and create numerous features based on information like the methods called, the complexity of the stack traces etc.. These features are then encoded into feature vectors, and supervised machine learning algorithms are leveraged in order to perform categorization of crash reports into frequent or not frequent.

Jiang et al. developed a capability-guided fuzzer, Evocatio [2], that explores crash reports and their circumstances, such as the input that caused the crash, to uncover possibilities in which the bug could be exploited by a malicious attacker. This helps in discovering the severity of bugs, which is one of the crucial criteria taken into account when prioritizing bugs.

Spanos et al. [3] leveraged text-mining techniques and supervised machine learning to predict the severity of a vulnerability using information extracted from its description in the National Vulnerability Database. Specifically, their work is focused on automatically predicting the CVSS and WVISS severity score for each vulnerability.

Chapter 8

Conclusion

In this thesis, we investigated the potential of the use of vector embeddings for crash report clustering, as a solution to the problem of bug categorization and prioritization. Our approach makes use of stack trace embeddings, which can be directly extracted from crash reports.

In the Architecture section we propose an architecture for collecting crash reports, producing and evaluating stack trace embeddings, and training clustering algorithms of our choice for categorizing the stack traces into groups, where each group represents stack traces that were caused by a common bug.

In the Implementation section, we follow the proposed architecture. We collect stack traces using fuzzing, an automated software testing technique, which is used to find vulnerabilities and bugs in software. By fuzzing various programs with AFL++, an open source state-of-the-art fuzzer, we generate a dataset of crash reports, from which the stack traces are extracted. Stack traces were also found from pre-existing datasets, and were pre-processed to follow a common format.

An object representation of stack traces was also implemented, allowing for the parsing of stack traces into python objects, facilitating the comparison of stack traces.

We produce vector embeddings by training a fastText word embedding model on our collected stack traces. We then evaluate the produced vector embeddings by comparing their ability to compute stack trace similarity. The cosine similarity of stack trace vector embedding pairs was directly compared to the Levenshtein ratio, a string comparison metric, of the corresponding stack trace objects.

Using the produced stack trace embeddings, we train and optimize various clustering algorithms on a manually created ground-truth label assignment, and then transfer the optimized algorithms to perform clustering for each of the collected datasets of stack traces.

In the Evaluation section, the quality of the produced clusters was evaluated using both internal and external cluster evaluation criteria, those being silhouette coefficient and adjusted-rand-index respectively. By evaluating both the vector embeddings and the quality of the produced clusters, we discover that there is potential in representing stack traces as vector embeddings, in order to facilitate the process of crash report categorization.

In the Future Work section, we identify weaknesses and deficiencies in our approach, and outline possible refinements to our proposed work that could lead to better results. We also briefly present possible research directions in this field that could lead to further discoveries and solutions to this problem.

Bibliography

[1] Dongsun Kim, Xinming Wang, Sunghun Kim, Andreas Zeller, Shing-Chi Cheung, and Sooyong Park. Which Crashes Should I Fix First?: Predicting Top Crashes at an Early Stage to Prioritize Debugging Efforts. *IEEE Trans. Software Eng.*, 37:430–447, 05 2011.

[2] Zhiyuan Jiang, Shuitao Gan, Adrian Herrera, Flavio Toffalini, Lucio Romerio, Chaojing Tang, Manuel Egele, Chao Zhang, and Mathias Payer. Evocatio: Conjuring Bug Capabilities from a Single PoC. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 1599–1613, New York, NY, USA, 2022. Association for Computing Machinery.

[3] Georgios Spanos, Lefteris Angelis, and Dimitrios Toloudis. Assessment of Vulnerability Severity Using Text Mining. In *Proceedings of the 21st Pan-Hellenic Conference on Informatics*, PCI '17, New York, NY, USA, 2017. Association for Computing Machinery.

[4] Peter Mell, Karen Kent, and Sasha Romanosky. Common Vulnerability Scoring System. 2006-12-29 2006.

[5] Georgios Spanos, Angeliki Sioziou, and Lefteris Angelis. WIVSS: a new methodology for scoring information systems vulnerabilities. In *Panhellenic Conference on Informatics*, 2013.

[6] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++ : Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020.

[7] Michał Zalewski. American Fuzzy Lop. `https://lcamtuf.coredump.cx/afl/technical_details.txt`, 2016.

[8] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. LAVA: Large-Scale Automated Vulnerability Addition. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 110–121, 2016.

[9] John Neystadt. Automated Penetration Testing with White-Box Fuzzing. https://learn.microsoft.com/en-us/previous-versions/software-testing/cc162782(v=msdn.10)?redirectedfrom=MSDN, February 2008.

[10] Michał Zalewski. The AFL approach. https://afl-1.readthedocs.io/en/latest/motivation.html#the-afl-approach, 2016.

[11] Michał Zalewski. AFL Mutation Process. https://afl-1.readthedocs.io/en/latest/user_guide.html#stage-progress, 2016.

[12] Scikit-Learn Clustering. https://scikit-learn.org/stable/modules/clustering.html#clustering.

[13] SAS/STAT 9.2 Users Guide. SAS Institute. Clustering Linkage Methods. https://support.sas.com/documentation/cdl/en/statug/63033/HTML/default/viewer.htm#statug_cluster_sect012.htm.

[14] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, page 226–231. AAAI Press, 1996.

[15] Mihael Ankerst, Markus Breunig, Peer Kröger, and Joerg Sander. OPTICS: Ordering Points to Identify the Clustering Structure. volume 28, pages 49–60, 06 1999.

[16] Darius Pfitzner, Richard Leibbrandt, and David M. W. Powers. Characterization and evaluation of similarity measures for pairs of clusterings. *Knowledge and Information Systems*, 19:361–394, 2009.

[17] Scikit-Learn Evaluating Clusters. https://scikit-learn.org/stable/modules/clustering.html#clustering-performance-evaluation.

[18] William M. Rand. Objective Criteria for the Evaluation of Clustering Methods. *Journal of the American Statistical Association*, 66(336):846–850, 1971.

[19] Ka Yee Yeung and Walter Ruzzo. Details of the Adjusted Rand index and Clustering algorithms Supplement to the paper "An empirical study on Principal Component Analysis for clustering gene expression data" (to appear in Bioinformatics). *Science*, 17, 01 2001.

[20] Peter J. Rousseeuw. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20:53–65, 1987.

[21] Laurens van der Maaten, Eric Postma, and Jaap van den Herik. Dimensionality Reduction: A Comparative Review. https://members.loria.fr/moberger/Enseignement/AVR/Exposes/TR_Dimensiereductie.pdf, 10 2009.

[22] Scikit-Learn Decomposition. https://scikit-learn.org/stable/modules/decomposition.html#decompositions.

[23] Ian T. Jolliffe and Jorge Cadima. Principal component analysis: a review and recent developments. *Philosophical Transactions of the Royal Society of London Series A*, 374(2065):20150202, April 2016.

[24] V. I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, February 1966.

[25] Levenshtein Python Module - Levenshtein Ratio. https://maxbachmann.github.io/Levenshtein/levenshtein.html#ratio.

[26] DeepAI - N-gram Definition. https://deepai.org/machine-learning-glossary-and-terms/n-gram.

[27] fastText Python library. https://fasttext.cc/.

[28] GitHub - fastText Repository. https://github.com/facebookresearch/fastText.

[29] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. Bag of Tricks for Efficient Text Classification, 2016.

[30] Joseph Carlos and fuzzstati0n. GitHub - Fuzzgoat. https://github.com/fuzzstati0n/fuzzgoat.

[31] Andrew Fasano, Tim Leek, Rahul Sridhar, and Brendan Dolan-Gavitt. Rode0day - A continuous bug finding competition. https://rode0day.mit.edu/.

[32] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan. LAVA: Large-Scale Automated Vulnerability Addition. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 110–121, Los Alamitos, CA, USA, may 2016. IEEE Computer Society.

[33] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan. Download Link - LAVA-1 and LAVA-M corpora. http://panda.moyix.net/~moyix/lava_corpus.tar.xz.

[34] GDB - The GNU Project Debugger. https://www.sourceware.org/gdb/.

[35] Scikit-learn: Min-Max scaling. `https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html`.

[36] Roman Vasiliev, Dmitrij Koznov, George Chernishev, Aleksandr Khvorov, Dmitry Luciv, and Nikita Povarov. TraceSim: A Method for Calculating Stack Trace Similarity, 2020.

Appendix A

Results

A.1 Visualization of stack traces before clustering

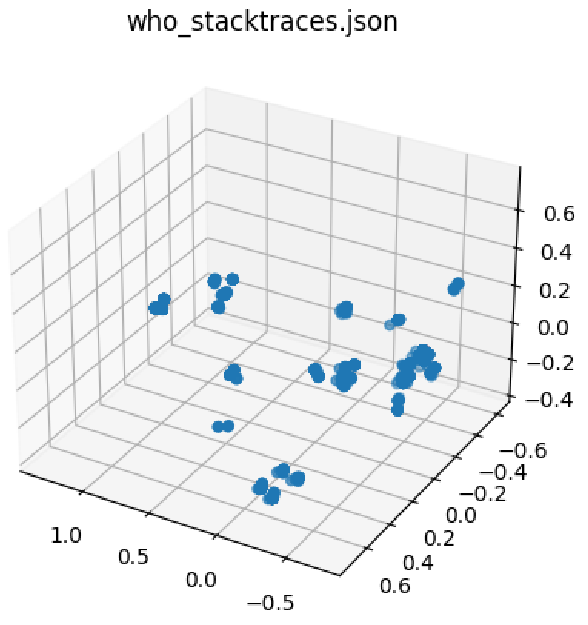


Figure A.1: Visualization of stack trace embeddings from `who` reduced to 3 dimensions

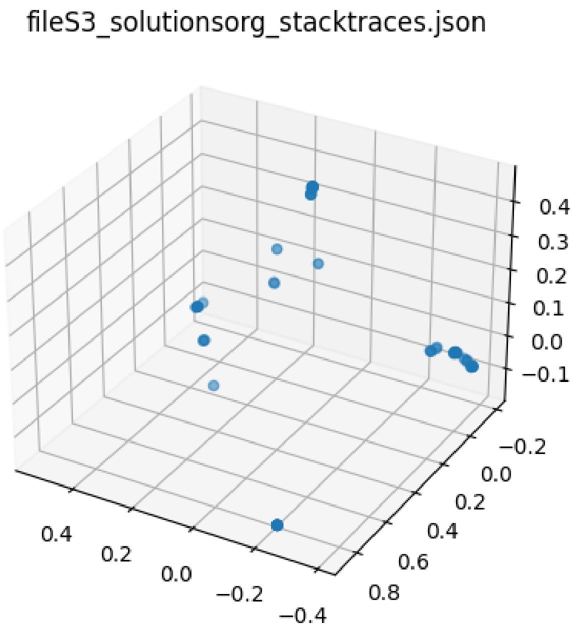


Figure A.2: Visualization of stack trace embeddings from `fileS3` reduced to 3 dimensions

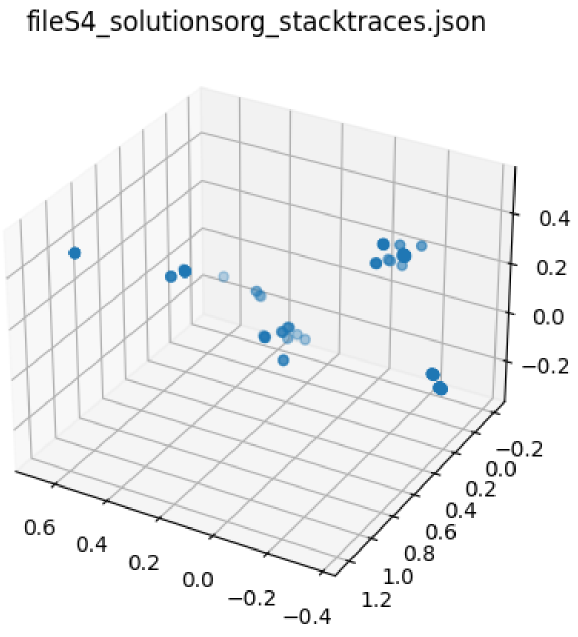


Figure A.3: Visualization of stack trace embeddings from `fileS4` reduced to 3 dimensions

file_stacktraces.json

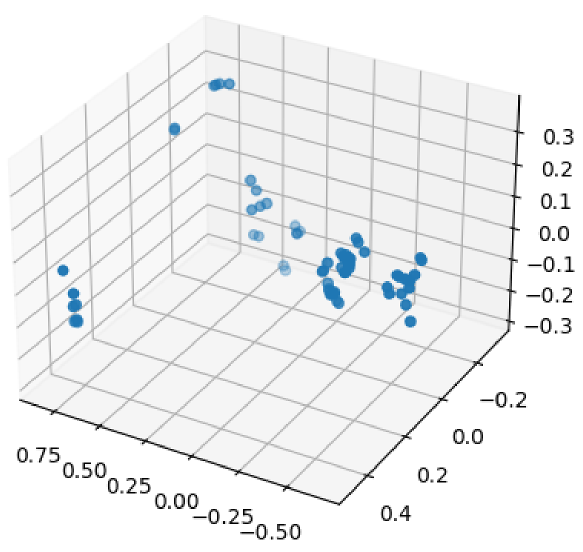


Figure A.4: Visualization of stack trace embeddings from `file` reduced to 3 dimensions

md5sum_stacktraces.json

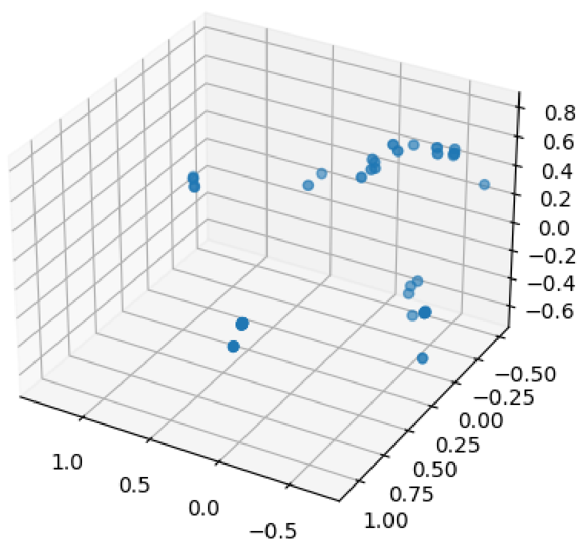


Figure A.5: Visualization of stack trace embeddings from `md5sum` reduced to 3 dimensions

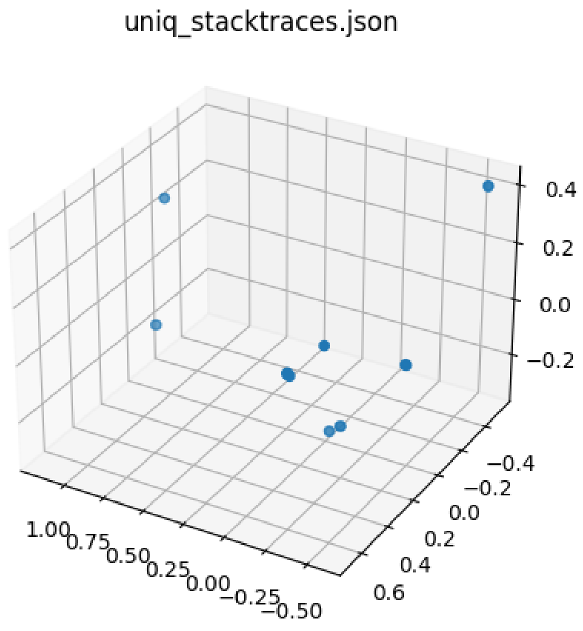


Figure A.6: Visualization of stack trace embeddings from `uniq` reduced to 3 dimensions

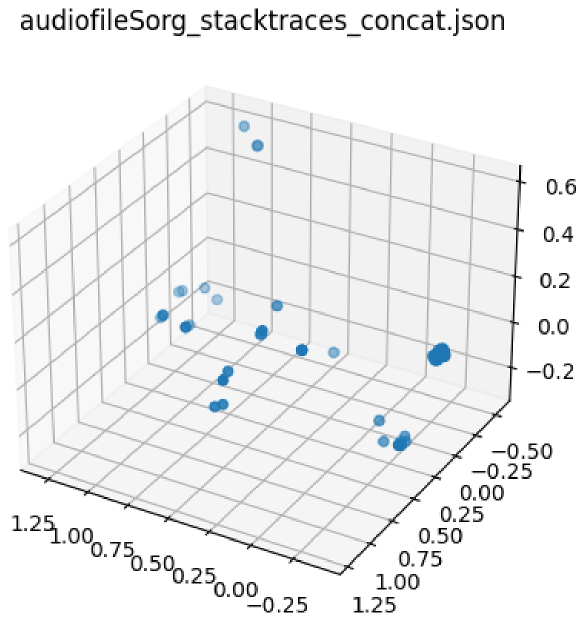


Figure A.7: Visualization of stack trace embeddings from `audiofileS` reduced to 3 dimensions

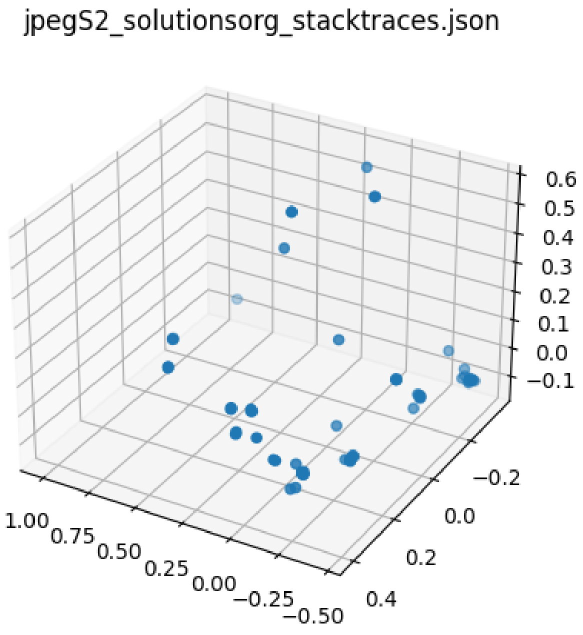


Figure A.8: Visualization of stack trace embeddings from `jpegS2` reduced to 3 dimensions

A.2 Ground-truth label assignments

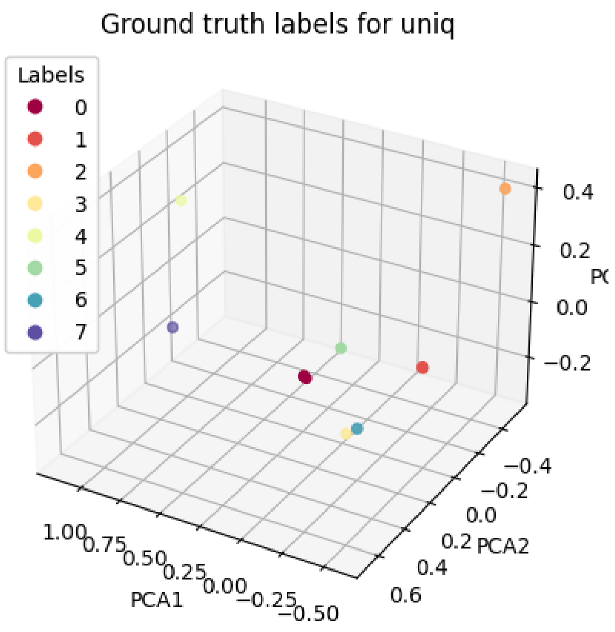


Figure A.9: Visualization of ground-truth label assignment of `uniq` reduced to 3 dimensions

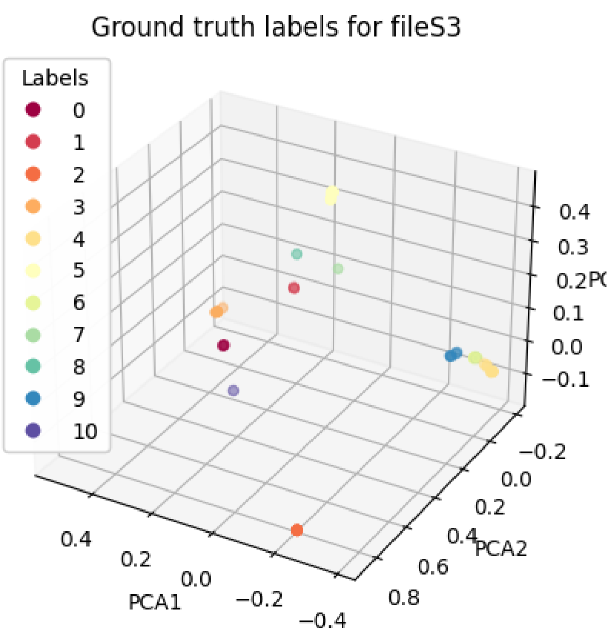


Figure A.10: Visualization of ground-truth label assignment of `fileS3` reduced to 3 dimensions

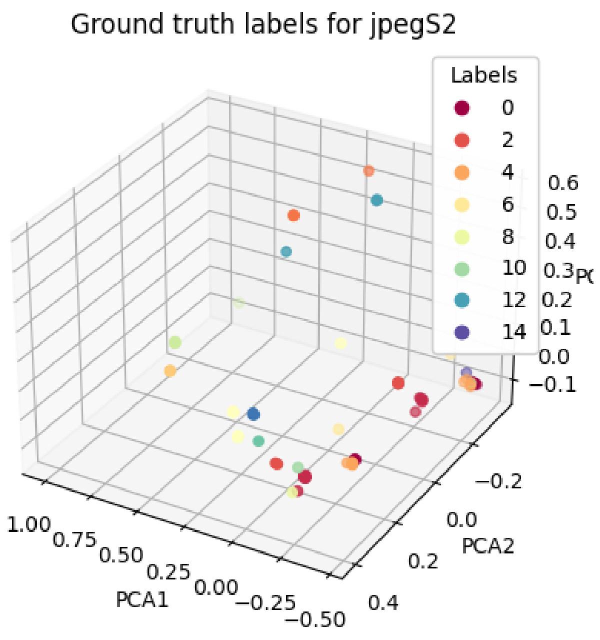


Figure A.11: Visualization of ground-truth label assignment of `jpegS2` reduced to 3 dimensions

A.3 Results of optimal DBSCAN clustering

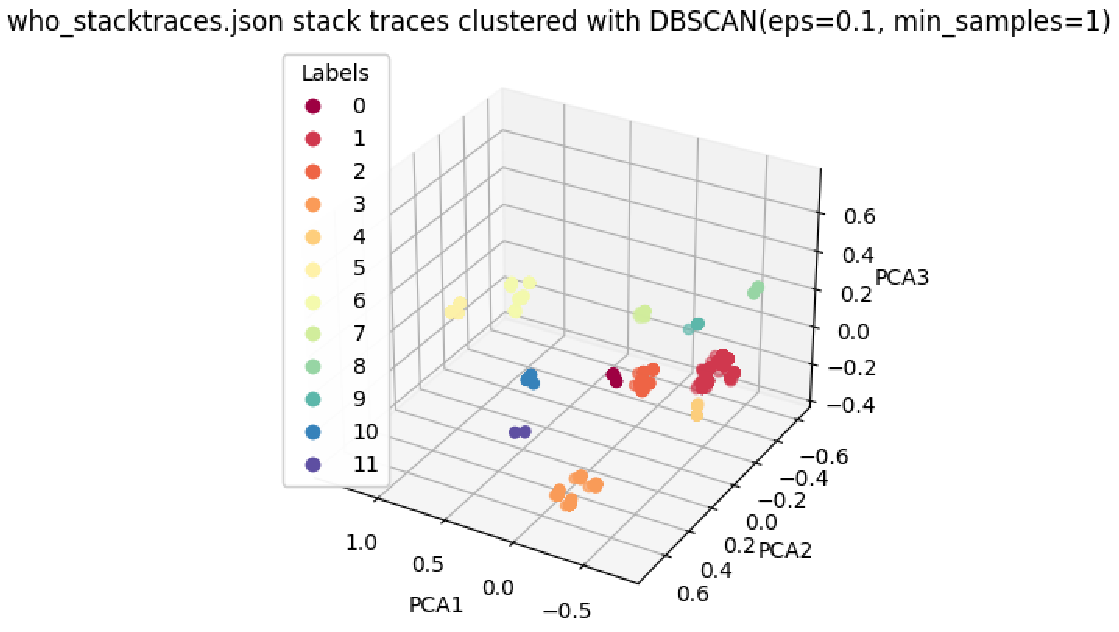


Figure A.12: Clustering of stack trace embeddings using DBSCAN from `who` reduced to 3 dimensions

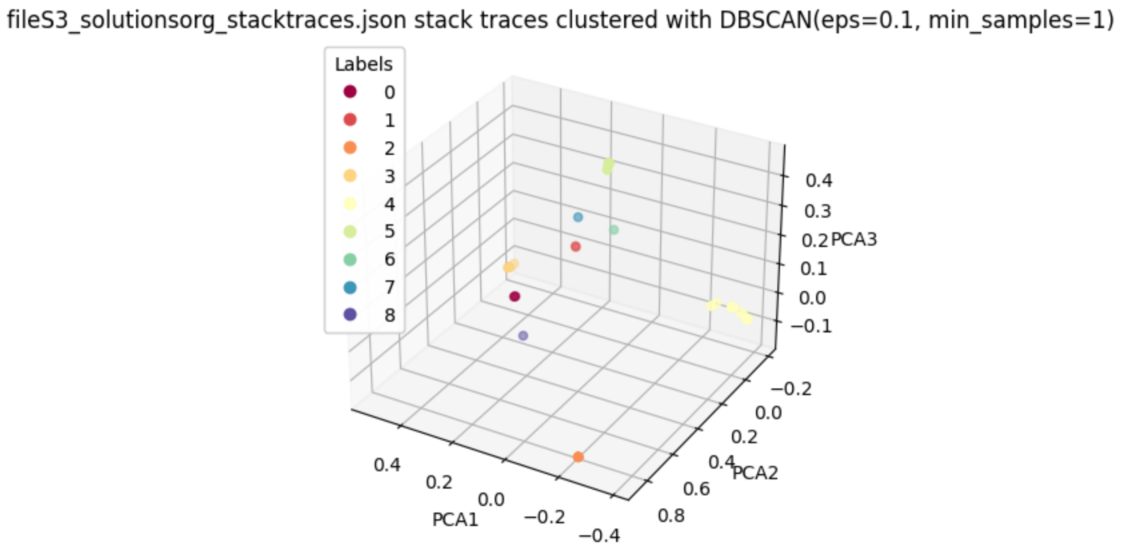


Figure A.13: Clustering of stack trace embeddings using DBSCAN from `fileS3` reduced to 3 dimensions

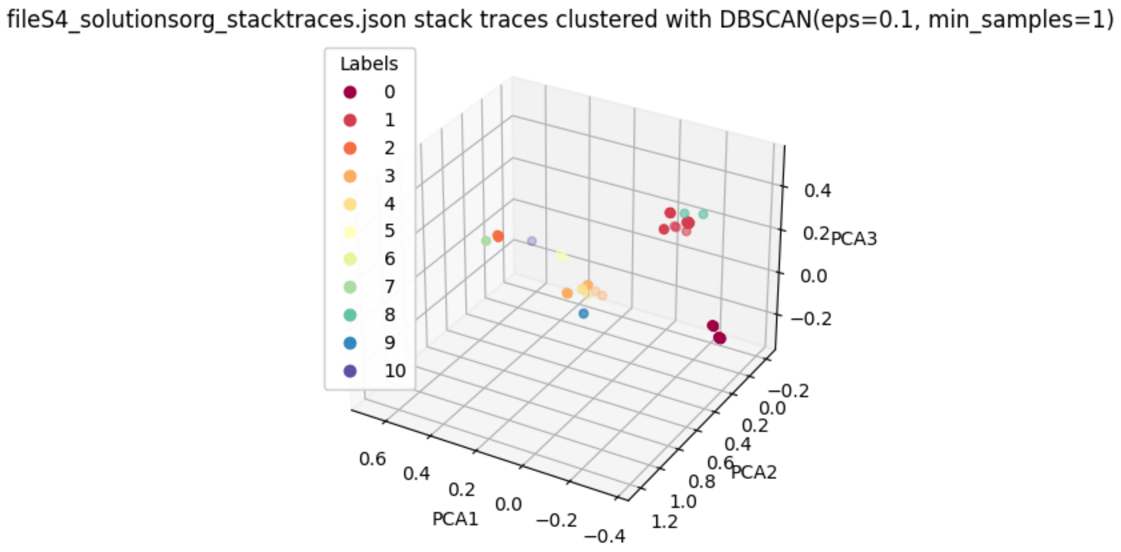


Figure A.14: Clustering of stack trace embeddings using DBSCAN from `fileS4` reduced to 3 dimensions

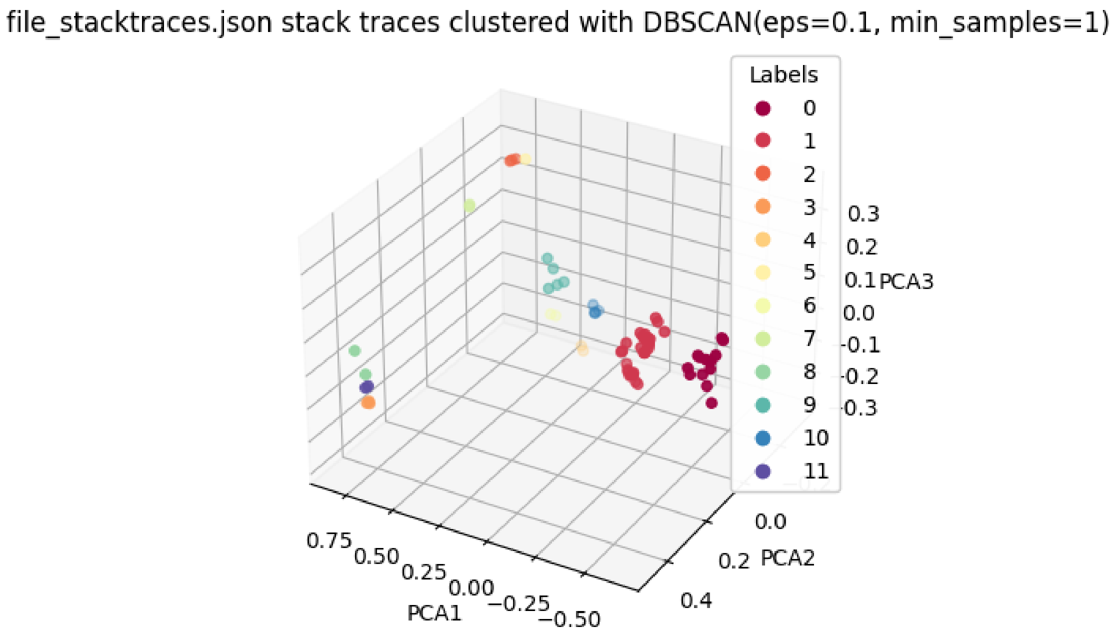


Figure A.15: Clustering of stack trace embeddings using DBSCAN from `file` reduced to 3 dimensions

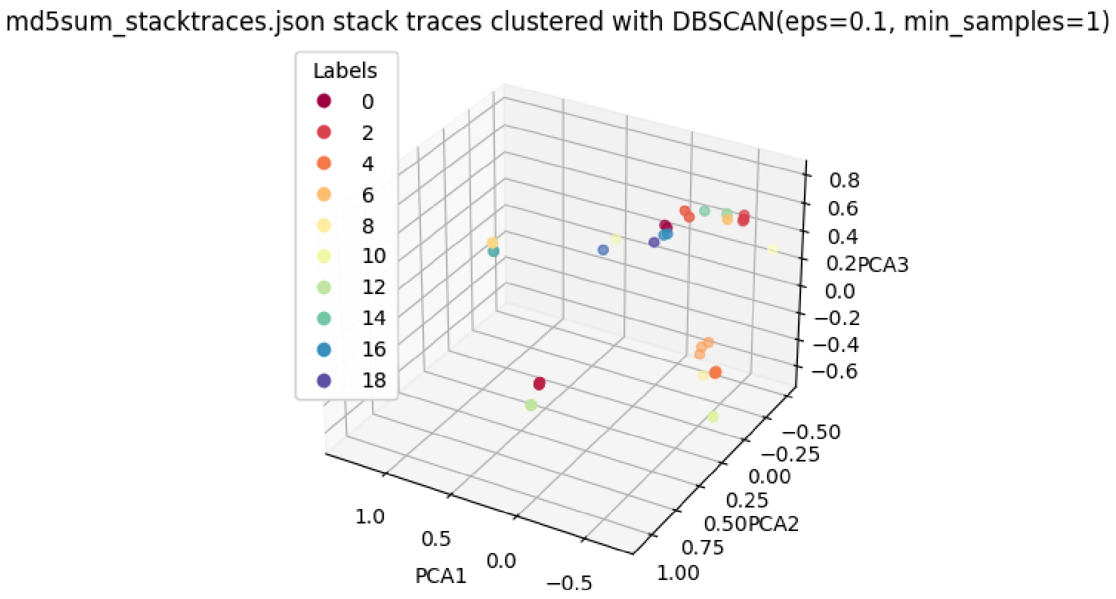


Figure A.16: Clustering of stack trace embeddings using DBSCAN from `md5sum` reduced to 3 dimensions

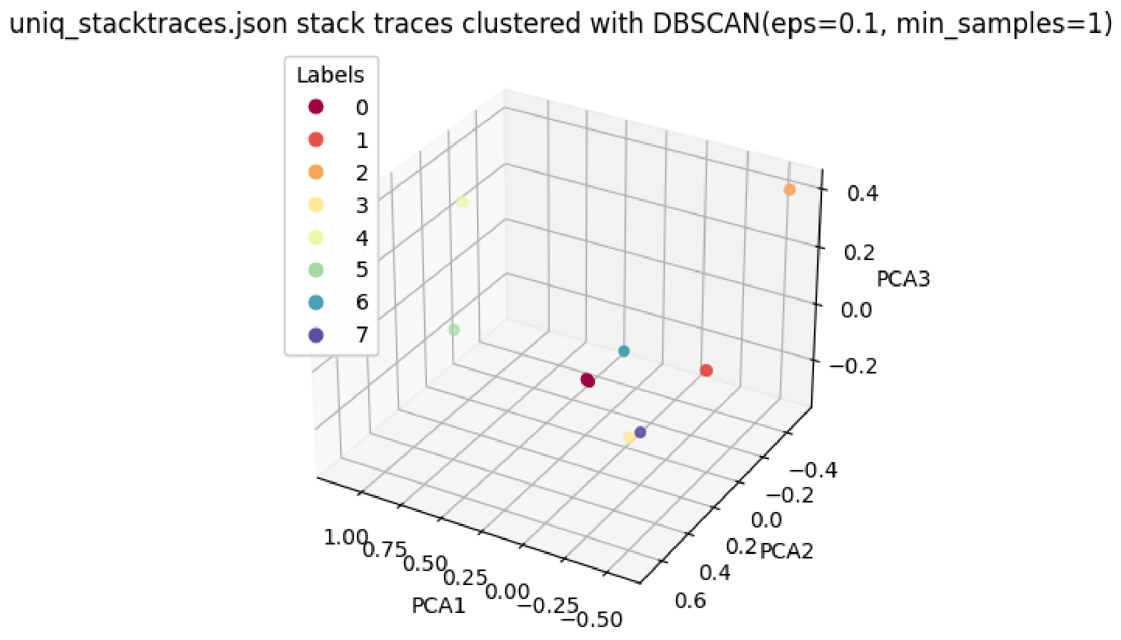


Figure A.17: Clustering of stack trace embeddings using DBSCAN from `uniq` reduced to 3 dimensions

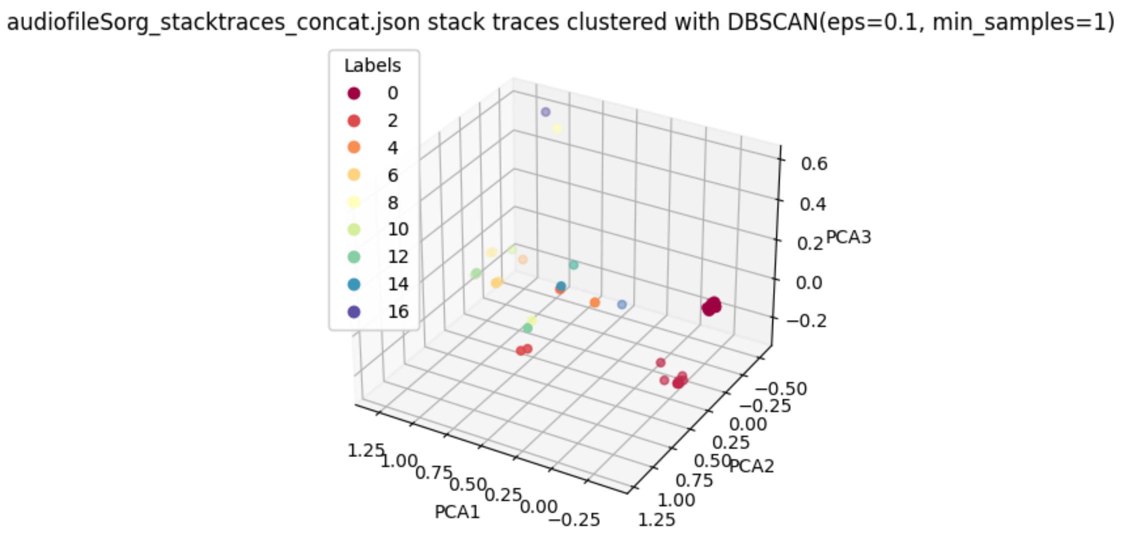


Figure A.18: Clustering of stack trace embeddings using DBSCAN from `audiofileS` reduced to 3 dimensions

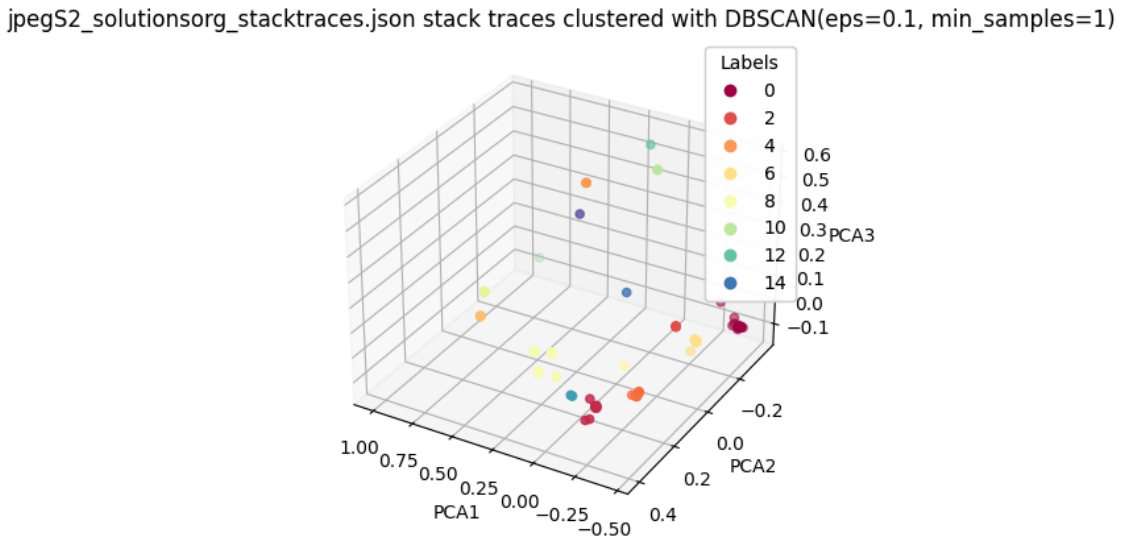


Figure A.19: Clustering of stack trace embeddings using DBSCAN from `jpegS2` reduced to 3 dimensions

A.4 Results of optimal Agglomerative clustering

A.4.1 Agglomerative Clustering with pre-defined number of clusters

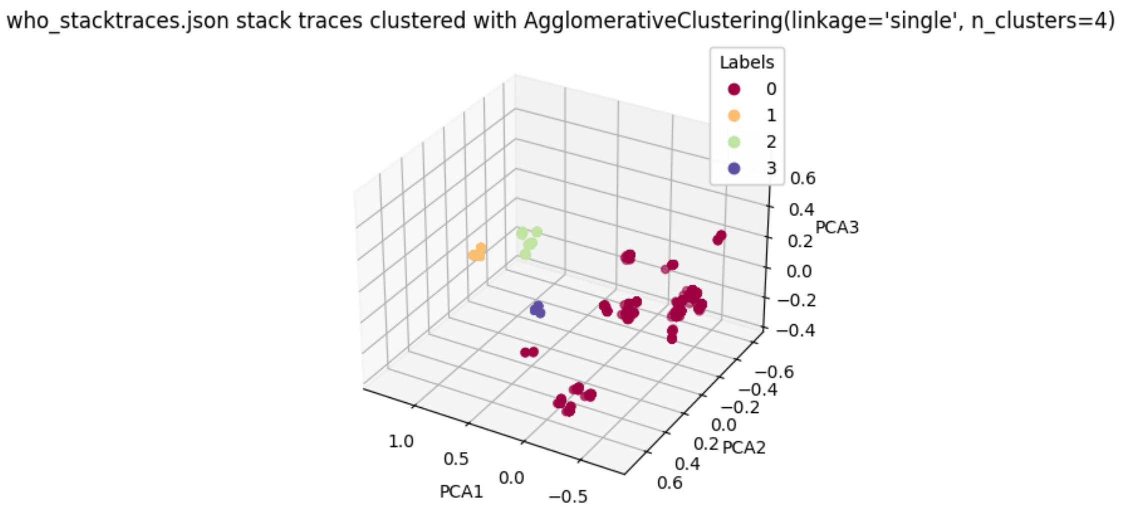


Figure A.20: Clustering of stack trace embeddings using Agglomerative Clustering from `who` reduced to 3 dimensions

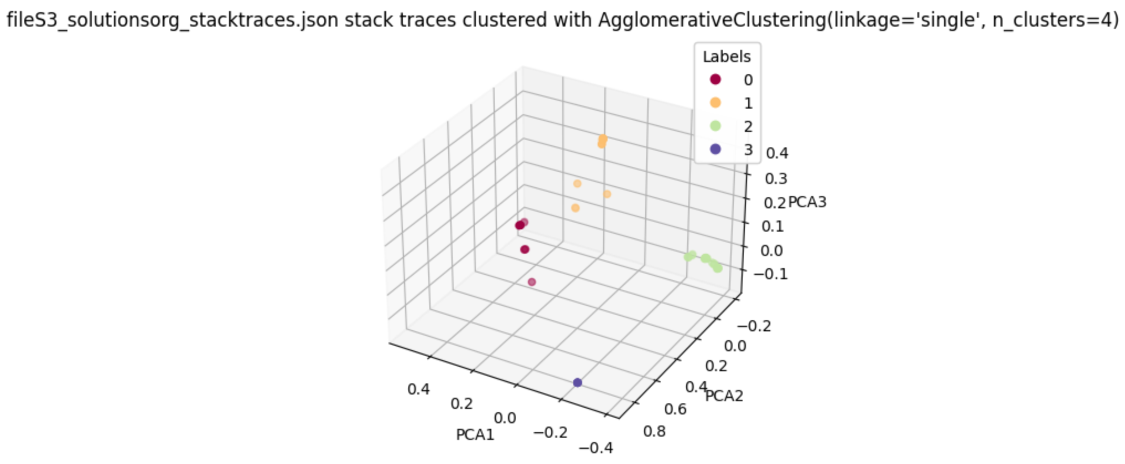


Figure A.21: Clustering of stack trace embeddings using Agglomerative Clustering from `fileS3` reduced to 3 dimensions

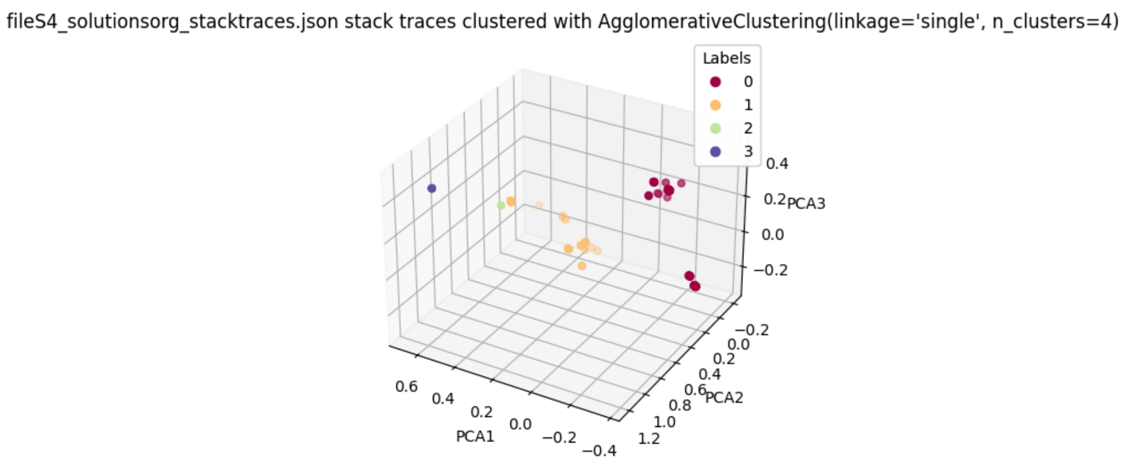


Figure A.22: Clustering of stack trace embeddings using Agglomerative Clustering from `fileS4` reduced to 3 dimensions

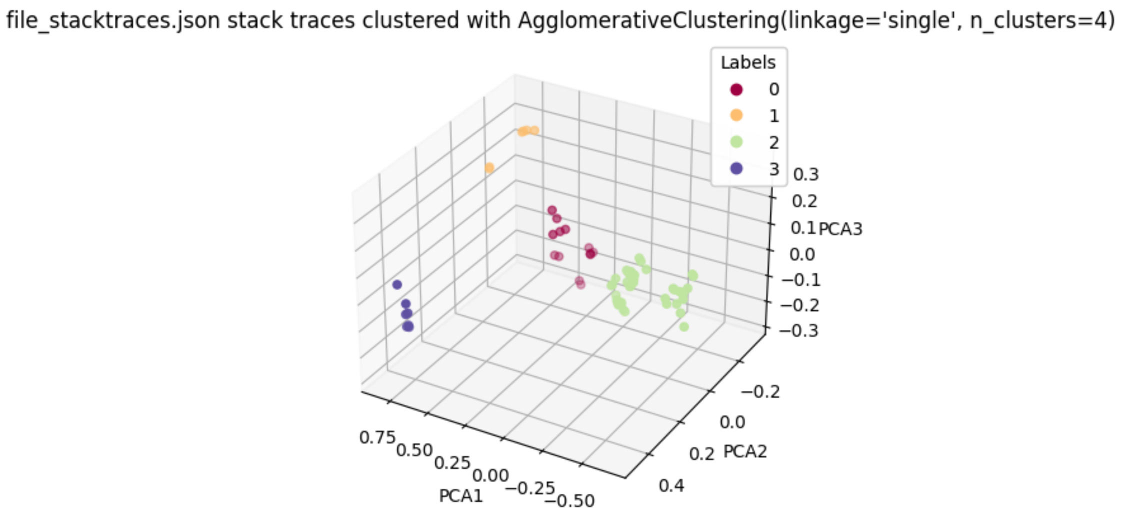


Figure A.23: Clustering of stack trace embeddings using Agglomerative Clustering from `file` reduced to 3 dimensions

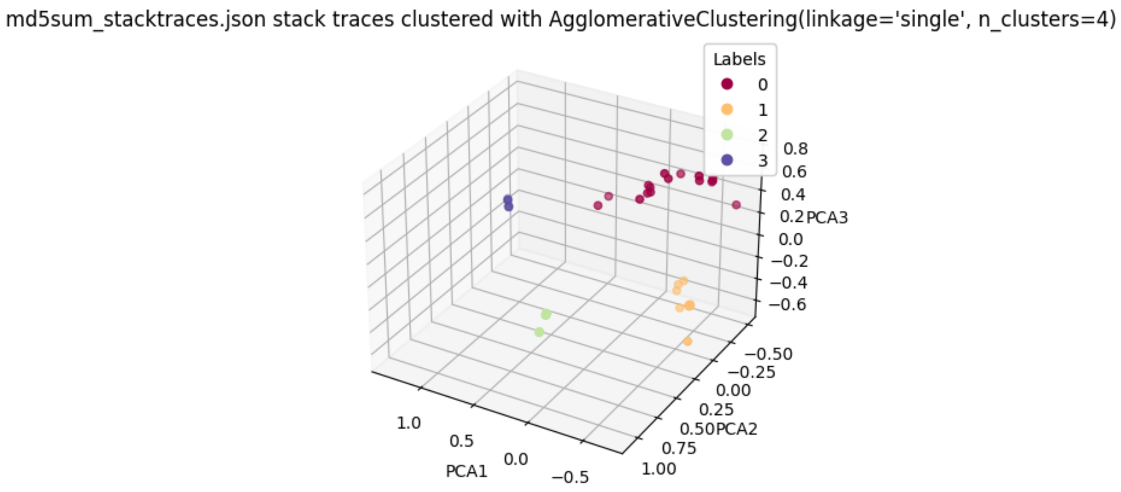


Figure A.24: Clustering of stack trace embeddings using Agglomerative Clustering from `md5sum` reduced to 3 dimensions

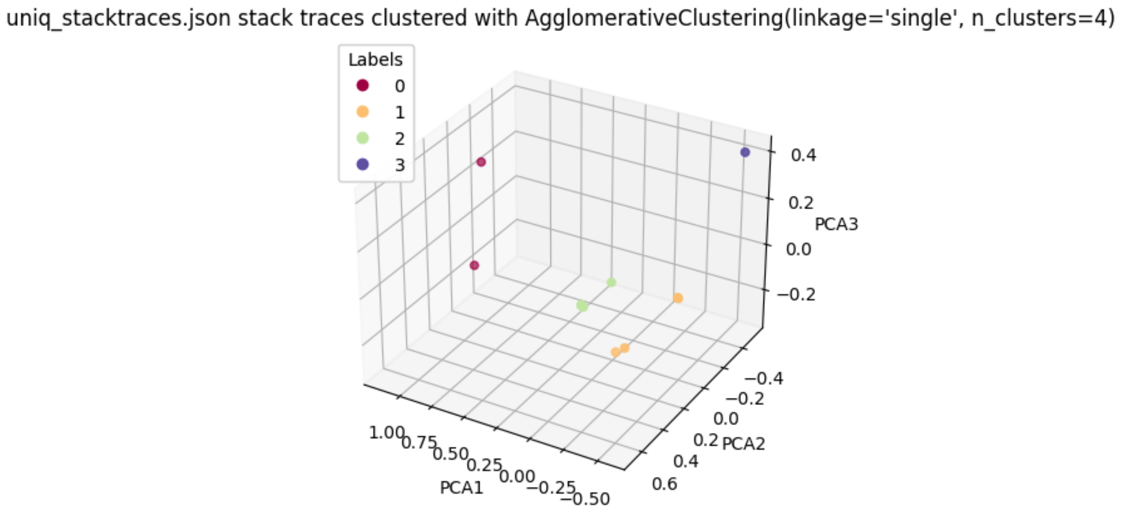


Figure A.25: Clustering of stack trace embeddings using Agglomerative Clustering from `uniq` reduced to 3 dimensions

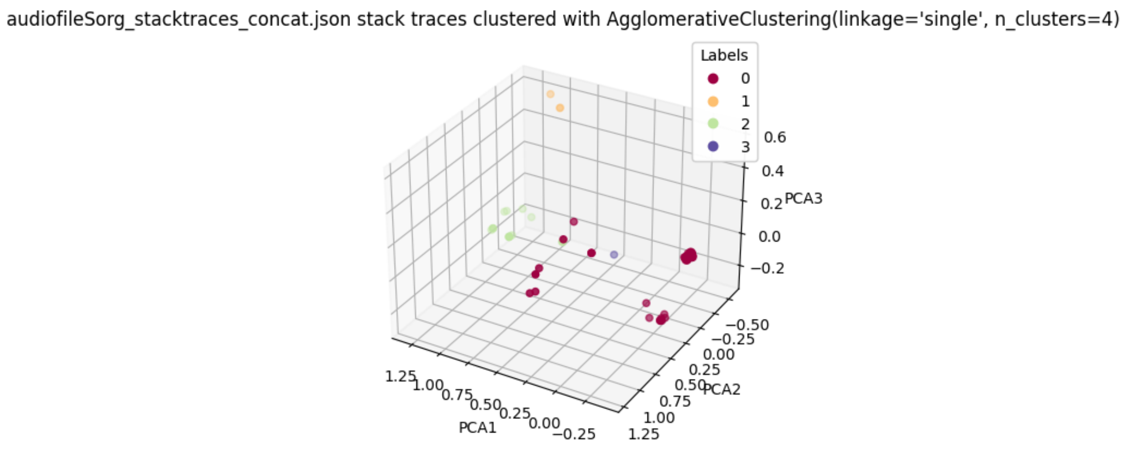


Figure A.26: Clustering of stack trace embeddings using Agglomerative Clustering from `audiofileS` reduced to 3 dimensions

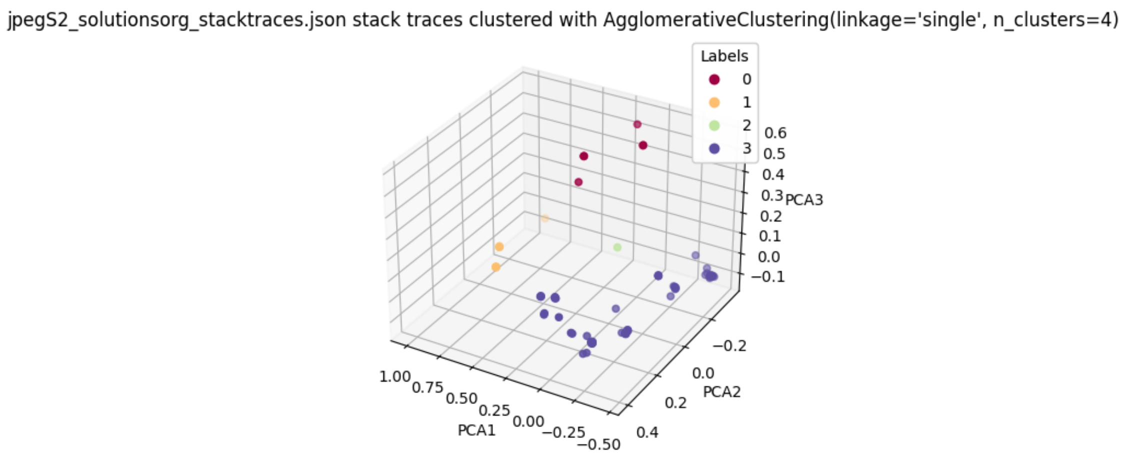


Figure A.27: Clustering of stack trace embeddings using Agglomerative Clustering from `jpegS2` reduced to 3 dimensions

A.4.2 Agglomerative Clustering without pre-defined number of clusters

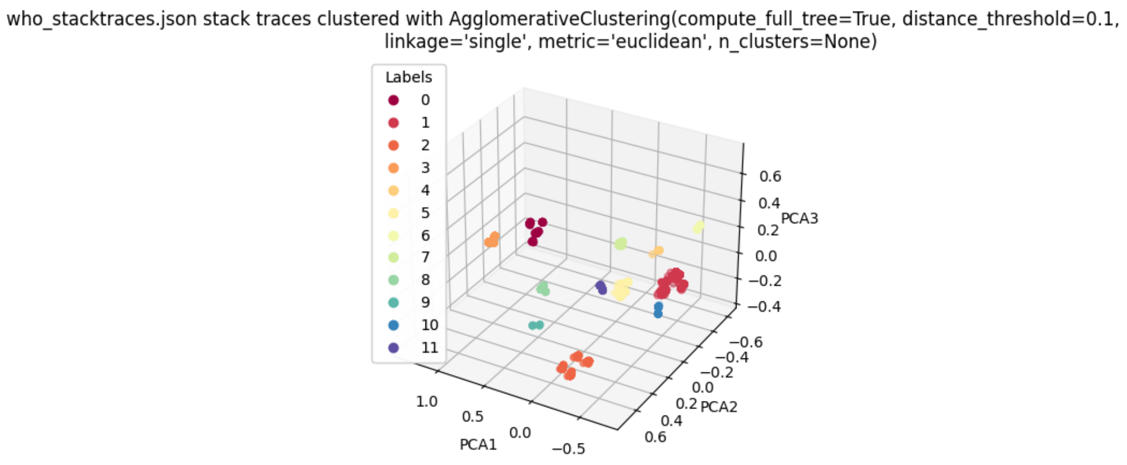


Figure A.28: Clustering of stack trace embeddings using Agglomerative Clustering from `who` reduced to 3 dimensions

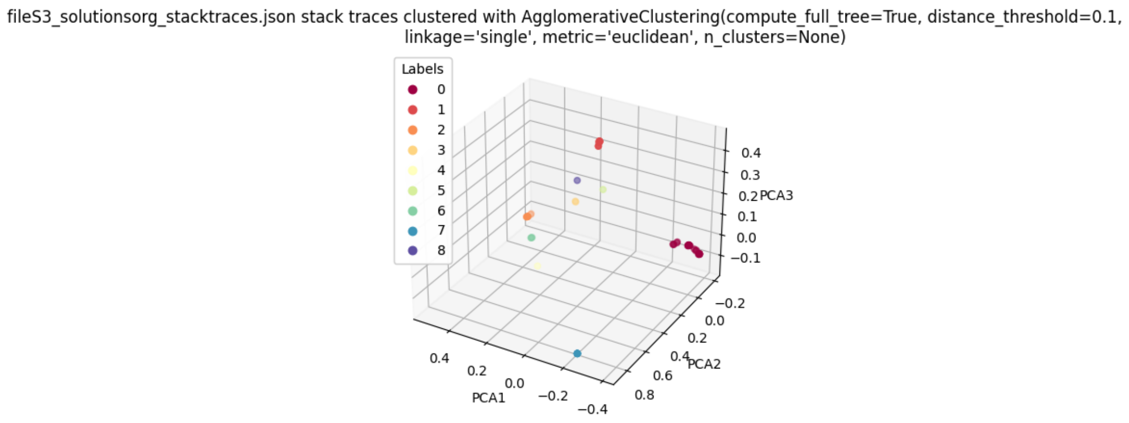


Figure A.29: Clustering of stack trace embeddings using Agglomerative Clustering from `fileS3` reduced to 3 dimensions

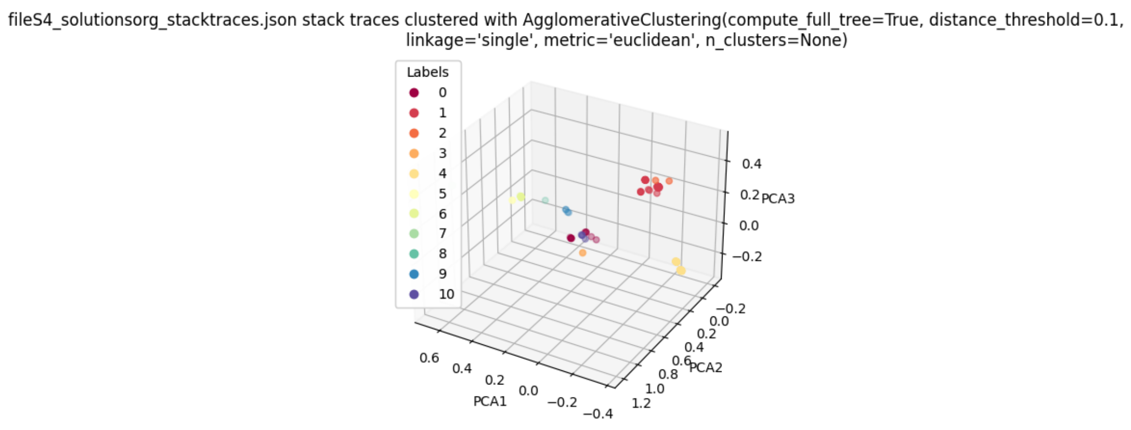


Figure A.30: Clustering of stack trace embeddings using Agglomerative Clustering from `fileS4` reduced to 3 dimensions

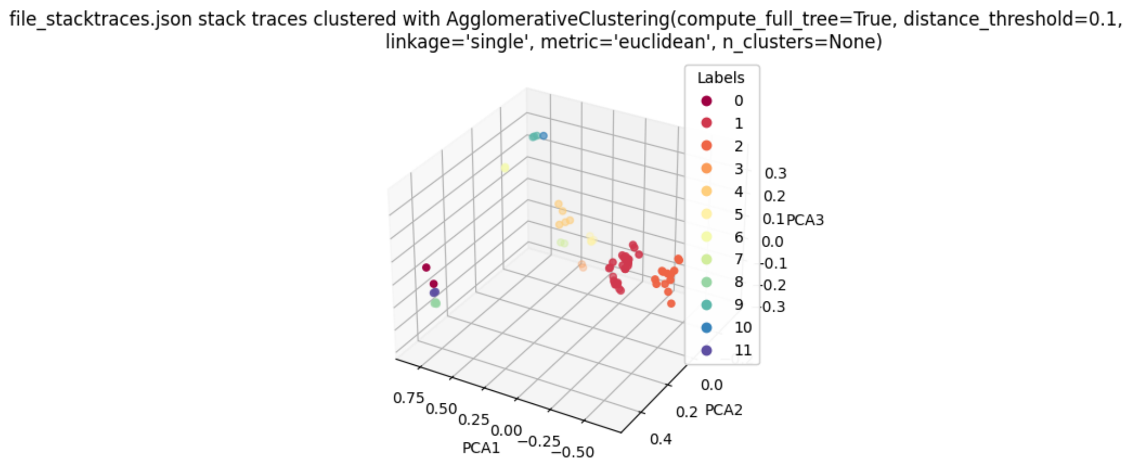


Figure A.31: Clustering of stack trace embeddings using Agglomerative Clustering from `file` reduced to 3 dimensions

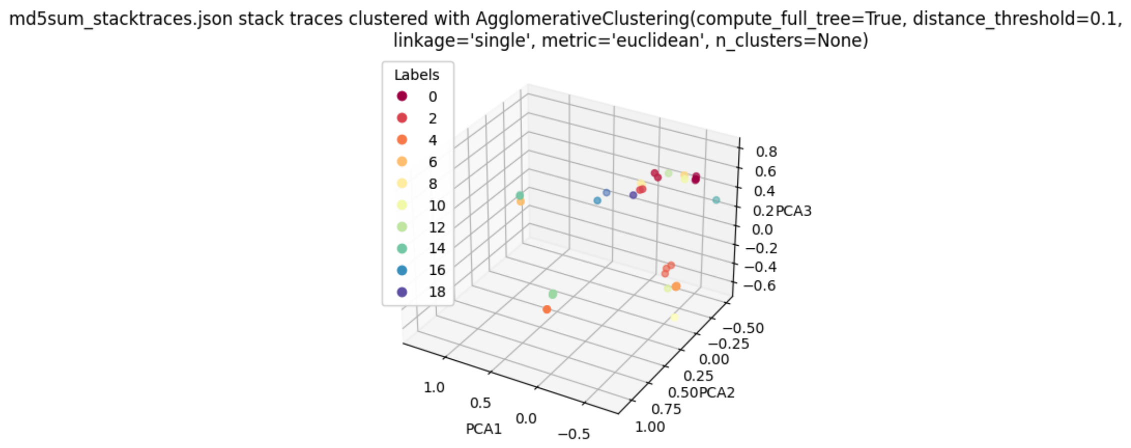


Figure A.32: Clustering of stack trace embeddings using Agglomerative Clustering from `md5sum` reduced to 3 dimensions

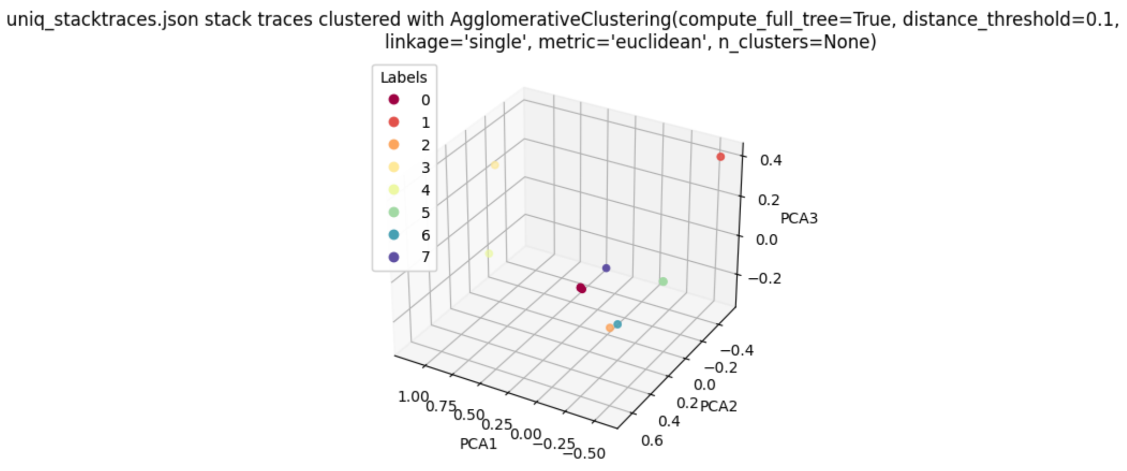


Figure A.33: Clustering of stack trace embeddings using Agglomerative Clustering from `uniq` reduced to 3 dimensions

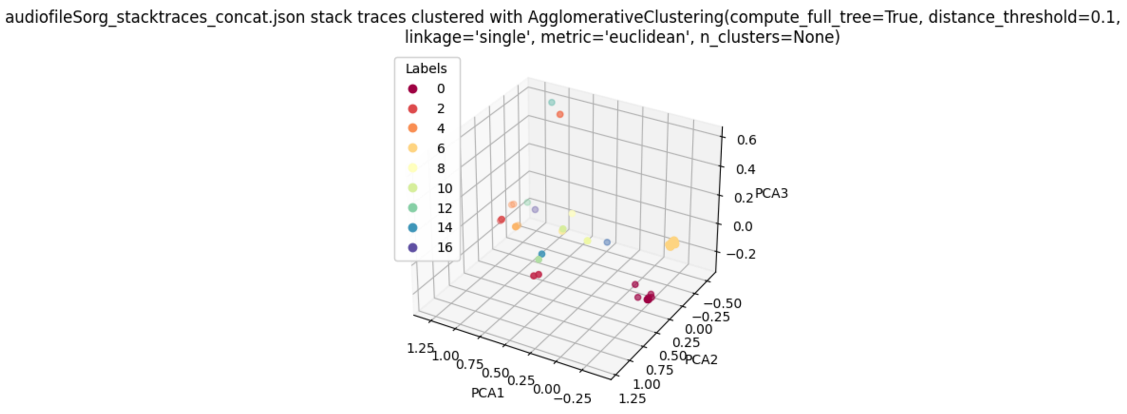


Figure A.34: Clustering of stack trace embeddings using Agglomerative Clustering from `audiofileS` reduced to 3 dimensions

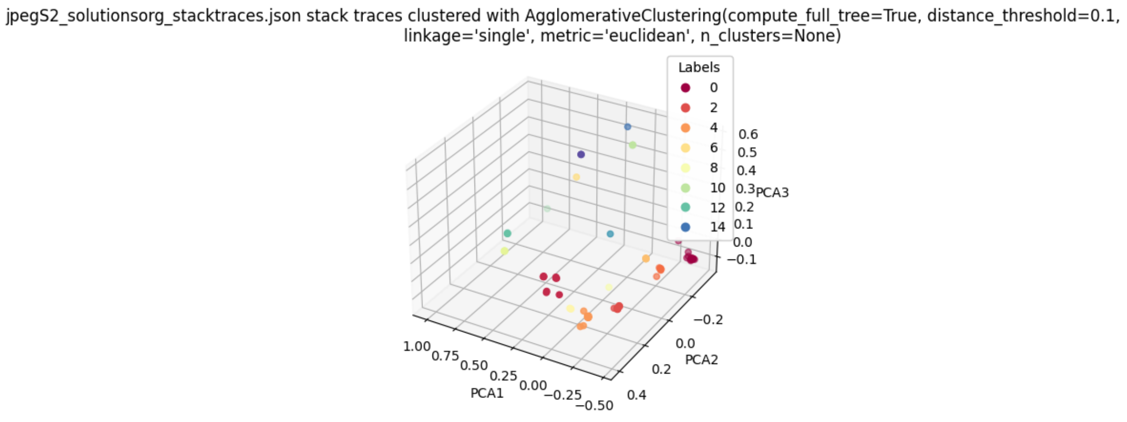


Figure A.35: Clustering of stack trace embeddings using Agglomerative Clustering from `jpegS2` reduced to 3 dimensions