Automatic Application Identification from Billions of Files

Kyle Soska
CMU
ksoska@cmu.edu

Chris Gates
Symantec
Chris_Gates@Symantec.com

Kevin A. Roundy
Symantec
Kevin_Roundy@Symantec.com

Nicolas Christin
CMU
nicolasc@cmu.edu

ABSTRACT

Understanding how to group a set of binary files into the piece of software they belong to is highly desirable for software profiling, malware detection, or enterprise audits, among many other applications. Unfortunately, it is also extremely challenging: there is absolutely no uniformity in the ways different applications rely on different files, in how binaries are signed, or in the versioning schemes used across different pieces of software.

In this paper, we show that, by combining information gleaned from a large number of endpoints (millions of computers), we can accomplish large-scale application identification automatically and reliably. Our approach relies on collecting metadata on billions of files every day, summarizing it into much smaller "sketches," and performing approximate k-nearest neighbor clustering on non-metric space representations derived from these sketches.

We design and implement our proposed system using Apache Spark, show that it can process billions of files in a matter of hours, and thus could be used for daily processing. We further show our system manages to successfully identify which files belong to which application with very high precision, and adequate recall.

1 INTRODUCTION

Back in the early days of computing, binaries, programs, and applications were often all synonyms. On modern computing systems however, applications consisting of a single standalone binary file are relatively rare. Instead, binary files are typically used in conjunction with other binary files to make up an application. For instance, executables rely on dynamic libraries; libraries may rely on certain operating system drivers to operate. A given piece of software, or application, requires all of these binary files to be present on the system to run properly. Furthermore, applications evolve over time as new functionality is added, and old bugs are fixed, creating new versions of the software. This may lead certain files packaged as part of a given application to be replaced or removed over time.

Despite the existence of such relationships between many files on a given system, most endpoint security and management software still focuses on individual files, instead of attempting to understand the broader context in which a specific file inscribes itself.

As an example, traditional static malware analysis examines a single file at a time, matching the file against known signatures for malware or extracting features to pass to a classifier. The ability to identify the primary executables, the libraries, and any kernel or system drivers would provide more context for what the file under examination is, and what it is being used for, which in turn would help to avoid mislabeling benign files as malicious, and to characterize malicious files more accurately. Likewise, system administrators working in an enterprise or government setting may want the ability to quickly and automatically check that all the machines under their purview run approved versions of all software. Such tasks are made even more complicated when employees are allowed to bring their own devices to use in the workplace. System administrators might also be interested in gaining the ability to quickly profile a machine from the applications it is running, and rapidly detect any deviations from that profile to fend off long-term compromises. For instance, a machine usually primarily used for typical office tasks, which all of sudden starts serving remote desktop connections to many other hosts, might need to be immediately inspected.

In short, there are many practical settings for which the ability to automatically detect which files belong to which application, and, conversely, which files an application uses, would be extremely useful.

In this paper, we introduce a novel approach to identify applications based on individual file reports across a large user base. We leverage information collected as part of Symantec's WINE dataset [15] to get a global view of several metrics, including the time of appearance of new executable files on end systems, as well as several types of file metadata. These reports enable us to infer relationships between files, to map files to software packages and also to identify the evolution of those files over time. Table 1 highlights an example for the WinRAR application: the application contains a number of files (executables and libraries) which are routinely modified. Some of these changes coincide with version changes (e.g., v5.2), but some do not (e.g., `rarext.dll` was apparently updated between v5.11 and v5.2).

Identifying sets of files related to *popular* benign software is relatively easy to do, by monitoring the installations or updates of popular packages in a controlled environment, or with a few simple heuristics that consider trusted installers and trusted signers. In fact, many current whitelisting techniques use this approach to

KDD'17, August 13–17, 2017, Halifax, NS, Canada.
 ISBN 978-1-4503-4887-4/17/08…$15.00
DOI: http://dx.doi.org/10.1145/3097983.3098196

[1]Version information collected based on release dates from http://filehippo.com/download_winrar/history/

First Seen	winrar.exe	unrar.exe	rarext.dll	rarext32.dll	rarext64.dll	uninstall.exe	version[1]	Aesop
2013-09-03	•	•	•	•		•	v5.0	3
2013-12-03	•	•	•	•	•	•	v5.01	2
2014-06-11	•							1
2014-09-01	•		•	•		•	v5.11	2
2014-11-03			•					0
2014-12-02	•	•	•	•		•	v5.2	1
2015-02-16	•	•	•	•	•	•	v5.21	3
2015-02-18	•		•	•				1
2015-02-20	•							1
2015-02-25	•		•			•		2
2015-11-24	•	•	•	•	•	•	v5.3	2
2015-12-02	•					•		2
2016-02-04	•		•	•	•	•	v5.31	x
2016-02-23						•		x
2016-03-05			•	•		•		x
2016-03-07			•		•			x
2016-03-29	•							x

Table 1: Example of a single cluster ("WinRAR") reconstructed from our data. The "first seen date" for each (variant of each) file in the cluster shows application evolution over time; the table highlights the most common files in the cluster. Missing entries do not mean that the file was absent in that version, but that the file was either not updated, or not present in the window of time we were considering. The Aesop column, for which data is only available until the end of 2015, shows, for each file, the number of unique Aesop clusters to which the file belongs. This highlights that our approach combines at least 20 Aesop clusters into a single group.

identify files to add to the whitelist. However, these techniques achieve only limited success on the long tail of less popular software, which includes many important enterprise specific tools and other niche software. As we will discuss later, this "long tail" accounts for the majority of all unique software packages we observe in the wild, and is thus very important to consider.

The problem we are attempting to solve is challenging for a number of reasons. First, there is no standard way to install or update software on a desktop. Many less popular programs are not consistently signed—if they are signed at all—and malicious software and greyware may actively attempt to undermine whatever monitoring is in place. Due to these limitations, we instead choose a more general approach that does not focus on the activity or relationships of files on a single endpoint, but instead considers file metadata across *many* endpoints to understand how files are related. The scientific challenge then becomes that of inferring these relationships very quickly (in a matter or milliseconds or faster), and at scale (i.e., across billions of observations).

The authors of Aesop [24] showed that reconstructing the co-occurrence relationships between files, based on information contributed from many endpoints, allows them to identify more malware instances in smaller amounts of time. They rely on "minhashing [7]," a form of locality sensitive hashing, to approximate the similarity of files based on the Jaccard similarity of their "machine sets," (i.e., the set of machines that host a given file). Files mapping to similar machine sets are grouped together and labels for one file affect other files in the group.

Aesop does have some limitations, though. Software updates, localization, shared libraries, among other factors, muddle the co-occurrence relationships to the point of capturing sets that are much more limited in size than the set of files that constitute (the specific version of) an application. For instance, consider two versions of the same application, with a number of common files across both versions. Assuming some systems have the first version, some have the second version, and some have both, then three sets of files will emerge—those unique to the first version, to the second version, and those files that exist in both versions. Even worse, since Aesop captures no file metadata and identifies each file solely based on a hash of its contents, two versions of the same file are considered to be completely different even if they differ in minor respects, such as the embedding of a license key.

We overcome some of these limitations as follows. First, we incorporate additional metadata from the file reports, including filename, path, signer, first seen dates, prevalence, file size, as well as co-occurrence information at a much shorter time scale. Using these features from a single endpoint is unreliable, but using them across many systems provides a stronger, more reliable signal.

Thus, given a short time window, we take billions of individual {file_id, machine_id, [meta data]} instances and reduce these to sketches for each file. The sketches summarize all of the different reports into a fixed amount of space per file, in effect, scaling the billions of requests down to tens of millions of file sketches for that short window.

These sketches allow us to construct a distance metric and cluster the files into more meaningful units that not only capture applications, but in many cases also allows us to identify version changes within that application.

Contributions: To our knowledge, our work is the first attempt to group all known software files into applications. We do so using scalable techniques over a massive dataset, with no restrictions that limit it to a subset of the known files (e.g., such as signed files, or those produced by particular installers). Our proposed system has the following desirable properties:

- Efficient sketching of large amounts for a relatively smaller number of primary objects (files) into fixed space per object;
- Use of a principled distance measure (cosine similarity) over the fixed-size sketches;
- Scalable and meaningful clustering even with near exact matches that can cause problems for the k-Nearest-Neighbor (kNN) neighborhoods; and
- Version extraction from large software groupings.

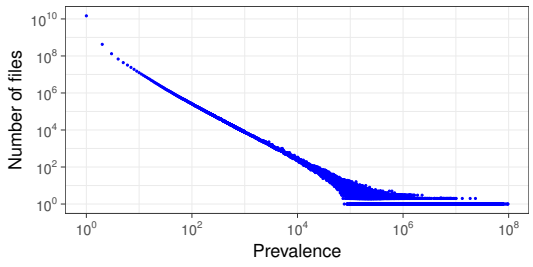

Figure 1: Number of files, ordered by prevalence. The prevalence denotes the number of times a given file has been seen across the network. The y-axis corresponds to the number of files which have that prevalence value.

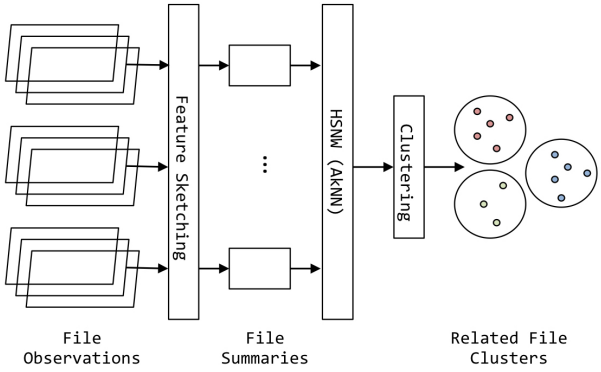

Figure 2: High-level overview of our system. Billions of file observations are compressed into file summaries, using feature sketching; these file summaries are subsequently used as a feed to a clustering algorithm

2 DATA DESCRIPTION

The data we used for this research is available through Symantec's WINE program [15], which makes various computer-security datasets collected by Symantec available to researchers, with the goal of making research efforts such as ours easier to reproduce. Here, we describe our data source in more details, as well as our data characteristics.

2.1 Data Source

Our data comes from Symantec's Norton Community Watch data, which is included in WINE, and which was also used by Aesop. This data is provided by participants that share metadata information about the files that appear on their systems, in exchange for additional protection, through a variety of data-mining algorithms [10, 24]. For our purposes, metadata of each file contains the following information, which uniquely identifies the file in question: a SHA-256 hash of the file contents, a file name, an anonymized path (corresponding to the location of the file on the machine that reported about it), and a country code reflecting the geographic location of the endpoint that reported about the file. In addition, the metadata also includes details of the public-key certificate with which the file was signed to certify its origin, if any such certificate exists.

We calculate additional features from this data, by aggregating hundreds of billions of file instances across machines. These additional features include the first time the file was ever seen, the total number of machines the file has been seen on, and co-occurrence relationships such as those calculated by Aesop [24].

2.2 Data Characteristics

An important popularity metric for our purposes is the notion of "prevalence," which characterizes the number of different times a given file has been reported on by one of the endpoints on the network. Despite the fact that high volume files account for most of the distinct file reports that the reputation system sees, Figure 1 shows that low and medium prevalence files constitute the vast majority of the unique files we see. Specifically, the figure shows that over 10^{10} files are only seen once, while one single file has been observed approximately 100,000,000 times. The relationship between prevalence and file count seems to roughly follow a power-law, with a long-tail distribution. In other words, the majority of software packages we observe appear relatively infrequently. This motivates the need to improve on Aesop [24], since Aesop operates by approximating the size of the intersection of the sets of machines on which distinct files have occurred, and thus struggles when files are of particularly low prevalence.

Likewise, (cryptographic) file signing is often insufficient to assert ownership by a given application. More than 250,000 different certificates observed in our dataset are used to authenticate the signature of a single file (and thus are impossible to directly correlate with the rest of an application). Conversely, one specific certificate has been used to vouch for the signatures of over 100,000,000 files. Furthermore, we discovered that most binary files are not signed. Worse, as we will see later, sometimes, files belonging to the same application are signed by different parties; and yet in other cases, applications consist of a mix of signed and unsigned files.

3 SYSTEM OVERVIEW

Figure 2 provides a graphical high-level overview of our system. We collect metadata on billions of files every day, which we summarize into *sketches*. These sketches are in turn used as a feed to a clustering algorithm, which outputs clusters of related files constituting applications. We next discuss each operation in turn.

Metadata and attributes. Every day, we observe over 10 billion instances of files and collect various metadata (or "telemetry") associated with the context of each observation. Disparate observations of the same file can be linked together by using the SHA256 hash of the file contents as a canonical index.

Table 2 describes the telemetry (or metadata) associated with each file, and consists of a set of attributes. Attributes denoted by *value* have a constant value across all observations corresponding to any particular file in the dataset. For example, there is only one correct value for the first time a file appeared in the data, and there is only one correct value for the prevalence (i.e., popularity) of that file. On the other hand, attributes such as the filename, or the client

Attribute	Type
SHA256	Value
Name	Distribution
Path	Distribution
Signer (Issuer)	Distribution
Signer (Subject)	Distribution
System GeoIP Country	Distribution
Prevalence	Value
Size	Value
First Seen Date	Value
Aesop [24]	Derived

Table 2: File Attributes and Features

machine's path under which the file is stored often differ considerably across observations. For these types of attributes, a *distribution* is formed over each file. Lastly Aesop [24] is a *derived* metric, specifically, a computed array of hashes useful for identifying files that usually co-occur on the same machines.

Sketching and summarizing. Due to the scale of the data we consider, reasoning about individual file observations is intractable. Instead, we condense disparate observations about a given file into a single object using data summarization techniques. Unfortunately, we cannot use exact techniques for summarization. For example, consider a popular file that is given a random name every time that it is installed: remembering these exact values would require an impractical amount of storage.

Instead of exact summarization techniques, we investigate approximations that produce a *sketch*. A useful sketch ought to concisely preserve the meaningful structure of the attribute distributions. Sketching techniques typically come with an (ϵ, δ)-guarantee that the resulting sketch is able to estimate some aspect of a data stream within error ϵ with probability $1 - \delta$.

We consider each *distribution* attribute of each unique file to be an individual data stream. Due to the scale at which new data is generated, we constrain ourselves to only looking at each file observation once. Furthermore we do not delete any observations after they are made, so ours is an insertion only stream setting.

We require our attribute summaries to be:

(1) **Efficiently computable:** The time required to update the sketch to reflect a new observation must be fast and preferably independent of the size of the data stream.
(2) **Efficiently represented:** The storage used to maintain a sketch must be sublinear in the size of the data stream with good constants such as $O(1/\epsilon)$.
(3) **Comparable:** We need to be able to estimate the similarity between the distributions of a particular attribute between two different files.
(4) **Ordering independent:** We make no assumptions about how the data might arrive to the algorithm and so it must be robust to re-ordering.

There are several well-studied techniques for summarizing data streams[3, 5, 6, 9, 12, 13, 22, 27]. Many approaches focus on identifying the most common elements in the data, or *heavy hitters*. While the precise definition of a heavy hitter differs slightly based the situation, the intuition is always to characterize the data stream by a small number of important elements.

Recent progress has been made in the asymptotic performance of heavy-hitter algorithms [3, 5, 6] but unfortunately these techniques often come with large constants. In the case of BPTree [5], the algorithm requires an estimate of the second moment [2] of the data stream which uses $O(\frac{1}{\epsilon^2} \log n \log \frac{1}{\delta\epsilon})$ bits of storage and has $O(\frac{1}{\epsilon^2} \log \frac{1}{\delta\epsilon})$ update time where n is the number of observations. We implemented BPTree and our experiments showed that even using 100 computers in parallel, the update time was too slow to keep up with new arrivals in the dataset for reasonable choices of (ϵ, δ).

Additionally, although the heavy-hitter intuition seems reasonable for many of our cases, it is sometimes woefully inadequate. Consider the example of the **path** attribute associated with a shared library. For instance, the Microsoft Visual C++ runtime is routinely bundled with applications, and therefore appears under a great number of different paths. The **path** for a file of this nature would likely not have any heavy hitters for a reasonable choice of a threshold, and therefore its distribution would probably fail to be summarized in a useful way.

As a result of these design constraints, we choose to use count-min sketch (CMS, [13]) for data summarization, which enjoys $O(1)$ update times and uses just $O(\frac{1}{\epsilon} \log \frac{1}{\delta})$ space. Our techniques utilize several details of CMS, which can be found in the original paper [13].

The following theorem states that, given two sketches provided by an (ϵ, δ) CMS, we can compute the cosine similarity of the item counts in the data streams within error ϵ with probability $1 - \delta$.

Theorem 3.1. *Let $count_a$ and $count_b$ be the count-min sketches of non-negative vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ respectively and let $S_C(\boldsymbol{a}, \boldsymbol{b}) = \frac{\boldsymbol{a} \cdot \boldsymbol{b}}{\|\boldsymbol{a}\|_1 \|\boldsymbol{b}\|_1}$ be the cosine similarity between $\boldsymbol{a}$ and $\boldsymbol{b}$. We can compute $\widehat{S_C(\boldsymbol{a}, \boldsymbol{b})}$ where $S_C(\boldsymbol{a}, \boldsymbol{b}) \leq \widehat{S_C(\boldsymbol{a}, \boldsymbol{b})}$ and with probability $1 - \delta$, $\widehat{S_C(\boldsymbol{a}, \boldsymbol{b})} \leq S_C(\boldsymbol{a}, \boldsymbol{b}) + \epsilon$ in time $O(\frac{1}{\epsilon} \log \frac{1}{\delta})$ using space $O(\log n) + O(\frac{1}{\epsilon} \log \frac{1}{\delta})$ where $n = \|a\|_1 = \|b\|_1$ and updates are performed in time $O(\log \frac{1}{\delta})$.*

Proof. From Theorem 3 of Cormode et al. [13], we can compute $\widehat{\boldsymbol{a} \cdot \boldsymbol{b}}$ as $\min_j \sum_{k=1}^{w} count_a[j,k] * count_b[j,k]$ where $\boldsymbol{a} \cdot \boldsymbol{b} \leq \widehat{\boldsymbol{a} \cdot \boldsymbol{b}}$ and with probability $1 - \delta$, $\widehat{\boldsymbol{a} \cdot \boldsymbol{b}} \leq \boldsymbol{a} \cdot \boldsymbol{b} + \epsilon \|\boldsymbol{a}\|_1 \|\boldsymbol{b}\|_1$.

The exact L_1-norm for each vector can be computed by simply incrementing a counter upon each insertion to the sketch from the data stream using $O(\log n)$ space and $O(1)$ update time.

$S_C(\boldsymbol{a}, \boldsymbol{b}) = \frac{\boldsymbol{a} \cdot \boldsymbol{b}}{\|\boldsymbol{a}\|_1 \|\boldsymbol{b}\|_1}$ is then estimated as $\widehat{S_C(\boldsymbol{a}, \boldsymbol{b})} = \frac{\widehat{\boldsymbol{a} \cdot \boldsymbol{b}}}{\|\boldsymbol{a}\|_1 \|\boldsymbol{b}\|_1}$ where $S_C(\boldsymbol{a}, \boldsymbol{b}) \leq \widehat{S_C(\boldsymbol{a}, \boldsymbol{b})}$ and with probability $1 - \delta$, $\widehat{S_C(\boldsymbol{a}, \boldsymbol{b})} \leq S_C(\boldsymbol{a}, \boldsymbol{b}) + \frac{\epsilon \|a\|_1 \|b\|_1}{\|a\|_1 \|b\|_1} = S_C(\boldsymbol{a}, \boldsymbol{b}) + \epsilon$.

The sketches require $O(\frac{1}{\epsilon} \log \frac{1}{\delta})$ space and time to estimate and $O(\log \frac{1}{\delta})$ time to update. The L_1-norm requires $O(\log n)$ space where n is the length of the data stream and is updated in time $O(1)$. Therefore the overall time requirement is $O(\frac{1}{\epsilon} \log \frac{1}{\delta})$ using $O(\log n) + O(\frac{1}{\epsilon} \log \frac{1}{\delta})$ space and updates in time $O(\log \frac{1}{\delta})$. □

Indexing. After the data has been sketched and assembled into a single feature vector for each unique file, we need a way to efficiently process and compare the points. Since we use the cosine

similarity measure to compare attribute distributions, which is not a metric, the space that the data points are mapped into is a non-metric space. Given points in a non-metric space, techniques for extracting patterns, such as Locality Sensitive Hashing (LSH, [17, 18]), have been utilized extensively by previous attempts [24] to address our problem, but yield a binary decision ("close" vs. "far"), which is too coarse-grained for our needs. Indeed, such techniques fail to capture rather common situations in which a file relates to multiple application clusters.

More appealing for our needs are mechanisms for approximating k-nearest neighbor search (akNN). One way to achieve an approximation to kNN is using Vantage Point Trees (VP-Trees) [26, 29] which have been successfully applied to the problem of searching billions of images [19]. Alternatively, graph-based approaches [14, 20, 21] have been empirically shown to perform very well on real-world data sets. We elect to use the Non-Metric Space Library (NMSLIB) [4] implementation of Hierarchical Navigable Small World graphs (HNSW, [21]).

Clustering. After the data has been indexed with HNSW, we want to cluster [1, 8, 16, 25] nearby files into groups of applications. Unfortunately, our performance requirements preclude the use of many appealing algorithms such as (most) hierarchical clustering schemes due to their $O(n^2)$ calls to the kNN subroutine. While other applications [19] have simply chosen to use a fixed distance threshold to cluster large sets of objects, we instead opt to use the DBSCAN [16] clustering algorithm, which makes only $O(n)$ queries to the kNN subroutine.

4 LOW-LEVEL IMPLEMENTATION

We next turn to a discussion of our implementation of the various algorithms described above. Our implementation attempts to ensure that our system can work in a completely automated manner, and process very large amounts of files in the smallest amount of time. We discuss the implementation of each of the components in turn.

4.1 Data Summarization

We process and sketch the raw file observations using Apache Spark and a map-reduce paradigm. Our data summarization process begins by reading in the set of file observations and shuffling the data such that all observations of any particular file reside on the same node. If a single file has too many observations to fit on any single Spark node at a time, we "page out" some observations to disk, and read them later. This step is critical to avoid memory exhaustion errors on the executors. We also load, during this initialization phase, any count-min sketches that we have previously computed for the observed files.

For each observation, we add all attributes from Table 2 of type *distribution* to their corresponding pre-existing sketches if one exists. If there does not exist a corresponding sketch, we create a new empty sketch and insert the data into that. A (δ, ϵ) count-min sketch is a matrix with height $\lceil \ln \frac{1}{\delta} \rceil$ and width $\lceil \frac{e}{\epsilon} \rceil$ whose entries come from $\mathbb{N} \cup \{0\}$ (we do not allow deletions). For our experiment we selected a matrix height of 5 and width of 64 yielding $\epsilon < 0.05$ and $\delta < 0.01$—in other words, we can compute the cosine similarity between sketched attributes within an error of 5% with greater than 99% probability.

The Spark subroutine for sketching a stream of file observations works by pairwise reducing attribute data. The routine accepts both attributes and sketches as inputs and outputs a sketch. When the two inputs are sketches, we reduce them using standard matrix addition, since the count sketches naturally compose. Non-matrix inputs are interpreted as attributes. We insert them into an empty matrix according to the rules for CMS and then reduce the resulting matrices by matrix addition.

The function is constructed in this way to minimize the maximum amount of memory that is consumed by the Spark node at any point in time. For example, since a sketch matrix is typically much larger than an attribute, converting all of the attributes to sketches and then reducing the sketches together would cause the nodes to run out of memory on files with lots of observations. Additionally from this construction, shuffling the data is very import since it allows each executor to reduce the data by growing a single matrix instead of different executors holding lots of sparse matrices and reducing them via communicating across the network. The just-in-time sketching in the reduce phase limits the amount of memory taken at any given time.

In some cases it makes sense to pre-process the attributes before sketching them. Due to the cosine similarity comparison of sketches, instances of attributes that are not identical will often fail to intersect, breaking intuitive notions of distance or similarity that may exist. Take for example the attribute **path** from Table 2. Recall that this attribute is the fully qualified path to the file in the context of the system it was observed on, and consider the following values:

File 1 Path: \program files\company\product\client

File 2 Path: \program files\company\product\utility

Despite the intuitive notion that these **path** values are similar and that the files should be close in feature space, cosine similarity between the sketches will fail to reflect this since the hash values of these strings are unrelated. To adjust for this observation, instead of simply inserting the attribute value into the sketch, we can sample a new distribution derived from the **path** value and insert those values instead. The exact number of values that are inserted into the sketch is unimportant since cosine similarity normalizes the result and it is the shape or distribution of the inputs that impacts the similarity between sketches.

More specifically, we compute the distribution to be sketched when we observe a value for **path** as follows. We start by constructing a table consisting of all directories and sub-directories that have been seen for files that were observed at least 10 times. We pair every directory with a counter which we increment when the directory is found to be either an exact or prefix match for a file's directory. We then normalize this counter by the total number of files observed to produce the probability that a randomly selected file resides in a directory that shares a prefix match with that path.

As an illustration, Table 3 shows the relative frequency for subdirectories corresponding to a particular Matlab file. We first break the instance of the **path** attribute into all of its prefixes. We automatically eliminate prefixes for which more than 2% of all files match—and do not assign any weight to such prefixes. This emphasizes the notion that sharing a common directory such as Program Files or Desktop is not a sufficient criterion to identify a common piece of software.

Path	Relative Frequency	Assigned Weight
\	1	0
\program files	0.39	0
\program files\matlab	.00211	.827
...\matlab\r2015b	.000403	.10704
...\matlab\r2015b\toolbox	.000211	.00656

Table 3: File Path Frequencies

We weight the remaining prefixes according to their information gain, that is, the reduction in entropy of the distribution of all observed files caused by fixing more of the path prefix. This procedure has the property of causing the sketch comparison to assign high similarity to pairs of files in application subdirectories since the number of files in an application subdirectory is small compared to the global number of files. There is typically a large reduction in entropy of file path when stepping from common system directories into application specific ones which is where a large volume of the mass is assigned.

In the case of Table 3, the system root and the PROGRAM FILES directory are too common and thus are not assigned any weight. The majority of entropy reduction occurs when stepping into Matlab subdirectory, and then again when moving into the R2015B folder which indicates the application version. Notice that the TOOLBOX directory is assigned relatively low weight using this approach which is ideal since intuitively this is simply an artifact of the 2015 version of Matlab. A file located in the TOOLBOX directory would have a file path sketch obtained by inserting file paths according to the distribution given by the assigned weight column.

4.2 Non-Metric Space and Clustering

We use NMSLIB [4] to implement the non-metric space for comparing summarized file attributes. We compute the distance between two vectors of file attributes by first evaluating their distance in each dimension separately. We then weight these different dimensions according to heuristics that we iteratively hand-tuned over the data set. The final distance between two files is the weighted sum of the distances in each dimension, or a weighted L_1 distance measure.

We implement the comparison between sketches using hand-tuned assembly that leverages the Intel AVX instruction set for vector floating-point operations. In our testing, using memory-aligned instructions resulted in an additional 20% throughput over non-memory aligned instructions. As a result, this operation is most efficient if the width of the sketch matrix is a multiple of 8; this justifies our choice of a width of 64.

Additionally, we extensively rely on pre-computations to make distance computations fast. For example, we normalize sketches by their L_1-norm when we project them into the space. This normalization step saves us an extra floating-point multiplication and division when computing similarity.

We give the highest weight to the **path**, **Signer (Subject)** and **prevalence** attributes (dimensions), each taking a weight of 1.5. We further rely on heuristics to improve the distance quality for several natural cases. For example, take the case of the **Signer (Subject)** attribute. Two files that have the same non-NULL signer are intuitively similar and thus assigned a distance of 0 in this dimension, and files with different signers are given a maximum distance of 1.5. In the event that both files are unsigned, that is their signer is reported as NULL, then although the signer information is technically identical between them, it is not a strong indicator that they are similar and in this particular case we assign a distance of 0.6.

We added several other such heuristics in response to observing the cases that commonly appear in the data. These include, but are not limited to, giving no weight to **size** unless **name** is a near-perfect match. This helps detect instances of a direct file update (newer version replacing an old version and using the same name) and reduces the weight of prevalence when the files are sufficiently popular.

Lastly, for clustering, we use an implementation of DBSCAN [16] with a neighborhood threshold of 3.2 and a support requirement of 4 neighbors. We selected these parameters by studying the local density of files in ground truth applications when projected into the space.

5 EVALUATION

We evaluate our system along the following lines: how fast data summarization is performed, and how much space it requires; how well the approximate kNN algorithm performs; and how well our clustering fares. We then delve into a set of case studies to illustrate the benefits of our approach.

5.1 Data Summarization

We processed our input file observations and related context telemetry using Apache Spark. We run our job on 100 executors, each with 16 GB of memory and a single 4-core 8-thread Xeon processor.

In all, we observed 498,290,979 unique files a total of 9,150,164,969 times. We stored the resulting sketches and L_1-norms for files in a hive table which was split across 3,440 files. We encoded the SHA256 identifiers as strings, the sketches as integer arrays, and the L_1-norm as an integer. In aggregate, each summarized attribute produced 41 GB of output data. The attributes that did not need to be summarized, such as file prevalence, roughly produced an additional 200 MB of data per attribute.

The entire operation was split into 3,440 tasks which computed in just over 2 hours. Of note, all but four of the tasks finished in under 30 minutes. Because file observations must be grouped by SHA256 hashes, and then reduced locally on the executors to avoid out-of-memory errors, the executors unlucky to be tasked with reducing the most popular files take a long time to finish. This difficulty is not fundamental and this instance of the "curse of the last reducer" (i.e., the last bit of computation taking a disproportionate amount of time) can be mitigated if the most frequent files are identified ahead of time and split across multiple executors.

The runtime efficiency of sketching makes it a viable operation for a nightly job, even at the scale considered. We could also reasonably expect to be able to catch up months or even years of unprocessed data, especially if we could increase the number of executors used.

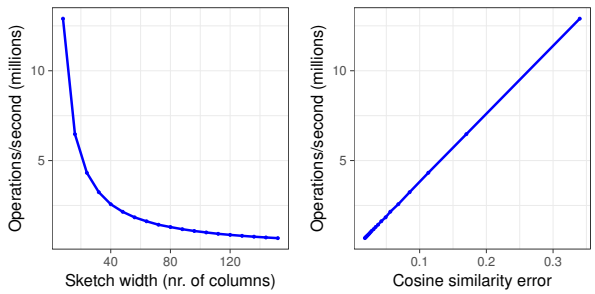

Figure 3: Number of space operations (distance computations) against sketch width (left) and cosine similarity error for 5 dimensions and $\delta = 0.01$ (right).

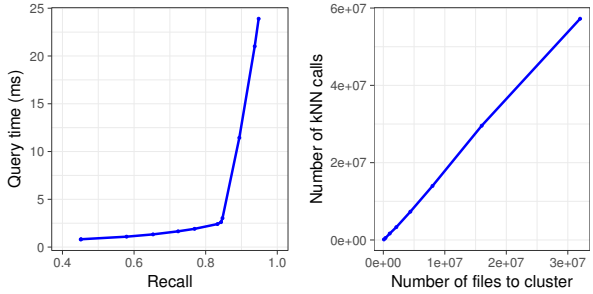

Figure 4: HNSW time/recall tradeoff (left) and clustering performance (right).

5.2 Approximate kNN

The performance of the approximate kNN solution can be characterized by the trade-off between runtime and recall—a higher recall should require a longer runtime. We first project the input data, namely the file sketches and other attributes into our custom non-metric space, by building on the NMSLIB implementation [4]. We benchmark the space using input data cached in memory with a single thread of an Intel Xeon E5-2630 v2 @ 2.60 GHz CPU.

Figure 3 shows the performance in number of operations per second (i.e., how many times the pairwise distance function can be computed per second) when using sketches of different sizes. This is the main subroutine used by Hierarchical Navigable Small Worlds (HNSW, [4]) to perform approximate kNN search, and so the performance of this subroutine will directly impact the performance of the rest of the system. All tests use 5 dimensions of sketched data and $\delta = 0.01$ which corresponds to matrices with 5 rows. We vary the number of columns, which corresponds to the choice of ϵ.

Figure 3 shows a strong linear relationship between the cosine similarity error ϵ and the number of operations per second that the space is able to support. If we are willing to allow more than 25% error in the distance comparisons between summarized attributes, then we can achieve over 10 million operations per second. For more conservative error terms, such as $\epsilon = 0.05$, the space supports about 2 million operations per second. For our experiments, we selected a sketch width of 64, which corresponds to $\epsilon \approx 0.0425$. In other words, the calculated cosine similarity between attributes in our space will be within 5% of the true value 99% of the time.

To put the performance of our implementation into context, regular cosine similarity over 100 dimensions can be computed at a rate of 13 million operations per second, while KL-divergence runs at approximately 4.8 millions operations per second. This implies that the performance of the sketch distance computation is quite reasonable compared to other well-known and widely used distance functions.

The Hierarchical Navigable Small Words graph is tunable with a few key parameters denoted by M, *efConstruction* and *efSearch*. M is the size of the initial set of neighbors during indexing. *efConstruction* governs the search depth while constructing the index. *efSearch* governs the search depth during a query. We vary these parameters and evaluate performance by constructing an index for 1 million points of our dataset for each choice of parameters. We measure the recall by performing brute force 10-NN search for 500 random points in the dataset and comparing them to the results given by the index. We obtain query time by measuring how long it takes the index to process the set of 10-NN queries. Figure 4 (left) shows the performance of HNSW, which achieves recall as high as 0.85 in less than 4 ms. That its run-time begins to increase significantly at higher recall rates is a well-known property of HNSW [21] and does not present a problem, as DBScan is able to achieve good clustering results despite imperfections in the kNN approximation.

5.3 Clustering

Figure 4 (right) shows a near-linear relationship between the number of files to cluster, and the number of kNN calls we have to make—with the latter being slightly less than twice the former. The linear relationship is typical of DBSCAN's $O(n)$ scaling and in particular the approach of scanning through the space and attempting to expand a cluster from each unlabeled point.

To compute the clustering of 32 million files requires roughly 57 million akNN queries, each of which can be satisfied in 4 ms with 0.85 recall. This scaling suggests that a well provisioned machine that recomputes clustering daily to account for new observations could scale to 100 million files. Additionally, files that were not previously clustered (were first observed after the last clustering) can be projected into the space at any time and assigned a cluster ID based on their neighborhood.

5.4 Case Studies

Due to the complexity and scale of the problem we are trying to solve, there are no openly available ground-truth datasets that we can use to evaluate our clusters against. Partial lists do exist, containing certain files for some specific software versions, but unfortunately those fall very short of what we would need for a proper comparative evaluation. Thus, instead of a principled comparison to known software packages that existed during the time of our analysis, we explore some of the clusters that emerged from our data as evidence that our system is working well.

The largest clusters. The largest cluster contains close to 200,000 unique file SHAs. This large cluster, unsurprisingly, represents Microsoft, and captures many files that exist in the operating system across many versions of Windows, and other related software. Because these files share path information, replace older versions during updates, are consistently signed by a few different signing

certificates, and appear on large numbers of systems, they are similar enough that they end up being clustered together. Microsoft is an outlier relative to the rest of the clusters, and while this cluster is very large, it is expected.

Updates. Other large clusters are modular and long-lived software that perform many updates over time. Table 1 in Section 1 gives one example of what we mean by an update: files that are part of the same cluster, appear to be replacing each other over time, and many of which share common dates. The first-seen dates are the first time that the file appeared in our dataset, even if that date is outside of our analysis window. Table 1 is interesting because it also shows some minor updates we are capturing that occur in between major releases. These are either related to beta or dev versions of the software, where the file does not get updated again in the official release, or these are patches that are applied and released outside of normal updates cycles. While this is difficult to tell given our current view of the data, seeing these files at the time of their release would have made this more clear.

Tracking similarity through updates was one of the limitations of Aesop [24], which only used co-occurrence information, and did not bring in additional information to get more complex measures of similarity. Witnessing these update patterns are some of the more useful relationships that emerge from the clusters that we examine.

Most recent version. Knowing the most recent version of specific types of software can be used to help manage a system. Enabling admins to verify and update policies which check that systems are using the latest software.

A certain, often exploited, PDF reader generated a single cluster with 7,684 files. Of those files, 42 were different versions of the primary executable used to launch the software. Sorting by the first seen date, it is easy to quickly check which files are the newest across all files and add them to a policy to allow their execution while revoking older versions that should be replaced. Our system is able to automatically detect when new versions are available with no input from an admin.

Another example is `putty.exe`, shown in Table 4. Putty is a standalone secure shell client on Windows. All versions that are reported during our analysis window appear in a single cluster, the older versions are reported for various reasons such as explicitly downloading an older version or attaching external or networked storage to a system, both resulting in the file being reported in our analysis window. We only track first seen dates since 2009, so the earlier versions of putty have an unknown first seen date. This takes 9 Aesop clusters which consist of only a single file, and merges them into 1 cluster. It also highlights the usefulness of tracking software versions, since all versions of putty prior to 0.69 currently[2] have security warnings due to known vulnerabilities. Identifying a new report of an old version, would allow us to automatically identify and alert a user based on information from the cluster.

Mixed signers. Some clusters are pure in the sense that all files are signed by a common certificate, but many clusters are not.

[2]Version 0.66 was the most recent version at the time of our analysis, 0.69 at time of writing

[3]Version information collected based on release dates from https://the.earth.li/~sgtatham/putty/0.XX/, where XX is the version

First Seen	putty.exe	Version[3]	Release date
unknown	•	v0.53	2002-10-01
unknown	•	v0.58	2005-04-05
unknown	•	v0.59	2007-01-24
unknown	•	v0.60	2007-04-29
2011-12-10	•	v0.62	2011-12-10
2013-08-06	•	v0.63	2013-08-06
2015-02-28	•	v0.64	2015-02-28
2015-07-25	•	v0.65	2015-07-25
2015-11-07	•	v0.66	2015-11-07

Table 4: All files in the cluster for ("Putty"), with corresponding first seen date, version, version release date. The 'unknown' first seen dates occur prior to our system tracking this data. There are missing versions due to the fact that some versions are old, and was not reported by a customer during our analysis window. Each version of putty existed in its own Aesop cluster, so this combines 9 Aesop clusters into a single cluster.

Even the example from Table 1 consists of files that were signed by two different certificates from two different, but similarly named, issuers. This is often the case as signing certificates expire and have to be renewed, breaking this chain. There are many clusters with this property.

Other clusters have a mix of signed and unsigned files. This means that even for signed software, using a rule based on a common signer would not be able to identify the relationships between these files, but due to the high similarity in other dimensions, our robust approach is able to group these files together. One of these clusters contains 8,408 files and represents a meeting and communications application, about half of the files are signed and half are not, but all are clearly related based off the other features.

Packet sniffing software. Popular open source packet sniffing software did not cluster into a single large cluster, but in three clusters, with approximately 180 unique file SHAs each; this may be due to the short time window we are using, and the rapid update cycles of the software. We also find several smaller clusters containing files for optional plugins. Because these files are nested deeper in the directory and are unsigned, our design parameters lead them to form their own cluster.

Nevertheless, we find these results highly encouraging: indeed, the clusters corresponding to this packet sniffing software do not include *any* files from any other application. Thus, merely detecting that some subset of the files belonging to these clusters is present on a given machine almost assuredly indicates that a packet sniffer is installed (and/or running) on that machine; this in turn may be an extremely valuable signal to an admin that certain attack tools exist on a system. Combining this insight with machine profiling would allow us to automatically raise alerts under for certain types of unexpected activity.

Unclustered Files. 121,717 files are not included in a cluster. The clustering technique we apply and the parameters we use are probable causes for the existence of these singletons. Indeed, a file needs at least four close neighbors to join a cluster. As a result, small sets of files that form islands of less than five files will not have enough support to be included. This includes programs with no dependencies that never update themselves, which is common practice for some developers.

If we decide to use less stringent conditions for cluster membership, we run the opposite risk of *overmerging*. Indeed, we then tend to group too many files together and clusters are merged together through shared libraries and other spurious relationships. Installers, in particular, usually end up in this group. Since we do not have access to the parent and child relationship when a file is written to disk, we do not have enough other information to group the installer with the main package. Installers are particularly challenging, as they are often downloaded to what appears to be random locations (in addition to traditional Desktop or Downloads folders); Aesop-style co-occurrence relationships do not hold up because many users who download a file will not run it immediately if ever, or it only appears because a backup drive was connected.

Interestingly to us, some files that we expected to be unclustered, such as `putty.exe`, actually ended up clustering with other versions of themselves as we saw in Table 4. Even though the update cycle is relatively slow, and it it only contains a single file, it is popular enough that we see many versions even for the relatively short window of our analysis.

6 DISCUSSION AND LIMITATIONS

The approach we describe in this paper is designed to generate more complete clusters compared to previous approaches. These more complete clusters help to provide additional context about the application, both related to the variety of files in the application as well as updates made over time. Understanding and exposing this information can be leveraged in many ways, but there are certainly limitations that should be acknowledged. We have already touched on some of these in the case studies, and elaborate further here.

By their very nature, shared DLLs are used by many different applications. The current approach does not allow these shared relationships to be expressed since our clustering only allows a file to exist in a single cluster. We should, in the future, improve the clustering algorithm robustness to allow files to be part of multiple groups, while maintaining favorable scaling with large numbers of points.

The scalability of the system is bottlenecked by the approximate kNN implementation. While the performance is quite impressive and allows for scaling to 100s of millions of files, another order of magnitude could be achieved by moving to a distributed approach [19]. The authors of HSNW [21] suggest that implementing a distributed extension may be straightforward and represents a promising direction for future work.

Last, given the current features, our technique does not guarantee robustness with respect to an adversary who actively tries to undermine it. In many cases, robustness in practice comes from the difficulty to adversarially manipulate several of the features, such as signer and issuer, prevalence, and co-occurrence. However, some other features, such as path and filename, can be trivially manipulated. As a result, an attacker could try to tamper with these features to force the clustering algorithm to group certain files in a cluster of the attacker's choosing. To mitigate this issue, we could consider extracting additional features from static file information; however, such additional features are not present in the current WINE dataset.

7 RELATED WORK

We considered many alternative techniques that bear relevance to the challenges we faced in performing application identification, many of which were discussed in Section 3. To the best of our knowledge, we are the first to attempt application identification on a global scale, with no restrictions that would limit us to identifying the application context of a subset of all known program binary files.

Ours was not, however, the first attempt to identify application-to-file relationships. The authors of Aesop [24] also used scalable techniques to label files as benign or malicious by placing them in the context of the other labeled files with which they are nearly always installed. Their methods were not suitable for application identification, however, as application identification was not their goal, and for this reason they made no effort to deal with updates or other issues that would cause the files of an application to not cluster together. Ye et al. [28] also used file relationships for malware detection, as do Chen et al. [11], but neither attempt to cluster or group files into applications as we do.

Possible applications of this work include assistance in predicting vulnerabilities that will be exploited in the future. In the web context, Soska and Christin [23] attempt to identify which software is powering a given website to determine whether the site is likely to be exploited or not; the work we propose here is helpful in precisely mapping out software versions running on a given host, and could be used as a feed to similar predictive classifiers.

8 CONCLUSION

In this work we sucessfully clustered billions of file observations into hundreds of thousands of meaningful application clusters using a novel combination of scalable data processing and approximation techniques. Our approach produces meaningful clusters that out outperform previous efforts in this space in terms of robustness and have significant applications to areas such as machine profiling and risk detection.

ACKNOWLEDGMENTS

This work was partially supported by a Symantec Research Labs fellowship. Many thanks to Matteo Dell'Amico for suggesting the combined use of the HNSW and DBSCAN algorithms to achieve scalable clustering. Also thanks to Leonid Boytsov for his helpful conversations and support with manipulating NMSLIB and hsnw.

REFERENCES

[1] Charu C Aggarwal, Jiawei Han, Jianyong Wang, and Philip S Yu. 2003. A framework for clustering evolving data streams. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*. VLDB Endowment, 81–92.

[2] Noga Alon, Yossi Matias, and Mario Szegedy. 1996. The space complexity of approximating the frequency moments. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*. ACM, 20–29.

[3] Arnab Bhattacharyya, Palash Dey, and David P Woodruff. 2016. An optimal algorithm for l1-heavy hitters in insertion streams and related problems. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. ACM, 385–400.

[4] Leonid Boytsov and Bilegsaikhan Naidan. 2013. Engineering efficient and effective non-metric space library. In *International Conference on Similarity Search and Applications*. Springer, 280–293.

[5] Vladimir Braverman, Stephen R Chestnut, Nikita Ivkin, Jelani Nelson, David P Woodruff, and Zhengyu Wang. 2016. BPTree: an heavy hitters algorithm using constant memory. *arXiv preprint arXiv:1603.00759* (2016).

[6] Vladimir Braverman, Stephen R Chestnut, Nikita Ivkin, and David P Woodruff. 2015. Beating CountSketch for heavy hitters in insertion streams. *arXiv preprint arXiv:1511.00661* (2015).

[7] Andrei Z. Broder. 1997. On the resemblance and containment of documents. In *Proceedings of Compression and Complexity of Sequences*. 21–29.

[8] Feng Cao, Martin Ester, Weining Qian, and Aoying Zhou. 2006. Density-Based Clustering over an Evolving Data Stream with Noise.. In *SDM*, Vol. 6. SIAM, 328–339.

[9] Moses Charikar, Kevin Chen, and Martin Farach-Colton. 2002. Finding frequent items in data streams. In *International Colloquium on Automata, Languages, and Programming*. Springer, 693–703.

[10] Duen Horng Chau, Carey Nachenberg, Jeffrey Wilhelm, Adam Wright, and Christos Faloutsos. 2011. Polonium: Tera-scale graph mining and inference for malware detection. In *Proceedings of the 2011 SIAM International Conference on Data Mining*. SIAM, 131–142.

[11] Lingwei Chen, William Hardy, Yanfang Ye, and Tao Li. 2015. Analyzing File-to-File Relation Network in Malware Detection. In *International Conference on Web Information Systems Engineering (WISE)*.

[12] Graham Cormode and Marios Hadjieleftheriou. 2008. Finding frequent items in data streams. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1530–1541.

[13] Graham Cormode and S Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75.

[14] Wei Dong, Charikar Moses, and Kai Li. 2011. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th international conference on World wide web*. ACM, 577–586.

[15] Tudor Dumitras. 2011. Field Data Available at Symantec Research Labs: The Worldwide Intelligence Network Environment (WINE). In *Proceedings of the ASPLOS Exascale Evaluation and Research Techniques Workshop*.

[16] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, and others. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise.. In *Proceedings of the ACM SIGKDD international conference on Knowledge discovery and data mining (KDD)*, Vol. 96. 226–231.

[17] Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. ACM, 604–613.

[18] Eyal Kushilevitz, Rafail Ostrovsky, and Yuval Rabani. 2000. Efficient search for approximate nearest neighbor in high dimensional spaces. *SIAM J. Comput.* 30, 2 (2000), 457–474.

[19] Ting Liu, Charles Rosenberg, and Henry A Rowley. 2007. Clustering billions of images with large scale nearest neighbor search. In *Applications of Computer Vision, 2007. WACV'07. IEEE Workshop on*. IEEE, 28–28.

[20] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. 2014. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems* 45 (2014), 61–68.

[21] Yuri Malkov and DA Yashunin. 2016. Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs. *arXiv preprint arXiv:1603.09320* (2016).

[22] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2005. Efficient computation of frequent and top-k elements in data streams. In *International Conference on Database Theory*. Springer, 398–412.

[23] Kyle Soska and Nicolas Christin. 2014. Automatically detecting vulnerable websites before they turn malicious. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security'14)*. San Diego, CA, 625–640.

[24] Acar Tamersoy, Kevin Roundy, and Duen Horng Chau. 2014. Guilt by Association: Large Scale Malware Detection by Mining File-relation Graphs. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '14)*. ACM, New York, NY, USA, 1524–1533. DOI:http://dx.doi.org/10.1145/2623330.2623342

[25] Thanh N Tran, Ron Wehrens, and Lutgarde MC Buydens. 2006. KNN-kernel density-based clustering for high-dimensional multivariate data. *Computational Statistics & Data Analysis* 51, 2 (2006), 513–525.

[26] Jeffrey K Uhlmann. 1991. Satisfying general proximity/similarity queries with metric trees. *Information processing letters* 40, 4 (1991), 175–179.

[27] David P Woodruff. 2016. New Algorithms for Heavy Hitters in Data Streams. *arXiv preprint arXiv:1603.01733* (2016).

[28] Yanfang Ye, Tao Li, Shenghuo Zhu, Weiwei Zhuang, Umesh Gupta Egemen Tas, and Melih Abdulhayoglu. 2011. Combining file content and file relations for cloud based malware detection. In *Proceedings of the ACM SIGKDD international conference on Knowledge discovery and data mining (KDD)*.

[29] Peter N Yianilos. 1993. Data structures and algorithms for nearest neighbor search in general metric spaces. In *SODA*, Vol. 93. 311–21.