CS560 Course Project

Project Title: Continuous Static Analysis based on dependency graph embeddings in Elasticsearch

Group Members: Arca Özkan 25524 - Fatih Öztank 25060

Supervisor: Cemal Yılmaz

Date: 19.01.2022

Sabancı Üniversitesi

Table of Contents

1. Abstract

Static analysis and continuous testing are two debugging methods that have been utilized for a long time now. Continuous static analysis is a novel approach which includes performing static analysis over a period of time on different releases of software under test(SUT). The SUT in this research was Elasticsearch, coded in Java. Graphs based on dependencies between classes in Elasticsearch were created, and embedded using FeatherGraph, for efficiency reasons. After the graph embeddings for each release of SUT were extracted, they were grouped using the clustering algorithm provided by sklearn. These continuous static analysis algorithms were applied to several classes of Elasticsearch, with different hyperparameter settings in order to find releases which introduced a new bug for the class we're investigating. Main idea behind our approach is once a release is put into a cluster, usually it should be in the same cluster as the previous release. If there's a violation of this case, we'll label that release as a candidate for introducing a new bug. This approach will likely reveal if there is a vulnerability issue emerging from newly added or broken dependencies in this release.

2. Introduction

Static analysis has been one of the main approaches in automated debugging throughout the years. Unlike dynamic analysis, the code doesn't need to be executed in static analysis.

For debugging purposes, static analysis is mostly used by checking common programming mistakes or typos. However, when it is performed continuously for a specific software over a period of time, it can also be used for detecting security issues.

Continuous static analysis is a novel approach to debugging, and it can be performed in various ways. The source code for the specified software will be extracted in each time step, and it can be analyzed based on multiple approaches, such as comparison of the dependency graphs of classes, modules or functions. After recognizing patterns in these multiple graphs, security issues can be detected by analyzing when certain patterns change, certain dependencies get lost. This idea was used in the project in order to find vulnerabilities in widely used, important frameworks in which vulnerabilities can cause a lot of damage.

Elasticsearch framework was choosen as the software under test (SUT) for performing continuous static analysis in this research, mostly because it is an open source project with many contributors. It is a RESTful search and analytics engine which can be used to store, search and manage data for logs, metrics, endpoint security etc. Also it uses Java language, which is one of the easiest languages for static analyses thanks to the tools come with the jvm.[1]

To perform continuous static analysis on Elasticsearch based on changes in class dependencies, dependency graphs were created and embedded for efficiency reasons. This process will be explained under the Approach and Evaluation sections in more detail.

3. Background Information and Related Work

Static analysis has been one of the most staple, quick and easy ways to find errors in the source code of a software. It is widely used and accepted throughout all kinds of different softwares. Static analysis can utilize different methods, such as using Control Flow Graphs or common programming mistakes for debugging the code without execution.

Continuous testing has also been explored throughout the years, by continuously running regression tests in the background while the developer edits the code. This continuous regression testing can make sure that newly added changes to a source code do not break anything that worked before the change.[2]

However, Continuous Static Analysis has been an area of exploration that has surfaced recently. Pronto is a framework that can be integrated with Github or Bitbucket to perform continuous static analysis. It is compatible with Ruby language. The analysis focuses on the relevant changes between the current head and the provided commit. Unlike our research, it doesn't focus on dependencies, it simply focuses on differences in the source code.[3]

Another tool that has included continuous static analysis as an option recently is JTest. It is an automated Java software testing framework making use of multiple different testing technologies like regression testing, static analysis and data flow analysis. Continuous static analysis in JTest is performed when a source file is opened or a new or modified code is saved. Rules and features can be added or modified by the developer to personalize the analysis even further.[4]

These are some of the few example frameworks exploring continuous static analysis. There are not many other papers/frameworks published about the concept yet, since it is a novel idea.

4. Approach

For developing a system where we can do continuous static analysis to source code, we developed a framework from Python which operates on an existing git folder. This framework receives a starting date as input and iterates through every single day until current day. For each day, it checks for whether or not there’s a release which was released on that day. If a release is found for that date, it will build the software and execute dependency analysis from the executables. Results of the dependency analyses are saved into a folder in txt format, so the dependency analysis step is executed only once.

When implementing our framework, the first thing we did was to create an interface. Depending on the project we’re testing, we implemented our continuous static analysis class from that interface. When choosing a project for our class implementation, we looked for two things. First, it has to be an open source project so we can access the source code easily and second is it has to be written in an easy to analyze language such as Java or Javascript.

For this paper, we picked the Elasticsearch project for implementing our framework. Elasticsearch is a distributed, RESTful search and analytics engine written in Java[1]. Reason for we picked Elasticsearch was because it was an open source project with lots of contributors and it was fully written in Java. Starting from a release which was released on 2018-02-06(v6.2.0) our framework managed to execute dependency analysis for 65 different releases until a release which released on 2022-01-13(v7.16.3)

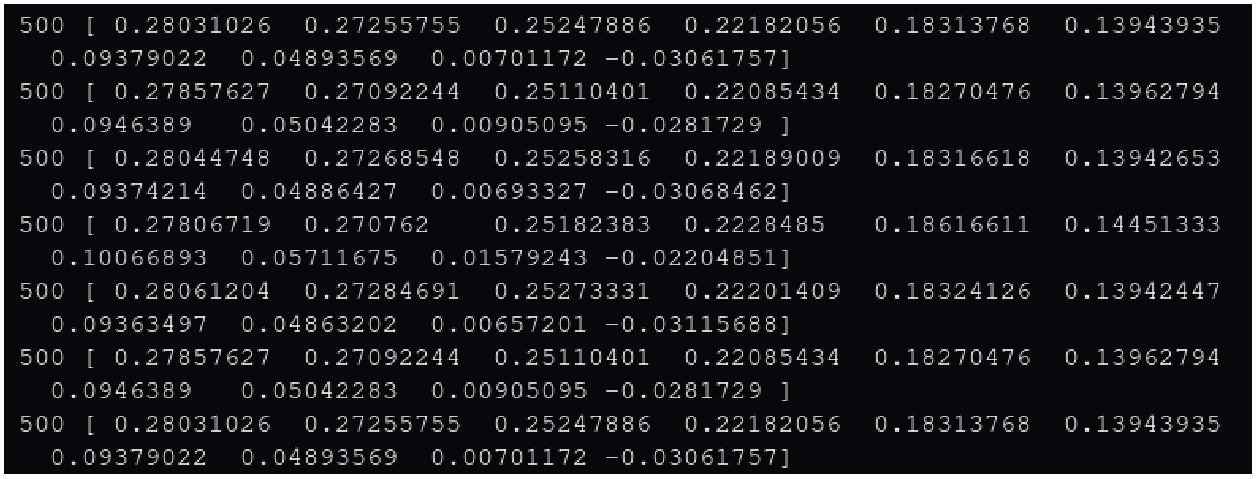
After the dependency analyses are complete, the next step of our approach comes into the play. For this step, we implemented a graph class based on the Networkx library in Python. By using dependency analysis reports, our class generates dependency graphs for each different release. Upon generating graphs for all of the releases, our framework creates graph embeddings from them by using FeatherGraph class, which is an implementation of FEATHER-G paper[5] on karateclub library for python. Reason for generating those graph embeddings is it makes it easier for graphs to apply machine learning techniques into them for anomaly detection.

By using OPTICS clustering algorithm from sklearn library, we managed to cluster graph embeddings generated from dependency graphs. OPTICS(Ordering Points To Identify the Clustering Structure) is a clustering technique which is suitable for datasets with a large number of features[6]. Since our embedding vectors had 500 elements in them, we had to use clustering algorithms like DBSCAN or OPTICS which are successful at clustering datasets with a large number of features.

Good thing about our framework is even though for large projects, executing dependency analyses and generating graphs seems like taking too much time, it has to be done only once. Once the dependency graphs are created, extracting class specific graphs from them doesn’t require to build the project again, instead it will just generate a subgraph from the existing dependency analysis reports according to the requested class with given depth value.

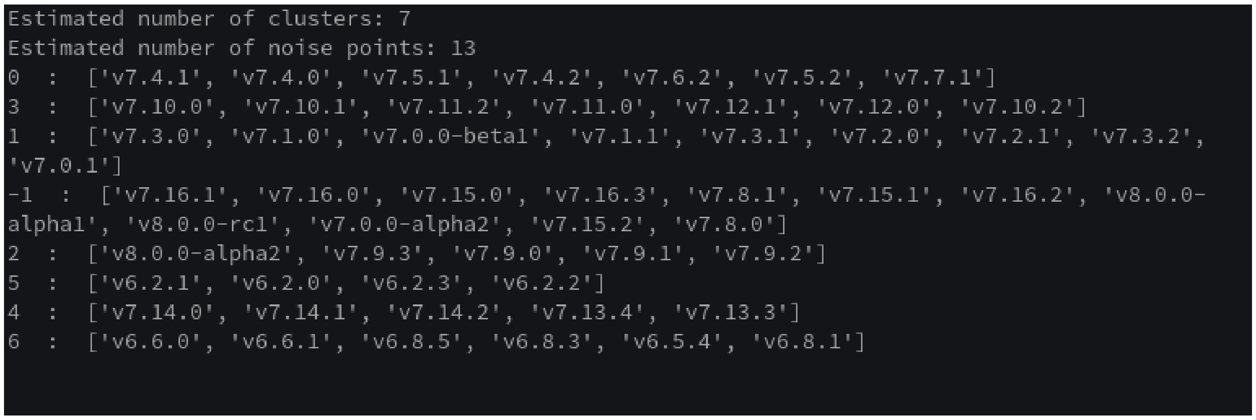
5. Results and Discussions

When the results of the graph embedding with FEATHER-G algorithm are analyzed, it can be seen that the algorithm generates an embedding vector from dependency graphs. Each vector has 500 elements in it. In the above figure, the first 10 elements were printed for a couple of embedding vectors, in order to get an idea of the output format. This representation of the graph is much easier to work with compared to the previous format, since machine learning algorithms work much more efficiently on smaller size inputs.



```
500 [ 0.28031026  0.27255755  0.25247886  0.22182056  0.18313768  0.13943935
  0.09379022  0.04893569  0.00701172 -0.03061757]
500 [ 0.27857627  0.27092244  0.25110401  0.22085434  0.18270476  0.13962794
  0.0946389   0.05042283  0.00905095 -0.0281729 ]
500 [ 0.28044748  0.27268548  0.25258316  0.22189009  0.18316618  0.13942653
  0.09374214  0.04886427  0.00693327 -0.03068462]
500 [ 0.27806719  0.270762    0.25182383  0.2228485   0.18616611  0.14451333
  0.10066893  0.05711675  0.01579243 -0.02204851]
500 [ 0.28061204  0.27284691  0.25273331  0.22201409  0.18324126  0.13942447
  0.09363497  0.04863202  0.00657201 -0.03115688]
500 [ 0.27857627  0.27092244  0.25110401  0.22085434  0.18270476  0.13962794
  0.0946389   0.05042283  0.00905095 -0.0281729 ]
500 [ 0.28031026  0.27255755  0.25247886  0.22182056  0.18313768  0.13943935
  0.09379022  0.04893569  0.00701172 -0.03061757]
```

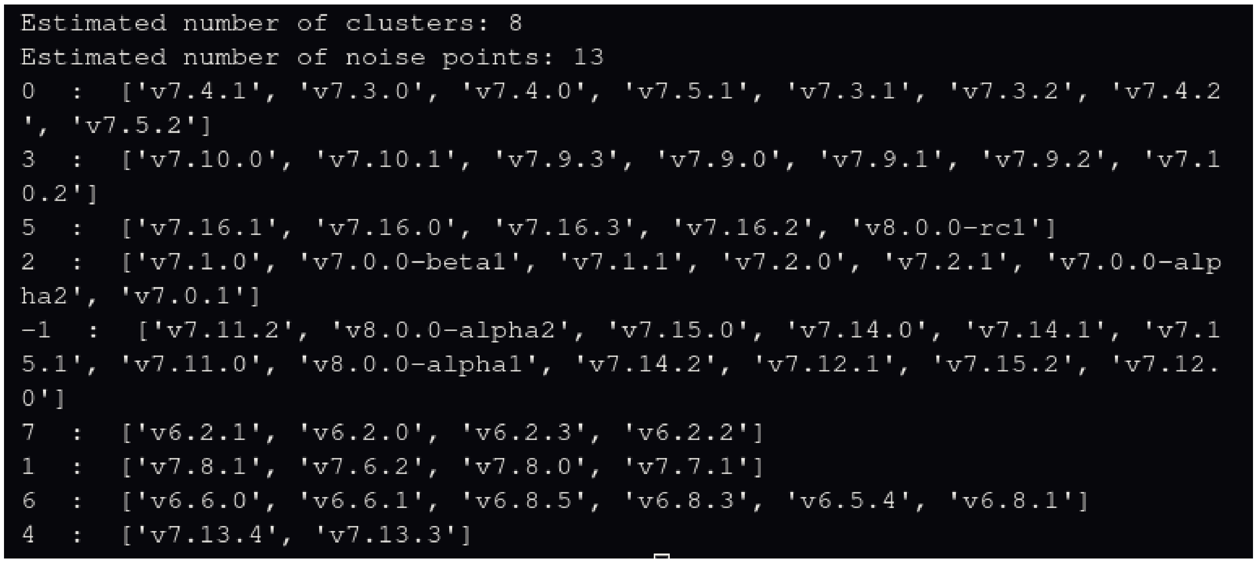
Output of Graph Embeddings(first 10 elements of the embedding vector)



```
Estimated number of clusters: 7
Estimated number of noise points: 13
0  :  ['v7.4.1', 'v7.4.0', 'v7.5.1', 'v7.4.2', 'v7.6.2', 'v7.5.2', 'v7.7.1']
3  :  ['v7.10.0', 'v7.10.1', 'v7.11.2', 'v7.11.0', 'v7.12.1', 'v7.12.0', 'v7.10.2']
1  :  ['v7.3.0', 'v7.1.0', 'v7.0.0-beta1', 'v7.1.1', 'v7.3.1', 'v7.2.0', 'v7.2.1', 'v7.3.2',
'v7.0.1']
-1  :  ['v7.16.1', 'v7.16.0', 'v7.15.0', 'v7.16.3', 'v7.8.1', 'v7.15.1', 'v7.16.2', 'v8.0.0-
alpha1', 'v8.0.0-rc1', 'v7.0.0-alpha2', 'v7.15.2', 'v7.8.0']
2  :  ['v8.0.0-alpha2', 'v7.9.3', 'v7.9.0', 'v7.9.1', 'v7.9.2']
5  :  ['v6.2.1', 'v6.2.0', 'v6.2.3', 'v6.2.2']
4  :  ['v7.14.0', 'v7.14.1', 'v7.14.2', 'v7.13.4', 'v7.13.3']
6  :  ['v6.6.0', 'v6.6.1', 'v6.8.5', 'v6.8.3', 'v6.5.4', 'v6.8.1']
```

Output of Clustering on Server Module

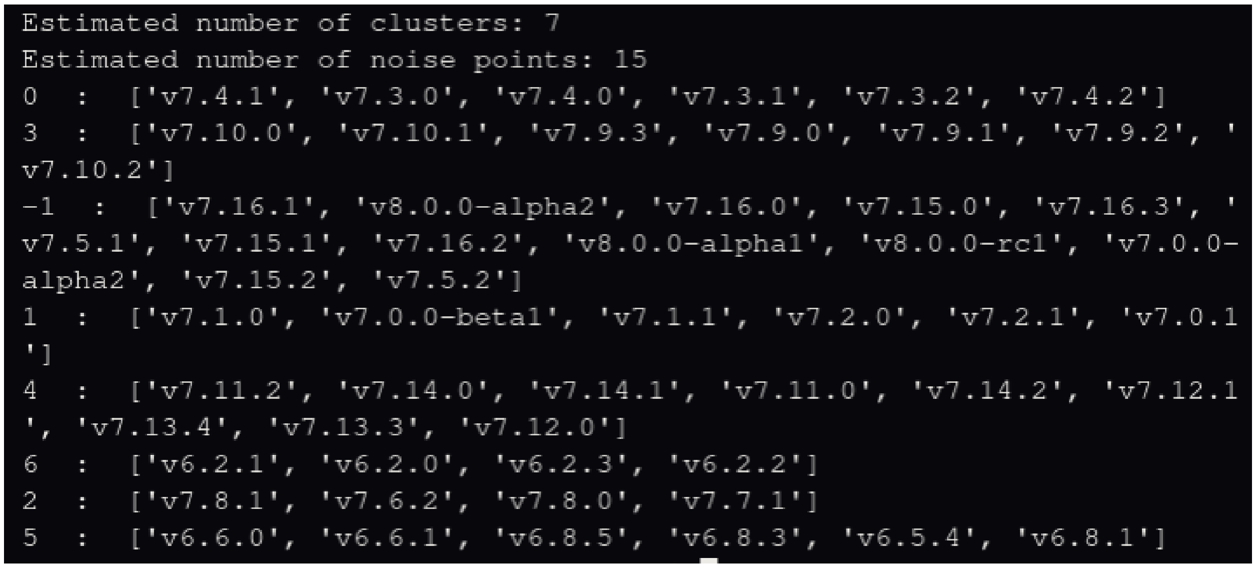
Clustering results for the server module in elasticsearch according to the releases is shown above. Cluster with ID -1 represents the outliers which are not able to be assigned to a cluster. When the clusters are analyzed, we can conclude that our approach managed to put closely released tags in the same cluster, without many mistakes.



```
Estimated number of clusters: 8
Estimated number of noise points: 13
0  :  ['v7.4.1', 'v7.3.0', 'v7.4.0', 'v7.5.1', 'v7.3.1', 'v7.3.2', 'v7.4.2
', 'v7.5.2']
3  :  ['v7.10.0', 'v7.10.1', 'v7.9.3', 'v7.9.0', 'v7.9.1', 'v7.9.2', 'v7.1
0.2']
5  :  ['v7.16.1', 'v7.16.0', 'v7.16.3', 'v7.16.2', 'v8.0.0-rc1']
2  :  ['v7.1.0', 'v7.0.0-beta1', 'v7.1.1', 'v7.2.0', 'v7.2.1', 'v7.0.0-alp
ha2', 'v7.0.1']
-1  :  ['v7.11.2', 'v8.0.0-alpha2', 'v7.15.0', 'v7.14.0', 'v7.14.1', 'v7.1
5.1', 'v7.11.0', 'v8.0.0-alpha1', 'v7.14.2', 'v7.12.1', 'v7.15.2', 'v7.12.
0']
7  :  ['v6.2.1', 'v6.2.0', 'v6.2.3', 'v6.2.2']
1  :  ['v7.8.1', 'v7.6.2', 'v7.8.0', 'v7.7.1']
6  :  ['v6.6.0', 'v6.6.1', 'v6.8.5', 'v6.8.3', 'v6.5.4', 'v6.8.1']
4  :  ['v7.13.4', 'v7.13.3']
```

Output of Clustering on action.admin.indices.validate.query Class When depth=2

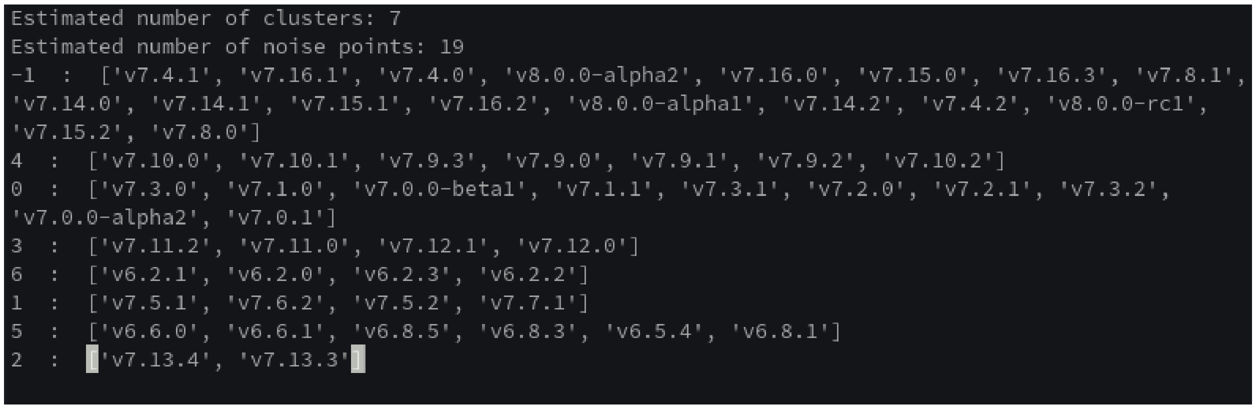
Next, we tried to apply the clustering algorithm on the embeddings for the action.admin.indices.validate.query class in elasticsearch with the depth = 2. This has added one more cluster while decreasing the number of release tags in each cluster, compared to the server module.



```
Estimated number of clusters: 7
Estimated number of noise points: 15
0  :  ['v7.4.1', 'v7.3.0', 'v7.4.0', 'v7.3.1', 'v7.3.2', 'v7.4.2']
3  :  ['v7.10.0', 'v7.10.1', 'v7.9.3', 'v7.9.0', 'v7.9.1', 'v7.9.2', '
v7.10.2']
-1  :  ['v7.16.1', 'v8.0.0-alpha2', 'v7.16.0', 'v7.15.0', 'v7.16.3', '
v7.5.1', 'v7.15.1', 'v7.16.2', 'v8.0.0-alpha1', 'v8.0.0-rc1', 'v7.0.0-
alpha2', 'v7.15.2', 'v7.5.2']
1  :  ['v7.1.0', 'v7.0.0-beta1', 'v7.1.1', 'v7.2.0', 'v7.2.1', 'v7.0.1
']
4  :  ['v7.11.2', 'v7.14.0', 'v7.14.1', 'v7.11.0', 'v7.14.2', 'v7.12.1
', 'v7.13.4', 'v7.13.3', 'v7.12.0']
6  :  ['v6.2.1', 'v6.2.0', 'v6.2.3', 'v6.2.2']
2  :  ['v7.8.1', 'v7.6.2', 'v7.8.0', 'v7.7.1']
5  :  ['v6.6.0', 'v6.6.1', 'v6.8.5', 'v6.8.3', 'v6.5.4', 'v6.8.1']
```

Output of Clustering on action.admin.indices.validate.query Class When depth=3

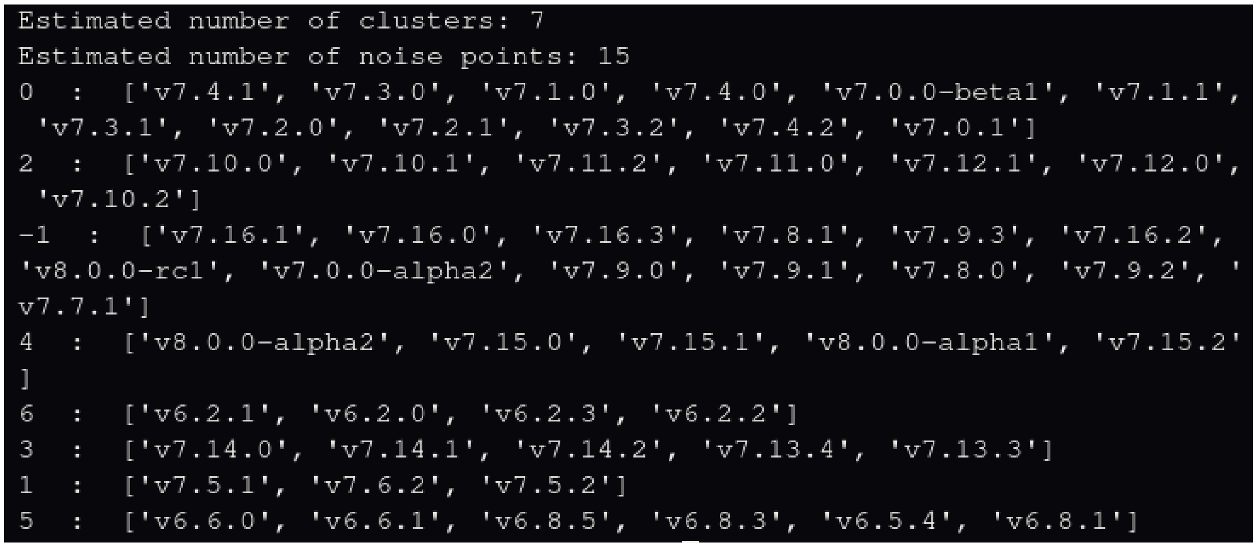
Then, we created clusters based on the dependency graph embeddings of the action.admin.indices.validate.query class again, but this time the graphs were created with the depth parameter as 3. This increase in depth causes the dependency graph to include more dependencies for the class, by also adding their neighbors dependency to the graph. This change in depth resulted in one more cluster and 2 more outlier points. We can say that for this class, the clusters created when depth=2 can be more useful to detect vulnerabilities, since there are less outliers in this case.



```
Estimated number of clusters: 7
Estimated number of noise points: 19
-1  :  ['v7.4.1', 'v7.16.1', 'v7.4.0', 'v8.0.0-alpha2', 'v7.16.0', 'v7.15.0', 'v7.16.3', 'v7.8.1',
'v7.14.0', 'v7.14.1', 'v7.15.1', 'v7.16.2', 'v8.0.0-alpha1', 'v7.14.2', 'v7.4.2', 'v8.0.0-rc1',
'v7.15.2', 'v7.8.0']
4  :  ['v7.10.0', 'v7.10.1', 'v7.9.3', 'v7.9.0', 'v7.9.1', 'v7.9.2', 'v7.10.2']
0  :  ['v7.3.0', 'v7.1.0', 'v7.0.0-beta1', 'v7.1.1', 'v7.3.1', 'v7.2.0', 'v7.2.1', 'v7.3.2',
'v7.0.0-alpha2', 'v7.0.1']
3  :  ['v7.11.2', 'v7.11.0', 'v7.12.1', 'v7.12.0']
6  :  ['v6.2.1', 'v6.2.0', 'v6.2.3', 'v6.2.2']
1  :  ['v7.5.1', 'v7.6.2', 'v7.5.2', 'v7.7.1']
5  :  ['v6.6.0', 'v6.6.1', 'v6.8.5', 'v6.8.3', 'v6.5.4', 'v6.8.1']
2  :  ['v7.13.4', 'v7.13.3']
```

Output of Clustering on index.query Class When depth=2

To see the relationship between depth value and clusters more clearly, we also created clusters for index.query class in Elasticsearch. When depth=2, there were 19 outlier points, which is higher than any of our previous attempts.



```
Estimated number of clusters: 7
Estimated number of noise points: 15
0  :  ['v7.4.1', 'v7.3.0', 'v7.1.0', 'v7.4.0', 'v7.0.0-beta1', 'v7.1.1',
 'v7.3.1', 'v7.2.0', 'v7.2.1', 'v7.3.2', 'v7.4.2', 'v7.0.1']
2  :  ['v7.10.0', 'v7.10.1', 'v7.11.2', 'v7.11.0', 'v7.12.1', 'v7.12.0',
 'v7.10.2']
-1  :  ['v7.16.1', 'v7.16.0', 'v7.16.3', 'v7.8.1', 'v7.9.3', 'v7.16.2',
'v8.0.0-rc1', 'v7.0.0-alpha2', 'v7.9.0', 'v7.9.1', 'v7.8.0', 'v7.9.2', '
v7.7.1']
4  :  ['v8.0.0-alpha2', 'v7.15.0', 'v7.15.1', 'v8.0.0-alpha1', 'v7.15.2'
]
6  :  ['v6.2.1', 'v6.2.0', 'v6.2.3', 'v6.2.2']
3  :  ['v7.14.0', 'v7.14.1', 'v7.14.2', 'v7.13.4', 'v7.13.3']
1  :  ['v7.5.1', 'v7.6.2', 'v7.5.2']
5  :  ['v6.6.0', 'v6.6.1', 'v6.8.5', 'v6.8.3', 'v6.5.4', 'v6.8.1']
```

Output of Clustering on index.query Class When depth=3

When the depth is increased to 3 for the index.query class, the 4 of the outliers are assigned to a cluster while the number of clusters remain the same, providing a better result.

The increase in depth has caused different effects on the clustering for different classes, so we could not come up with a recommended depth value that works for all SUT's and classes. Depth values need to be adapted accordingly, for each SUT and class. After finding an appropriate depth value and a class which provides the least possible amount of outliers; if a new release of the SUT can be assigned to the clusters with close dates, it can be concluded that the release is not likely to contain any vulnerability issues.

6. Threats to Validity

There were lots of unassigned releases due to the build problems while building elasticsearch using Gradle. For this reason, dependency graphs for some releases are missing, which makes the analysis less reliable because those missing releases could change the clustering results. Since most of these unassigned points are recent releases, this problem might be fixed when more versions are released to form a new cluster with the unassigned ones.

The main focus for this research was continuous static analysis based on dependencies between classes. Other dependencies or different factors were not considered for analyzing the software under test. The results may be improved by creating the clusters based on factors other than dependencies between classes.

Another threat is our approach is not tested with exact bugs, because most of the bugs reported to elasticsearch project don't have their root causes with them. That’s why the efficiency of our approach is not fully measured and it will require another project with a clear description of bugs in order to measure the effectiveness of our approach.

7. Concluding Remarks and Future Work

Continuous Static Analysis is a novel approach to find vulnerabilities in each release of a SUT. In this project, we focused on the dependency graphs of the classes of Elasticsearch. A graph embedding algorithm, FeatherGraph, was used to make the graphs easier to use with the OPTICS clustering algorithm. The clusters we got as output were not perfect, there were a lot of outliers and the number of releases in each cluster varied a lot. These issues may decrease/increase for other softwares and/or other dependency graphs. However, we successfully created the clusters and they can be used to have an idea on whether a new version of the SUT is likely to contain vulnerabilities based on broken or newly created dependencies between classes.

Approaches other than focusing on dependencies can also be applied and the results can be compared to see which approach provides more accurate results. Also, various different softwares can be tested with the same approach, to see if the approach is suitable for all softwares and to find out if there is a pattern in depth values. Using a method that can reduce the number of outliers will probably improve the results.

Another future work for this project could be refactoring the source code of this project, so source code would be easier to read and maintain. Also, adding a visualization side and interactiveness for this project in the long run could provide an easy to understand, interactive tool for continuous static analysis which allow its users to visualize those dependency graphs and visualize the possible bugs on a project.

8. References

[1] *Elasticsearch*. (2010). Github. https://github.com/elastic/elasticsearch

[2] Yılmaz, Cemal. "CS560: Automated Debugging Lecture Slides." Sabancı University.

[3] Prontolabs. *Pronto.* (2021). Github. https://github.com/prontolabs/pronto

[4] "Continuous Static Analysis - Parasoft Jtest 2021.1." *Parasoft Documentation*, https://docs.parasoft.com/display/JTEST20211/Continuous+Static+Analysis.

[5] Rozemberczki, B., & Sarkar, R. (2020, October). Characteristic functions on graphs: Birds of a feather, from statistical descriptors to parametric models. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management* (pp. 1325-1334).

[6] Ankerst, Mihael, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. "OPTICS: ordering points to identify the clustering structure." ACM SIGMOD Record 28, no. 2 (1999): 49-60.