Design and analysis of incremental clustering algorithms for large dynamic similarity graphs

Dávid Maliga Levente Buttyán[1]

Abstract: This work focuses on efficiently clustering large-scale, dynamically growing malware similarity graphs that represent malware repositories, with the primary goal of reducing storage requirements for malware samples. As these malware repositories receive new samples every day, their similarity graph is dynamically growing, therefore, requiring a dynamic graph clustering approach. We propose two incremental clustering approaches that process new samples without reclustering the entire graph from scratch every time new samples are added. The proposed methods efficiently support a differential storage scheme, which achieved a nearly 50% reduction in required storage space.

Keywords: incremental clustering, malware similarity, dynamic graphs

1 Introduction

Large malware repositories such as VirusTotal[2], MalwareBazaar[3] and Kaibou Repo[4] are essential resources of the cybersecurity community, providing a comprehensive collection of malware samples for analysis and research purposes. However, efficiently storing the hundreds of millions of samples that they contain presents significant challenges.

To store samples efficiently, one approach is to use differential storage, where only the differences between similar malware samples are stored rather than storing each individual malware sample as a separate file. Since a significant portion of malware is not entirely new but rather modified variants of earlier versions, there is significant overlap between the binary content of malware samples, especially those belonging to the same family. By identifying and grouping similar malware samples together, one can store such groups differentially, reducing storage requirements while still allowing for lossless retrieval of every sample in the repository.

Differential storage methods organize these groups of similar malware samples into a structure known as a delta tree. A delta tree stores the differences between similar samples in a hierarchical structure. The root of the tree is a representative sample, stored as a whole, while the nodes lower down in the tree do not contain the entire sample but only the difference relative to their immediate parent.

One way to find similar malware samples is to use a similarity digest scheme, such as Trendmicro Locality Sensitive Hash (TLSH) [1]. TLSH generates a hash value for a given input such that similar inputs produce similar hash values. TLSH also provides a function to compare similarity hashes and quantifying their difference. One can think of this as quantifying the dissimilarity of inputs, where a smaller TLSH difference means lower dissimilarity, therefore, higher similarity.

By using TLSH, one can efficiently compute the (dis)similarity between large numbers of malware samples, which allows for building a so-called similarity graph. In a similarity graph, each node represents an individual malware sample, and there is an edge between two nodes if their (dis)similarity is above (below) a certain threshold.

Graph clustering algorithms can then be applied to the similarity graph to identify groups of similar malware samples, which can be stored more efficiently using differential storage.

[1]All authors are with the Laboratory of Cryptography and System Security (CrySyS Lab), Department of Networked Systems and Services, Budapest University of Technology and Economics

[2]https://www.virustotal.com

[3]https://bazaar.abuse.ch

[4]https://ukatemi.com/products/kaibou/

However, since malware repositories are dynamic and receive new samples every day, traditional static graph clustering algorithms are not efficient. This is because they would require re-clustering the entire graph from scratch every time new samples are added, which is inefficient. This is where incremental clustering can help, allowing for efficiently updating the clusters in the similarity graph as new malware samples are added, without recomputing the entire clustering.

In this work, we propose two incremental clustering algorithms: the Incremental Join Closest Cluster Head (I-JCCH) algorithm and a modified version of I-JCCH with a limited cluster size n (I-JCCH-n). Our algorithms are designed to efficiently update the clusters in the similarity graph as new malware samples are added. We evaluate the performance of these algorithms on a large dataset of malware samples and compare the compression achieved by a differential storage scheme and the time required to retrieve the differentially stored samples.

2 I-JCCH

First, we introduce our I-JCCH clustering algorithm. The main idea behind I-JCCH is to maintain a set of clusters and their cluster heads, where a cluster is a group of malware samples that are stored together differentially and a cluster head is the root of the delta-tree of the cluster. I-JCCH updates the clustering when new samples are presented to it by considering the similarity of the new samples to the actual cluster heads.

An initial set of clusters and cluster heads can be obtained in an early stage, when the number of malware samples and the size of their similarity graph are still manageable, by clustering the similarity graph using a static graph clustering algorithm. For instance, this algorithm may randomly select a node and all its neighbors in the similarity graph to form a cluster, and repeat this step until every sample is assigned to a cluster. The resulting clusters can then be stored differentially, and the roots of the delta-trees of the clusters become the cluster heads.

When a new malware sample N is encountered, I-JCCH identifies the cluster head H that is the most similar to N. If the similarity between H and N is above a certain threshold, then N is added to the cluster of H. Otherwise (i.e., if the new sample is not similar enough to any existing cluster head), N forms a new cluster and becomes its own cluster head. In this way, I-JCCH can efficiently update the clusters as new malware samples are encountered, without having to recompute the entire clustering from scratch.

3 I-JCCH-n

In I-JCCH, a cluster can grow indefinitely as new samples are added, leading to large clusters where adding new samples to the delta-tree and retrieving stored samples can become inefficient. To address this problem, we introduce a variant of I-JCCH that we call I-JCCH-n. The main idea behind I-JCCH-n is to limit the size of every cluster by n.

New samples are typically added in batches. When a batch of new samples is encountered, I-JCCH-n updates the clusters iteratively by processing every new sample in the batch, one after the other. For each new sample N in the batch, I-JCCH-n identifies the closest cluster head, similar to I-JCCH. However, if the cluster of the closest cluster head has already reached its maximum size n, then N cannot be added to that cluster. In this case, I-JCCH-n tries to add N to the cluster of the next closest cluster head. If that fails, then I-JCCH-n tries to add N to the cluster of the next closest cluster head again, and so on. If N cannot be assigned to any existing cluster, it will form a new so-called **virtual cluster** and become a new **virtual cluster head** itself.

Virtual clusters are handled in the same way as actual clusters while processing the batch of new samples. When the entire batch is processed, and every new sample is placed in an actual

or a virtual cluster, the actual clusters are updated with their assigned new samples, whereas the virtual clusters are turned into actual clusters by differentially packing them.

This way, I-JCCH-n maintains bounded cluster sizes, thereby improving the efficiency of adding new samples to the delta tree and retrieving stored samples.

4 Results

The dataset used for our measurements was provided by Ukatemi Technologies and comes from the Kaibou Repo. From the dataset, we randomly selected $100,000$ samples, resulting in a subset totaling approximately 116.2 GB in size. This subset was used for all the measurements described in the following.

The proposed methods were evaluated in two scenarios. In the first scenario, we randomly selected $50,000$ samples, applied the static clustering algorithm (described above) to them, and then added the remaining $50,000$ samples incrementally in batches of $1,000$, clustering each batch with the I-JCCH algorithm. The second scenario used the I-JCCH-n algorithm with different cluster size upper-bounds n for our measurements: 25, 50, 100, 200, and 400. To eliminate the random effects of splitting the dataset into two for bootstrapping and incremental processing, as well as the random effects of choosing the batches, we repeated every experiment 10 times and present the results in box-plots. We use two TLSH dissimilarity threshold values for both scenarios: the empirically promising 40 [2] and 86 [1].

First, we examined how efficiently the dataset was compressed. The results are shown in Figure 1. Differentially packing clustered samples reduced the original dataset size by almost 50%. The best compression, 48% was achieved with I-JCCH and a TLSH difference threshold of 86.

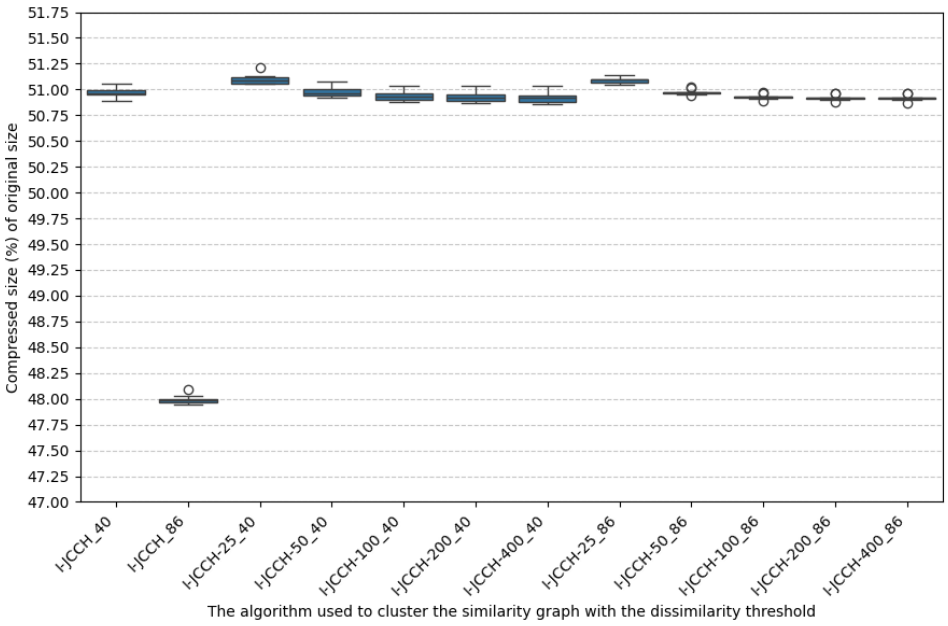

Figure 1: Boxplots of compressed size percentage (relative to original size) for different clustering algorithms and dissimilarity thresholds

Then, we measured the sample retrieval time for each algorithm. For this, we randomly selected $20,000$ samples and measured the time required to retrieve each differentially stored sample. The results are shown in Figure 2. Here, I-JCCH-n achieved the best results, especially with small upper-bounds on the cluster sizes. As it can be seen, the retrieve time of I-JCCH

changes significantly when the TLSH difference threshold is changed, while I-JCCH-n remains stable for every TLSH difference threshold.

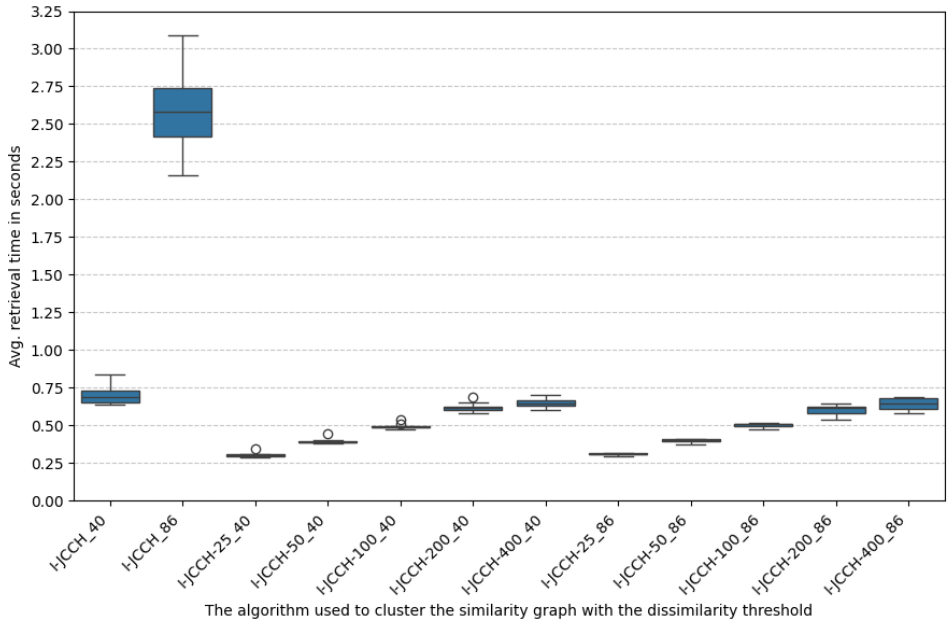

Figure 2: Boxplots of average retrieval time for different clustering algorithms and dissimilarity thresholds

In conclusion, the results show a trade-off between storage efficiency and retrieval time, which is a natural consequence of a higher TLSH dissimilarity threshold. A higher TLSH dissimilarity threshold creates more edges by accepting greater dissimilarities and allowing samples to be clustered with other samples, leading to larger clusters. Larger clusters can improve storage efficiency but also increase retrieval time. At a lower dissimilarity threshold, samples can end up clustered as singletons (clusters containing only one node). In this case, differential storage does not offer any compression benefit, but at the same time, there is no retrieval cost. Selecting the appropriate clustering algorithm and dissimilarity threshold could depend on specific application requirements.

Acknowledgements

The research presented in this paper was carried out in the context of project no. 2023-1.1.1-PIACI_FÓKUSZ-2024-00030 funded by the National Research, Development and Innovation Office of Hungary.

References

[1] J. Oliver, C. Cheng and Y. Chen, "TLSH – A Locality Sensitive Hash," *2013 Fourth Cybercrime and Trustworthy Computing Workshop*, Sydney, NSW, Australia, 2013, pp. 7-13, doi: 10.1109/CTC.2013.9.

[2] L. Buttyán, R. Nagy and D. Papp, "SIMBIoTA++: Improved Similarity-based IoT Malware Detection," *2022 IEEE 2nd Conference on Information Technology and Data Science (CITDS)*, Debrecen, Hungary, 2022, pp. 51-56, doi: 10.1109/CITDS54976.2022.9914145.