# HAC-T and Fast Search for Similarity in Security

Jonathan Oliver
*TrendMicro Research*
Australia
jon_oliver@trendmicro.com

Muqeet Ali and Josiah Hagen
*TrendMicro Research*
USA
{muqeet_ali,josiah_hagen}@trendmicro.com

***Abstract*—Similarity digests have gained popularity for many security applications like blacklisting/whitelisting, and finding similar variants of malware. TLSH has been shown to be particularly good at hunting similar malware, and is resistant to evasion as compared to other similarity digests like ssdeep and sdhash. Searching and clustering are fundamental tools which help the security analysts and security operations center (SOC) operators in hunting and analyzing malware. Current approaches which aim to cluster malware are not scalable enough to keep up with the vast amount of malware and goodware available in the wild. In this paper, we present techniques which allow for fast search and clustering of TLSH hash digests which can aid analysts to inspect large amounts of malware/goodware. Our approach builds on fast nearest neighbor search techniques to build a tree-based index which performs fast search based on TLSH hash digests. The tree-based index is used in our threshold based Hierarchical Agglomerative Clustering (HAC-T) algorithm which is able to cluster digests in a scalable manner. Our clustering technique can cluster digests in $O(n\ logn)$ time on average. We performed an empirical evaluation by comparing our approach with many standard and recent clustering techniques. We demonstrate that our approach is much more scalable and still is able to produce good cluster quality. We measured cluster quality using purity on 10 million samples obtained from VirusTotal. We obtained a high purity score in the range from 0.97 to 0.98 using labels from five major anti-virus vendors (Kaspersky, Microsoft, Symantec, Sophos, and McAfee) which demonstrates the effectiveness of the proposed method.**

***Index Terms*—Clustering, Hierarchical Agglomerative Clustering (HAC), Approximate Nearest Neighbour, Fuzzy Hashing, Trend Locality Sensitive Hashing (TLSH)**

## I. Introduction:

Similarity is a powerful tool for information security defense. Similarity measures detect novel artifacts that are similar to others of interest, whether good or bad. In this manner, defenders can detect variants of malicious software, shellcode, or spam. Similarity can be used to cluster like artifacts together, helping to group goods with goods and bads with bads, and finding the bads hiding like goods. Clustering unknown data in some interpretable way enables analysts to quickly investigate latent groups, the clouds around them, and the distant dissimilar outliers. For the problems of search and clustering, a good similarity measure provides great leverage to group similar bytestreams (malware), and discover previously unknown malware by examining outliers.

In this paper, we aim to address the problem of aiding security analysts to quickly sift through large corpus of data, in order to find samples of interest, or draw meaningful association between different malware/goodware. Fast search and clustering are two fundamental tools which can help analysts to hunt down the malware samples, and improve the process of security analysis/malware hunting. While different approaches have been utilized in the past to cluster malware, however, they seem to lack in two important aspects, namely scalability and resistance to evasion. Previous approaches do not scale to hunting amidst billions of samples that exist in the wild owing to their algorithmic characteristics. These approaches either depend upon features which are hard to compute, or use clustering techniques which are inherently not useful for clustering malware at large scales. In Section II we provide more details on previous work, and highlight the differences with our work. Secondly, previous work relies on features which are based on metadata instead of whole bytestreams (as is the case with TLSH digests) and are more prone to evasion attacks as the attacker can subvert the whole clustering if one or more of the metadata involved is compromised.

The main contributions of the paper are the design of a fast search technique which is based on tree-based index and utilizes the properties of the TLSH. The tree-based index is then used to perform threshold based Hierarchical Agglomerative Clustering (HAC-T) which is able to cluster TLSH hash digests in $O(n\ logn)$ time on average. We have performed an empirical evaluation of HAC-T on up to 10 million samples from VirusTotal using commodity cloud compute, and show that it outperforms other standard and recent clustering techniques while obtaining clusters which are of high quality. In summary we make the following contributions in this paper:

- We present a fast search technique which is based on nearest neighbors techniques as elaborated in Section V.
- We build on the fast search technique to do threshold based clustering based on Hierarchical Agglomerative Clustering.
- We have done a comparative analysis of different well-known clustering techniques, and show that none of these techniques is well-suited to cluster TLSH digests at scale.
- We show the HAC-T is highly scalable; it can be used to cluster datasets with 10s of million of samples (and potentially more) while still obtaining good quality of clusters. We obtained a high purity score in the range of 0.97 to 0.98 for five well-known anti-virus vendors.

The organization of the rest of the paper is as follows. In Section II we present details of prior work. In Section III we give an overview of TLSH. In Section IV we present our system overview and experimental design. In Section V we present our work on fast search based on nearest neighbors, and in Section VI we provide details of our clustering method (HAC-T). Section VII gives an overview of the empirical evaluation conducted, and finally in Section VIII we present concluding remarks.

## II. Prior Art:

TLSH is a high performance similarity hash digest, as it gives excellent detection rates for finding similar byte streams/files while minimizing the false positives. [1]. Authors in [2] arrive at the conclusion that TLSH is particularly well-suited at identifying variants of software when minor source code changes occur. Recent research measuring the effectiveness of different similarity hashing techniques at detecting code re-use and identifying code similarity of programs compiled with different optimizations demonstrates that TLSH performs well on these tasks [3]. Other similarity measures like Lempel-Ziv Jaccard (LZJ) are useful but simply not scalable enough to be used in a large-scale study [4]

There are many works in the literature which aim to cluster malware samples in order to make classification and analysis of malware easier for the analysts. Authors in [5] present a clustering technique which scales to cluster billions of files. It is restricted to using meta-data about the files. It uses sketches to represent files, and DBSCAN is used to obtain final clusters using approximate nearest neighbors. Another recent paper presents a variant of DBSCAN making use of approx. nearest neighbors approach based on HNSW (Hierarchical Navigable Small World) graphs [6]. In Section VII we demonstrate a Hierarchical Agglomerative Clustering (HAC-T) approach that scales better algorithmically and empirically than these clustering approaches.

A variety of hierarchical clustering techniques have been used on malware, and most of the approaches fail to scale to millions of samples or more. A study using Hierarchical Agglomerative Clustering (HAC) evaluates the clustering effectiveness of ssdeep, sdhash, and mv-HashB [7]. This work suggests improvements in the design of distance metrics to obtain better quality clusters. However, the dataset was quite small (only 1146 samples were considered). Authors in [8] use Locality Sensitive Hashing (LSH) technique based on approximate Jaccard similarity to cluster malware based on hierarchical clustering technique. The dynamic features of the malware are used to obtain the behavior profile when is then used to compute the LSH. Again, the sample set was rather limited (only around 70,000). A prototype-based agglomerative clustering algorithm is used in another study which only clusters the prototypes based on static features like opcode [9]. The sample set consisted of approx. 130,000 samples. Another paper uses feature hashing to compactly represent each malware [10]. It uses co-clustering to identify subsets of malware to be grouped together. They used a sample set of 1000 for single-node implementation and approx. 655,000 for distributed implementation. These studies [7]–[10] either did not evaluate their techniques on a large-scale or their estimates of throughput take on the order of days to scale to a million samples. Hierarchical clustering is also used in other studies to cluster malware [11]–[13]. One study uses hierarchical clustering to group together android-based malware based on shared code segments [11]. Hierarchical clustering is used also in [12] to cluster malware samples based on features extracted from the command line, like process name, and command line arguments. Hierarchical clustering is also evaluated with different distance metrics and linkage criterion in [13]. The sample size for these works [11]–[13] does not exceed 100,000.

A comparative analysis of several clustering techniques based on different distance and evaluation metrics is presented in [14]. It found hierarchical and density-based approaches as winners. A comparison study of different clustering techniques is also conducted in [15]. The study concludes that BIRCH clustering algorithm followed by hierarchical clustering works best for clustering malware. BIRCH is a general clustering technique which can be combined with other clustering algorithms and usually works with Euclidean based distances. Our study is based on non-Euclidean based distances, therefore the direct applicability of BIRCH is not possible. A deep learning based architecture based on auto-encoders is used to cluster malware samples of Portable Executable (PE) files [16]. A set of 54 features are used which are based on the format of the PE file. The dataset was close to 96,000 malicious samples, and 41,000 benign samples. The output from the auto-encoder is used to project onto a sample space which is then used for clustering. A byte frequency based approach is used in [17] to cluster malware samples. The byte frequency is represented as time-series data which is then converted to a sequence of symbols using symbolic aggregation approximation. The works [16], [17] are not directly comparable to our work as they depend on complex techniques and models which are difficult to scale. Neither of these works have evaluated their techniques on a million samples or more. Also, most techniques rely on a set of features to obtain clustering and are more susceptible to evasion as attackers can possibly trick such techniques by hiding or changing a few features which they could control. With TLSH, it is much harder to evade detection as the hash is computed over the whole byte stream/file involved [18].

## III. The Design of TLSH:

TLSH is a locality sensitive hash which produces a fixed-length hash digest based on the input bytes. The standard TLSH hash (which is used throughout in this paper) comes out

to be 70 characters long. TLSH hash digest has the property that two similar inputs would produce a similar hash digest (the hash computation is based on statistical features of the input bytes). The hash digest is a concatenation of the digest header and digest body. The following steps are involved in computation of the standard TLSH hash:

- All 3-grams from a sliding window of 5 bytes are used to compute an array of bucket counts, which are used to form the digest body.
- Based on the calculation of bucket counts (as calculated above) the three quartiles are calculated (referred to as $q1$, $q2$, and $q3$ respectively).
- The digest body is constructed based on the values of the quartiles in the array of bucket counts, using two bits per 128 buckets to construct a 32 byte digest.
- The digest header is composed of a checksum, the logarithm of the byte string length and a compact representation of the histogram of bucket counts using the ratios between the quartile points for $q1$:$q3$ and $q2$:$q3$

Two different TLSH hash digests are compared using the TLSH distance. The TLSH distance of zero represents that the files are likely identical, and scores greater than that indicate greater degrees of dissimilarity (please see the original paper for more details on the computation of the distance). [1] Through extensive experimentation it is shown in the paper that TLSH is good at producing similar hash digests of two similar files which have undergone small changes. The design choices of TLSH take into consideration criterion like runtime performance, anti-evasiveness, and false positive rate. Creating a digest of relatively small fixed length guarantees predictable consistent runtimes across arbitrary length byte streams. Using $K$-skip $N$-grams improves anti-evasiveness at the cost of runtime, and so the default version of TLSH use 2-skip 5-grams as a knee of the curve value.

## IV. Experimental Setup:

In this section we describe the experimental setup including data sources utilized, the measurements and evaluation metrics, and the hardware used.

### A. The Data:

We source data from VirusTotal data feeds. We used a random sample of VirusTotal data downloaded between September 2019 and Feb 2020. The VirusTotal data feed gives information about the scan results of all the major anti-virus vendors, and includes some auxiliary information (like SHA256, md5 etc.) for each file. We calculate the TLSH for each Win32 PE file that is submitted to VirusTotal, as provided by the data feed. The scan includes information on whether the file is malicious or benign, and also the result of the scan (the format is type.family.variant). We use the TLSH and scan results of five major anti-virus vendors (Kaspersky, Microsoft, Symantec, Sophos and McAfee) when doing clustering analysis to ascertain the quality of the clustering.

### B. Measurements/Evaluation:

We used silhouette coefficient to measure the clustering quality of different well-known clustering techniques, including K-Means, K-Medoid, CLARANS, and DBSCAN. Silhouette coefficient varies from -1 to +1 with scores closer to +1 showing good cluster separation while scores closer to -1 show that clusters are confused and overlap. The advantage of using the silhouette coefficient is that it can be calculated without relying on any external label to ascertain the quality of the cluster. The problem with measuring clustering quality with silhouette coefficient is that this metric is $O(n^2)$, and difficult to scale when data size grows to millions of items. Therefore, we were unable to use silhouette when measuring results of clustering on millions of samples.

In evaluating the HAC-T clustering technique presented here we used purity score. Purity score varies from 0 to 1 with scores closer to 1 showing that the cluster quality is good. Besides cluster homogeneity and separation, another important criteria for the usefulness of the clustering techniques is the percentage of data items which are clustered and the complementary number of data items which are marked as 'noise' and not assigned to clusters. Usually there are tradeoffs involved between percentage of data items clustered, and the quality of the clusters obtained. For example, for DBSCAN, the choice of parameters *min points* and *epsilon* affects the percentage of data items marked as 'noise' and also the quality of the clusters. In this work, we provide guidance for parameters which balance cluster coverage and cluster quality for our Hierarchical Agglomerative Clustering approach (HAC-T).

### C. Hardware:

Our experimental setup consists of a commodity cloud 32-core machine with 128 GB memory, and AMD EPYC 7000 series processor (with an all core turbo clock speed of 2.5 GHz). We parallelized the implementation of some algorithms (K-Means, K-Medoid, and CLARANS) to make the comparative evaluation of these algorithms more tractable (these were parallelized using python programming language, and make use of message passing libraries). We implemented DBSCAN in C/C++ and run on a single core as a direct comparison with HAC-T is presented which relies on single-core implementation done in C/C++.

## V. Fast Search using TLSH Forest:

The fast search problem can be stated as a type of nearest neighbor search:

- We have a dataset, $D$, of items and a similarity or distance measure. For the remainder of the paper we will assume a distance measure $Dist(d1, d2)$.
- We have a new data item, $S$, where we want to find $dmin$, the element of $D$ which is closest to $S$. That is the element of $D$ which has minimum $Dist(dmin, S)$.

- We need to avoid comparing $S$ to all of the elements of $D$. Ideally, we require that we only compare $S$ to a small subset of $D$, for example to compare $S$ to a logarithmic number of elements.

There are a number of (approximate) nearest neighbor approaches. Special case fast algorithms exist for low dimension and problems with specific geometry. The generalized approaches include (i) Locality Sensitive Hashing (LSH) where the algorithm searches through data points which are mapped onto the same bucket [19], and (ii) greedy search in proximity neighborhood graphs [20] and (iii) vector approximation files [21].

The malware problem (and computer security problems in general) require a general solution. The traditional LSH approach is to rely on bucket collions to identify near matches [19]. We note that this approach has a significant problem for security, namely an adversary has a target to aim for; once they generate sufficient change to a malicious item to belong to a new bucket, then they may have defeated the search algorithm. Ideally, we want a search algorithm which is not brittle in this way.

Here we present a different approach to the traditional LSH methods. We can build a search tree where we may associate 1 or more items from $D$ at the leaf nodes. Given a new item, we can trace a path down the search tree and compare with the nodes there. Depending on our computational requirements, we can use the search algorithm from Vantage Point trees (with backtrack) [22] to do search guaranteed to find the closest item. Or we can do approximate nearest neighbor search by employing a forest of trees; an advantage of this approach is that we know the computational cost of a lookup which is proportional to the depth of the deepest tree times the number of items in the leaves. We describe the algorithm below assuming that the TLSH distance is used, but any appropriate distance measure can be used[1].

### A. Building Trees:

We now describe how we can build such a search tree from a dataset $D$. First of all, we define a SplitMethod as shown below which has inputs a node N (where we associate $N.data$ with a subset of $D$) and outputs $(Y, T, X1, X2)$. If the distance measure has the appropriate characteristics, then we can ensure that the size of $X1$ and $X2$ are approximately the same size.

We can add an additional requirement that Split Method should only be applied to nodes where the resulting threshold, $T$, which splits the set into two should be greater than some parameter. For example, if we find that a node has a $T = 1$, then it is very likely that there is no benefit in splitting $N$.

We can then build a tree by setting $root.data \leftarrow D$ and calling $TreeBuild(root)$:

[1]TLSH is an approximate distance metric (within a constant of a metric), so it can be employed in a single tree with backtrack, or it can use a forest of search trees (which can be a randomly constructed forest to ensure that they are uncorrelated trees).

**Algorithm 1** $SplitMethod(N)$

$nitems \leftarrow size(N.data)$
**if** $nitems < nitemsInLeaf$ **then**
  **return** $NULL$
**else**
  $Y \leftarrow$ random element of $N.data$
  Find threshold $(T)$ s.t. $size(X1) \approx size(X2)$ where
    $X1 \leftarrow \{xi \in N.data : Dist(xi, Y) \leq T\}$
    $X2 \leftarrow \{xi \in N.data : Dist(xi, Y) > T\}$
  **return** $(Y, T, X1, X2)$;
**end if**

**Algorithm 2** $TreeBuild(N)$

$Res \leftarrow SplitMethod(N)$
**if** $Res \neq NULL$ **then**
  $(Y, T, X1, X2) = Res$
  $N.Split \leftarrow Y$
  $N.Threshold \leftarrow T$
  $N.LC.data \leftarrow X1$
  $TreeBuild(N.LC)$
  $N.RC.data \leftarrow X2$
  $TreeBuild(N.RC)$
**end if**

We can search through the tree for items closest to an item of interest, S, by performing $Search(root, S)$ which returns the closest item to S and the distance from S to that item:

**Algorithm 3** $Search(N, S)$

**if** $isLeaf(N)$ **then**
  $X \leftarrow closestItem(N, S)$
  $d \leftarrow Dist(X, S)$
  **return** $(X, d)$
**else**
  $thisDist \leftarrow Dist(N.Y, S)$
  **if** $thisDist \leq T$ **then**
    **return** $Search(N.LC, S)$
  **else**
    **return** $Search(N.RC, S)$
  **end if**
**end if**

### B. Implementation:

We have implemented the search tree and search forest as described in the previous section. The search speed is very fast and we compared these with linear search.

### C. Requirements for a Fast Search Measure:

The fast search approach uses characteristics of the distance measure to achieve high performance. In particular we need to be able to split arbitrary sets of elements in two so that the partitions are approximately equal in size. If this is not achieved, then the tree will be unbalanced and we will fail to achieve logarithmic search times. This characteristic for fast

search can be achieved by criteria which have a smooth ROC curves [23]. In particular, similarity measures such as Ssdeep and Sdhash suffer from this as shown in the ROC curve in Figure 1 of [1].

We note that we have seen a number of papers convert TLSH (distance metric like) into a similarity score. This is done, for example, by mapping score in the range 0 – 100 to [0, 1] and all scores $> 100$ to 0 similarity. We strongly advise against turning distance metrics into similarity scores in this way; doing so breaks the properties which enable fast search and scalable clustering [2].

## VI. Threshold Based Hierarchical Agglomerative Clustering (HAC-T):

**Algorithm 4** *HAC-T*$(D, CDist)$

```
Step 1: Preprocess data
for each distance d ∈ [0, CDist] do
   ListPair(d) ← Empty
end for
for each item A ∈ D do
   (B, d) ← Search(root, A)
   if d < CDist then
      Insert (A, B) into ListPair(d)
   end if
end for
Step 2: Cluster data
for each item A ∈ D do
   Put A in its own cluster
end for
for each distance d ∈ [0, CDist] do
   for each pair (A, B) ∈ ListPair(d) do
      if cluster(A) <> cluster(B) then
         Merge(cluster(A), cluster(B)
      end if
   end for
end for
```

We now consider approaches to clustering large datasets. We consider standard approaches such as Kmeans, Kmediods (CLARANS) and density-based clustering (DBSCAN). The arguments for the Kmeans / Kmedoids is often stated as because standard agglomerative clustering methods can be very slow. The straight forward way to calculate the distance matrix requires $O(n^2)$ distance calculations, where $n$ is the number of items in $D$.. However, the methods which require the number of clusters be given in advance can be unsuited for security applications, where such knowledge cannot be known since an adversary will be creating the cluster which are the most important. In addition, small clusters may be very important, and small clusters which contain a legitimate file and a malicious variant of that file are very important to identify. We also consider the straight forward hierarchical clustering approach. If we use a fast search approach then we can evaluate the closest few data points (the closest 2 points in our implementation) to each other point, generating a distance matrix in $O(n\ logn)$ time. This makes the straight forward hierarchical approach very attractive. The approach can be readily adapted to a range of linkage approaches including complete linkage or single linkage clustering.

The algorithm to do single link hierarchical agglomerative clustering (HAC) on dataset D, with a threshold distance between clusters CDist is the HAC-T shown in Algorithm 4. The HAC-T algorithm clusters data in 2 steps. Step 1 creates a data structure with close matches; this is done in a computationally efficient way by using the fast Search algorithm from Algorithm 3. In this step, we exploit the discrete nature of LSH distances to create a convenient data structure. Step 2 does the same merge operations as a traditional HAC algorithm. The algorithm can be extended to other linkage criteria (such as complete linkage) or continuous distances in straight forward ways.

### A. Computational Complexity of HAC-T Procedure:

The HAC-T procedure is $O(n\ logn)$ as the running time is dominated by step 1, determining the distance matrix. We have confirmed this for the TLSH distance measure[3] using two methods: (i) by monitoring the relative running times of various parts of the algorithm on large datasets; and (ii) profiling the program using gprof.

## VII. Experimental Evaluation:

In this section we will first present a comparative analysis of different well-known clustering techniques, and then show that HAC-T as presented in this paper is much more scalable as compared to other techniques, and also produce good quality clusters. Next, we present a comparative analysis of different clustering methods.

### A. Comparison of Clustering Techniques:

We present an analysis of different clustering techniques which we used to cluster TLSH hash digests of PE files. For comparison purposes, we use a sample size of up to 10,000 samples. The clustering quality is measured with silhouette coefficient. We use four clustering techniques including $K$-Means, $K$-Medoid [24], CLARANS [25] and DBSCAN [26]. We finally present an evaluation of HAC-T using 10,000 samples which demonstrates that HAC-T is a clear winner in terms of performance (as measured by run times) and also produces good quality clusters. The $K$-Means approach relies on computation of mean which is obvious in case of Euclidean coordinates and distance metrics. One cannot directly compute the mean of TLSH hash digests but we make use of a heuristic which approximates the mean. The heuristic works by first preprocessing a TLSH hash digest of

[2]TLSH was designed with this characteristic to enable fast search.

[3]We did code optimization in version 3.11.0 of TLSH and got the evaluation time of the hash to be comparable with cryptographic hashes. We perform 1 million distance calculations in 147 msec on amazon linux 2 instance(https://aws.amazon.com/amazon-linux-2/).

70 characters to 70 dimensional Euclidean coordinates with ASCII values. The mean of each cluster is chosen as the hash digest which has closest TLSH distance to the mean as calculated by computing mean of 70-dimensional vectors in each cluster. We used 100 as the max. iterations parameter for $K$-Means and simple $K$-Medoid algorithms.

Note that the three algorithms ($K$-Means, $K$-Medoid, and CLARANS) were run on a 32-core machine, and were parallelized to make the experiments more tractable. We parallelized the calculation of mean in $K$-Means, and medoid in $K$-Medoid. For CLARANS, we parallelized the cost function when selecting a new neighbor.

| | n | k=10 | k=20 | k=30 | k=40 | k=50 |
|---|---|---|---|---|---|---|
| **K-Means** | 3000 | 0.07 (5.82) | 0.09 (10.38) | 0.08(14.92) | 0.09 (19.06) | 0.06 (23.51) |
| | 5000 | 0.11 (6.56) | 0.10 (11.26) | 0.07 (15.98) | 0.07(20.72) | 0.07(24.62) |
| | 10000 | 0.12(7.5) | 0.09(13.74) | 0.08(19.43) | 0.07(24.52) | 0.06(29.13) |
| **K-Medoid** | 3000 | 0.09(15.78) | 0.1(21.09) | 0.08(27.58) | 0.08(21.06) | 0.08(15.96) |
| | 5000 | 0.12(26.93) | 0.12(35.78) | 0.09(27.29) | 0.07(25.96) | 0.09(44.12) |
| | 10000 | 0.13(61.12) | 0.13(98.47) | 0.11(80.84) | 0.10(96.54) | 0.14(139.41) |
| **CLARANS** | 3000 | 0.13(3.89) | 0.12(42.43) | 0.11(59.66) | 0.09(168.65) | 0.09(162.91) |
| | 5000 | 0.13(18.44) | 0.11(42.23) | 0.10(239.08) | 0.10(142.22) | 0.10(386.44) |
| | 10000 | 0.15(53) | 0.11(429.56) | 0.10(1303.81) | 0.10(1270.66) | 0.10(5073.19) |

TABLE I
COMPARISON OF SILHOUETTE SCORE FOR K-MEANS, (SIMPLE) K-MEDOID AND CLARANS WITH RUNTIME (SEC)

As can be seen from Table I all these clustering techniques do not perform well for different values of $n$ and $k$. The interesting clusters are small in size, and it is quite challenging to find the small interesting clusters using the above techniques. In particular, it is hard to fine-tune the parameter $k$ to hunt down the clusters which are well-formed. Another difficulty is that these techniques try to cluster all the data without separating out outliers or noise which don't belong to any well-formed clusters. The result is that the clusters found are coarse-grained, and not useful for grouping together similar files. Next, we present DBSCAN which produces better quality of clusters but has scalability issues.

| | epsilon =10 | epsilon=20 | epsilon=40 | epsilon=60 | epsilon=80 |
|---|---|---|---|---|---|
| **Silhouette** | 0.92 | 0.90 | 0.82 | 0.70 | 0.58 |
| **time (s)** | 99.88 | 99.91 | 99.88 | 99.73 | 100.24 |
| **Noise (%)** | 82.51 | 79.03 | 72.95 | 66.11 | 58.34 |

TABLE II
SILHOUETTE SCORE WITH TIME(S) AND NOISE (%) AT VARIOUS VALUES OF *epsilon* AND *min points*=2 FOR DBSCAN ($n$=10000)

DBSCAN has two parameters *min points* and *epsilon*, which affects the percentage of items clustered, and the resulting quality of the clusters obtained. Table II shows the tradeoff involved between these two quantities at various values of *epsilon*. While DBSCAN is useful for finding good quality clusters it suffers from scalability issues as it is essentially $O(n^2)$ algorithm and does not scale well to millions of items.

Next, we present the clustering quality obtained with HAC-T at various values of $T$

| | T =10 | T=20 | T=40 | T=60 | T=80 |
|---|---|---|---|---|---|
| **Silhouette** | 0.86 | 0.83 | 0.77 | 0.69 | 0.61 |
| **time (s)** | 1.68 | 1.64 | 1.65 | 1.66 | 1.66 |
| **Noise (%)** | 76.94 | 72.05 | 66.19 | 59.84 | 53.39 |

TABLE III
SILHOUETTE SCORE WITH TIME(S) AND NOISE (%) AT VARIOUS VALUES OF $T$ FOR HAC-T ($n$=10000)

As can be seen from Table III HAC-T is much faster as compared to DBSCAN and also produces comparable cluster quality (note that both DBSCAN and HAC-T are run on a single-core for this comparison). Next, we will evaluate HAC-T on up to millions of items, and show how it scales on large input sizes. We also evaluate the quality of the clusters by using external labels as obtained by VirusTotal.

### *B. Scalability of HAC-T:*

We present experimental evaluation for larger data sizes. In the table below we report run-times for running HAC-T on data sizes of up to 10 million items.

| | n=1000000 | n= 5000000 | n=10000000 |
|---|---|---|---|
| **time (s)** | 478 | 3434 | 7974 |
| **Noise (%)** | 29.6 | 34 | 34 |

TABLE IV
RUNNING TIME(S) AND NOISE (%) WITH $T$=50 FOR HAC-T

As can be seen from Table IV, HAC-T based clustering presented in this paper can scale to 10 million items on a single core. A dataset of 10 million items can be clustered in approximately 2 hours and 10 mins using this HAC-T technique using commodity hardware. The percentage of items clustered are around 66 percent (for 10 million items). For comparison, the clustering technique presented in [5] can cluster 32 million items using 57 million approx. kNN queries each of which takes 4 ms to complete, which would be approx. 63 hours. Assuming the linear relationship between no. of queries and no. of files to cluster as claimed in the paper, we estimate that it would take approx. 20 hours to cluster 10 million items. Therefore, the clustering technique presented in this paper scales much better.

The maximum size of the dataset considered for experimental evaluation in [6] is around 2 million items (with euclidean distance metric). It takes around 27498 (s) to build the index and cluster this dataset which is approx. 7 hours and 38 mins. Note that using the approach presented in this paper, one can cluster 10 million items in around 2 hours and 10 mins. This comparison shows that the approach presented here has better runtime performance than other approaches have achieved. This better run-time performance is a result of the design of HAC-T which is able to perform fast search and clustering in $O(n\ logn)$ time on average, and allows HAC-T to scale to a large number of samples.

Below we tabulate clustering quality evaluated using labels gathered from VirusTotal involving five major anti-virus vendors (Kaspersky, Microsoft, Symantec, Sophos and McAfee). We made use of five anti-virus vendors as quality of the labels varies for each vendor across different data samples. We report the clustering quality using purity, and data size is 10 million items. A high purity score in the range from 0.97 to 0.98 shows the consistency of our proposed method across different vendors.

| AV vendor | Kaspersky | Microsoft | Symantec | Sophos | McAfee |
|---|---|---|---|---|---|
| purity | 0.979 | 0.983 | 0.986 | 0.980 | 0.978 |

TABLE V
PURITY SCORE FOR DIFFERENT AV VENDORS WITH $T$=50 FOR HAC-T ($n$=10000000)

Table V shows a purity score of between 0.97 to 0.98 when data is clustered using HAC-T and evaluated using scan labels from five major AV vendors. The high purity score achieved shows the proposed technique is able to produce well-formed clusters, and separates out the PE files based on malicious/benign labels.

## VIII. CONCLUSION:

In this paper we have shown that TLSH hash digests can be used to do fast search and clustering such that the clusters obtained exhibit a high degree of similarity. Previous work on malware clustering often relies on hierarchical clustering to obtain the clusters, and as such suffers from scalability issues. While some recent approaches [5], [6] claim to use more scalable techniques, but as demonstrated in the paper these approaches are still less scalable, and also more prone to evasion attacks as they are based on features which rely on metadata as opposed to utilizing the whole bytestream. Using TLSH hash digest has the advantage of using a fixed length digest which is easy to compute, performs well on range of different domains as the hash is computed at the bytestream level, and is optimized for performance and security considerations. Our fast search technique utilizes the properties of TLSH, and when augmented with our theshold based clustering technique (HAC-T) results in a fast and scalable method which can scale to large number of samples. For future work, we plan to do more experiments using more data sources in addition to the VirusTotal feed, so as to be able to demonstrate the broad applicability of our proposed technique.

## REFERENCES


[1] J. Oliver, C. Cheng, and Y. Chen, "Tlsh–a locality sensitive hash," in *2013 Fourth Cybercrime and Trustworthy Computing Workshop*. IEEE, 2013, pp. 7–13.

[2] F. Pagani, M. Dell'Amico, and D. Balzarotti, "Beyond precision and recall: understanding uses (and misuses) of similarity hashes in binary analysis," in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, 2018, pp. 354–365.

[3] J. Coffman, A. Chakravarty, J. A. Russo, and A. S. Gearhart, "Quantifying the effectiveness of software diversity using near-duplicate detection algorithms," in *Proceedings of the 5th ACM Workshop on Moving Target Defense*, 2018, pp. 1–10.

[4] E. Raff and C. Nicholas, "Lempel-ziv jaccard distance, an effective alternative to ssdeep and sdhash," *Digital Investigation*, vol. 24, pp. 34–49, 2018.

[5] K. Soska, C. Gates, K. A. Roundy, and N. Christin, "Automatic application identification from billions of files," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2017, pp. 2021–2030.

[6] M. Dell'Amico, "Fishdbc: Flexible, incremental, scalable, hierarchical density-based clustering for arbitrary data and distance," *arXiv preprint arXiv:1910.07283*, 2019.

[7] Y. Li, S. C. Sundaramurthy, A. G. Bardas, X. Ou, D. Caragea, X. Hu, and J. Jang, "Experimental study of fuzzy hashing in malware clustering analysis," in *8th Workshop on Cyber Security Experimentation and Test ({CSET} 15)*, 2015.

[8] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, "Scalable, behavior-based malware clustering." in *NDSS*, vol. 9. Citeseer, 2009, pp. 8–11.

[9] X. Hu, K. G. Shin, S. Bhatkar, and K. Griffin, "Mutantx-s: Scalable malware clustering based on static features," in *Presented as part of the 2013 {USENIX} Annual Technical Conference*, 2013, pp. 187–198.

[10] J. Jang, D. Brumley, and S. Venkataraman, "Bitshred: feature hashing malware for scalable triage and semantic analysis," in *Proceedings of the 18th ACM conference on Computer and communications security*, 2011, pp. 309–320.

[11] Y. Li, J. Jang, X. Hu, and X. Ou, "Android malware clustering through malicious payload mining," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2017, pp. 192–214.

[12] O.-B. Boţocan and G. Czibula, "Hacga: An artifacts-based clustering approach for malware classification," in *2017 13th IEEE International Conference on Intelligent Computer Communication and Processing (ICCP)*. IEEE, 2017, pp. 5–12.

[13] A. Mohaisen, O. Alrawi, and M. Mohaisen, "Amal: High-fidelity, behavior-based automated malware analysis and classification," *computers & security*, vol. 52, pp. 251–266, 2015.

[14] H. Faridi, S. Srinivasagopalan, and R. Verma, "Performance evaluation of features and clustering algorithms for malware," in *IEEE International Conference on Data Mining Workshops (ICDMW)*, 2018, pp. 13–22.

[15] G. Pitolli, L. Aniello, G. Laurenza, L. Querzoni, and R. Baldoni, "Malware family identification with birch clustering," in *2017 International Carnahan Conference on Security Technology (ICCST)*, 2017, pp. 1–6.

[16] C. K. Ng, F. Jiang, L. Y. Zhang, and W. Zhou, "Static malware clustering using enhanced deep embedding method," *Concurrency and Computation: Practice and Experience*, vol. 31, no. 19, p. e5234, 2019.

[17] N. Singh and S. S. Khurmi, "Bytefreq: Malware clustering using byte frequency," in *2016 5th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions)(ICRITO)*. IEEE, 2016, pp. 333–337.

[18] J. Oliver, S. Forman, and C. Cheng, "Using randomization to attack similarity digests," in *International Conference on Applications and Techniques in Information Security*. Springer, 2014, pp. 199–210.

[19] "Locality sensitive hashing," https://en.wikipedia.org/wiki/Locality-sensitive_hashing, [Online; accessed 24-Mar-2020].

[20] Y. Malkov and D. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *IEEE transactions on pattern analysis and machine intelligence*, 2018.

[21] R. Weber and S. Blott, "An approximation based data structure for similarity search," Citeseer, Tech. Rep., 1997.

[22] J. Uhlmann, "Satisfying general proximity/similarity queries with metric trees," *Information processing letters*, vol. 40, no. 4, pp. 175–179, 1991.

[23] J. Oliver, "On criteria for evaluating similarity digest schemes," https://github.com/trendmicro/tlsh/blob/master/2015_Euro_DFRWS_Criteria_Sim_Digests.pdf, 2015, presented at DFRWS Europe 2015.

[24] H.-S. Park and C.-H. Jun, "A simple and fast algorithm for k-medoids clustering," *Expert systems with applications*, vol. 36, no. 2, pp. 3336–3341, 2009.

[25] R. T. Ng and J. Han, "Clarans: A method for clustering objects for spatial data mining," *IEEE transactions on knowledge and data engineering*, vol. 14, no. 5, pp. 1003–1016, 2002.

[26] M. Ester, H.-P. Kriegel, J. Sander, X. Xu *et al.*, "A density-based algorithm for discovering clusters in large spatial databases with noise." in *Kdd*, vol. 96, no. 34, 1996, pp. 226–231.