An Alternative to NCD for Large Sequences, Lempel-Ziv Jaccard Distance

Edward Raff
edraff@lps.umd.edu
Laboratory for Physical Sciences
Neuromorphic Computing Research Team

Charles Nicholas
nicholas@umbc.edu
University of Maryland, Baltimore County
Dept. of Computer Science and Electrical Engineering

ABSTRACT

The Normalized Compression Distance (NCD) has been used in a number of domains to compare objects with varying feature types. This flexibility comes from the use of general purpose compression algorithms as the means of computing distances between byte sequences. Such flexibility makes NCD particularly attractive for cases where the right features to use are not obvious, such as malware classification. However, NCD can be computationally demanding, thereby restricting the scale at which it can be applied. We introduce an alternative metric also inspired by compression, the Lempel-Ziv Jaccard Distance (LZJD). We show that this new distance has desirable theoretical properties, as well as comparable or superior performance for malware classification, while being easy to implement and orders of magnitude faster in practice.

KEYWORDS

malware classification, cyber security, Normalized Compression Distance, Lempel-Ziv, Jaccard similarity

1 INTRODUCTION

The Normalized Compression Distance (NCD) [19] is a general purpose method of measuring the similarity between any two arbitrary objects. The NCD works via the use of compression algorithms, using the sizes of compressed objects, individually and then when concatenated, to compute a similarity between the two input objects. The NCD algorithm will be described in detail a little later in this paper. Since compression is the basis of the NCD, it has proven effective for comparing a wide variety of data objects, and requires no domain knowledge to apply. In addition to its intuitive appeal, the NCD also has theoretical underpinnings in terms of Kolmogorov complexity that may inspire additional confidence in it.

These practical and theoretical properties make the NCD appealing in the context of malware detection and malware family classification, which we will jointly refer to as malware classification. Malware detection is a binary classification problem in which one tries to determine if a binary is benign or malicious, and family classification attempts to label a known malicious binary as a member of one (or more) malware families. A number of others have used the NCD to do these tasks successfully using different features, including API call sequences and the raw byte contents of a file [2, 4, 15, 28]. We are particularly interested in NCD for malware since it can be used on raw bytes, requiring the use of little, if any, domain knowledge.

The minimization of domain knowledge is desirable for this task for a number of other reasons as well. Particularly, malware classification is subject to concept drift, meaning the nature of malware changes over time. This means our feature extraction process must often change with it, requiring some level of ongoing maintenance work. Malware itself will often intentionally break rules regarding format specification or attempt undefined behavior, requiring additional overhead for feature extraction which is compounded by the changing nature of malware. The more advanced domain knowledge approaches use dynamic analysis, which involves running the malware in a virtualized environment. This adds significant complexity in practice, as malware can detect that its in a virtiualized environment and alters its behavior, and the virtual environment may have many inconsistencies with real environments that prevent a system from generalizing in practice [23].

Malware classification is also an excellent test bed for any-purpose metrics such as NCD. Not only is malware classification an important problem in improving cyber security, but it is a domain for which recent advanced in Machine Learning and Deep Learning have yet to yield significant gains. This is in contrast to problems such as image and signal processing, where NCD has been applied previously but is no longer needed [17, 27]. The nature of a real malicious adversary makes feature selection and engineering particularly difficult for this domain, which NCD can partially side-step through its use of compression.

Unfortunately there exists a number of shortcomings with the NCD that make its application to malware classification difficult. While the theoretical underpinnings of NCD say it will behave like a metric if certain conditions are met, it is often difficult to meet them in practice [9]. The nature of how compression algorithms work also causes problems for larger input sequences [5]. Most critically though, the computation time for the NCD is significant. This has limited its application to malware datasets of 10,000 samples or less[4]. We resolve the runtime and metric issues of NCD with the new Lempel-Ziv Jaccard Distance (LZJD), which is a valid distance metric and computationally efficient to use in practice. We perform extensive validation of our new technique by using

We would like to thank Mark McLean and the Laboratory for Physical Sciences for providing access to data and compute resources used in this work. A special thanks to Luke McDowell for reading several drafts and his helpful feedback.

ACM 978-1-4503-4887-4/17/08...$15.00
http://dx.doi.org/10.1145/3097983.3098111

multiple datasets (with over 500,000 files), with different byte representations, for both malware detection and family classification, and for both Microsoft and Android malware. In contrast, most works in malware classification use only one dataset (often 40,000 samples or less), choose one representation, and for one operating system [e.g. 18, 24, 26, 29].

The remainder of our paper is organized as follows. We will review the definition of NCD in section 2, and then introduce our new distance metric in section 3. Focused on malware classification, we provide several experiments in section 4 that show our new distance to be more accurate and orders of magnitude faster to apply in practice. Given the accuracy advantage we observe with our new LZJD metric, we analyze two theoretical differences in behavior of NCD and LZJD in section 5. Our conclusions are then presented in section 6.

2 NORMALIZED COMPRESSION DISTANCE

The inspiration for the NCD comes from Kolmogorov complexity. Given some arbitrary sequence x and its length $|x|$, the Kolmogorov complexity function $K(x)$ will return the length of the shortest possible program that outputs x as a result of its execution. Similarly, the conditional Kolmogorov complexity function $K(x|y)$ will return the length of the shortest possible program that outputs x, given y as an input to the function that it may use. Intuitively, the Kolmogorov functions capture notions of compression, entropy, and their relationships. The closer $|x|$ and $K(x)$ are the more random or uncompressable the sequence x must be. Using these notions, Li et al. [19] define the Normalized Information Distance (NID) (1), which returns a distance in the range $[0, 1]$.

$$\mathrm{NID}(x, y) = \frac{\max\left(K(x|y), K(y|x)\right)}{\max\left(K(x), K(y)\right)} \tag{1}$$

Intuitively, given two items that are near duplicates, the second can be represented as a small set of changes from the first, which would result in a small increase in compressed size (and thus a small distance). Given two inputs that are purely random, and do not have any overlap, one gives us no information about the other. Thus the sizes will remain large, and the numerator will become equal to the denominator (as $K(x|y) = K(x)$ if y gives no information about x), resulting in the maximal distance. The NID is a valid distance metric, in the sense that for any possible inputs x, y, and z, the following properties of metrics hold:

- $d(x, y) = 0$ if and only if $x = y$. (identity)
- $d(x, y) = d(y, x)$ (symmetry)
- $d(x, y) + d(y, z) \geq d(x, z)$ (triangle inequality)

Unfortunately, since $K(\cdot)$ and $K(\cdot|\cdot)$ are uncomputable functions, the NID cannot be used in practice. Given this issue, Li et al. [19] proposed to approximate the function $K(\cdot)$ using any compression algorithm. Defining a new function $C(x)$, which returns the compressed length of x in bytes, we get the NCD distance (2), where $C(xy)$ indicates the compressed size of sequences x and y concatenated together.

$$\mathrm{NCD}(x, y) = \frac{C\left(xy\right) - \min\left(C(x), C(y)\right)}{\max\left(C(x), C(y)\right)} \tag{2}$$

The quality of this approximation to NID depends on the compression algorithm used for $C(\cdot)$, where a better compression algorithm will result in better accuracy. For malware analysis, it has generally been found that LZMA [20] and similar compression algorithms tend to work best [2, 5]. Regardless of the compression algorithm used, in practice the NCD is not a true distance metric, since from time to time all three properties listed above may be violated. Li et al. [19] realized this and also introduced the concept of a *normal compressor*, and showed that the NCD will behave like a distance metric so long as the compressor used maintains certain normalcy properties. These properties are not intrinsic to any compression algorithm, but are a function of the compression algorithm and the input given. For this reason, many have found that empirically the normal compressor properties do not hold in practice [5, 9]. It is even the case that NCD often returns values larger than the theoretical maximum distance of one [10]. A compounding issue is that the NCD is computationally expensive. While the values $C(x)$ and $C(y)$ can be computed once for each datapoint, the conjoined term $C(xy)$ cannot be pre-computed, and is the most computationally demanding of the terms in (2). This has made it difficult to apply NCD to larger datasets.

Despite these issues NCD has been quite popular for many domains, including classification and clustering of EEG signals, pose estimation, text datasets, and more [16, 17]. The use of compression distances also has strong ties to Machine Learning, where compression distances can be seen as a new feature space[25] and the concept of compression can be used for learning bounds[13, 14]. Numerous works have proposed modifications of the terms in NCD, but it has been found that most of these changes will result in equivalent orderings and only change the normalizing terms [25]. Given the wide success of NCD, we seek to address its major issues of computational overhead and lack of metric properties.

3 LEMPEL-ZIV JACCARD DISTANCE

Inspired by the use of compression in NCD, we develop a new distance metric called the Lempel-Ziv Jaccard Distance (LZJD). We base this new distance on two insights about the use of NCD, which allow us to simplify the process as a whole. First, that the most accurate compression algorithms for NCD, such as LZMA, make use of the Lempel-Ziv (LZ) technique for creating a compression dictionary of previously seen sub-sequences[31, 32]. Second, that we do not care about the actual compressed output of any compression algorithm when computing NCD. Compression is merely a means to the end goal of measuring similarity or distance between two objects.

This second insight allows us to ignore the many technical details of LZMA used to efficiently represent the encoding, bookkeeping needed for decoding, block sizes for efficiency, and any additional steps required for effective compression. Instead we can focus on just the act of obtaining a LZ dictionary. Thus we use a simplified version of the LZ77 [31] to get a set of sub-sequences, as shown in Algorithm 1, which defines the LZSet method to convert a byte sequence into a set of byte sub-sequences. This method works by building a set of previously seen sequences. The set starts out empty, and a pointer starts at the beginning of the file looking for a sub-sequence of length one. If the pointer is looking at a

Algorithm 1 Simplified Lempel-Ziv Set

```
1:  procedure LZSet(Byte sequence $b$)
2:      $s \leftarrow \emptyset$
3:      $start \leftarrow 0$
4:      $end \leftarrow 1$
5:      while $end < |b|$ do
6:          $b_s \leftarrow b[start : end]$
7:          if $b_s \notin s$ then
8:              $s \leftarrow s \cup \{b_s\}$
9:              $start \leftarrow end$
10:         end if
11:         $end \leftarrow end + 1$
12:     end while
13:     return $s$
14: end procedure
```

sub-sequence that has been seen before, we leave it in place and increase the desired sub-sequence length by one. If the pointer is at a sub-sequence that has not been seen before, it is added to the set. Then the pointer is moved to the next position after the sub-sequence, and the desired sub-sequence length reset to one.

Once we have the LZSet method, we can turn any sequence of bytes into a set of sub-sequences. The similarity between two sets can then be measured using the familiar Jaccard similarity,

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{3}$$

The Jaccard similarity is the cardinality of the intersection of two sets divided by the cardinality of their union (3). The Jaccard Distance, which is a valid distance metric, is simply $D_J(A, B) = 1 - J(A, B)$. We can then combine the Jaccard similarity and the LZSet algorithm to produce our new Lempel-Ziv Jaccard Distance (LZJD),

$$\text{LZJD}(x, y) = 1 - J(\text{LZSet}(x), \text{LZSet}(y)) \tag{4}$$

Since the LZSet method consistently maps any byte sequence to a set, and the Jaccard distance is a valid distance metric, then the LZJD is also a valid metric. Because we are unconcerned with the extra work of performing full compression, the LZJD turns out to be faster to compute in practice.

3.1 Storage and Compute Efficient LZJD via Min Hashing

The set of sub-sequences extracted by LZJD requires memory proportional to the size of the input strings. This makes it impractical to store all the resulting sets in memory for every data point, forcing us to do redundant computations. While this can be partially alleviated through caching schemes, we can instead exploit one of many approximation algorithms for the Jaccard similarity. In this way we can compute an approximate LZJD with high accuracy, throughput, and minimal memory usage. In particular, we use min-hashing to create compact representations of the input strings, and the same min-hashing lets us approximate the distances between sets of sub-sequences.

Let $h(a)$ be a hash function that returns an integer given some object a, and $h_{min}(A) = \min_{a \in A} h(a)$ returns the minimum hash value over every object a in a set A. Then it is known that [6]

$$P(h_{min}(A) = h_{min}(B)) = J(A, B)$$

That is, for two sets A and B, the probability that the min hash value of A and B are the same is equal to the Jaccard similarity of the sets. This observation could be used to approximate the Jaccard similarity by collecting multiple hash values for different hash functions. Instead, we can be more computationally efficient by selecting the minimum k hashes from the set [7]. Using $h^n_{min}(A)$ for the n'th smallest hash value from the set A, we then get

$$J(A, B) \approx J\left(\bigcup_{j=1}^{k} h^j_{min}(A), \bigcup_{j=1}^{k} h^j_{min}(B)\right)$$

We can use this approximation to reduce time and memory requirements for computing LZJD. The error of this approximation is probabilistically bounded above by $O(1/\sqrt{k})$ if the minimum k hash values are used. We can then use $k = 1024$ to reduce the approximation error to around 3%. Each dictionary d (derived from LZSet applied to some input string) is mapped to a new dictionary d^k which contains the k smallest hash values. This means that any string we wish to use as input to LZJD will take on the order of 4KB to store in memory, which is much smaller than the multiple megabytes binary files may require. This gives us the following procedure for a faster and more memory efficient approximation:

(1) Convert byte sequence B_i to sub-sequence set C_i using Algorithm 1
(2) Convert C_i to a set of integers, via some hash function $h(\cdot)$
(3) Obtain integer set C_i^k by keeping only the k smallest values from the set
(4) Approximate LZJD(B_i, B_j) as $\approx 1 - J(C_i^k, C_j^k)$

We will denote our min-hash approximation of LZJD as LZJD$_h$. By reducing the memory use to just a fixed 4KB, we are able to greatly reduce both the storage and compute requirements for our approach. We also note that since LZJD is a metric, so is the approximation LZJD$_h$. To see this, note that the min-hash set used is a fixed function converting one set into another set. The approximate distance is then computed using the Jaccard similarity, which is a metric, and so LZJD$_h$ is also a metric.

4 EXPERIMENTS

Having defined the LZJD distance, we describe a number of experiments which show that LZJD is competitive with NCD in terms of quality of results, while having superior run-time characteristics. In all experiments we will apply NCD and LZJD to the raw byte contents of a binary as our features, unless stated otherwise. We will generally use the k-Nearest Neighbor algorithm (k-NN) [11] in these experiments to perform classification. This is a well-known algorithm that is intuitive, and a good fit for our distance metrics. Given a query point q, we find the k training data-points closest to q. The label we assign the query is then the majority label for the k nearest neighbors, where ties are broken arbitrarily. For all experiments, we do not perform a search for the most accurate value of k as it is time intensive to run experiments for NCD and normal LZJD, and results were generally insensitive to changes in k.

For our NCD implementation, we use the XZ compression algorithm. XZ is a container for LZMA and LZMA2 compression, and has an additional compression filter specifically for binary code data. This makes it an especially good fit for our goal of malware classification from raw binaries. The per-file compression sizes were cached after first use to avoid redundant computation.

We implemented LZJD and LZJD$_h$ in Java with JSAT[21], without any significant attempt at performance optimization. For computing the min-hash sets we used the MD5 hash function. All results were run on a single machine with 64 CPU cores and 2TB of RAM, and we report the time taken as the time spent on all CPU cores added together (which is reported by the unix *time* command). This form of measurement is valid for this task as the k-NN and distance computations can be done independently, meaning there is minimal communication overhead. This is done in part to avoid differences in load balancing, where differing file size lengths and compressibility can result in uneven distribution of workloads. In terms of performance comparisons, our setup gives the maximal advantage to NCD (which is using an optimized implementation of XZ compression), where our LZJD implementation is naive and unoptimized. LZJD's run-time could be further enhanced by using a disk based cache of the LZSet instead of re-computing for each distance comparison. Both LZJD and LZJD$_h$ could enjoy further speedup by the use of a rolling hash function to compute the LZSet. Given the time intensive nature of our experiments, we tested NCD and LZJD on random sub-samples of each dataset. For each method, the maximum subset size was determined by a one week runtime limit on individual runs. When a dataset was sub-sampled, both the training and testing data were sub-sampled. Otherwise the test-set sizes alone would exceed our runtime capacities.

4.1 Microsoft Malware Detection

We first demonstrate the performance of LZJD with the task of malware detection, where we try to distinguish between benign and malicious binaries. While most prior results have restricted the use of NCD to data-sets of 2000 or less samples, we use the much larger data from [22], which divides the data into two different training sets (Group A and Group B) and three different testing sets (Group A, B, and Open Malware). The data in each group is collected in a different manner from a different source, and is used to better estimate the generalization error by minimizing common biases between train and test sets. We will use the Group B training set, as [22] found that models performed best when trained on that group. We will report run-times when using Group B training data, and evaluating against all 3 test sets. The Group B training set has 400,000 unique files, half benign and half malicious. Combined with the over 220,000 files in the test sets this represents 275.5GB of data. Our results represent a data-set two orders of magnitude larger than what NCD has ever, to our knowledge, been used with before.

For this experiment we use balanced accuracy [8] and Area Under the Curve (AUC) to evaluate on the test sets for this task. For k-NN, we choose $k = 9$ over smaller values of k so that we can more accurately measure the AUC, which is not well defined for $k = 1$.

In Figure 1, we can see the balanced accuracy of all three distances run on training subsamplings of varying sizes. We use the balanced accuracy where each class receives equal total weight in the calculation, to ease comparisons across datasets. As can be seen, the LZJD metric has higher accuracy than NCD across all sample sizes and comparable AUC. For the Group B test set (Figure 1b), we can see that all distances have increasing accuracy as the sample size increases. This is to be expected, as k-NN theory indicates that the error rate approaches the Bayes optimal error rate as the training set size increases. While the accuracy of NCD closes the gap with LZJD with larger samples, its computational cost means it cannot reach the same accuracies as LZJD$_h$. The Group A test set (Figure 1a) does not have quite the same behavior, since its data comes from a different distribution, but we still see the same overall trend: LZJD obtains better accuracies and LZJD$_h$ allows us to use more data.

We point out that LZJD$_h$ has no significant impact on the classification accuracy of our approach. In terms of AUC, LZJD and NCD tend to go back and forth, with relatively minor differences. The exception being early on the Group B test set, where NCD performs much worse than LZJD in all respects.

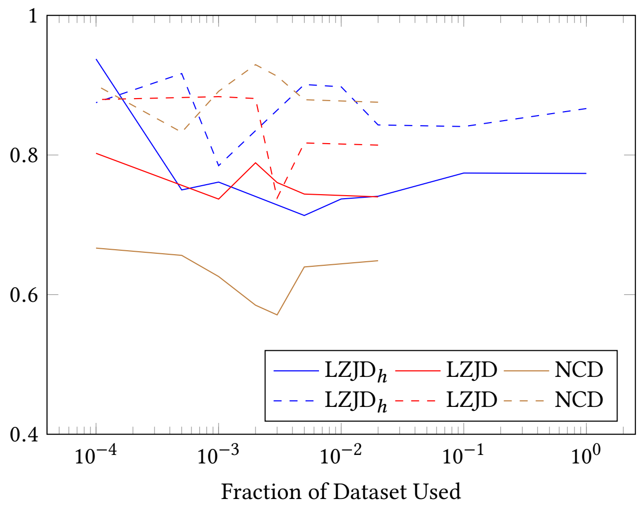
Table 1. **Balanced accuracy results for distance metrics on all three test sets. Results given for using 2% of training (and test) data and 100% of data. Includes best results from [22] in last column**

	2% of data			100% of data	
Test Set	NCD	LZJD	LZJD$_h$	LZJD$_h$	Byte 6-grams
Group A	64.9	74.0	74.1	77.4	87.3
Group B	76.8	81.8	79.3	85.9	94.5
Open Malware	21.9	64.1	59.5	67.8	81.1

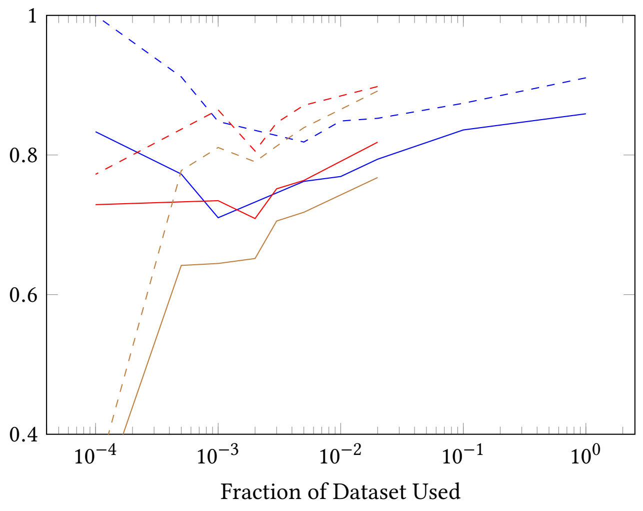
The accuracies at 2% and 100% of the data from Figure 1 are shown in Table 1. In every case LZJD performs better than NCD, and performance improves as more data is used. While the byte n-gram approach used in previous work [22] performs better than LZJD, LZJD allows a wider variety of uses in clustering, similarity search and requires less effort to apply[1]. Because LZJD is a distance metric, we can apply it to various existing techniques, but such questions are beyond the scope of this paper. The increased practicality will also allow investigating improved classification methods, such as Radial Basis Function networks, that were too expensive with NCD.

The total single-threaded run-time for this evaluation is presented in Figure 2, again as a function of how much of the corpus was used. At 0.1% of the corpus, LZJD$_h$ is 216 times faster than NCD to perform the classification. It is also clear that LZJD$_h$ has a lower slope than NCD, and by 2% of the corpus, LZJD$_h$ was 3,572 times faster than NCD. This shows that LZJD$_h$ is several orders of magnitude faster than NCD, making it practical for larger datasets. In addition, creating the min-hash set of the data took 90.2% of the computational time. For a system that will classify new items

[1]The byte n-graming approach is computationally demanding. Using *all* 64-cores of the same server it required multiple days of out-of-core processing for the same data, and needed over 3TB of scratch space. This makes scaling problematic, and its software took 6 months of engineering to be this efficient. In contrast, LZJD$_h$ was initially written in a few hours.



(a) **Results on Group A test set**



(b) **Results on Group B test set**

Figure 1. **Balanced Accuracy and AUC on the y axis, presented for the Group A and Group B test datasets. The same legend applies to each plot. Solid lines are for balanced accuracy, dashed lines are for AUC. Values with a smaller fraction of training data had higher variance, but runs were not repeated due to computational burden.**

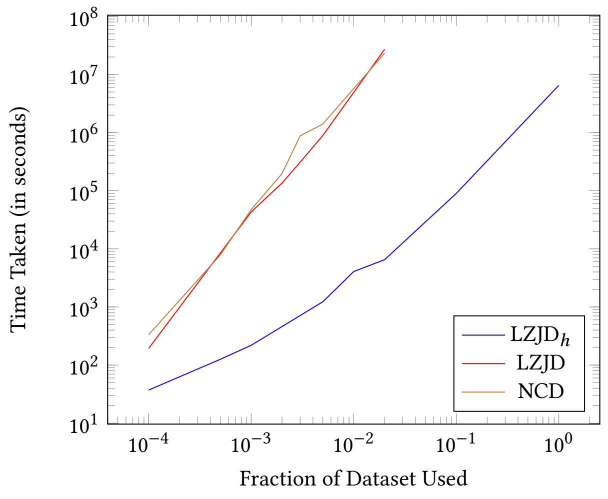


Figure 2. **Time taken to perform 9-NN classification on Group A, B, and Open Malware test sets with the Group B training set.**

against an existing database, the min-hashing is a one-time cost, making deployment of LZJD$_h$ more realistic as well.

4.2 Malware Family Classification

In our second set of experiments we consider malware family classification, where we are given known malware and need to determine what family of malware a sample belongs to. We will evaluate this with two data sets, one for Windows binaries and one for Android applications. For each dataset the distribution of families is skewed, so we use k=1 for k-NN. Larger values of k tended to reduce the resulting accuracy since most samples belong to only a small set of malware families. Each dataset is evaluated using 10-fold cross validation with balanced accuracy as the target metric. Due to the computational time required for NCD and LZJD, we only evaluate them on 10% of each dataset. For LZJD$_h$ we evaluate on 10% and 100% of each dataset, since it is much faster than NCD or LZJD.

Our Windows dataset was provided by Microsoft for a 2015 Kaggle competition [1]. This dataset contains 9 malware families in 10,868 training files at 50.8GB in size. We evaluate two different feature options that were provided. First, similar to subsection 4.1, we use the raw byte contents of the files[2] as the inputs to our distance metrics, and refer to it as "Kaggle Bytes". Microsoft also provided the disassembled versions of each file using industry standard software. These disassembled versions contain ASCII representations of not only the assembly from binary code (.text sections), but ASCII representations of all other sections of the binary as well. This includes additional human-readable annotations when possible (such as resolving import names and function signatures). We include the raw disassembled versions of each file as another input feature type to show that LZJD is not specific to binary data. The disassembled version of the dataset takes up 147GB of disk space, and we refer to it as "Kaggle ASM".

For our Android APK malware we use the Drebin dataset [3], but remove any malware family that had less than 40 samples[3]. This leaves us with 20 malware families and 4664 files with a collective size of 6.4 GB. Android programs are referred to as *Android application packages* (APKs). APKs are in fact zip files, which may include some level of compression, of the Dalvik bytecode and other application resources. We note that the default Android toolkit often

[2]Microsoft provided the raw contents, but with the file header removed so that one could not accidentally run the malware samples. Since these headers are not recoverable, we used these header-less versions.
[3]Many of the malware families had less samples than cross-validation folds, which would have made evaluation difficult

applies little or no compression when creating the zip files. Compressed input can be a challenge for NCD, since it depends on the input objects being (even more) compressible in order to work. For this reason we evaluate the dataset in two ways, one with the raw APK files and one with the APKs uncompressed and its contents combined into a single tar archive (i.e., no compression). We refer to these versions as "Drebin APK" and "Drebin TAR" respectively. Drebin TAR is 8.6 GB uncompressed. We note that three of the files could not be unzipped due to a malformed APK, and these three files were removed.

Table 2. **Balanced accuracy results on each data and feature set. Evaluated with 10-fold CV, standard deviation in parenthesis.**

	10% of data			100% of data
Dataset	NCD (%)	LZJD (%)	LZJD$_h$ (%)	LZJD$_h$ (%)
Kaggle Bytes	58.1 (3.6)	98.2 (1.2)	94.4 (5.0)	97.6 (1.5)
Kaggle ASM	71.8 (6.1)	92.9 (4.6)	95.6 (4.1)	97.1 (2.0)
Drebin APK	67.2 (7.8)	81.4 (5.5)	80.5 (5.8)	80.8 (2.6)
Drebin TAR	81.0 (6.5)	85.0 (6.6)	82.0 (7.0)	87.2 (2.8)

The results of running 1-NN on these datasets are given in Table 2, where we can see two major trends. First, the NCD distance performs significantly worse than both variants of LZJD. Second, LZJD$_h$ typically performs slightly worse than LZJD when using only 10% of the data.

The performance difference between LZJD and LZJD$_h$ is always within a standard deviation, with two cases where LZJD performs better and two where it performs worse. Additionally, being able to use 100% of the data with LZJD$_h$ naturally reduces the variance of the error. We suspect the slightly reduced performance of LZJD$_h$ is caused by errors when the nearest neighbor with the correct label, and a second nearest neighbor with an incorrect label, are almost equidistant from the query. Because some of the malware families in each corpus are related to each other, this is a source of errors even when not using the approximated distances. This situation could easily change the nearest neighbor due to the 3% approximation error of LZJD$_h$, which can cause a significant change in output since we only consider the nearest neighbor. When comparing accuracy to NCD, we will refer to the *worse* accuracy result between LZJD and LZJD$_h$ as just "LZJD" for brevity.

For the Kaggle data set NCD's performance using the raw binaries and assembly is better than the random guessing rate of 11%, but is still 21 to 36 whole percentage points behind LZJD. This dramatic drop in accuracy would be an indication that the compression is simply not effective when trying to distinguish the finer details between malware families. We see evidence of this when examining the classification errors made by NCD. For example, on the Kaggle Bytes dataset, NCD could not distinguish between Kelihos version 3 and version 1, which resulted in errors both ways. For Kaggle ASM such errors were not as prevalent, but NCD still had difficulty with the malware families that had few samples being miss-classified as other, larger, families. We note that while NCD gains over 13 percentage points by using the more verbose disassembled Kaggle dataset, LZJD is relatively unaffected by the change, especially when run on 100% of the data. This suggests that NCD is more sensitive to the data's representation than desired, and that LZJD possesses a greater invariance to the data encoding.

On the Drebin datasets, NCD performs better, but still trails LZJD in accuracy. When tested with the uncompressed Drebin TAR data, NCD is within a percentage point of LZJD's accuracy. But NCD is over 6 points behind when we consider that we can use all of the data for LZJD$_h$. Consistent with prior results, moving from Drebin TAR to the compressed Drebin APK causes the performance of NCD to trail LZJD by nearly 14 percentage points, making it considerably worse than LZJD in all cases. This also provides additional evidence that LZJD is more robust in the face of higher entropy data, as the drop in accuracy from Drebin TAR to Drebin APK is not as dramatic, losing only 6.4 points when the whole dataset is used.

Table 3. **Total evaluation time for each method 10-fold CV. Time presented in seconds.**

	10% of data			100% of data
Dataset	NCD	LZJD	LZJD$_h$	LZJD$_h$
Kaggle Bytes	1.20×10^7	2.95×10^6	1.22×10^3	1.73×10^4
Kaggle ASM	2.83×10^7	1.16×10^7	4.94×10^3	4.85×10^4
Drebin APK	1.79×10^5	4.22×10^5	7.41×10^2	7.17×10^3
Drebin TAR	3.59×10^5	4.41×10^5	8.33×10^2	7.65×10^3

In Table 3 the CPU time on each dataset is given in seconds. Our un-optimized implementation of LZJD is sometimes faster and sometimes slower than NCD, depending on the dataset. But our ability to apply min-hashing makes LZJD$_h$ orders of magnitude faster. In every case, LZJD$_h$ can perform 10 fold CV on all the data faster than NCD can be applied to one tenth the amount of data. The speedup of LZJD$_h$ ranges from 241 to 9,836 times that of NCD, nearly four orders of magnitude, with the larger speedups being obtained on the larger Kaggle datasets. This runtime improvement greatly extends the utility of LZJD over NCD, representing the difference between 327 CPU days for the Kaggle ASM dataset to under two CPU hours.

5 DIFFERENCES BETWEEN NCD AND LZJD

We have shown that the LZJD distance, for byte-based malware classification, has superior accuracy to NCD. Combined with min-hashing, it is also orders of magnitude faster while retaining the desirable properties of being a metric. Superficially, it may seem surprising that the NCD and LZJD have meaningfully different results, given that in practice they both use the Lempel-Ziv compression scheme as a core component. Here we present two ways in which the behavior of these two distances are different.

5.1 High Entropy Files

One important difference between LZJD and NCD is the value returned when faced with a compressed, encrypted or otherwise random looking file. Such processing results in a high byte entropy, and is a common scenario for malware classification. Malware will often encrypt or compress portions of itself to obfuscate its true intentions and reduce its footprint to avoid detection. This is referred to as packing, and over 90% of Microsoft malware uses some form of packing [30]. We will first discuss how NCD and LZJD differ in this scenario, and then explain how the impact can be seen on our results.

When NCD encounters high entropy regions that cannot be compressed, these areas will become additive constants to the compressed size of each file, and neither file will have information that can help compress the high entropy areas of the other (assuming the high entropy regions in the two files are not near duplicates). This will result in an increase in their distance. When two different files cannot be compressed, the maximal possible distance (of 1.0) is returned. We emphasize that because a single high entropy file will not help compress or be further compressed by any other file (including ones that are not compressed), high entropy files will become maximally far and equidistant from all other data points in practice.

For LZJD, we build the LZ dictionary which, in the presence of non-compressible randomness, will begin collecting all possible shortest length sequences into the set. This is because each sequence is equally likely to be observed, and corresponds to the worst case scenario for LZ compression. This will generate a dictionary set with a maximal number of elements. Since we use the Jaccard distance between these sets, two different files composed of random sequences will likely have a near-zero distance from each other, which is the opposite behavior of NCD. In contrast, when computing the distance between a high and low entropy file, LZJD's distance will become larger (but not maximal), the value of which will depend on the ratio of small to large sub-sequences in the non-compressed sequence. The lower entropy a sequence is, the easier it is to accumulate longer sub-strings, thus increasing the distance to higher entropy sequences. Thus, LZJD will tend to compute very small distances between two compressed files, but not between a compressed file and a non-compressed file.

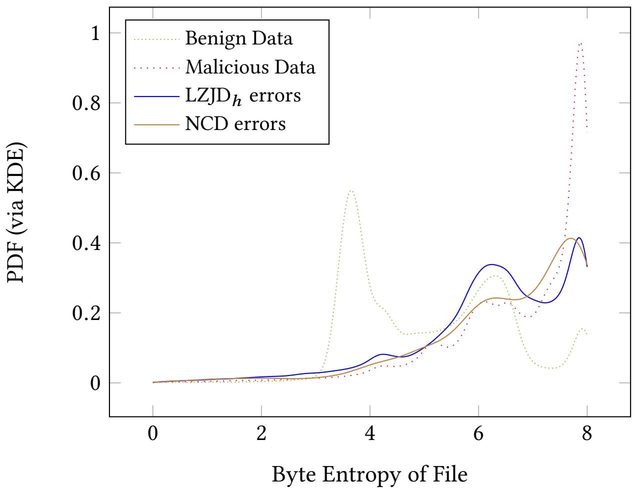


Figure 3. **Distribution of file entropy for datapoints that were misclassified (solid lines) in the test set, and benign vs malicious subsets (dotted lines). Results with respect to all three test sets from subsection 4.1.**

The impact of these entropy-related effects is clearly evidenced by our results with the Android Drebin malware, where the only difference between Drebin TAR and Drebin APK is compression (at a ratio of 1.34). NCD has a nearly 14 point drop in performance on the compressed Drebin APK dataset compared to the TAR version. This is a significant performance gap for what amounts to two versions of the same data. LZJD is considerably more robust to the change, with only a 2-6 point drop in performance. Because the only difference between these two versions is compression, we can attribute the better performance of LZJD to the manner in which it handles higher entropy (caused by compression) data.

The better performance of LZJD can also be seen by looking at the entropy of files which are misclassified, as shown in Figure 3 for the malware detection results from subsection 4.1. The Probability Density Function (PDF)[4] is shown for the test data, and compared against the PDF of the misclassified test data for each metric. Recall that the PDF is normalized to integrate to one, and so we are comparing the shapes of PDF curves, not their magnitude.

Since NCD has difficulty with high entropy files, it is more likely to misclassify those files as compared to files of lower entropy. This results in the highest proportion of errors near the rightmost end of the distribution. The PDF for NCD thus also increases with entropy, as NCD(a,b) approaches 1 when either a or b are of sufficiently high entropy.

Because many, if not most, high entropy binaries ($\geq$ 7 entropy) are malicious, it is easier for LZJD to properly classify many of these files. Under LZJD such high entropy files will have a small distance from each other, but a larger distance compared to the lower entropy files. The majority of neighbors will then be malicious based on the population density, and give the label of malicious (which is usually correct). Errors will then come from high entropy benign files, which can be seen in Figure 3. The test PDF errors for LZJD$_h$ in the ≥ 6 entropy range matches the shape of the benign data (dashed green line), indicating that LZJD's errors with high entropy files are failures in separating the minority of benign packed files.

The behavior of LZJD in this case results in improved accuracy, but could be seen in both a positive and negative light. On one hand, two files that are comprised of different random bytes are intrinsically different and have no overlapping similarity — thus making it appropriate that they receive the maximal possible distances from each other. At the same time, both files are similar in the fact that they appear random and incompressible, making it appropriate to place them closer together distance wise. While there may be other datasets and domains where the way LZJD handles high entropy sequences is undesirable, it has clearly resulted in improved accuracy for malware classification and provides more meaningful nearest neighbors compared to NCD.

5.2 Sensitivity to Sequence Length Repetition

While the behavior of LZJD with high entropy sequences can be argued in either direction, there is one way in which the theoretical behavior of LZJD does *not* match our intuition of how a distance metric should behave. This is when we are given two sequences where one sequence is a repetition of the other. We have no reason to suspect this scenario occurs in our data or constitutes a significant impact on our data and results, but find the exercise informative

[4]PDF is estimated with a Kernel Density Estimator using a Gaussian kernel, bandwidth selected using Silverman's method.

to the differences between NCD and LZJD. NCD has the desired theoretical behavior in this scenario, as we will show below, but does not deliver upon this behavior in practice. LZJD lands in the middle ground, where its behavior is not what we would desire but is better than NCD in practice. Devising ways to rectify this theoretical shortcoming may be a way to improve LZJD as a whole in future work.

Let us assume we have a sequence of bytes α, and represent the duplication of a sequence n times as $\alpha^{(n)}$, where $\alpha^{(1)} = \alpha$. Intuitively, we would desire the distance between $\alpha^{(n)}$ and α to be small, as they are intrinsically similar. There is effectively no true difference in content, only in repetition of the same data.

Using the theoretical Kolmogorov complexity $K(\cdot)$, NCD does match this intuition. We would expect that $\forall n > 1, \text{NCD}(\alpha, \alpha^{(n)}) < \epsilon$. This is because, for most cases, $K(\alpha^{(n)}) \approx K(\alpha) + \log(n)$, as we can generally represent the duplication of the string α with a minimal amount of additional programming that repeats the original sequence, and simply need to know how many times to repeat that sequence (the value of which takes a logarithmic number of bits to represent)[5]. Then applying this to NCD we expect to get

$$\begin{aligned}\text{NCD}\left(\alpha, \alpha^{(n)}\right) &= \frac{K(\alpha\alpha^{(n)}) - \min(K(\alpha), K(\alpha^{(n)}))}{\max(K(\alpha), K(\alpha^{(n)}))}\\ &= \frac{K(\alpha^{(n+1)}) - K(\alpha)}{K(\alpha^{(n)})}\\ &\approx \frac{K(\alpha) + \log(n+1) - K(\alpha)}{K(\alpha) + \log(n)}\\ &= \frac{\log(n+1)}{K(\alpha) + \log(n)}\end{aligned}$$

It is easy to see that $\lim_{K(\alpha)\to\infty} \frac{\log(n+1)}{K(\alpha)+\log(n)} = 0$. Thus, the NCD between two files that are of widely different lengths, that differ only in how many times the same sequence α is repeated, should result in a small distance. The distance will increase slowly as the repetition n increases, due to the log terms. This behavior matches our intuition that α and $\alpha^{(n)}$ should have a small distance and are intrinsically similar.

This is the result with the theoretical Kolmogorov complexity $K(\cdot)$. In practice, we have to use some compression algorithm $C(\cdot)$, in which case for large inputs α, $C(\alpha^{(n)}) \approx nC(\alpha)$. This is due to the fact that compression algorithms (like LZMA) usually use a window size for compression, and for larger sequences the window size will be smaller than the length of the files. By the time the window reaches into the second sequence, information from the first is mostly out of the window. This means little information about one sequence is used for the compression of the second [5]. Using this we would instead get the result $\text{NCD}\left(\alpha, \alpha^{(n)}\right) \approx \frac{(n+1)C(\alpha)-C(\alpha)}{nC(\alpha)} = 1$. This is the opposite of the theoretical behavior we would expect.

We have now shown that NCD has a theoretically desirable behavior, but in practice has the worst possible behavior. We now show that LZJD's behavior (which does not have a disconnect between theoretical and real performance) falls between these two opposing ends. It does not match our intuition for what good behavior is, but avoids marking sequences as equidistantly maximally far away. By the definition used in Algorithm 1, it is easy to see that the dictionary is monotonically increasing in size, irrespective of the amount of repetition in the source sequence. For a minimal increase in the dictionary size (and thus, minimal change in distance), we want a sequence of all the same character. This means the size of the next sub-string added to the dictionary will always increase by 1. If the length of a string α is $|\alpha|$, and k is the minimal number of items added to a set, we get $|\alpha| = \sum_{i=1}^{k} i$. Solving for k reveals that we get at least $\frac{1}{2}(\sqrt{8|\alpha|+1} - 1)$ sub-strings. Applying this to the Jaccard distance we obtain a lower bound on the distance

$$\begin{aligned}\text{LZJD}\left(\alpha, \alpha^{(n)}\right) &= 1 - \frac{|\text{LZSet}(\alpha) \cap \text{LZSet}(\alpha^{(n)})|}{|\text{LZSet}(\alpha) \cup \text{LZSet}(\alpha^{(n)})|}\\ &= 1 - \frac{|\text{LZSet}(\alpha)|}{|\text{LZSet}(\alpha^{(n)})|}\\ &\geq 1 - \frac{\sqrt{8|\alpha|+1} - 1}{\sqrt{8n|\alpha|+1} - 1}\end{aligned}$$

By taking the limit $\lim_{|\alpha|\to\infty} 1 - \frac{\sqrt{8|\alpha|+1}-1}{\sqrt{8n|\alpha|+1}-1} = 1 - \frac{1}{\sqrt{n}}$, we see that the distance will start off near 0.3 for just one repetition, and grow relatively quickly as the repetition is increased. Repeating this with assumptions on a faster growth rate of the dictionary size increases the value of the limit and increase the distance between $\alpha^{(n)}$ and α.

This is in many ways counter to our intuition that α and $\alpha^{(n)}$ are intrinsically similar, and so should receive a small distance. The LZJD distance also grows more rapidly with repetition than it does in the case of NCD. This is worse than NCD in theory, but better in practice since $C(\cdot)$ is often not a good enough approximation of $K(\cdot)$ when dealing with large sequences like binaries. More succinctly, with regards to this scenario, LZJD is worse than NCD in theory, but better than NCD in practice. This is because the theoretical behavior of NCD is unobtainable. Fortunately the scenario of duplicated sequences does not seem to occur in practice, but the results are informative to the behaviors of these distances. Understanding and rectifying this theoretical weakness this may allow us to devise improvements to LZJD in future work.

6 CONCLUSIONS AND FUTURE WORK

We have introduced the novel LZJD distance as an alternative to NCD when dealing with large byte sequences, particularly for malware classification. LZJD has comparable or better accuracy than NCD, when using raw bytes for Microsoft Windows files and Android applications, ASCII disassembly, and moderately compressed Android APKs. We have also shown two theoretical differences in behavior between these distances, despite similar inspiration. Our new distance allows the use of min-hashing to obtain speed improvements of up to four orders of magnitude. This has allowed us to apply LZJD to datasets orders of magnitude larger than were previously possible with NCD. This comes with improved accuracy compared to NCD, yet retains the desirable distance metric properties that NCD lacks.

[5]This is not true in all cases, so we avoid absolute statements. But for most files this will be approximately correct

In this work we have focused primarily on computing distances between raw byte sequences for malware classification. In future work, the differences between NCD and LZJD should be explored in other domains. While they share a critical component of using LZ compression, it is not obvious that LZJD will be superior in all domains. Similarly, our sequences are fairly long, which is an area of weakness of NCD. The differences in performance and accuracy may be smaller for tasks involving shorter sequences. Finally, we note that a benefit of our approach is that it can exploit the rich existing literature in approximating the Jaccard similarity between sets. Combining our work with other approximations such as Locality-Sensitive-Hashing [12], and balancing a trade-off between approximation and exact computation, may enable even more applications of LZJD that are not possible for NCD. That LZJD is a true distance, and is fast to compute, also enables new follow up research that would not have been practical for NCD. This includes evaluations in clustering, large scale similarity search, visualization, and other classification algorithms that require only a distance metric between points.

REFERENCES

[1] 2015. Microsoft Malware Classification Challenge (BIG 2015). (2015). https://www.kaggle.com/c/malware-classification/

[2] Nadia Alshahwan, Earl T Barr, David Clark, and George Danezis. 2015. Detecting Malware with Information Complexity. (2 2015). http://arxiv.org/abs/1502.07661

[3] Daniel Arp, Michael Spreitzenbarth, Hubner Malte, Hugo Gascon, and Konrad Rieck. 2014. Drebin: Effective and Explainable Detection of Android Malware in Your Pocket. *Symposium on Network and Distributed System Security (NDSS)* February (2014), 23–26. https://doi.org/10.14722/ndss.2014.23247

[4] Michael Bailey, Jon Oberheide, Jon Andersen, Z Morley Mao, Farnam Jahanian, and Jose Nazario. 2007. Automated Classification and Analysis of Internet Malware. In *Proceedings of the 10th International Conference on Recent Advances in Intrusion Detection (RAID'07)*. Springer-Verlag, Berlin, Heidelberg, 178–197. http://dl.acm.org/citation.cfm?id=1776434.1776449

[5] Rebecca Schuller Borbely. 2015. On normalized compression distance and large malware. *Journal of Computer Virology and Hacking Techniques* (2015), 1–8. https://doi.org/10.1007/s11416-015-0260-0

[6] Andrei Z Broder. 1997. On the Resemblance and Containment of Documents. In *Proceedings of the Compression and Complexity of Sequences 1997 (SEQUENCES '97)*. IEEE Computer Society, Washington, DC, USA, 21–29. http://dl.acm.org/citation.cfm?id=829502.830043

[7] Andrei Z Broder, Moses Charikar, Alan M Frieze, and Michael Mitzenmacher. 1998. Min-wise Independent Permutations (Extended Abstract). In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing (STOC '98)*. ACM, New York, NY, USA, 327–336. https://doi.org/10.1145/276698.276781

[8] Kay Henning Brodersen, Cheng Soon Ong, Klaas Enno Stephan, and Joachim M Buhmann. 2010. The Balanced Accuracy and Its Posterior Distribution. In *Proceedings of the 2010 20th International Conference on Pattern Recognition (ICPR '10)*. IEEE Computer Society, Washington, DC, USA, 3121–3124. https://doi.org/10.1109/ICPR.2010.764

[9] Manuel Cebrián, Manuel Alfonseca, Alfonso Ortega, and others. 2005. Common pitfalls using the normalized compression distance: What to watch out for in a compressor. *Communications in Information & Systems* 5, 4 (2005), 367–384.

[10] Manuel Cebrin, Manuel Alfonseca, and Alfonso Ortega. 2007. The Normalized Compression Distance Is Resistant to Noise. *IEEE Transactions on Information Theory* 53, 5 (5 2007), 1895–1900. https://doi.org/10.1109/TIT.2007.894669

[11] T Cover and P Hart. 2006. Nearest Neighbor Pattern Classification. *IEEE Trans. Inf. Theor.* 13, 1 (9 2006), 21–27. https://doi.org/10.1109/TIT.1967.1053964

[12] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry - SCG '04*. ACM Press, New York, New York, USA, 253–262. https://doi.org/10.1145/997817.997857

[13] Ofir David, Shay Moran, and Amir Yehudayoff. 2016. Supervised learning through the lens of compression. In *Advances In Neural Information Processing Systems 29*, D D Lee, U V Luxburg, I Guyon, and R Garnett (Eds.). Curran Associates, Inc., 2784–2792. http://papers.nips.cc/paper/6490-supervised-learning-through-the-lens-of-compression.pdf

[14] Thore Graepel, Ralf Herbrich, and John Shawe-Taylor. 2005. PAC-Bayesian Compression Bounds on the Prediction Error of Learning Algorithms for Classification. *Machine Learning* 59, 1 (2005), 55–76. https://doi.org/10.1007/s10994-005-0462-7

[15] Matthew Hayes, Andrew Walenstein, and Arun Lakhotia. 2008. Evaluation of malware phylogeny modelling systems using automated variant generation. *Journal in Computer Virology* 5, 4 (2008), 335–343. https://doi.org/10.1007/s11416-008-0100-6

[16] Eamonn Keogh, Stefano Lonardi, and Chotirat Ann Ratanamahatana. 2004. Towards Parameter-free Data Mining. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '04)*. ACM, New York, NY, USA, 206–215. https://doi.org/10.1145/1014052.1014077

[17] Eamonn Keogh, Stefano Lonardi, Chotirat Ann Ratanamahatana, Li Wei, Sang-Hee Lee, and John Handley. 2007. Compression-based data mining of sequential data. *Data Mining and Knowledge Discovery* 14, 1 (2007), 99–129. https://doi.org/10.1007/s10618-006-0049-3

[18] J Zico Kolter and Marcus A Maloof. 2006. Learning to Detect and Classify Malicious Executables in the Wild. *Journal of Machine Learning Research* 7 (12 2006), 2721–2744. http://dl.acm.org/citation.cfm?id=1248547.1248646

[19] Ming Li, Xin Chen, Xin Li, Bin Ma, and Paul M.B. Vitanyi. 2004. The Similarity Metric. *IEEE Transactions on Information Theory* 50, 12 (2004), 3250–3264. https://doi.org/10.1109/TIT.2004.838101

[20] Igor Pavlov. 2007. LZMA SDK (Software Development Kit). http://www.7-zip.org/sdk.html. http://www.7-zip.org/sdk.html

[21] Edward Raff. 2017. JSAT: Java Statistical Analysis Tool, a Library for Machine Learning. *Journal of Machine Learning Research* 18, 23 (2017), 1–5. http://jmlr.org/papers/v18/16-131.html

[22] Edward Raff, Richard Zak, Russell Cox, Jared Sylvester, Paul Yacci, Rebecca Ward, Anna Tracy, Mark McLean, and Charles Nicholas. 2016. An investigation of byte n-gram features for malware classification. *Journal of Computer Virology and Hacking Techniques* (9 2016). https://doi.org/10.1007/s11416-016-0283-1

[23] Christian Rossow, Christian J. Dietrich, Chris Grier, Christian Kreibich, Vern Paxson, Norbert Pohlmann, Herbert Bos, and Maarten van Steen. 2012. Prudent Practices for Designing Malware Experiments: Status Quo and Outlook. In *2012 IEEE Symposium on Security and Privacy*. IEEE, 65–79. https://doi.org/10.1109/SP.2012.14

[24] M.G. Schultz, E. Eskin, F. Zadok, and S.J. Stolfo. 2001. Data Mining Methods for Detection of New Malicious Executables. In *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*. IEEE Comput. Soc, 38–49. https://doi.org/10.1109/SECPRI.2001.924286

[25] D Sculley and Carla E Brodley. 2006. Compression and Machine Learning: A New Perspective on Feature Space Vectors. In *Proceedings of the Data Compression Conference (DCC '06)*. IEEE Computer Society, Washington, DC, USA, 332. https://doi.org/10.1109/DCC.2006.13

[26] M. Zubair Shafiq, S. Momina Tabish, Fauzan Mirza, and Muddassar Farooq. 2009. PE-Miner: Mining Structural Information to Detect Malicious Executables in Realtime. In *Recent Advances in Intrusion Detection*. 121–141. https://doi.org/10.1007/978-3-642-04342-0{_}7

[27] Nicholas Tran. 2007. The normalized compression distance and image distinguishability. In *Proc. SPIE 6492, Human Vision and Electronic Imaging XII*, Bernice E. Rogowitz, Thrasyvoulos N. Pappas, and Scott J. Daly (Eds.), Vol. 64921D. https://doi.org/10.1117/12.704334

[28] Stephanie Wehner. 2007. Analyzing Worms and Network Traffic Using Compression. *J. Comput. Secur.* 15, 3 (8 2007), 303–320. http://dl.acm.org/citation.cfm?id=1370628.1370630

[29] Wing Wong and Mark Stamp. 2006. Hunting for metamorphic engines. *Journal in Computer Virology* 2, 3 (2006), 211–229. https://doi.org/10.1007/s11416-006-0028-7

[30] Wei Yan, Zheng Zhang, and Nirwan Ansari. 2008. Revealing Packed Malware. *IEEE Security and Privacy* 6, 5 (9 2008), 65–69. https://doi.org/10.1109/MSP.2008.126

[31] Jacob Ziv and Abraham Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory* 23, 3 (5 1977), 337–343. https://doi.org/10.1109/TIT.1977.1055714

[32] Jacob Ziv and Abraham Lempel. 1978. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory* 24, 5 (9 1978), 530–536. https://doi.org/10.1109/TIT.1978.1055934