APPROVAL SHEET

Title of Thesis: Malware Detection and Cyber Security via Compression

Name of Candidate: Edward Raff
Computer Science, 2018

Thesis and Abstract Approved: ______________________
Charles K. Nicholas
Professor
Department of Computer Science and
Electrical Engineering

Date Approved: ____________________________

ABSTRACT

Title of dissertation:	Malware Detection and Cyber Security via Compression
	Edward Raff, Doctor of Philosophy, 2018
Dissertation directed by:	Professor Charles K. Nicholas Department of Computer Science and Electrical Engineering

As society becomes increasingly interconnected and dependent on computing systems, so does the importance of cyber security and the prevention of malware. Beyond just the home computer, smart-phones, routers, printers, and all kinds of devices now run operating systems that could be potentially infected. This represents an explosion in the potential attack surface for a malicious actor. The tools currently available to security professions are improving, but limited. Each tool is designed for one software platform, making their scope limited. Adapting these tools to new platforms and hosts requires years of effort and introduces a significant lag time to protecting any new platforms that will arise in the future. Further, malware often involves an adversary intentionally violating format specification and rules. These violations may be intended to slow reverse engineering efforts, hide intent or attribution, or simply be part of an exploit that is part of the malware's functionality.

In this thesis, we develop a new approach for tackling problems related to malware detection and cyber security in general. Specifically, we develop new methods

inspired by compression algorithms that support a wide range of tasks. The compression background allows the methods we develop to be applied to any file format, operating system, or platform. This provides a single method which can be used in all circumstances, and dramatically reduces the potential lag time to protect new platforms. Not only does this provide a wide range of flexibility, but we will also show that our approach significantly improves upon the existing methods available to practitioners today.

Malware Detection and Cyber Security via Compression

by

Edward Raff

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, Baltimore County in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2018

Acknowledgments

I owe a great many number of people a great amount of thanks for helping me reach this moment in my life. I am eternally grateful to you all for your persistent assistances, support, and guidance.

To my advisor, Professor Charles Nicholas, I owe you a great many thanks. You've gone above and beyond in your role, providing guidance and assistance both in my studies and work. Especially your patience in reading all of the many paper and thesis drafts it took to reach this point.

My colleagues at the Laboratory for Physical Sciences, this would not have been possible without your assistance and support. Both in providing the technical resources needed, but also the welcoming and friendly environment that allowed this work to sprout from. In particular, to Mark McLean for your unwavering support through the trials and tribulations. You've provided incredible mentorship and opportunity for me that I don't know how I will repay.

My colleagues at Booz Allen Hamilton, are owed a special note of thanks and support. To Dr. Ryan Lance, Stephanie Beben, Matt Hutchinson, and Steven Mills, your unwavering support for my dual-endeavors in work and school helped to give me the confidence to pursue this thesis. To my officemates Richard Zak and Russell Cox, you've been instrumental to my sanity. Dr. Jared Sylvester and Dr. Robert Brandon, I owe you both a great deal for helping to fill in my weakness and knowledge gaps in pursuing this line of research. I'd also like to thank Booz Allen Hamilton for its financial support in attending UMBC and the wonderful conferences

I've attended through this process. A special note of thanks to Dr. Josh Sullivan, for recruiting me into the firm and supporting my idiocy enough to serve on my committee.

I owe an incredible thanks to my mother, Beryl Raff, for putting up with hours of venting and kvetching about reviewer # 2 and emotional support through this journey. Last, but certainly not least, my father Dr. Barry Raff. Without you I would have never headed down this path or have been this successful.

I don't have the space to individually thank all the other wonderful people in my life who have made it special, and are equally deserving of thanks as I reach this milestone in my life and career. There are so many people I have been lucky to know and call a friend or family member. Thank you all for everything.

Table of Contents

List of Figures

List of Abbreviations

OS	**O**perating **S**ystem
AV	**A**nti-**V**irus
PE	**P**ortable-**E**xecutable
NN	**N**eural **N**etwork
NCD	**N**ormalized **C**ompression **D**istance
LZJD	**L**empel-**Z**iv **J**accard **D**istance
IID	**I**dentically and **I**ndependently **D**istributed

Chapter 1: Introduction and Related Work

In this thesis we tackle the problems of detecting malware (is this file malicious or benign?), malware classification (which family is this known malware from?), and similarity search (given this file, have I seen one like it before?). All of these problems are important to a malware analyst and to the broader domain of cyber security.

While tools exist today for all of these tasks, their breadth or utility is often limited. In particular, most are built on domain knowledge based features: using our knowledge of the file formats involved and the methods of malicious actors to manually extract information. This requires extensive expertise and has had difficulty scaling to the ever growing amount of malware being produced. When a new platform emerges and needs similar protection, porting an anti-virus system can take months and even years to re-create. By developing a new approach based upon compression, we can circumvent this domain knowledge requirement: giving us a tool that can be deployed into any new domain once data is available.

Before delving into the details of this thesis, we must review some pertinent information. In particular, there does exist a relatively small body of related work that performs malware detection using little domain knowledge. We will review these methods related to the thesis as a whole in section 1.1, and in each chapter of this work review any chapter specific related work in greater detail. While we will discuss some of the accuracies and results that these prior works have obtained, there exists no standard benchmark corpus in the field. This makes comparing results across papers, and to previous work, challenging. Further still, as we will show in chapter 3, there is a severe dataset design flaw that has been overlooked by many prior works. For this reason we will first detail the data used in this thesis in section 2.2.

1.1 Related Work

There is little existing work in performing malware detection with the explicit goal of using minimal domain knowledge. Doing so requires us to work with the raw byte contents of the binary, and there exist three primary approaches to doing so: byte n-grams, entropy analysis, and compression distances. We will review all three methods, as well as some related but less frequently used approaches. We note that we will refer to the reported accuracies of a number of works, but emphasize that the numbers are not comparable and may not even be meaningful. This is because of a data quality issue that causes overfitting, which we will discuss more in chapter 3.

1.1.1 String Features

Possibly the simplest approach to obtaining features without exploiting any kind of domain knowledge is to extract the ASCII or Unicode strings from a binary. This can be done easily using the Unix *strings* command. One can then create features based on the presence or absence of a string, and may choose to use some kind of term and frequency weighting if a string occurs multiple times within a file.

While strings were one of the first feature types considered in initial research by Schultz, Eskin, Zadok, *et al.* [1], they have not been widely used since. This is likely because of the packing issue, which makes it trivial for the malware author to hide all (or some) strings from trivial extraction. However there has been some more recent works that have used string based features in conjunction with others to improve performance though. Islam, Tian, Batten, *et al.* [2] combined strings with statistics describing the functions within a binary for malware family classification. Saxe and Berlin [3] used statistics about the strings found, rather than the actual string values, in conjunction with three other feature types.

1.1.2 Entropy Information

A method that has not gained wide use as a detector on its own is that of Entropy based classifiers. Given n discrete bins, and the probability of a ball falling into bin i as p_i, the entropy of the bins is given by (1.1).

$$\text{Entropy} = -\sum_{i=1}^{n} p_i \log_2(p_i) \tag{1.1}$$

At a coarse level, the entropy of an entire file (measure over all 256 possible byte values as the bins) has a decent amount of information. At first pass, a high entropy file is likely to be compressed or encrypted. A normal executable program will have an entropy within a range of 4.9 to 5.3 [4]. If a file has an entropy of 2.0

or 7.5, this is so far outside the expected range that we would have good reason to investigate or treat the binary suspiciously.

However, information is needed at a finer level to begin making judgments about a binary's potential maliciousness. The most immediate option is to create a windowed measure of entropy, where the entropy is computed over a finite subset of the binary. This window is then moved across the binary to create a one dimensional time series of the file's overall entropy.

This approach is common, and is often combined with somewhat elaborate dynamic-programing solutions to develop a classifier [5]–[8]. Dynamic programing suffers from considerable computational cost though, which limits its scalability. Others have looked at Haar wavelet based approaches [9], [10], but not found them to perform at a level congruent with state-of-the-art predictors. They however can be used in conjunction with other features as part of a larger system or ensemble.

1.1.3 Byte N-Grams

Byte n-gramming is the most popular method of building a machine learning model from a executable binary without using domain knowledge, and is widely popular in general. For this reason it has a long history of use that we will review.

The method of applying byte n-grams is quite simple, which is likely a major factor in its wide use. N-grams have been widely used in malware classification, starting with the work of [11] that connected the methods being used with those in the domain of Natural Language Processing (NLP). Since then, n-grams have been one of the most popular feature processing methods for malware classification, and have been used for processing bytes, assembly, and API calls [12] into bag-of-words type models. To give a more concrete example of this process, the byte sequence *0xDEADBEEF* would have the 2-grams *DEAD*, *ADBE*, and *BEEF*. At training time all possible 2-grams would be counted, and each 2-gram found would map to an index in a high-dimensional feature vector. The feature vector for *0xDEADBEEF* would have 3 non-zero values, the specific values determined by some feature weighting scheme such as TF-IDF or Okapi[13], [14], though a binary present/absent value is popular as well.

Byte n-grams in particular just use the raw bytes of an executable file as the source to extract n-grams from. Kephart, Sorkin, Arnold, *et al.* [15] provide one of the earliest instances of byte n-grams for malware analysis, using byte 3-grams to classify infected boot-sectors.

For the case of byte n-grams for Microsoft PE binaries, Schultz, Eskin, Zadok, *et al.* [1] provide the earliest work of which we are aware. Shultz et al. considered DLL imports, Strings, and byte n-grams as features, evaluating them using a number of different classifiers. In their work a Naive Bayes classifier had the best overall accuracy at 97.1%, followed by an ensemble of Naive Bayes classifiers using byte n-grams at 96.9%. Their n-gram based model also had the highest detection rate of malware. Shultz et al. compared against a simple signature based approach, which achieved only 49.3% accuracy, showing the importance of expanding beyond signatures in defending against new, unseen malware. Abou-Assaleh, Cercone, Keselj, *et al.* [11] made the connection with techniques in Natural Language Processing work and using a feature selection step, reporting 98% cross validation scores using a nearest neighbor classifier.

Kolter and Maloof [16], [17] looked exclusively at byte n-grams for classifying benign vs. malicious executables, as well as classifying malicious EXEs by payload method. They performed their initial work on a smaller set of 1,037 files, by which they settled on using 4-grams for their features and AdaBoost [18] with C4.5-style decision trees[19] consistently provided the best results. In addition they used Information Gain to prune the set of 4-grams down to the top 500 used for their model. This approach obtained an AUC of 0.984. Testing on a larger set of 3,622 files reached an impressive AUC of 0.996 for the benign vs malicious task. In attempts to explore the information captured by their model, Kolter and Maloof did discover evidence of string features being extracted by their model, but made no effort to quantify their significance to the overall model. Their work has been regularly replicated, for example, by Jain and Meena [20]. We will also replicate their work in chapter 3, where we will more throughly evaluate the efficiency of byte n-grams.

Most work on using byte n-grams for malware classification follow the overall method set by Kolter and Maloof: choose a value of n, use some feature ranking scheme to select a few hundred (up to say one thousand) n-grams, and then evaluate (using cross validation or with a random training / testing split) with one or more classifiers. Since their work, there has been a significant amount of follow up work using byte n-grams, often as a major component in a larger system or improving a component of the process.

A number of works, such as Elovici, Shabtai, Moskovitch, *et al.* [21], Menahem, Shabtai, Rokach, *et al.* [22], and Masud, Khan, and Thuraisingham [23] have looked at combining byte n-grams with other features. Masud, Khan, and Thuraisingham fused byte n-grams, opcode n-grams, and DLL function imports into a larger classification system. They also evaluated on two datasets, similar to our data, which we will discuss shortly in section 2.2. Masud, Khan, and Thuraisingham's two datasets have overlap with each-other, where ours are kept completely disjoint. In their work 4 and 6-grams performed almost equally and obtained 95.4% and 93.6% on their two datasets. The hybrid approach presented in their work obtained 96.3% and 97.6% for 6-grams respectively, indicating that while n-grams did not perform the best they still performed well compared to an approach which required domain knowledge features. For feature selection and model construction Masud, Khan, and Thuraisingham used the same approach as Kolter and Maloof. We note as well that the assembly n-grams in Masud, Khan, and Thuraisingham perform worse than the byte n-grams for $n \geq 2$.

Another line of research, to which this thesis is a contribution, has included the use of larger datasets. Moskovitch, Stopel, Feher, *et al.* [24] pushed their dataset up to 30,000 files for n-gramming. Masud, Al-Khateeb, Hamlen, *et al.* [25] used a distributed system to process 105,388 files, obtaining an accuracy of 97.2% for their best model. This was achieved using 2000 4-grams selected by Information Gain, and using an ensemble of C4.5 decision trees.

Researchers have looked at building more sophisticated systems using byte n-grams as a significant component. Though not directly comparable, the information we discover about n-grams is relevant to the underpinning of these methods, since they are built upon byte n-grams. Perdisci, Lanzi, and Lee [26] developed a multi-stage system using both whole file byte n-grams and byte n-grams from binary code extracted via dynamic analysis, obtaining a final AUC of 0.977. Tahan, Rokach, and Shahar [27] looked at 3-grams to select and match "segments" which were used as their final features. Another related area that we do not compare directly against is the task of distinguishing between subfamilies of malware using byte n-grams, as is done by Zhang, Yin, Hao, *et al.* [28] and Stolfo, Wang, and Li [29].

1.1.4 Byte Sequence Distances

Another general method of comparing files is to define some distance measure by which to compare byte sequences. Work in this area has not been wide spread, perhaps in part due to the difficulty of the task. Defining a similarity measure over arbitrary byte sequences is at face value daunting, and must deal with a potentially unbounded space of possible inputs.

One approach to this has been the use of similarity preserving hash functions[30]–[33]. These hash functions have been developed heuristically for forensic purposes, and are generally designed to support a very high precision but at the expense of low recall. They are most often used to find near exact duplicates, and are often not widely usable as a generic measure of similarity. Their specificity has made them usable for malware family clustering, but where out performed in accuracy and runtime by a more classical machine learning approach [34].

A more principled approach is the Normalized Compression Distance (NCD)[35].

The NCD is a metric inspired by Kolmogorov complexity, and uses compression algorithms as a means of approximating this uncomputable function. Li, Chen, Li, *et al.* [35] provided theoretical backing for their new distance measure, showing that it would behave like a distance metric when certain properties are meet. Given this general purpose ability, there has been considerable interest in using the NCD metric for malware classification [36]–[40]. However, NCD is not without its shortfalls. In particular it is computationally demanding, which limits the scale at which it can be applied. We will review the details of NCD in more detail in chapter 5, where we will also propose a new alternative to NCD that alleviates its issues.

1.2 Outline

Given this overview of the state of domain-knowledge free learning, we can begin the rest of this thesis. As mentioned at the begging of this chapter, data is important and no standard benchmark exists in this space. For this reason we begin in chapter 2 by describing the data we use throughout this work as a whole.

The first experiments and results of our work will begin with the classic n-gram based approach. In chapter 3 we will perform an extensive evaluation of them, improving how byte based n-gram models are built. This provides significant performance improvement on our data compared to the classic approach, but still shows byte n-grams not performing as well as previously thought. We extend this in chapter 4 to show that adding domain knowledge to byte n-graming does not necessarily improve performance. This shows the non-triviality of using domain knowledge and helps to justify the need for a new approach.

We develop our new compression based approach, the Lempel-Ziv Jaccard Distance (LZJD) in chapter 5. We show that LZJD is both orders of magnitude faster and more accurate than its inspiration NCD. We then turn LZJD into an effective and practical malware classifier in chapter 6, providing better accuracy than the byte n-gram approach.

This gives us tooling for classification problems like malware detection and classification. Our next need is in similarity search. We show in chapter 7 that LZJD outperforms prior approaches to measuring the similarity between arbitrary files, and also introduce optimizations to LZJD to make is another 4 to 10 times faster. Because LZJD, unlike its predecessors, is a true distance metric, we can use metric index structures to further accelerate it. We show this in chapter 8, while also developing new incremental construction strategies that allow us to build metric indexes. These are necessary for the incremental construction and querying requirements of a malware analyst.

The culmination of these chapters represents our contribution in this thesis. The conclusion of which is presented in chapter 9.

1.2.1 Chapters and Prior Publication

This thesis represents the culmination of a number of published works. Below we list the publications that contribute to the contents of this thesis.

1. E. Raff, R. Zak, R. Cox, *et al.*, "An investigation of byte n-gram features

for malware classification," *Journal of Computer Virology and Hacking Techniques*, Sep. 2016, ISSN: 2263-8733. DOI: 10.1007/s11416-016-0283-1. [Online]. Available: http://link.springer.com/10.1007/s11416-016-0283-1

- Used for chapter 2 and chapter 3.

2. R. Zak, E. Raff, and C. Nicholas, "What can N-grams learn for malware detection?" In *2017 12th International Conference on Malicious and Unwanted Software (MALWARE)*, IEEE, Oct. 2017, pp. 109–118, ISBN: 978-1-5386-1436-5. DOI: 10.1109/MALWARE.2017.8323963. [Online]. Available: http://ieeexplore.ieee.org/document/8323963/

   - Used for chapter 4.

3. E. Raff and C. Nicholas, "An Alternative to NCD for Large Sequences, Lempel-Ziv Jaccard Distance," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '17*, New York, New York, USA: ACM Press, 2017, pp. 1007–1015, ISBN: 9781450348874. DOI: 10.1145/3097983.3098111. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3097983.3098111

   - Used for chapter 5.

4. E. Raff and C. Nicholas, "Malware Classification and Class Imbalance via Stochastic Hashed LZJD," in *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, ser. AISec '17, New York, NY, USA: ACM, 2017,

pp. 111–120, ISBN: 978-1-4503-5202-4. DOI: 10.1145/3128572.3140446. [Online]. Available: http://doi.acm.org/10.1145/3128572.3140446

- Used for chapter 6.

5. E. Raff and C. K. Nicholas, "Lempel-Ziv Jaccard Distance, an effective alternative to ssdeep and sdhash," *Digital Investigation*, Feb. 2018, ISSN: 17422876. DOI: 10.1016/j.diin.2017.12.004. [Online]. Available: https://doi.org/10.1016/j.diin.2017.12.004

   - Used for chapter 7.

6. E. Raff and C. Nicholas, "Toward Metric Indexes for Incremental Insertion and Querying," *ArXiv*, 2018. [Online]. Available: http://arxiv.org/abs/1801.05055

   - Used for chapter 8.

Chapter 2: Data and Quality

In this chapter we will discuss the datasets that we use throughout the thesis as a whole. A chapter is dedicated to this discussion due to the extreme importance data plays in our results, and the challenges in obtaining it. First we will discuss the difficulties and challenges associated with getting good labeled data in section 2.1, and then describe the data we use in particular in section 2.2.

2.1 Data Quality Challenges

As with many applications, the first task to building a machine learning model is to obtain data that accurately represents the distribution of binaries that will be observed. It is indeed well known that obtaining more and better labeled data is one of the most effective ways to improve the accuracy of a machine learning system [47], [48]. However, by its very nature the potential scope of what a binary can do is unbounded. There is no way for us to randomly sample from the binaries that exist in the world and we have little way to meaningful measure how much of the "space" of binaries we have covered with any given data-set. Beyond the unbounded scope, the malware domain poses a number of unique challenges to data collection.

When obtaining data, it is often the case that malware is the easiest to get.

Not only are there websites dedicated to collecting malware sent in by volunteers [49], [50], but it is reasonable for a researcher to obtain their own malware through the use of honeypots [51]. A honeypot is a system connected to the Internet that intentionally tries to get infected by malware, often by leaving open security holes and foregoing standard protections. At the same time, both of these sources of malware can have data quality issues. Honeypots will have data biased toward what the system is capable of collecting, as malware may require interaction from the honeypot through specific applications in order to successfully infect a machine [52] (e.g., a malware sample's infection vector may rely on a specific version of Firefox or Chrome to be running), and it may not be possible to account for all possible application interactions. Malware may also attempt to detect that a potential target is in fact a honeypot, and avoid infection to defer its detection [53]. The issues that bias what malware is collected by honeypots are also likely to impact the quality of larger malware repositories, as users may run honeypots and submit their catches to these larger collections. Malware repositories will also have a self-selection bias from those who are willing to share their malware and take the time to do so.

Benign data, or "goodware", has proven to be even more challenging to physically obtain than malware. This is in part because malware actively attempts to infect new hosts, where as benign applications do not generally spread prolifically. As far as we are aware, no work has been done to quantify the diversity or collection of benign samples, or how to best obtain representative benign data. Most works take the easiest avenue of data collection, which is to simply collect the binaries found on an instillation of Microsoft Windows. This tactic can lead to extreme over-fitting, where models literally learn to find the string "Microsoft Windows" to make a determination [41]. The population of binaries from Windows share too strong a common bias to be able to generalize to real data, as the model learns to classify everything that does not come from Microsoft as malware. This bias is strong enough that even using only a subset of the information will still lead to over-fitting [54]. This issue is particularly wide spread, and occurs in almost all cited papers in this survey. The notable exception to this are papers produced by corporate entities that have private data they use to develop anti-virus software. When this goodware bias issue is combined with the fact that there is no standard data-set for the task of malware detection, it is almost impossible to compare the results from different papers when different datasets are used. In addition, prior work using benign samples from clean Microsoft installications may significantly over-estimate the accuracy of their methods.

Once data has been obtained, labeling the data must follow (when labels do not come "for free" like they do with honeypots). The issue of labeling malware into families, or determining if an unknown binary is or is not malware, is labor intensive and requires significant domain knowledge and training. This is in contrast to many current machine learning domains, like image classification, where labeling can often be done by individuals with no expertise and with minimal time. For example, an expert analyst can often take around 10 hours to characterize a malicious binary [55]. This makes manually labeling large corpora impractical, and a major roadblock to future applications.

For benign vs malicious labeling, many have attempted to circumvent this issue through the use of anti-virus (AV) products. One popular strategy is to upload binary to websites like Virus Total[1], which will run several dozen AV products against each binary, and return individual results. If more than 30% of the AVs claim a file is malicious, it is assumed malicious. If none of the AVs say it is malware, it is assumed benign. Anything less than 30% but non-zero is then discarded from the experiment [56]. While this selection is easy to perform, the labels will be intrinsically biased to what the AV products already recognize. More importantly, binaries marked by only a few AV products as malicious are likely to be the most important and challenging examples. This middle ground will consist of either benign programs which look malicious for some reason (false positives), or malicious binaries that are not easy to detect (false negatives). Removing such examples will artificially inflate the measured accuracy, as only the easiest samples are kept. Removing such difficult to label points will also prevent the model from observing the border regions of the space between benign and malicious. This issue also hampers effective model evaluation, as we are skewing the data and thus the evaluation to an easier distribution of benign and malicious samples. This causes an artificially high accuracy to be reported.

Inferring labels from AV output is even more problematic for determining malware family, as such labels are not standardized and different AV products will often disagree on labels or type [57]. While more advanced methods exist than simple thresholding for determining benignity [58] and malware family [59], the use of many AV products remains the only scalable method to obtain labels.

[1]`https://www.virustotal.com/`

Beyond the issue of collecting data, there is also the fact that binaries exhibit *concept drift*, meaning the population as a whole changes over time. This is true of both benign and malicious binaries, as changes will percolate through the population as the API of Windows changes, code generation changes with newer compilers, libraries fall in and out of favor, and other factors. It then becomes important to investigate the performance of a classification system as change occurs [25], which is not widely explored. The distribution of malware in particular drifts at a faster rate, as malware authors attempt to modify their code to avoid detection. This is referred to as an adversarial scenario, and only further complicates the development of a long term solution [60], [61].

2.2 Data for Experiments

As we have now discussed, the issue of data quality is paramount to obtaining good results. For this reason we take care to be explicit with how and where we obtained our data.

Our primary data is divided into two higher level groups, *Public* and *Industry*, that are collected in different manners and from different sources. Every file in both datasets is a valid PE binary for either x86 or x64 Windows. We do not intermingle these data for training and keep them separate when testing so that we can better evaluate the generalization of our models. We do this because evaluating on held-out data that was collected in the same manner as training data may not adequately evaluate generalization, since both sets are biased by the same collection mechanism.

Having separate sets collected in different ways helps avoid this issue. Since not all of our training and test sets have the same ratio of benign to malicious files, we always report a weighted accuracy such that the two classes have equal total weight in the score. Across all training and testing sets, no two samples in our data have the same MD5 checksum.

Public's malware is taken from Virus Share [49], and we also use data from Open Malware [50] as a separate malware only test set. Public's benign files (or "goodware") mostly come from various Microsoft Windows operating systems, including Windows XP, Window 7, and Windows 10. A smaller collection of goodware files were also downloaded from `portablefreeware.com` and from the Cygwin and MinGW installations. We use a held out test set from Windows 8.

We arbitrarily chose Windows 8 for the test set. We wanted to avoid having files from the same version of Windows in both the training and testing set to minimize any potential information leakage across the sets. We used as many files from VirusShare in the training set as we could given our computational resources, and used the remainder that we had downloaded for testing. The split of Virus Share in training and testing was random, since we had no additional information to improve the way data was split. During evaluation, Open Malware is reported as a separate line and is not included in the Public numbers. Again, this is so we can better judge generalization since no data from Open Malware was in the training set. The Open Malware data is collected in a manner similar to the Virus Share data, so we would expect their data to be more similar to each other than our other data.

The use of a public malware corpus combined with Windows files and a selection of other commonly installed applications is the same strategy used to build the training and testing sets in most previous work [e.g., 1], [16], [24], [26], [62]. Our results for models trained on the Public data should be representative of the results other works would have obtained.

During initial testing, models trained on Public were not performing well on new data, despite encouraging cross validation scores. We sought out an industry partner to share data of a higher quality and more representative of the larger population, and is the source of our Industry data. Santos, Penya, Devesa, *et al.* [63] also used a private corpus from an industry partner, sampling 1,000 benign and malicious files. We received data from our partner in two batches. The testing set represents the first batch of data obtained, while the second and larger batch was used as the training set.

Both groups of data were randomly sampled from a larger set of benign and malicious EXEs that are meant to be representative of what is often seen on desktop computers (excluding Microsoft EXEs). The total number of files for all our data, Groups A and B, can be found in Table 2.1.

2.2.1 Industry Label Quality

Since the Industry data is not publicly available, we attempt to provide extra details about the contents for readers who are interested. The results of the n-gram analysis convince us that the resulting model has more utility than one constructed

	training		testing	
Public	malicious	benign	malicious	benign
Virus Share	175,875	—	43,967	—
Open Malware	—	—	81,733	—
MS Windows	—	268,236	—	21,854
Misc.	—	1,195	—	—
total	*175,875*	*269,431*	*125,700*	*21,854*
Industry				
Industry Partner	200,000	200,000	40,000	37,349
total	*200,000*	*200,000*	*40,000*	*37,349*

Table 2.1: Breakdown of the number of malicious and benign training and testing examples in each data group, along with the sources they were collected from. "Misc." comprises portablefreeware.com, Cygwin and MinGW.

from Public style data, as many prior works have done. One may wonder then how much of the generalization gap incurred by the Industry model is due to differences in the data distribution, rather than the weaknesses of byte n-grams, or potential label errors. Fully answering such a question is beyond the scope of this work, however we hope the additional details may be of use to the reader to better understand the results as a whole and dissuade any concern of label quality.

To obtain a rough estimate of label errors we ran all Industry data (goodware and malware) through ClamAV.[2] We chose ClamAV because it is freely available to everyone and usable on all operating systems. If the labels are correct, we expect to see that most goodware is not flagged by ClamAV and that most malware is. Because ClamAV is not the most advanced of anti-virus (AV) products, we upload a random sample of 50 files in which ClamAV's output disagreed with our labels to Virus Total for further confirmation. Virus Total uses an ensemble of anti-virus

[2]https://www.clamav.net/

systems to asses each sample it is given. This produces more reliable labels, as well as giving us a proxy for confidence in those labels: a file which is identified as malicious by 20 of Virus Total's constituent programs is more likely to be malignant than one identified by only a single anti-virus program. While it would have been preferable to upload all of our data to Virus Total, a rate limit on the API means it would take multiple months to process the entire corpus. The results of this examination can be seen in Table 2.2. The first line shows the percentage of files marked as malware by ClamAV, and the remaining lines are the statistics of potentially mislabeled data sent to Virus Total (VT).

		Industry Train		Industry Test		Public Train	Public Test
	Label	Goodware	Malware	Goodware	Malware	Malware	Malawre
	ClamAV says Malware	0.4%	81.2%	1.4%	78.3%	66.6%	66.5%
# say malware	0 AVs	82%	—	12%	—	—	—
	[1,5] AVs	12%	—	22%	—	—	6%
	[6,15] AVs	6%	—	36%	12%	2%	—
	[16,25] AVs	—	96%	24%	10%	2%	—
	26+ AVs	—	4%	6%	78%	96%	94%

Table 2.2: Percentage of Industry data, and Public Malware, marked as malware by ClamAV. Cases where ClamAV and the label disagreed were uploaded to Virus Total to help confirm the label quality.

We can see for the goodware Industry data, ClamAV marks almost all files as benign, with the test dataset having a higher conflict of 1.4% of the data. Even if all of such data was in fact mislabeled malware, the percentage would be small enough that a robust machine learning system should be able to learn from it. From the sample sent to Virus Total, we can see that most files had no or only a handful of anti-virus systems flag the files, giving us confidence that ClamAV was throwing false positives on the goodware data it identified as malicious. We note as well that anti-virus products may, in general, throw false positives for more challenging benign samples. A benign application with more sophisticated code (e.g, for performing encryption, just-in-time compilation, or disk formatting) may seem malicious to an AV product for good reason, despite having no malicious purpose as an application. We have observed that a number of the Industry test goodware files, marked by multiple AVs, are products for encryption that do not have a clear malicious intent. We avoid explicitly naming these products due to privacy concerns.

While the Industry goodware samples sent to Virus Total are more anomalous compared to those from the training set, we are still confident in the majority of the labels since we obtained. If we assume that all samples marked by 6 or more AVs is in fact malware, then Industry's test set goodware would contain only 0.9% mislabeled files. The maximal change in test accuracy is thus less than one percentage point, and then would not meaningfully impact any of the results or conclusions of our work.

For the malware datasets, ClamAV fails to recognize a much higher percentage as malicious: around 19% of files for Industry and 33% for Public. When uploading samples that were not caught by ClamAV, every sample was marked by at least one anti-virus. Most were marked by a plethora of products (26 or more), with only a handful being marked by less than 10. This again gives us confidence that our labels are correct, and that ClamAV is throwing false negatives.

As mentioned in section 2.1, the construction and evaluation of a high quality dataset for this task is still an open problem. We believe we have provided ample evidence that our Industry data is of a better quality than the datasets normally used (i.e., Public type data), but there is room for improvement in validating and constructing yet larger corpora for this task.

Table 2.3: ClamAV labels found in only the malware portion of the Public and Industry data. "Assorted Other" includes obscure labels that had 5 or fewer occurrences in any dataset.

ClamAV Label	Industry Train	Industry Test	Public Train	Public Test
Dos.Trojan	0	0	18	4
BC.Win.Trojan	0	4	32	8
BC.Win.Virus	217	26	0	0
Heuristics.Encrypted	0	1	9	4
Heuristics.Trojan	1	10	400	104
Heuristics.W32	1	478	59	15
Js.Adware	16	0	0	0
Html.Trojan	0	14	0	99
Legacy.Tool	0	0	9	2
Legacy.Trojan	140	133	604	145
Pdf.Exploit	0	1	8	2
Win.Adware	77,613	7,551	1,924	454
Win.Downloader	306	353	14,250	3,599
Win.Dropper	74	67	2,834	672
Win.Exploit	15	16	588	180
Win.Ircbot	0	0	83	15
Win.Joke	0	0	63	26
Win.Keylogger	0	0	7	4
Win.Malware	10	3	5	4
Win.Proxy	0	0	148	31
Win.Spyware	75	285	8,579	2,089
Win.Tool	12	1	455	110
Win.Trojan	83,701	18,672	80,246	20,033
Win.Virus	52	573	177	42
Win.Worm	191	3,132	6,181	1,589
Assorted Other	5	1	15	6

We also look at the labels ClamAV produces for the files it does recognize as malignant in the malware data. These are shown in Table 2.3. We caution that the labels should not be taken as an absolute ground truth; ClamAV did not recognize a significant percentage of each dataset as malware, and anti-virus products in general do not always agree on the type of, or label for, an individual specimen. It is also important to note that some of the labels are quite unexpected, indicating JavaScript, PDF, and HTML malware. We reiterate that all files in all of our datasets are valid PE files. These labels are either faulty in their designation, or an indication of our executables containing malware of a considerably different nature would be anticipated.

Looking at the Public and Industry data separately, and comparing within group train and test sets, we see fairly consistent patterns. Looking across groups, we do see some distributional similarities and differences. Both Public and Industry are comprised mostly of Trojans, and do contain a significant amount of Adware. In Industry, Adware is a close second for the malware type, where Droppers make a distant second for Public. Public seems to have, in general, a wider array of malware types. It is possible that the differences in distribution account for some portion of the decrease in generalization when applying a model trained on Industry to data from Public. At the same time, it is also important for a model to generalize to novel data, which in this case includes malware of a type never seen before.

2.2.2 Other Datasets Used in Experiments

While our Public and Industry data covers the majority of experiments in this thesis, we will also make sure of some additional datasets for malware family classification. This is a highly related problem to malware detection. Instead of determining if a binary is benign or malicious, we are given a known malicious binary and attempt to identify which family it belongs to. We use these additional datasets to show that some of the new methods developed in this thesis have utility wider than just the domain of windows binaries without domain knowledge. Doing so increases confidence in our results for malware detection, as it provides evidence that our newly developed algorithms are not overfit to the primary problem of interest.

2.2.2.1 Microsoft Kaggle Malware Data

In 2015 Microsoft provided a corpus of malware data for a Kaggle competition [64]. This dataset contains 9 malware families in 10,868 training files at 50.8GB in size. We will use two different feature options that were provided as part of the competition. First is the raw byte contents of the files[3] which can be used for byte n-grams, NCD, or any other similar approach. We will refer to this version as "Kaggle Bytes".

Microsoft also provided the disassembled versions of each file using industry standard software IDA Pro. 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). The disassembled version of the dataset takes up 147GB of disk space, and we refer to it as "Kaggle ASM".

[3]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.

2.2.2.2 Android Drebin Malware

We will also make use of a dataset of Android APK malware called Drebin [65], but for practicality we will remove any malware family that has less than 40 samples[4], unless stated otherwise. 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 applies little or no compression when creating the zip files. For this reason we use 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).

Having described the data that will be used throughout this work, we now move to on to our analysis of that data.

[4]Many of the malware families had less samples then cross-validation folds, which would have made evaluation difficult

Chapter 3: Weakness of Byte N-Grams

Byte n-grams have been used as features in a number of works, and are one of the most common feature types used for static malware analysis [66]. By treating a file as a sequence of bytes, byte n-grams are extracted by looking at the unique combination of every n consecutive bytes as an individual feature. Most experiments from other works range from $n = 1$ to $n = 8$ bytes, and are generally reported to be effective for any $n \geq 2$ with papers determining various different values of n as performing the best [66]. Byte n-grams are particularly attractive since they require no knowledge of the file format, do not require any dynamic analysis, and could potentially learn information from both headers and the binary code sections of an executable[67]. This would satisfy our desires both for a method that is effective at malware detection, and avoids the use of potentially expensive domain knowledge.

Given these benefits combined with reported accuracies of 95% or better, in this chapter we investigate what features are learned by byte n-grams and why they seem to perform so well. Our experiments will examine their performance for $n \in \{4, 6\}$. We find 6-grams to perform best, and use them as the basis of our investigation into what concepts are actually being learned by our model.

Overview of chapter contributions

Some of the potential shortcomings with byte level n-grams in the realm of malware classification have been discussed before [68], but we are not aware of any work that attempts to assess their true effectiveness or generalization to new data sets. To do this assessment, we use multiple separate sources of data for our experiments, divided into two higher level groups (*Industry* and *Public*) and as described in section 2.2. Doing so allows us to show that prior works have likely suffered from considerable overfitting by using data in the style of Public.

We begin our investigation by attempting to reproduce previous work, but our larger collection of data results in more potential features. For this reason we performed an evaluation of feature selection methods (section 3.1). As part of improving the feature-selection process we used Elastic-Net regularized Logistic Regression as our classifier, which performs implicit feature selection. In subsection 3.1.1, we examine both the final model performance, and the regularization path, where we discover significant over-fitting of our n-gram models and a possible methodological flaw in the data-collection process of some, if not most, previous papers.

In section 3.2 we investigate the nature of our features, and determine why they don't work as well as expected. We present evidence that our n-grams are learning string-like features rather than information from the code or other sections of an EXE file. Based on these results, we devise the Multi-Byte Identifier in subsection 3.2.1 as a technique to help further evaluate n-grams for EXE files. The final experiment in our investigation is covered in subsection 3.2.2, where we provide evidence that most of the *generalizable* information may be coming from ASCII strings. Given these surprising results, we discuss what we believe are the major weaknesses of the byte n-gramming approach in section 3.3.

3.1 Feature Selection and Model Building

An issue not adequately addressed in previous work is the feature selection process. For the 400k total files in Industry's training set, there are 4,289,759,510 unique 4-grams and 35,953,973,975 unique 6-grams observed. Storing the 6-grams naively, with 32 or 64-bit integers for count information, would take 503 or 791 GB of RAM, respectively. This issue alone is a significant road-block to applying byte n-gram features in practice.

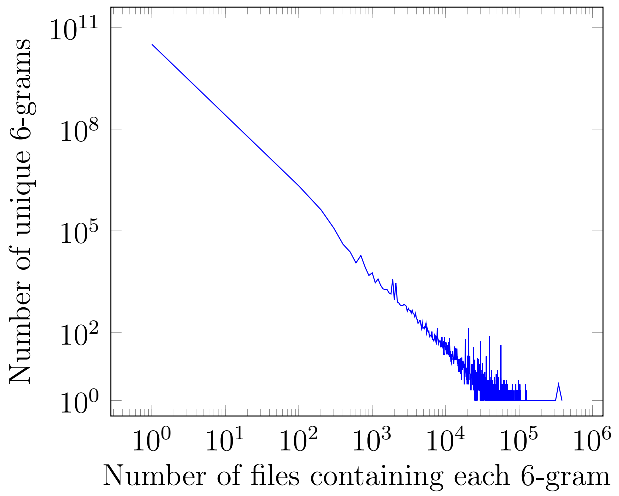


Figure 3.1: From training-set of Industry, the number of 6-grams that occurred in x many files.

The feature selection process is made somewhat easier by the frequency of individual n-grams, as shown in Figure 3.1. We observe that they tend to follow a power-law type distribution, with 87.72% of 6-grams occurring only once, 97.58%

6-grams occurring ten or fewer times, and 99.61% with 100 or fewer occurrences. This is not surprising, since n-grams from NLP applications tend to follow a power-law (Zipfian) distribution as well. We felt the presence of such a distribution was worth confirming, since there is no reason n-grams would have to follow such a distribution when applied to different domains. We can reduce our set of candidate n-grams in general by selecting a minimum number of occurrences based on coverage in our dataset. For example, selecting 6-grams that occur in at least 1% of the aforementioned 400k files results in just under 1.6 million 6-grams (a reduction of more than 99.99%).

Learning from 1.6 million 6-grams is still a computational burden and provides strong potential for over-fitting, so additional feature selection is necessary. Most previous work [e.g., 20], [26] use Information Gain criteria (3.1) or some other simple ranking scheme to select a fixed subset of n-grams. Our approach is to first do a coarse feature selection down to 200k n-grams, followed by a final feature selection during model construction.

We compared a number of ranking schemes to choose a subset of 200k n-grams which we list and briefly describe. For the equations below, g_j indicates the presence of n-gram j, and m_j and b_j are the number of malware and benign files that had g_j present. M denotes the malware class, B denotes the goodware or "benign" class, and P(x) is the probability of x given the training data. N_M and N_B indicate the number of n-grams found in malicious and benign files respectively. We tested each of the below methods, such as Information Gain (3.1), to select the initial 200k subset of n-grams.

$$IG(g_j) = \sum_{v \in \{g_j, \neg g_j\}} \sum_{C \in \{\mathrm{M},\mathrm{B}\}} \mathrm{P}(v, C) \cdot \log_2 \left(\frac{\mathrm{P}(v, C)}{\mathrm{P}(v) \cdot \mathrm{P}(C)} \right) \tag{3.1}$$

As an alternative to Information Gain, we introduced two simple scores that prefer features occurring in only one of the classes. Malice Score (3.2), which is biased toward features more common in malware and Benign Score (3.3) to favor features found in goodware.

$$\text{Malice Score}(g_j) = \mathrm{P}(g_j|\mathrm{M}) - \mathrm{P}(g_j|\mathrm{B}) \tag{3.2}$$

$$\text{Benign Score}(g_j) = \mathrm{P}(g_j|\mathrm{B}) - \mathrm{P}(g_j|\mathrm{M}) \tag{3.3}$$

To test favoring lop-sided occurrence rate in either direction we added the Absolute Malice Score (3.4). A simple variant on the Absolute Malice Score is Root Malice Score (3.5), which prefers more "pure" features based on the class in which it occurred.

$$\text{Absolute Malice Score}(g_j) = |\, \mathrm{P}(g_j|\mathrm{M}) - \mathrm{P}(g_j|\mathrm{B}) \,| \tag{3.4}$$

$$\text{Root Malice Score}(g_j) = \left| \sqrt{\mathrm{P}(g_j|\mathrm{B})} - \sqrt{\mathrm{P}(g_j|\mathrm{M})} \right| \tag{3.5}$$

We also evaluated ranking based on the Gini coefficient (3.6) and KL-divergence

(3.7). Our Gini tests add c artificial observations to each n-gram, because without this modification many millions of n-grams all received the same maximal score.

$$\text{Gini}_c(g_j) = \frac{2(m_j + c)(b_j + c)}{(m_j + b_j + 2c)^2} \tag{3.6}$$

$$\begin{aligned} \text{KL}(g_j) = &\frac{m_j}{m_j + b_j} \cdot \log_2 \frac{m_j(N_M + N_B)}{N_M(m_j + b_j)} \\ &+ \frac{b_j}{m_j + b_j} \cdot \log_2 \frac{b_j(N_M + N_B)}{N_M(m_j + b_j)} \end{aligned} \tag{3.7}$$

Much previous work used feature ranking schemes like these to do all of their feature selection. A shortcoming of this approach is the need to then estimate how many features to select. Prior works usually selected only a few hundred to one thousand n-grams, and then trained a model on the selected subset. Determining the appropriate value of k becomes its own expensive process, as noted in Kolter and Maloof [16] where the number of n-grams was chosen based on analysis of a subset of the data. In our approach (detailed in 3.1.1), the initial coarse feature selection is mostly for computational convenience. This is because we have chosen a Machine Learning model that does implicit feature selection as part of the model building process.

To compare these feature selection methods, we built models using each of them according to the method described in subsection 3.1.1 and sorted the models by their cross validated (CV) accuracy. The results are shown in Table 3.1. We see that simply selecting the n-grams that occurred in the most files performed best, with most methods resulting in only a minor difference. Specifically, all of the tested feature selection methods, except the Gini measure (3.6) and KL-divergence (3.7), obtained 90%+ accuracy. We note that of the simple sorting methods we tested — equations 3.2 through 3.5 — the Root Malice Score did slightly worse than the others. The square root term in (3.5) causes a slight preference for purity of label, that is to say the equation prefers to select n-grams that occur only in benign or malicious files but not both. This is a property shared by (3.6) and (3.7). Overall this would seem to indicate a need for common, high-frequency n-grams in order for our models to perform well.

Much of the previous work [16], [24] in using n-grams has suggested the use of Information Gain (3.1) based on its success in text classification and other NLP domains. Our results indicate that while Information Gain does work well, we can use much simpler approaches by choosing a model that has feature selection built-in to the model's training.

Selection Method	CV Accuracy
Frequency	96.6%
Malice Score	96.3%
Abs Malice Score	96.3%
Benign Score	96.3%
Info Gain	95.2%
Root Malice Score	94.6%
Gini_{32}	85.0%
KL	78.7%
Gini_{256}	77.2%

Table 3.1: 10-fold CV accuracy rate on Industry training data using the top 200k features selected by different methods. Test set errors had similar ranking.

A downside not addressed in previous works is that adding more data does not significantly change the number of n-grams that become viable model features, causing diminishing returns as data is added. This is counter to the argument presented in Schultz, Eskin, Zadok, *et al.* [1]. For example, for the results of our whole model building process: a model built from the training set of Industry obtains a weighted accuracy of 87% on all data, excluding Open Malware, from Public. Training from the much smaller test-set of Industry gets an accuracy of 84.4% on the same data. Evaluating the recall on only the Open Malware data, the performance only drops from 81% to 80%. This puts n-grams in a poor place, as doubling the amount of data can double the amount of resources required but produces only marginal improvement in outcome. This makes it difficult to exploit the phenomena that adding more data tends to provide significant improvements to accuracy [47], [48], [69], which is what we would normally expect. The minor increase in accuracy could be an indication that our features do work well or are nearing the representational capacity of our features and model. We don't believe this is the cause, as we observe evidence of overfitting in the remainder of this section, as exemplified in Table 3.2 and discussed through the rest of this chapter.

3.1.1 Elastic-Net Models of N-Grams

We build our models with Logistic Regression using either of two different regularization methods: Lasso (also called L_1) [70] and Elastic-Net [71]. The objective function of both can be defined using equation 3.8: for Lasso, $\alpha = 1$ and for Elastic-Net, $\alpha = \frac{1}{2}$. The value C in the loss function is our regularization parameter. Larger values of C decreases the strength of the regularization. As $C \to \infty$, (3.8) approaches the behavior of standard Logistic Regression. Smaller values of C reduce the flexibility and effective degrees of freedom of the model, encouraging the solution vector w to approach zero.

$$\begin{aligned} f(w) = \alpha||w||_1 + (1-\alpha)\frac{1}{2}||w||_2^2 + \\ C\sum_{i=1}^{N} \log(1+\exp(-y \cdot w^{\mathsf{T}} x_i)) \end{aligned} \tag{3.8}$$

We use these methods for two reasons. First, they provide a principled method of feature selection that is robust to extremely high dimensional data with many irrelevant features [72] while also being computationally tractable. Their feature selection property is a direct result of the model building process, as exact zeros will occur in the optimal solution of w as a result of the $\alpha||w||_1$ term in the objective function. Second, we can compute a regularization path, where we look at the properties of the model (e.g. accuracy, number of non-zero weights, or coefficient values) as a function of C. This regularization path provides insights into the model's performance and the quality of the selected features.

For our training process, we start with the 200,000 6-grams based on total number of files they occurred in (i.e., an n-gram is counted only once per file). Feature vectors are constructed as binary vectors, with zero indicating the absence of the feature in the file, and one indicating its presence.

We also experimented with using a larger number of features to start (up to one million), with using TF-IDF style weightings for occurrences, and with counting n-grams by total number of occurrences in a file when considering the top selection.

None of these experiments resulted in a noticeable change in accuracy, or the behavior of the regularization path when trained. Any one of these changes could result in a different subset of n-grams being selected by our final model, with up to 50% of them changing. We are concerned that up to half of the selected features could change with no discernible impact on accuracy and believe this is an indication of a weakness of byte n-gram features.

To create our regularization path, we need a minimum and maximum value for C to consider. We first compute C_0, the value of C that would result in a weight vector of all zeros as specified in Yuan, Chang, Hsieh, *et al.* [73]. We then use a starting value of $2\,C_0$ as the strongest regularization we evaluate, and set the weakest regularization to be $2\,C_0 \times 10^5$. We compute the regularization path along 300 logarithmically spaced points along this range, following the basic warm-start strategy in Friedman, Hastie, and Tibshirani [74]. The warm-starting strategy allows us to build these models sequentially at only incremental cost. However, our search is over five orders of magnitude, where Friedman's search only covered two orders of magnitude. We use this larger range due to unusual behavior observed in the regularization path, and highlight the issues below. We confirmed our results using two implementations of Elastic-Net Logistic Regression, one an extension of the new GLMNET [75] to the Elastic-Net case and another using the extension to OWL-QN presented in Gong and Ye [76].

In Figures 3.2 and 3.3, we show the number of non-zero values in the weight vector and the accuracy over the regularization path using 6-gram features. These results give us several reasons to suspect that our n-gram features are over-fitting the data.

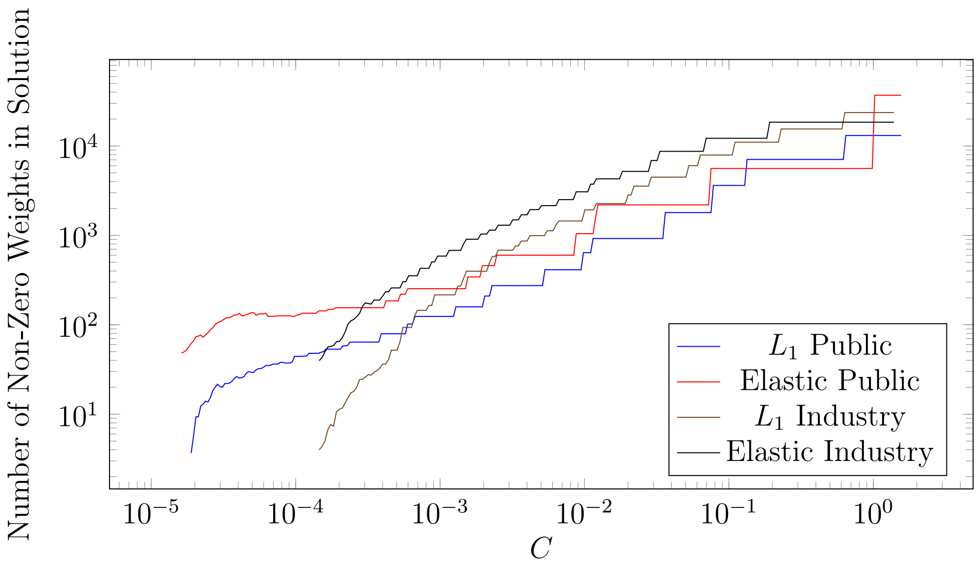


Figure 3.2: Average number of non-zero weights in solution vector based on 3-fold cross validation across regularization path.

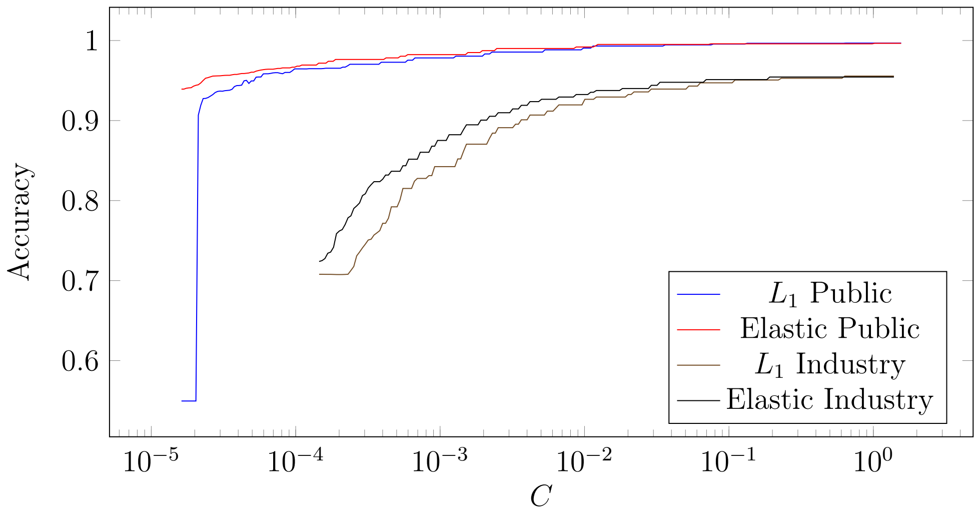


Figure 3.3: Average accuracy based on 3-fold cross validation across regularization path.

Our data requires extreme levels of regularization to obtain an informative regularization path, which is the main reason we use such a large search range for C. In Yuan, Ho, and Lin [75] the smallest regularization value considered is 2^{-4}, which is already near the maximum of our evaluated range and at the point of diminishing returns. If we had used the ranges suggested in other papers our regression model would have exhibited almost no change in the weight vector for all tested values.[1]

Even if we ignore the abnormally high amount of regularization, the behavior of the regularization paths of both Public and Industry models are irregular. Looking at Figure 3.2, the most pronounced issue is the step-ladder addition of features in large groups at a time, rather than more continuous additions of features a few at a time. This behavior is obvious in the Public models, but also occurs in the Industry models. In Figure 3.3, the Public models exhibit higher accuracies over the whole range of regularization, much higher than we would expect. The differences between the models' CV error rate gives a strong indicator of data quality issues.

In addition, we note that the L_1 Industry model in Figure 3.3 has a CV accuracy of 71% using an average of only four features. The L_1 Public model gets 91% using just ten. The Elastic-Net Industry & Public models use a similarly small 40 features to obtain 72% and 94% accuracy respectively. A priori, it seems highly unreasonable that as few as ten 6-grams should be able to obtain such high accuracies. Both Industry models are entering a plateau of 95% accuracy by 10,000 features, and both Public models plateau of 99%+ by just 2,000 features. If n-grams were effectively learning features from the binary sections of an executable, it seems unlikely that millions of malware and goodware EXE files would be forced to use such a small subset of binary code. We confirm in section 3.2 that the n-grams are

[1]In extended testing, neither the accuracy or number of non-zeros increased when testing larger values of C.

not effectively learning binary features.

		Public			Industry		
		L_1 (%)	Elastic (%)	KM (%)	L_1 (%)	Elastic(%)	KM (%)
6-gram	OM	96.9	97.2	96.2	81.1	81.2	49.5
	Public Test	99.7	99.6	99.1	87.3	87.0	72.8
	Industry Test	68.1	68.1	62.0	94.5	92.5	85.5
4-gram	OM	97.0	97.3	97.3	67.1	64.1	52.6
	Public Test	99.6	99.6	99.3	84.9	84.7	74.3
	Industry Test	70.5	68.8	65.8	90.6	89.7	86.6

Table 3.2: Performance of Kolter and Maloof (KM), L_1 and Elastic-Net regularized models trained on both groups of data, and applied to all testing data. Open Malware (OM) is recall, others are weighted accuracy. Column indicates which data was used for training the model used. Row indicates the test set and whether 4 or 6-gram features were used.

The final testing accuracy is shown in Table 3.2. We use a weighted accuracy so that the malware and benign samples in the test set have equal total weight. Since the Open Malware test set comprises only malicious files, we list the recall rate rather than accuracy. The Open Malware files are not included in the Public Test scores.

In our results the models trained on Public do not generalize to the data in Industry, getting an accuracy much lower than what was achieved in all previous works. Due to the bias in how Public's data was collected, we believe that the Public model has actually learned an "is it a windows executable?" classifier, rather than "is it malware?". This would explain the high recall on Open Malware. The model learns to say "no, not Windows" (i.e. malware) for all the data, since none of it came from Windows and the whole collection is malware. However, saying "no" for the Industry goodware (which also did not come from Windows) results in an error rate approaching random guessing. The lopsided errors in the Industry test set corroborate our suspicion that the models defaults to classifying most inputs as malware by default, and then use features present to switch to a decision of goodware. This is further confirmed by looking at the false-positives on the portablefreeware, Cygwin, and MinGW files. Since these were a part of the training set, we would expect them to be predicted correctly by the model. However we found that MinGW and Cygwin had a false positive rate of 39%, and portablefreeware data had a false-positive rate of 43%. These values are extraordinarily high and do not reflect the Public test set accuracies, providing strong evidence that the model is not accurately learning the desired concept.

Models trained on Industry generalize better to Public testing than vice versa, though the accuracy on the Public test set is lower than the Industry test set, and the recall on Open Malware is down to the low 80s. Despite the drop, this overall behavior is consistently better — as it is generalizing past the training distribution. This indicates the 6-gram model trained on this data has meaningfully captured some information. The spread in test set performance, combined with the behavior exhibited in Figures 3.2 and 3.3 give us reason to suspect that there may still be some level of over-fitting occurring. This is corroborated by the higher CV accuracies in Table 3.1, and suggests that the I.I.D. assumption of cross validation is too strongly violated in our corpora to be used for evaluation. Given the similarity of our Public data to others, and its increased scale, this brings many previous results into question — especially when used as the only validation as in Abou-Assaleh, Cercone, Keselj, *et al.* [11]. We also note that the Industry test performance drops to around 81% accuracy when considering the generalization to the Open Malware data. While this does not necessarily represent the accuracy we would expect when deploying this model to various users, it is considerably below the mid to high 90s that are reported by most others. This indicates that the true effectiveness of binary n-grams for malware classification has been considerably over-estimated.

Comparing to the baseline approached used in Kolter and Maloof [16] (KM), our use of Elastic-Net and Lasso regularized Logistic Regression has provided a dramatic performance improvement when trained on the Industry data. Because KM use boosted decision trees, which can learn non-linear decision surfaces, we know the difference in accuracy is not a capacity issue of the chosen model since we used a simpler linear model. The KM approach trained and tested on Industry performs worse, likely due to a lack of features. By using a model with the feature selection process built in, our performance has improved generalization and simplified the feature selection task. We also note that the KM's approach loses generalization accuracy at a quicker rate when moving from the Industry test data to Public test, and from Public test to Open Malware. Considering the extreme levels of regularization we are using for a linear model, it is likely that the KM approach's difficulty in generalizing past the training data is caused by the use of a more powerful and flexible discriminative model. That is to say, the non-linear model used in KM has greater capacity to overfit the data — which we believe to be the issue since our simpler model is also showing evidence of overfitting.

4-gram vs 6-gram Performance

On the issue of n-gram size, we note that there is no significant difference in the overall behavior of the regularization path between 4 and 6-grams, though there is an apparent difference in generalization accuracy. In Figure 3.4 (corresponding to Figure 3.2) we see the same general pattern of the Industry models initially selecting more features than the Public models, before reaching a common plateau. In Figure 3.5 (corresponding to Figure 3.3) we again see that the models trained on Public immediately reach a higher accuracy, with the Industry models not reaching as high and taking longer to reach their plateau. We note that the cross validation scores on the Industry data are reaching higher accuracies for 4-grams (97.4%) than was obtained for the 6-grams (95.6%). Yet 4-grams have lower test set accuracies than 6-grams in Table 3.2. This means the 4-grams are exhibiting even higher degrees of overfitting compared to 6-grams.

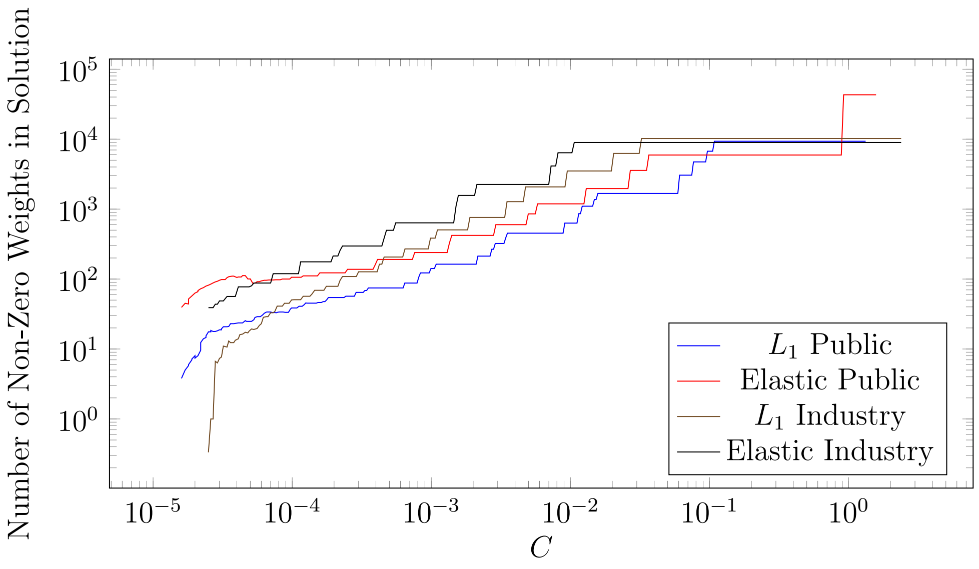


Figure 3.4: Average number of non-zero weights in solution vector based on 4-gram features and 3-fold cross validation across regularization path.

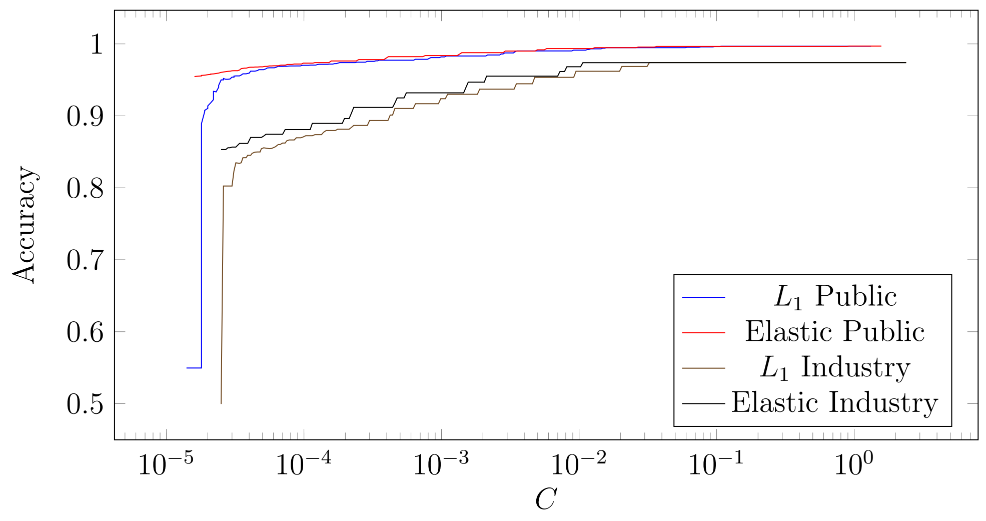


Figure 3.5: Average accuracy based on 4-gram features and 3-fold cross validation across regularization path.

Looking at the numbers in Table 3.2, we see that despite having the same general behavior — 6-grams trained on Industry generalized considerably better than 4-grams to the Open Malware data, with 4-grams trailing by 14 percentage points. The difference is not as large for the Public and Industry test sets, but the 6-grams do continue to perform better by 2 to 4 points. This would seem to validate our general preference for 6-grams over 4-grams. Looking only at the models trained on Public, the difference in accuracy mostly disappears. Considering that the Public models are overfitting to the Microsoft Windows data, it is understandable that their performances would converge. This would also explain why most prior works, using data similar to Public, have shown little performance difference for n-grams when testing various values of n.

3.2 N-Gram Evaluation

Given the poor level of performance compared to previous results when given more data, we sought to evaluate what the n-grams extracted were actually learning. The starting point of this was to use the 6-grams selected by the Elastic-Net model trained on Industry to obtain some simple statistics. We use the 6-gram model since 6-grams had better performance than 4-grams in subsection 3.1.1.

First, we look at the entropy of where our 6-grams occurred. Because different modalities of information have different average entropies [4], looking at these statistics gives us an idea of what information may be captured by the n-grams. Since 4 and 6-grams are too small to compute a meaningfully entropy measure, we estimate the entropy of an n-gram from a window of bytes around where it occurred in a file. We compute the Shannon entropy (1.1, repeated below for reference), where p_i is the proportion of byte i in the given window.

$$S = -\sum_{i=1}^{256} p_i \cdot \log(p_i)$$

In our testing we used a window of 128 bytes, though results were not sensitive to exact window size. The distribution of n-gram entropies are presented in Table 3.3. 26% of our n-grams occurred in entropy regions more often associated with plain-text (i.e. $S \leq 4$) [4]. From this table, it would appear that up to 74% of our n-grams are occurring in entropy regions associated with executable code (that may be packed or encrypted). This is initially encouraging, as it indicates our 6-grams do occur in regions associated with code and hence, may be learning to discriminate between benign and malicious code.

We also looked at how well distributed our features are among the test data. In Figure 3.6 we plot the fraction of 6-grams the model selected that occurred in the Industry testing data. If our 6-gram features are working effectively, we would like to see a relatively even distribution of feature occurrences, but the unfortunate trend is that our 6-grams are not evenly distributed through the test data. Instead a few files have a significant fraction of features present, quickly trailing off toward almost none of the features being present. For n-grams to generalize well, we need to see our features occur in new files on a consistent and regular basis, as they are intrinsically critical to making the classification decision. The likelihood of observing our n-grams can only decrease when tested on data from a different source, making this trend a significant issue.

Entropy	Proportion	Cumulative
0	0.4%	0.4%
1	1.5%	1.9%
2	3.8%	5.7%
3	11.1%	16.8%
4	9.2%	26.0%
5	19.2%	45.2%
6	48.3%	93.5%
7	5.7%	99.2%
8	0.8%	100%

Table 3.3: Percentage of n-grams that occurred within an entropy window (rounded to nearest integer).

These statistics give us some higher level information, but do not explain any of our results. Since so few features are needed to obtain high CV accuracies, and based

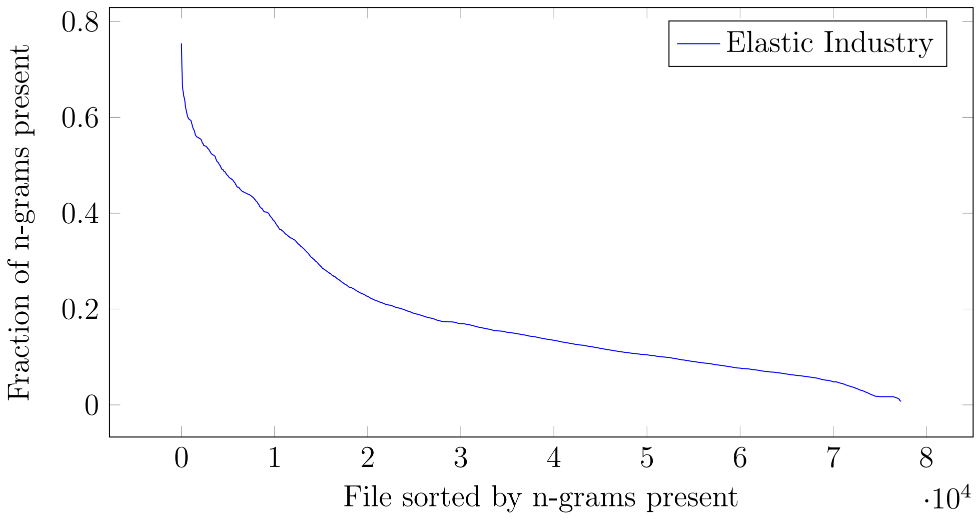


Figure 3.6: Fraction of 6-grams used in the model that are observed in each testing point.

on the proportion of our n-grams occurring in low entropy regions, we examined the ASCII decoding of the n-grams selected at the beginning of the regularization path. A subset of the 4 and 6-grams selected by the L_1 regularized models are presented in Table 3.4. Sixteen 6-grams and 4-grams were selected by the models trained on Public; thirteen 6-grams and 27 4-grams were selected for the models trained on Industry. Note, the number of non-zero features selected is larger than what is shown in Figure 3.2 because those numbers are the average number of non-zeros from the CV models, whereas this is the actual model trained on all data at that regularization strength.

Looking at the ASCII decodings, we can see that the models trained on Public are selecting parts of the text "Microsoft Corporation", which is embedded in most of the executables that come with any installation of Microsoft Windows, often if not always with symbols such as "®" and "©". Because L_1 regularization does not like selecting highly correlated features [71], it has difficulty obtaining the n-grams to complete the string. This overfitting to the concept of "from microsoft" was hypothesized by Seymour [77] who showed prior published models trained in this manner did not generalize to new benign files.

The model trained on Industry 6-grams appears to be picking up items from the header and import tables, selecting most of the import "KERNEL32.DLL" and a `GetProcess` function. By looking at the Elastic-Net 6-grams, since it has no issue selecting correlated features, we can confirm that our hypothesis is correct and that this behavior extends out into the beginning of the regularization path's search (such tables can be found in the appendix). The model trained on Industry 4-grams is still picking up string information, but seems to prefer some different information. It does not tend to select strings that make function imports, like the 6-grams do. We suspect this is an issue with the smaller 4-grams matching too many other tokens as well, losing some of their discriminative ability. The items selected by the 4-grams are generally selected by the 6-gram model later in the regularization path. Overall, the most discriminative items being selected appear to be string features. This result appears to contradict Schultz, Eskin, Zadok, *et al.* [1], which posited that n-gram features provide additional robustness to the model since they are harder to avoid than string-based features.

3.2.1 Multi-Byte Identifiers

Based on the behavior of the n-grams observed in Table 3.4 we develop the hypothesis that informative malicious or benign segments in the data are longer than

Public		Industry	
6-gram	ASCII	6-gram	ASCII
`20004D006900`	`␣Mi`	`000047657450`	`GetP`
`00720070006F`	`rpo`	`657450726F63`	`etProc`
`006F00720070`	`orp`	`00004C6F6164`	`Load`
`43006F007200`	`Cor`	`6B65726E656C`	`kernel`
`000100560061`	`Va`	`00004B45524E`	`KERN`
`004400000001`	`D`	`4C33322E444C`	`L32.DL`
4-gram	ASCII	4-gram	ASCII
`4D006900`	`Mi`	`69726541`	`ireA`
`7400AE00`	`t®`	`6C3D2272`	`l="r`
`00720070`	`rp`	`3C736563`	`<sec`
`20004300`	`␣C`	`2E637274`	`.crt`
`00010056`	`V`	`696F6E3E`	`ion>`
`72794100`	`ryA`	`3C2F7365`	`</se`

Table 3.4: Selection of 4 and 6-grams chosen by the most strongly regularized L_1 models. Whitespace denoted using the '␣' symbol

our n-grams. If this is the case, we should see sequences of adjacent or overlapping n-grams when processing our files, and it would explain why increasing n to even higher values does not tend to cause any significant drop in performance. We call any such sequence a Multi-Byte Identifier, or MBI. We search for MBIs using the subset of n-grams chosen by our Elastic-Net model, since any larger sequences of n-grams will be highly correlated and thus less likely to be selected by Lasso.

It is difficult to manually evaluate how n-grams are being used as more are added to the model. MBIs can help us to better understand what higher level concepts are being learned by our model. After manually inspecting MBIs from a few files, we used a simple heuristic to separate MBIs into three categories: Strings, Simple, and Other. Examining the contents and statistics of these categories helps us better understand what types of information are being captured by our n-gram models.

1. **Strings**: Any MBI where more than half of the bytes are ASCII printable characters.
2. **Simple**: Any MBI whose hex representation has two or less distinct ASCII characters.
3. **Other**: Any MBI not in the other two categories.

Using this simple strategy, we find that 16.8% of all extracted MBIs are strings, 43.0% are simple, and 40.2% belong to "other". If restricted to unique MBIs, we get 20.6%, 26.8%, and 52.7% respectively. The changes in percentage of MBIs come mostly from the simple section, which has the most repetition. Of the string MBIs, most are learning to find items like "GetProcAddress" and "WIN32.DLL" that are detecting imports or section names such as "`.text`" and "`.rsrc`".[2] Some strings appeared in data or code sections, an example being "PADDINGXX" repeated 26 times. Most simple MBIs were interleaved sequences of *0x00* and *0xFF* of varying lengths, sometimes over 1 KB in size. We were not able to ascertain the nature of the other MBIs. An attempt was made to disassemble these MBIs, but they did not always produce meaningful assembly code. While we were able to occasionally find useful MBIs in this category,[3] we were not able to determine how meaningful this larger category was as a whole.

[2]Amusingly, we discovered one instance of malware using the non-standard "`.virus`" section name

[3]An interesting example is the MBI *0x33C9B104014C24*, which we discovered is used by ClamAV as a signature (see `https://github.com/eqmcc/clamav_decode/blob/master/db/daily.ndb#L363`).

These results indicate that n-grams may not be learning strong features. The percentage of MBIs that are very simple is especially concerning given how frequently they appear in both benign and malicious files. We believe many such MBIs, like *0xFF00FFFF000000*, are an artifact of the overfitting we have observed. In generalizing to new data, they would then act as noise in the decision process. We note that there is the possibility for meaningful MBIs in the simple category, such as *0x0C0C0C0C* [78] or the sequence *0xCCCCCCCC* for repeated `int3` calls to interfere with a debugger. Unfortunately we did not tend to see these in the MBIs our models learned, though they do exist in our data set.

Objectively, any MBIs found in the code section of a file will be brittle, especially if they call any function, as the addresses called can easily change based on changes in the header. In this case, our n-grams would act more as a signature for previously seen malware rather than a feature to predict novel malware. The accuracy on Industry's test set could be partially explained by the model learning a set of smaller signatures, which are applicable to test data from the same collection. While this could be useful for systems such as the one in Griffin, Schneider, Hu, *et al.* [79], signatures are intrinsically not generalizable features. Given the number of string MBIs and the n-grams chosen by our models in Table 3.4, we suspect that n-grams are obtaining most of their generalizable information from the PE header and plain-text strings present in the file, similar to the features used in Shafiq, Tabish, Mirza, *et al.* [68]. We also note that while $n = 6$ bytes is large enough to capture the instruction and operands for about 97.6% of instructions[80], a valid x86 instruction can be up to 15 bytes in length. The variable-length nature of the instructions appears to be a general mismatch for fixed-length n-grams.

We were able to obtain the same MBI results using the n-grams from our L_1 regularized models as well. We hypothesize that this is an artifact of the lack of informative features. The L_1 model, as C increases, is forced to select increasingly larger groups of correlated features simultaneously. This would explain the unusual regularization paths in Figure 3.2 and the plateau in Figure 3.3 and that we are able to obtain MBIs with Lasso. However more testing is needed to be conclusive. Given these results, it may be interesting to explore using MBIs as features themselves in future work.

3.2.2 String Features

To determine how much of the discriminative power comes from n-grams picking up on string features, we perform another test using the occurrence of strings as features. We use the GNU `strings` command to extract all ASCII strings ≥ 4 characters from our training sets. We then create a regularization path using Lasso and Elastic net in the same way we did for n-grams, and plot the cross-validated accuracy as well as the test accuracies on Public and Industry as a function of C. For comparison, in each plot are two dashed lines representing the performance of the 6-gram features on the Public and Industry test sets.

Once trained, we note that both models used a comparable number of string features as compared to the n-gram features. Both models trained on Public strings selected around 5,500 out of 200,000 strings and the models trained on Industry se-

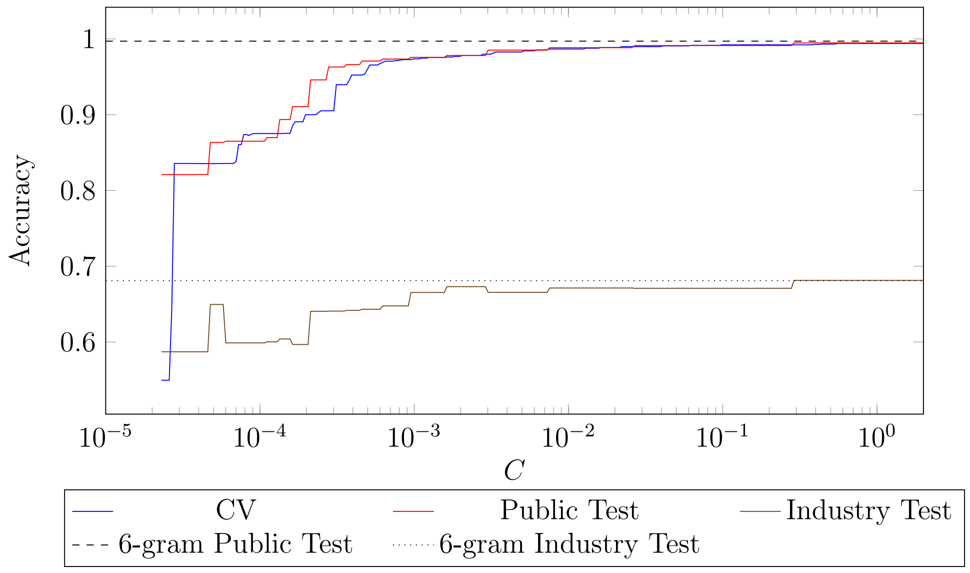


Figure 3.7: 3-fold CV and test set accuracy using string features. Trained using L_1 regularization from Public data

lected around 24,000. In Figure 3.7 we see the cross-validated and test set accuracies of training an L_1 regularized model on Public training data using string features. As C increases, we see the test set accuracies reach the same values as the n-gram results. However, in the same plot in Figure 3.8 we see the CV score go above previous results, while the test set scores both fail to reach the same accuracies achieved when trained with n-grams.

Based on these results, it seems reasonable to conclude that n-gramming does learn some non-string features, but can degrade to learning only string features when given poor data. This further calls into question all previous results collected in the same manner as our Public data. The gap in performance between n-grams and strings features trained from Industry does open the possibility that n-grams are learning some useful features that are not strings.

Another potential explanation of the performance gap is that exact extraction

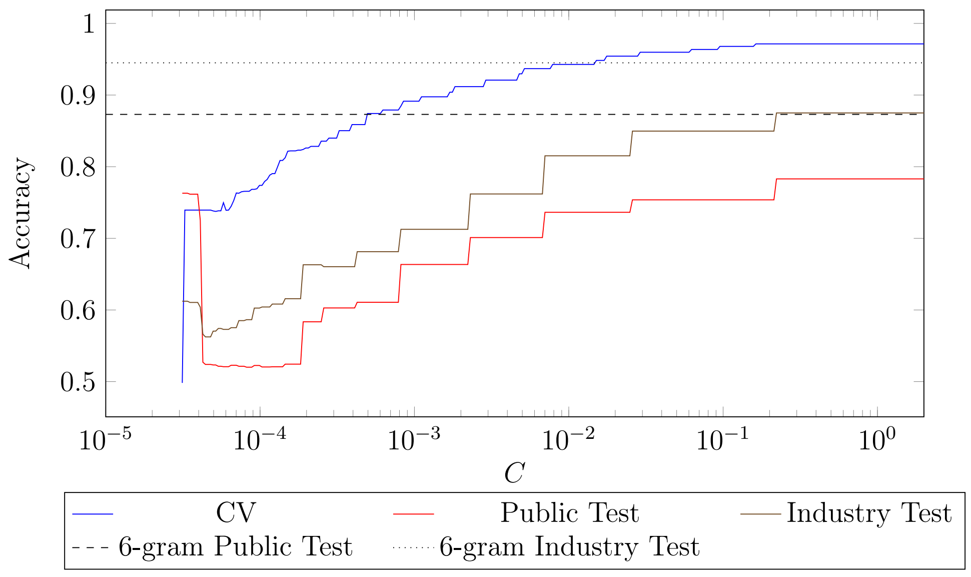


Figure 3.8: 3-fold CV and test set accuracy using string features. Trained using L_1 regularization from Industry data

of strings fails to capture certain information that is captured by n-grams of strings. Some sub-strings may be more generalizable than the whole string they come from. Many of our strings are also sub-strings of larger strings we extracted. A most-common-substring may better capture their relationship and match other instances appended with characters that we had not seen, and therefore would not catch. Normalizing by common sub-string could also move items up in total count and therefore warrant consideration in the model, when it would have been removed for being too infrequent before. A similar issue we see is strings that are mostly the same, but have an edit distance of one or two. In some sense, n-grams learning from strings better handle these problems by limiting themselves to a fixed size, though we must rely on the model building process to adequately select enough components of these strings and properly weight them. We have not yet tested this hypothesis as a detailed analysis of string-based features is beyond the scope of this work.

3.3 Discussion

We have performed a significant investigation into the performance of n-grams, and found them to be learning less and performing worse than previous work would have suggested. We hypothesize that n-grams, at least used on the whole executable, have a number of intrinsic issues that have not been adequately discussed.

First is the discovered issue that n-grams appear to be learning mostly from string content in an executable, and items from the PE header (which also contains strings). We believe this is an issue intrinsic to n-grams. While there are billions of potential n-grams, we need to select the features that occur frequently enough to occur in new data as well. Selecting with a predisposition to frequency then encourages us to select lower entropy features, which consist primarily of strings and padding. However padding alone is not a particularly strong signal, and as mentioned, strings could have been obtained in a much simpler fashion.

As mentioned in subsection 3.2.1, a possible preference for 6-grams may be that they are large enough to regularly capture a whole x86 instruction, but still fall short 2.4% of the time. This makes processing with n-grams problematic in creating a mismatch between our features and the data. This is important since, assuming no instruction in isolation is an indicator of maliciousness, we need to capture multiple instructions in a single feature. To even attempt to consistently capture three sequential assembly instructions we would need to produce and process n-grams of at least 12 bytes in length for most cases, and up to 45 for extreme cases. This alone is simply too computationally demanding a task to perform. Even if we had the computational resources to do so, there would still be a problem of trading off between specificity and generalization. We want our features to be large enough that they are not likely to occur by chance in a new file, but small enough that they are not specific to the training data as this would degenerate into a signature-based approach, which would have maximum specificity but no generalization. When considering the aforementioned frequency bias, it becomes even harder to imagine a large n-gram being selected by the model.

Another intrinsic issue is the loss of location information when using n-grams. Given our observed performance and the common use of string features, we believe that this limitation is likely playing a role in their weakness. For example, in our Public malware we observed that at least 5% of the data had inconsistent section names compared to the permissions set in the PE header (e.g. a section named `.data` but marked as executable in the header).[4] The occurrence of a feature in a non-executable section, when normally it would be found in an executable section, could have a significantly different meaning that would never be recognized by n-gram approaches. This type of location mismatch could occur both based on a section's string name in the header, and what the section's true identity is based on the permissions set.

Finally, we believe that the drop in generalization performance from our Industry model can be explained in part by an intrinsic brittleness to n-gram features. Regardless of what our n-grams learn, to apply them we must obtain an exact

[4]We did not perform an exhaustive test for consistency; this was merely a property found when testing some simple hypotheses. The true number could be much higher.

match when classifying a new datum. This means any minor change will make the feature "disappear" in terms of its impact on our model. Consider that we see `GetProcAddress` as a common MBI in Industry, indicating that this Windows function is a common feature of malware. We would then like to see our n-grams learn to identify the `call` or `jmp` instructions, followed by the address of the `GetProcAddress` function as a feature. However, in the import table to the PE the address of any function can be arbitrary defined, thus making it impossible for any of our n-grams containing a `call` or `jmp` plus an address to generalize to new files.

The other part to the lack of generalization is an intrinsic potential for over-fitting that will be found with n-grams. Estimating model parameters when the number of features is near or larger than the number of samples is a classic scenario for over-fitting due to the curse of dimensionality [81]–[83]. Since each file is one sample and based on the power-law observation of the n-gram distribution, we should reasonably expect that every new executable will give us thousands to tens of thousands of previously unobserved n-grams (depending on the size of the file). Thus we are in a scenario that will always produce many more features than samples. We selected Lasso and Elastic-Net for their robustness in this scenario and many of the features can be removed by frequency counts, but these techniques are not entirely immune to the curse of dimensionality.

3.3.1 On Using Multiple Training and Testing Sets

Because the issue of overfitting is critical to our results we also discuss why we have used multiple, independent datasets for our evaluation, which may seem unusual at first. This is important to understand, a a theme that will occur in multiple chapters of this work. We do this because the space of possible malware and goodware is extraordinarily large, and it is not possible for us draw a well-distributed random sample of real files from this space. Executables from Microsoft Windows are the most readily available source of benign files. Since these all come from the same source, they all share a strong common bias. Another collection strategy is to include commonly installed applications, such as third-party web browsers, as a source of goodware. This obtains only a few thousand EXEs, which is not enough to represent the larger population of different miscellaneous applications people may have. If such limited data were an adequate representation of the class of goodware then it would be easy enough to construct a white list of known safe applications.

It is easier to obtain large amounts of Malware thanks to resources such as Virus Share and Open Malware, and from the use of honeypots. Yet these sources are also biased. They are run by volunteers, and the samples are provided by volunteers. So each source already has an intrinsic bias in the malware that is provided by those who make the effort to do so. There may be individuals or organizations that see malware and are not able or willing to share it, and thus that data won't make it into the collection. Private data collections used by anti-virus corporations, such as our industry partner, are also biased by their data collection mechanism and contractual agreements with clients and customers.

Given all of these potential biases and the large space of possible EXEs, we must be careful in our inference of the true generalization of our results. Using multiple evaluation sets that avoid sharing common biases helps us to better determine generalization. For example, if Public was sufficient to learn the benign versus malware problem, we would expect a model's cross validated accuracy on Public and test set accuracy on Industry to be similar; however, it is not. Said another way, we want to see models trained on one dataset generalize to new datasets. If the model is able to generalize to a completely new dataset, that gives us higher confidence it will continue to perform well on new and novel malware. If it fails to do so, then we have little confidence that the model will continue to perform well against new and novel malware.

If we instead merged all our data into one larger dataset, and performed cross validation, the aforementioned biases will be in all folds, and we would obtain little information about the true generalization of the model. This is an issue with the assumption that data is independently and identically distributed. While the I.I.D. assumptions is rarely entirely true in practice, it is violated in too strong a manner for cross validation to be valid for our data. If it were appropriate to merge our Public and Industry data and use cross validation, we would expect to see similar performance across datasets and similar features learned. We have shown in subsection 3.1.1 that the performance is not consistent across datasets, and in section 3.2 that the features learned are different depending on the training dataset.

Having explained the importance of this evaluation strategy, we will use it repeatedly through this thesis. This gives us valuable information about the generalization of our models that would not necessarily be detectable otherwise.

3.4 Conclusions

Previous work has reported byte n-grams to be highly effective for malware classification. The goal of this chapter was to investigate this feature type and determine how it performed so well on such a difficult task. By applying and evaluating Elastic-Net and L_1 regularized Logistic Regression and the novel Multi-Byte Identifier to the n-gramming of a corpus of previously un-reported size, we have shown significant issues with the use of n-gram features. They are computationally expensive, exhibit diminishing returns with more data, are prone to over-fitting, and do not seem to carry information much stronger than what is more readily available from the PE header and ASCII strings. While n-grams do have some merit as a feature for executable files, their results have been significantly over-estimated in the literature. Overall, it seems that we could obtain the same performance using much simpler and more interpretable techniques. We have also observed a larger issue, to wit, many papers are evaluating with a corpus of benign files collected mostly from Windows installations, which is too simple a subset to accurately represent the goodware vs malware problem. Our work highlights the importance of investigating what a model has actually learned, rather than simply accepting held out or cross validated scores as true performance.

The information present in n-grams are still not fully explored, and may still be useful in a more restricted context. For example, while using only n-grams is not very effective, can n-gramming a limited portion of an executable be informative in conjunction with other features? It is also an open question as to whether n-gramming with domain knowledge could improve results, where n-grams extracted from the `.text` section of a EXE file are processed and treated differently then n-grams from other sections. Of course this may nullify one of the prime advantages of n-gramming, to wit, that it requires no domain knowledge to apply. Future avenues of research include evaluating the viability of n-grams in other domains, and exploring what other approaches may work for EXE files but require limited or no domain knowledge. Based on the information we have extracted, we plan to do an investigation in the use of n-grammed disassembled instructions combined with the header information that were used by the n-grams. The results based on our multiple data sources also leads to a larger open question: how do we evaluate the quality of an executable corpus?

While we have looked at the effectiveness of n-grams without domain knowledge. An intuitive follow-on item is to ask, how could n-grams be improved by the application of domain knowledge? We will investigate this in the next chapter.

Chapter 4: N-Grams With More Domain Knowledge

In the chapter 3 of this work, we performed an extensive investigation into the use of byte n-grams for malware detection. In this chapter, we extend this investigation to answer the question: can n-grams be improved via the application of domain knowledge? The surprising answer is that predictive accuracy may actually decrease when applying some obvious “better” approaches to the use of n-grams in a static analysis setting. This non-intuitive result highlights the potential difficulties in applying domain knowledge to the problem of malware detection, and highlights why we wish to have domain-knowledge free alternatives.

4.1 Introduction

In this chapter we provide two primary contributions to the understand of what n-grams can learn, both for byte n-gramming and assembly n-grams. First, we provide evidence that byte n-grams are able to learn from the code sections of a binary if forced to learn from *only* the code sections. However, the information learned from the code regions seems to be highly correlated with that of the import and header sections of PE binaries, suggesting the actual amount of information learned about the assembly instructions may be limited. Further, subdividing the byte n-grams into section types based on the PE header reduces predictive accuracy. Second, we note that the discriminative ability of assembly n-grams is less than that of byte n-grams from the code section. By using multiple datasets that do not share common biases, we observe that assembly n-grams seem to suffer from severe over-fitting, by being unable to generalize past data similar to the training distribution. This suggests that the disassembly process and n-gramming strategies commonly used may be losing important discriminative information, and not learning what was previously thought. The overall takeaway from this extended investigation is that there is, contrary to conventional wisdom, considerable predictive ability in using domain knowledge free approaches. Our assumptions about what should work better may be wrong in significant ways, leading to worse performance for Windows EXE data and restricting the approach to only Windows EXEs. This is not to say that adding domain knowledge can not our should not improve model performance. The point is that it is not a forgone conclusion, and the application of domain knowledge for malware detection is subtler and more difficult to use effectively.

The remainder of this chapter is organized as follows. In section 4.2 we review relevant related work. Next we give an overview of the byte n-gram approach in section 4.3, and detail which sub-regions we use and how we extracted them. In section 4.4 we will discuss the different ways in which assembly n-grams may be extracted, and introduce an additional novel representation. In section 4.5 we will review the classification algorithms that will be used in this work and training and parameter tuning procedures. The evaluation methodology and results will be presented in section 4.6, followed by a discussion of those results and conclusions in section 4.7 and section 4.8 respectively.

4.2 Related Work

Many works have looked at the use of byte n-grams for malware detection, and were considered in the first work on Microsoft Windows malware detection Schultz, Eskin, Zadok, *et al.* [1]. One of the most thorough previous investigations of byte n-grams was done by Kolter and Maloof [16], who evaluated several different linear and non-linear classification and feature selection algorithms to use with byte n-grams. A number of works follow the same general approach of Kolter and Maloof when using byte n-grams: pick a feature selection method similar to Information Gain on 4 or 6-grams, followed by a non-linear classifier [20], [21], [62]. Work by Raff, Zak, Cox, *et al.* [41] sought to explore this feature type more deeply, and discovered evidence that prior results were overfitting. We use the model building strategy suggested in their work in ours, using byte 6-grams and elastic-net regularization to perform implicit feature selection with a linear model.

This same general strategy has also been used for assembly (and opcode) based classification. In the static analysis case, the binary is disassembled using some tool, such as IDA Pro, and the extracted assembly features are used to create n-grams. It is often the case that creating n-grams from a whole instruction (with its arguments) is not efficient, and so a number of variants have been used in practice, which we discuss more in section 4.4. This basic strategy is similar to the byte n-gram approach but at a higher level of representation, and has been popular in practice [84]. Also popular for assembly instructions is to use Hidden Markov Models (HMMs) [85], [86]. The Markov assumption is that one item in a sequence can be predicted with information from only the previous m items. Thus a HMM approach of the m^{th} order has many functional similarities to the n-gram approach explored in this work.

There have been a few works attempting to combine assembly n-grams with byte n-grams and other feature types. These works have all focused on building a system with higher accuracy, where in our work our goal is to determine what kinds of information are being learned or used. Masud, Khan, and Thuraisingham combined these with function imports into one larger feature set, and found it to perform better than binary of assembly features independently. On one of their datasets, they report accuracies of 96.5%, 94.6% 87.1% when using the combined features, byte, and assembly features respectively. They obtained similar performance for both boosted decision trees and Support Vector Machines. Masud, Khan, and Thuraisingham's work is similar to our own in combining features of different types, though it differs in method and reasoning. No analysis was given in their work to determine which type of feature had more or less impact on the improved performance. Similarly, Menahem, Shabtai, Rokach, *et al.* [22] combined a wider array of feature types and classification algorithms into one larger ensemble to maximize performance, but did not attempt to investigate which features contributed in which ways. Though not combining feature types, Yan, Brown, and Kong [87] also looked at both byte and assembly n-grams for malware detection. They performed a wide search over feature selection and classification methods to determine which configuration worked best, but used only 300 binaries for their experiments.

The aforementioned works all used static analysis, as we do in this work. It is also possible to obtain assembly instructions via dynamic analysis, which has been popular as well. More closely related to our work, Damodaran, Troia, Visaggio, *et al.* [86] looked at using assembly instructions to train Hidden Markov Models (HMMs) from both static and dynamic analysis (among other approaches as well). While they did not perform the same type of malware detection, they found dynamic analysis could increase the detection effectiveness, as measured by Area Under the Curve (AUC), by as much as 20 percentage points. They also found that dynamic analysis reduced the number of distinct opcodes observed, which reduces the training time and indicates that many instructions present in a static analysis may not be relevant to functionality.

In conducting the literature review for this work, we did not find any previous malware detection work that uses data similar to Industry (i.e., production data from a corporation) and uses static assembly features[3], [12], [88], [89]. This is important, as most works use benign data from Microsoft Windows installations, which can result in overfitting to the concept of "Microsoft vs not-Microsoft". We believe the overfitting that occurs when using Microsoft binaries for training and testing may be the root of positive results with assembly-grams for malware detection that have been previously reported. This gives us some optimism that the issues we discover is not a problem with our data, but that a weakness with assembly-gram features was not published due to a bias against negative results [90], [91]. The lack of publicly available, high quality, datasets for this task will be a hindrance toward reaching a consensus on this issue.

4.3 Sectional Byte N-Gram Features

The primary effort of this work looks at byte n-grams and what types of information can be learned from them. To investigate this, we perform byte n-gramming on sub-sections of the PE file that correspond to different regions and thus types of data. The PE format specifies a variety of different sections types for storing information needed for the program to execute. Some typical sections found in many PE files are: `.text` for the section of an EXE that contains executable code, `.data` data for initialized variables, `.rodata` data for read-only variables, and `.idata` for the import table.

However, the name for any given section is arbitrary and does not impact how a section is loaded or used. Some compilers put the executable code in a `.code` section instead of the traditional `.text`. This makes the section name an unreliable method of determining type. Instead, one can use the information encoded in each section to more accurately determine a section's purpose. Each section has a number of flag bits which indicate properties of the section, particularly if it is: (1) Executable, (2) Read-only, or (3) Read-write. Newer versions of Windows don't allow a section that is marked executable to be also marked as read-write. To separate out the executable section, all that is needed is to find sections marked as executable. Most files have just one such section, although some have two or more. In the case of UPX-packed binaries, the executable sections are `.UPX0` and `.UPX1`. Using these section flags, and the other fields of the PE header, we perform byte n-gramming on four high level section types: PE-Header, data, imports, and executable code. We obtain the bytes for these four sections in the following manner.

For the parsing of the PE files, we use the PortEx library **PortEx** to discover all sections and the section offsets in the raw file. This library was also used to process the section bit flags needed for the other portions of this work. Once identified, we concatenated all the bytes associated with the PE-Header into one longer sequence. This sequence was then used as the PE-Header feature source for byte n-grams. It generally corresponds to low-entropy information, but is encoded in variable length fields (some fields are single bits, some are multi-bit flags, and some are integers varying between 4 and 64 bits in length). This makes it a good match for byte n-grams in terms of being low entropy, but a poor match in terms of the variable length nature with respect to the fixed size n for n-gramming.

For the executable sections of a binary, we checked every section within the binary for whether or not its executable bit was set. All sections found with such a property were concatenated together. For most files there was only one section marked executable. This corresponds to the theoretical worst case for byte n-grams. The contents of the executable regions are higher entropy, and the encoding of x86 instructions is variable length. Our expectation is that this section will have the worst performance.

The remaining two region types we are interested in are imports and data. In general, the vast majority of sections that were not marked executable corresponded to either a data section or an import section. For benign applications, it is usually easy to separate the two, as the PE-Header will point to the section in which imports are stored. This was not true in all cases for goodware though, and was rarely true for malware. For example, many samples had the import table address point to an address beyond the file size, to areas partway into an existing section, or areas that did not appear to contain any imports at all after manual inspection. If the import table address was present, the address was used as an offset into the section. From the offset to the end of the section was treated as the import table.

To remedy the situation when the import table offset wasn't valid, we used rudimentary string matching to detect regions of the binary that appeared to be containing import information. This was done by comparing the byte content with frequent DLL and function names, and deciding that these sections are import sections. This approach successfully extracted 90% of the imports from the non-executable data, which was verified by manual inspection of randomly selected binaries.

Finally, after identifying the byte sequences corresponding to the PE-Header, executable, and import sections of a binary, all other sections were assumed to be data sections. This gives us all four regions of interest for our experiments. We note that these extractions are not perfect, but are more than accurate enough to allow for informative experiments.

4.4 Assembly N-Gram Features

Before creating n-grams of assembly instructions, we must first select a subset of base n-gram representations to choose from. In assembly code, each line is generally represented by the instruction name and a number of parameters for the instruction. Just as n-grams are a sliding window of consecutive bytes, we define n-grams of assembly as a sliding window of *lines* of assembly code. A number of options have been proposed, which we will review below.

The relationship between byte and assembly n-grams has been noted before [23], [84]. An important consideration not widely discussed is that not all instructions with the same name map to the same binary opcode[1]. To illustrate, the `cmp` instruction's binary opcode can begin with 0x3C, 0x3D, 0x3A, 0x3B, 0x80, 0x81, 0x83, 0x38, or 0x39, depending on the arguments given. In this and all previous works in malware detection known to us, these are all treated as the same instruction based on the common `cmp` name. We will refer to distinguishing instructions based on their binary opcode as disambiguation. Little work has been done with such disambiguated opcodes in related tasks of malware family classification[92] and function identification[93], but neither work quantifies the importance or significance of using opcodes. We perform the first such comparison to determine the impact of disambiguation assembly n-grams in subsection 4.6.3, where we will show their impact is critical to obtaining generalizable results.

4.4.1 Instruction Only

The simplest approach is to capture only the instruction used as the base. If encountering the instruction `mov eax, 4` we simply reduce it to `mov`. This ap-

[1]Some works have used the term "opcode" to describe the instruction name, such as `cmp`. We avoid this terminology, and instead use "opcode" only to refer to the binary encoding of the instruction.

proach is used by Shabtai, Moskovitch, Feher, *et al.* [84]. They argue that this representation will generalize better, as small perturbations in the arguments (due to a change in location) can be functionally equivalent, but no longer found by an n-gram. For brevity, we will refer to this form of assembly n-grams as “OI” for “Only Instructions”.

4.4.2 Instructions with Parameter Type

This method was used by Masud, Khan, and Thuraisingham [23], and generally appears to be a common preference [94], [95]. They noted that an instruction will have some number of parameters and each parameter is coalesced into a location type, either memory, register, or constant corresponding to where the parameter came from: either an access to memory, directly from a register, or the immediate value from the call to an instruction. For example, the instruction `mov eax, 4` would be coalesced to `mov.register.constant` and `mov [eax], 4` to `mov.memory.constant`. We note that in this form it does not matter that a register was used in the first parameter, it is that the parameter came from a memory accesses that determines the type. We will refer to this form of assembly n-grams as “IPT” for “Instructions with Parameter Type”.

4.4.3 Instructions with Function Resolution

Many works have noted that the APIs called also have predictive power [23], [96], and we have found that byte n-grams tend to pick up on these features as well[41]. Inspired by these observations, we developed a novel feature representation of assembly where all matching constants are replaced with the function name being called, when available from the import table. All other operands are left in their raw form, and we attempt to match exact instruction sequences, with numerical constants replaced by the function name when a match is detected. This is a more viable alternative to using the raw pointer values, as the pointer to a function may change from one binary to the next, even if calling the same function. Our shorthand for this type of assembly n-gram will be "IFR" for "Instructions with Function Resolution". Doing so allows us to perform tests using as much of the raw disassembly as possible, and reducing many instructions to a canonical form. These instructions sequences are logically equivalent between binaries, but would not have been matched correctly if the addresses were not resolved.

4.5 Machine Learning Models

Now that we have reviewed the features that will be used in this chapter, we discuss the method of using them. At a base level, we will be continuing to use the Elastic-Net regularized Logistic Regression from the previous chapter. This was defined in (3.8), and has the benefit of automatic feature selection and robustness to irrelevant features [72]. For simplicity, we will only investigate $\alpha = \frac{1}{2}$ in this chapter.

More important, we introduce the ensemble method we will use to combine models for our byte n-gram results. Creating ensembles is a common method in

```
call 0x401040 ; address that was not found in the import table
lea edx, [esp + 0x20]
lea eax, [esp + 0x120]
push edx
push 0x4c2244
push eax
call 'MSVCRT.dll:sprintf' ; direct call to function
mov ecx, 0x40
xor eax, eax
lea edi, [esp + 0x2c]
add esp, 0x14
rep stosd es:[edi], eax
mov edi, 'KERNEL32.dll:GetPrivateProfileStringA' ; indirect call to
 ↪  function, first loaded into edi register
lea ecx, [esp + 0x118]
push ecx
lea edx, [esp + 0x1c]
push 0x100
push edx
push 0x4cceb0
push 0x4c223c
push 0x4c223c
call edi ; then called via register later
```

Figure 4.1: Excerpt of our diassembly with function resolution

machine learning to produce a more accurate model by exploiting the uncorrelated errors made by members of the ensemble [97], [98].

Stacking

Given that we want to understand if the information being learned in each section is different, or some variant of the same information, we also apply the Stacking ensemble technique [99] to combine the models from all four byte regions. When performing Stacking, we have a set of base classifiers that make the ensemble, which are all trained independently. The predictions of these classifiers are then used to create a new feature set, of dimension equal to the number of base models used. Any other classifier can be used as the combiner, which uses this new feature set to learn the same problem. The combiner model can be as simple or complex as desired. It is common to use a linear model for the combiner, in which case stacking learns what is essentially a weighted average vote of the constituent base classifiers.

Stacking is often an effective method to increase the predictive performance for a problem, at the cost of using multiple models (and thus more memory and compute time). Though the connection to stacking was not made, the strategy has been applied to malware detection before [100]. Like most ensemble methods, it relies on the base classifiers having some degree of variation and performs best if their errors are uncorrelated. We tested this with a number of combiners, including Random Forests (RF) [101], Linear Support Vector Machines (SVM) [102], and a simple Neural Network (NN). Given this wide array of stacking models, if the errors are uncorrelated and useful, either in a linear or non-linear way, we should see a boost in performance.

4.6 Evaluation and Results

As in the previous chapter, we will use the same Public and Private data corpora. All models will be evaluated with both balanced accuracy [103] and Area Under the ROC Curve (AUC) [104]

During feature processing, many binaries could not be disassembled. Many of these errors occurred due to the binary in question having no sections as being marked executable. These binaries ended up being DLL files with translation strings for localized copies of Windows, or DLL files containing icons or other data for applications. Some errors can also occur due to the dissembler erring on challenging inputs. Any file with such an issue was removed from both the training and testing datasets. Since balanced accuracy and AUC are not sensitive to class proportion, we can meaningfully compare those metrics with the results from our byte n-gram experiments. We note that we have confidence in our extracted disassembly, as we are able to resolve addresses across binaries to which function they are calling. These would not resolve or make sensible disassembly if we had an error in our disassembly process. We also removed .Net files, as these files don't contain machine readable code, but rather are interpreted by the .Net or open-source Mono runtime environments.

4.6.1 Byte 6-Gram Results

We use byte 6-grams for our evaluation as they were found to perform best compares to 4-grams, and larger values are beyond our computational capacity. We compare using byte 6-grams on the entire file as the control, versus byte 6-grams extracted from one of only four sub-regions of the binary. All five models were trained in the same manner as outlined in Raff, Zak, Cox, *et al.* [41], and the results can be found in Table 4.1. This also includes the results of using Stacking to combine the four different section types with various different combiner models.

We note that in all cases the 6-grams of the entire binary performed best in both accuracy and AUC. For the second best model, the PE-Header was best when

	Public		Industry		Open Mal
PE-Section	Acc (%)	AUC (%)	Acc (%)	AUC (%)	Acc (%)
PE Header	76.4	*98.2*	87.7	*95.6*	53.9
Data	69.6	94.9	84.4	92.9	52.7
Import	*83.8*	92.9	*88.7*	94.0	*74.3*
Code	80.5	94.6	88.1	95.2	60.7
Whole File	**87.0**	**98.4**	**92.5**	**97.9**	**81.2**
Stacking SVM	80.8	78.4	87.8	88.4	56.2
Stacking NN	83.1	81.4	89.0	89.8	60.7
Stacking RF	83.6	81.6	89.8	90.4	62.9

Table 4.1: Accuracy and AUC when using byte 6-grams. First four rows are results using 6-grams from only one section of a PE file, with the fifth row showing result from 6-gramming the whole file. Best numbers in **bold**, second best in *italics*. Last three rows show results when using Stacking (with different models for the combiner) to combine models from all four feature sections.

using the AUC metric, and the Import section best when considering accuracy. While it may seem unusual for these metrics to differ, this is not an uncommon scenario [105]. The best sub-region for 6-grams region was generally 3 to 5 percentage points behind that of using the whole file. We also note that the near 50% accuracies for Open Malware are not indicative of the model randomly guessing, as the Open Malware set contains only malware. It is likely those models are biased towards declaring a binary as benign, and so would get considerably higher accuracy rates if there were benign files to include with the Open Malware files. This is confirmed by looking at the precision on other datasets. For example, the PE-Header model had a precision of 99.7% and recall of 72.8%.

Regarding the hypothesis that byte n-grams only learn from the import and header sections, our evidence argues against this hypothesis . The 6-gram models were able to learn reasonable models from all four section types, though the models from the Import and PE-Header were the ones that performed best. That these two sections would perform best is not unreasonable, as the lower entropy content of these regions means that an n-gram found in the training set is more likely to be found in the testing set.

What is more interesting is that the Stacking models, which combine the predictions of each of the four section-based models into a final prediction, generally perform worse than models built on only the import and PE-Header sections. Due to how we portioned each binary into the four sections, the entirety of information available to the Stacking model and a byte-gram model trained on the entire binary should be equivalent. Theoretically, the Stacking model has an advantage in intrinsic information about region type from the constituent ensemble members.

The decrease in test accuracy from ensembling suggests that the predictions of the models are highly correlated, as we would expect performance to increase if the predictions were uncorrelated. Differences in prediction output (and confidence) then act only as a noise in the decision process, rather than as signal that helps improve accuracy. We suspect that this means the 6-grams from the code section of a binary are learning the same kind of information that is contained within the Import and Header sections of the binary. This information leakage could potentially be assembly instructions or operands that are correlated with particular functions or settings that may be found in those sections, respectively.

4.6.2 Assembly N-Gram Results

Given that we have evidence that byte n-grams can learn from the code sections of a binary, we wish to compare and understand the performance differences in what byte n-grams from code learn and what n-gramming of the disassembled code sections. We evaluate the three assembly n-gram strategies discussed in section 4.4 on the same datasets, using Industry for training. We emphasize that an assembly n-gram does not correlate well with a byte n-gram in terms of how much information is captured in a single features. A single assembly instruction can range in size from one byte to an extreme of 15 bytes, and so we do not concern ourselves with trying to compare bytes vs assembly based on the value of n. For each assembly n-gram type, we evaluate up to and including the largest value of n that we could manage in terms of memory on our workstation[2].

The results for all assembly grams are given in Table 4.2. The most striking result of these accuracy numbers is a dramatic drop in performance compared to the byte 6-grams. We also observe considerable overfitting when using the assembly features, where the Industry test set performance is reasonable (yet still lower than the byte grams), and drops in accuracy and especially AUC when evaluated against the other test sets. Our expectation would have been that assembly grams would perform equal to or better than byte grams of assembly, as the disassembled version is a higher level representation of the data. Assembly-grams obtaining an AUC lower than 50% (which would be the threshold of random guessing), combined with the

[2]The workstation used for this experiment has 128GB of RAM

Assembly-gram		Public		Industry		Open Mal
Type	n	Acc (%)	AUC (%)	Acc (%)	AUC (%)	Acc (%)
IFR	1	67.6	44.0	76.5	86.8	30.0
	2	67.9	42.6	78.1	88.5	35.7
IPT	1	64.0	37.8	69.0	78.6	31.6
	2	68.1	44.2	76.8	87.3	36.6
	3	67.1	41.1	76.8	87.8	34.6
	4	64.8	36.0	76.3	87.9	20.9
OI	1	61.6	33.3	64.3	70.3	30.8
	2	65.0	38.4	73.6	85.4	26.5
	3	66.7	40.6	77.8	88.6	28.7
	4	65.5	37.5	77.6	88.3	24.6
	5	64.7	35.9	76.1	87.2	21.5
	6	64.0	34.2	75.5	87.3	19.2

Table 4.2: Balanced Accuracy and AUC for each test set, with models trained on Industry. Using assembly n-grams of varying types.

behavior of marking most binaries as benign, indicates the assembly model trained on Industry has no actual generalization to the other datasets.

Given this surprising result, we hypothesize a number of ways in which discriminatory information may have been lost for assembly-grams compared to byte-grams. These stem from differences in assumption between performing byte n-gramming and assembly n-gramming.

First we note, as discussed in subsection 4.4.1, that different byte op-codes get mapped to the same higher level assembly instruction when performing the disassembly process. It is possible that the specific version of an instruction is in fact discriminative, and is thus lost when using the assembly grams. We will test this first hypothesis in the following section.

Second, we observe that byte n-grams may start and end in the middle of an assembly instruction, where as an assembly n-grams will cover exactly n assembly instructions. This gives byte n-grams an odd form of flexibility and specificity. A byte-gram could start at one instruction, and reach into only the op-code of the next instruction (touching some or none of the operands). Or a byte n-gram could start in the middle of a instruction, considering the lower order bits of the operands of one instruction and then whole or part of the preceding instruction.

4.6.3 Assembly-Grams with Disambiguation

Existing code infrastructure allows us to test the importance of opcode disambiguation with relative ease. The Capstone Engine API we used for disassembly allows us to obtain the first byte of the opcode for a particular instruction, and so we can produce an "enhanced" disassembly, an example of which can be seen in Figure 4.2. While the first byte of the opcode may not be sufficient in all cases, it already allows us to distinguish between multiple different versions of the same instruction. We treat these as a new instruction set, and repeat our assembly experiments on the same data.

We note that the disambiguation necessarily increases the size of the feature space. This intrinsically makes learning harder for the algorithm, as the impact of the curse of dimensionality is only increasing. Thus any additional discriminative information from disambiguation must be non-trivial in order to increase performance. This also increases computational burden and memory use, which prevents us from testing n-gram sizes as large as the preceding section.

```
jmp_eb 0x4010eb
push_68 0x10024b78
lea_8d ecx, dword ptr [esp + 4]
call_ff dword ptr [MFC71.DLL:None]
push_53 ebx ; three different pushes
push_56 esi
push_57 edi
push_68 0x10024c05
lea_8d ecx, dword ptr [esp + 0x14]
call_ff dword ptr [MFC71.DLL:None]
lea_8d ecx, dword ptr [esp + 0x24]
mov_bb ebx, 1
push_51 ecx
mov_88 byte ptr [esp + 0x20], bl
call_e8 0x41f8ec
mov_8b edx, dword ptr [eax]
```

Figure 4.2: Example of disassembly with opcode disambiguation. Note that it is now clear four different `push` instructions are being called, two different `call` instructions, and three different `mov` instructions.

Assembly-gram		Public		Industry		Open Malware
n-gram type	n	Accuracy (%)	AUC (%)	Accuracy (%)	AUC (%)	Accuracy (%)
IFR	1	65.5 (+2.1)	69.2 (+25.2)	78.1 (+01.6)	86.4 (−00.4)	48.4 (+18.4)
	2	69.8 (+1.9)	69.2 (+26.6)	78.8 (+00.7)	85.4 (−03.1)	44.7 (+09.0)
IPT	1	66.1 (+2.1)	73.7 (+35.9)	78.1 (+09.1)	88.4 (+09.8)	46.9 (+15.3)
	2	70.8 (+2.7)	73.7 (+29.5)	80.4 (+03.6)	90.9 (+03.6)	42.9 (+06.3)
	3	66.0 (−1.1)	66.9 (+25.8)	79.6 (+02.8)	90.4 (+02.6)	31.1 (−03.5)
OI	1	61.9 (+0.3)	69.4 (+36.1)	74.4 (+10.1)	84.5 (+14.2)	46.1 (+15.3)
	2	69.7 (+4.7)	72.9 (+34.5)	76.6 (+03.0)	89.9 (+04.5)	43.8 (+17.3)
	3	65.8 (−0.9)	68.3 (+27.7)	80.1 (+02.3)	91.3 (+02.7)	39.2 (+10.5)
	4	64.9 (−0.6)	63.5 (+26.0)	77.6 (+00.0)	89.2 (+00.9)	26.9 (+02.3)

Table 4.3: Balanced Accuracy and AUC for each test set, with models trained on Industry. Using assembly n-grams of varying types with partial opcode disambiguation. Difference in scores from Table 4.2 in parentheses.

The results can be seen in Table 4.3, where the disambiguated opcodes had an almost uniformly positive impact. Most notably, the AUC on the Public test data improved dramatically, by at least 25 points in every case. A large positive impact was also obtained on the Open Malware test set, and in general for every statistic on every test set when considering 1-grams. This indicates that the specific opcode of the instruction, and not just the instruction type, contains significant discriminative information for malware detection. The disambiguation has improved assembly grams from overfitting to the training data with almost *zero* generalization ability, to being able to show moderate generalization, but still subject to a non-trivial amount of overfitting.

4.7 Discussion

We have now tested byte n-grams by section type, to help us better understand what byte n-grams have learned. In this process we also tested assembly n-grams as a comparison point to byte n-grams of the executable sections of a binary. In doing so we have discovered a number of interesting results not previously reported, as far as we are aware, for both byte and assembly n-grams.

By byte n-gramming different sections of the binary, we were able to show that they can learn discriminative information from executable regions, which prior work hypothesized was not possible [41]. However, a surprising result is that they do not appear to be learning much about the code contents, but rather, leaked information about imports, strings, and any other lower entropy feature content that was discovered previously. We can draw evidence for this conclusion from the lack of improved accuracy (and in-fact, degraded accuracy), when creating an ensemble of classification models. If the information content used was different, errors should be uncorrelated with byte n-grams from other regions, and thus result in an improved model. In future work we hope to test and better understand the correlations between different types of feature information. One hypothesis as to how this information may be leaked, is that certain imports are strongly correlated with certain code patterns that get reused. This may not be broadly informative about higher level information such as malware author or source language, but are correlated enough with the imports to allow use as a proxy for the import itself.

Ultimately, it seems that byte n-grams are robust in their ability to learn, but weak in what types of information they are able to learn. That is to say, our results seem to confirm that byte n-grams are beholden to using certain types of low entropy information that can already be more easily extracted from the imports and PE-header sections of a binary. But regardless of where byte n-grams are applied, if such information exists or is leaked from other features, it appears byte n-grams will be able to find, extract, and use that information (though with potentially reduced effectiveness).

Somewhat more surprising, and not part of the original goal of this work, was what was learned about the effectiveness of assembly n-grams. It appears that assembly-grams, at least when obtained from static analysis, are uniformly less effective than byte n-grams. This is due to significant overfitting to the Industry data distribution, with a failure to generalize well to the other test sets. We have obtained significantly improved results by incorporating opcode disambiguation, a strategy which we are not aware of any prior works using for malware detection. Even with our improved disambiguated instructions, their performance still lags that of byte n-grams on the executable sections of a binary.

This result is counter to our intuition, as we would believe disassembly to be raising the feature representation up to a higher level, and thus making the job of the learning algorithm easier. Yet our disambiguation results clearly indicate that this may be inadvertently hiding important discriminative information. It is also possible that the higher level representation afforded by assembly grams is in some way enabling greater overfitting to the original data. If so, this could indicate a bias in the Industry data that is highly specific.

As discussed in subsection 4.6.1, byte n-grams also have the unique property that they do not care about instruction alignment. It is thus common to have a byte n-gram that starts in one instruction, and ends in another. It is possible that this accounts for the remainder of the performance gap between byte and assembly n-grams, and we hope to explore this in future work. Given that a byte 6-gram can be representing less information than an assembly 1-gram, we suspect that this scenario is a significant component of byte n-gram's learning ability when forced to learn from only the code section of a binary.

Our results are also impacted by the existence of files that could not be disassembled, which has happened before [106]. There may also be files with varying portions of erroneous disassembly, as disassembly of malware is not trivial. The difficulties and potential obfuscations that can prevent accurate disassembly is recognized as a challenging problem [107]–[111]. Others have noted there are different methods to performing disassembly that may produce differing results [87], with no one method being necessarily "better" than another. It seems clear from our work that a more thorough investigation of assembly n-grams is warranted, including use of different disassemblers, using disassembly obtained from dynamic analysis (which was not studied in this work), figuring out the best method for handling failure cases, and collective impacts these situations have on malware detection.

It would also be good to further evaluate assembly n-grams in the context of malware family classification. While some have used assembly-grams successfully for this task before [85], [112], a more thorough investigation is warranted. In particular, we hypothesize that assembly-grams may be more effective for family classification then for malware detection. We come to this theory by noting that the Industry test accuracies, which are from the same training distribution, provide reasonable performance. If assembly features processes greater specificity, this may be advantageous in family identification, where the classification labels are intrinsically more specific than the broader benign vs malicious task we have evaluated.

4.8 Conclusions

Following our results from chapter 3, we have delved further into understanding what types of information they can learn from Microsoft executable binaries. In doing so we make two unexpected conclusions. First, that byte n-grams can learn from higher entropy regions, such as the code section, of a binary — though it may not be learning much information beyond than what is found in the PE-Header and Import sections. Getting byte-grams to learn from these sections required limiting them to only these sections. Second, that assembly n-grams do not appear to generalize to new data, performing worse than byte n-grams learned from the code regions of a binary. Further, that the standard approach to creating assembly-grams is throwing away useful discriminative information.

These results help to further elucidate the need for new approaches to learning without domain knowledge. The methods used in this chapter represent “obvious” approaches to “improve” classification performance for malware, yet result in degraded accuracy — dramatically so on the case of assembly features. As such we now have evidence that the common wisdom of building malware detectors has been lead astray by the use of Public style data. For the remainder of this thesis, we will show that forgoing these assumptions will allow us to build a system that is more flexible, faster, and more accurate.

Chapter 5: Lempel Ziv Jaccard Distance

The Normalized Compression Distance (NCD) [35] 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 chapter. 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 [36], [38]–[40]. 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 [113].

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 [114], [115]. 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 meet, it is often difficult to meet these conditions in practice [116]. The nature of how compression algorithms work also causes problems for larger input sequences [37]. 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 [39]. We resolve the runtime and metric issues of NCD with the new Lempel-Ziv Jaccard Distance (LZJD), which is a valid distance metric (i.e., obeys identity, symmetry, and triangle inequality properties) and computationally efficient enough to use in practice. We perform extensive validation of our new technique by using 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. 1], [16], [68], [117].

The remainder of this chapter is organized as follows. We will review the definition of NCD in section 5.1, and then introduce our new distance metric in section 5.2. Focused on malware classification, we provide several experiments in section 5.3 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 5.4. Our conclusions are then presented in section 5.5.

5.1 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, Chen, Li, *et al.* [35] define the Normalized Information Distance (NID) (5.1), which returns a distance in the range $[0, 1]$.

$$\text{NID}(x, y) = \frac{\max\left(K(x|y), K(y|x)\right)}{\max\left(K(x), K(y)\right)} \tag{5.1}$$

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, Chen, Li, *et al.* [35] 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 (5.2), where $C(xy)$ indicates the compressed size of sequences x and y concatenated together.

$$\text{NCD}(x, y) = \frac{C(xy) - \min(C(x), C(y))}{\max(C(x), C(y))} \tag{5.2}$$

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 [118] and similar compression algorithms tend to work best [36], [37]. 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, Chen, Li, *et al.* [35] 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 [37], [116]. It is even the case that NCD often returns values larger than the theoretical maximum distance of one [119]. 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 (5.2). 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 [115], [120]. The use of compression distances also has strong ties to Machine Learning, where compression distances can be seen as a new feature space[121] and the concept of compression can be used for learning bounds[122], [123]. 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 [121]. Given the wide success of NCD, we seek to address its major issues of computational overhead and lack of metric properties.

5.2 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[124], [125]. 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.

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 $end \leq |b|$ 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
```

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 [124] 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 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,

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} \tag{5.3}$$

The Jaccard similarity is the cardinality of the intersection of two sets divided by the cardinality of their union (5.3). 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),

$$\text{LZJD}(x,y) = 1 - J(\text{LZSet}(x), \text{LZSet}(y)) \tag{5.4}$$

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.

5.2.1 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 [126]

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

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 [127]. 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.

5.3 Experiments

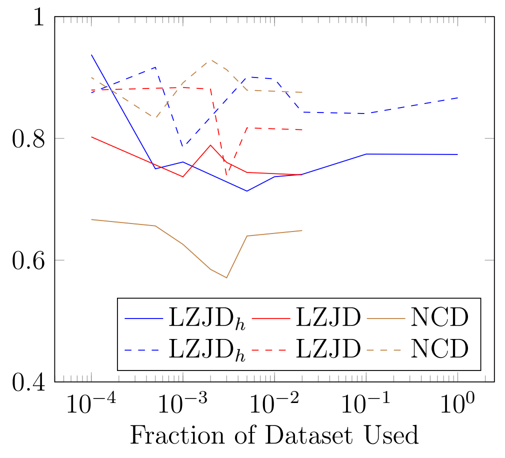
Having defined the LZJD distance, we describe a number of experiments which show that LZJD is competitive with NCD in terms of quality of results, while having superior run-time characteristics. In all experiments we will apply NCD and LZJD to the raw byte contents of a binary as our features, unless stated otherwise. We will generally use the k-Nearest Neighbor algorithm (k-NN) [128] 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[1]. 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.

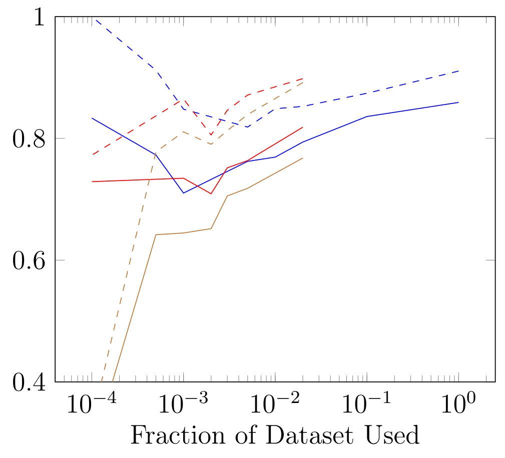
We implemented LZJD and LZJD$_h$ in Java, without any significant attempt

[1] `https://tukaani.org/xz/`

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.



(a) Results on Public test set



(b) Results on Industry test set

Figure 5.1: Balanced Accuracy and AUC on the y axis, presented for the Public and Industry 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.

5.3.1 Microsoft Malware Detection

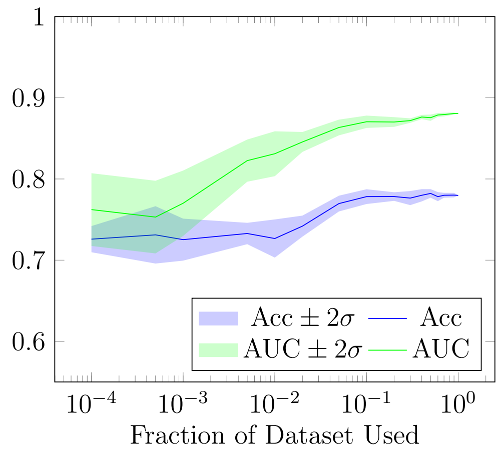
We first demonstrate the performance of LZJD with the task of malware detection, where we try to distinguish between benign and malicious binaries. While most prior results have restricted the use of NCD to data-sets of 2000 or less samples, we use the much larger data from Industry for training, and test on Public, Industry, and Open Malware test sets. We will report run-times when using Industry training data, and evaluating against all 3 test sets. These 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 [103] 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 $k = 1$.

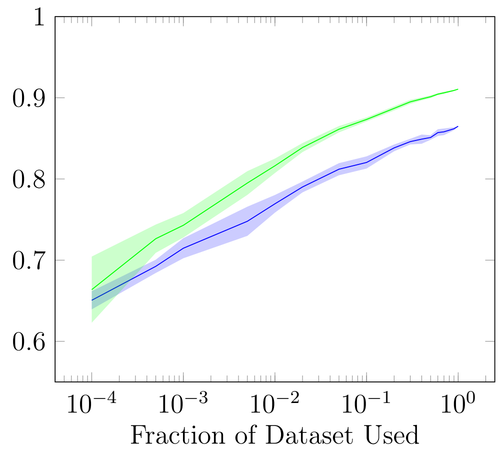
In Figure 5.1, we can see the balanced accuracy of all three distances run on training subsamplings of varying sizes. We use the balanced accuracy where each class receives equal total weight in the calculation, to ease comparisons across datasets. As can be seen, the LZJD metric has higher accuracy than NCD across all sample sizes and comparable AUC. For the Industry test set (5.1(b)), 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 Public test set (5.1(a)) 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.

The large variation in Figure 5.1 comes from sub-sampling both the training and testing distribution. This was a requirement due to the extreme computational burden of running NCD against even the whole test set. The purpose of this experiment is to show that the LZJD variants generally dominate NCD's performance across the spectrum of dataset sizes.

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 Industry test set, where NCD performs much worse than LZJD in all respects.



(a) Results on Public test set



(b) Results on Industry test set

Figure 5.2: Balanced Accuracy and AUC on the y axis, presented for the Public and Industry test datasets. The same legend applies to each plot. Here we use the whole test set at every fraction, and look at the variance of the accuracy as more data is used.

In the aforementioned plots, the fractioning of data also used a fraction of the test set. This was a computational requirement due to the extensive compute required for NCD and LZJD. Because LZJD$_h$ is computationally tractable, it does not have this same restriction. Thus we can look at the average accuracy and AUC for LZJD$_h$ measured against the whole test set as a fraction of the amount of training data used. We run this test 10 times for each fraction of data in order to produce a mean and standard deviation, and is plotted in Figure 5.2.

This allows us to unambiguously show the accuracy improvement as more data is used, and that the variation present in Figure 5.1 is purely a function of the sub-sampled test set combined with the sub-sampled training set. LZJD$_h$'s performance improves uniformly as more data is used.

The accuracies at 2% and 100% of the data from Figure 5.1 are shown in Table 5.1. In every case LZJD performs better than NCD, and performance improves

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

Table 5.1: 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 byte n-gram results from chapter 3 in last column

as more data is used. While the byte n-gram approach used in chapter 3 performs better than LZJD, LZJD allows a wider variety of uses in clustering, similarity search and requires less effort to apply[2]. Because LZJD is a distance metric, we can apply it to various existing techniques, but such questions are beyond the scope of this thesis. The increased practicality will also allow investigating improved classification methods, such as Radial Basis Function networks, that were too expensive with NCD.

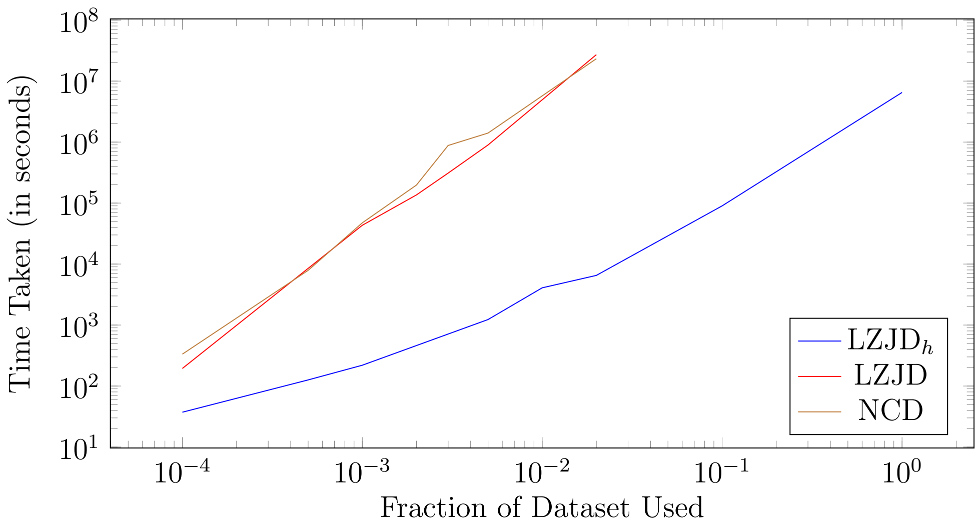


Figure 5.3: Time taken to perform 9-NN classification on Public, B, and Open Malware test sets with the Industry training set.

[2]The byte n-graming approach is computationally demanding, and requires multiple days of out-of-core processing for the same data. This makes scaling problematic.

The total single-threaded run-time for this evaluation is presented in Figure 5.3, 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 against an existing database, the min-hashing is a one-time cost, making deployment of LZJD$_h$ more realistic as well.

5.3.2 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.

Staying with Microsoft PE files, we will use the 2015 Microsoft Kaggle data discussed in subsubsection 2.2.2.1. Similar to subsection 5.3.1, we use the raw byte contents of the Kaggle Bytes version of the dataset to evaluate how well NCD handles the same type of data (Microsoft binaries), but with a different task. We will also evaluate on the Kaggle ASM data, to show that the approach developed is not overfit to binary content, but can work on ASCII disassembly as a feature type as well.

We will also use the Drebin dataset discussed in subsubsection 2.2.2.2. Doing so provides additional evidence that the approach is not overfit to Microsoft and x86 data. The performance difference between the Drebin APK and Drebin TAR versions of the dataset is also particularly informative for comparing LZJD and NCD. This is because the Drebin APK version has some level of compression due to the zip format, which raises entropy. NCD is known to be sensitive to higher entropy sequences, and since the only difference between Drebin APK and Drebin TAR is compression, it provides information about the relative robustness of NCD and LZJD to its impact.

	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)

Table 5.2: Balanced accuracy results on each data and feature set. Evaluated with 10-fold CV, standard deviation in parenthesis.

The results of running 1-NN on these datasets are given in Table 5.2, 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.

	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

Table 5.3: Total evaluation time for each method 10-fold CV. Time presented in seconds.

In Table 5.3 the CPU time on each dataset is given in seconds. Our unoptimized 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.

5.4 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.

5.4.1 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 [129]. 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.

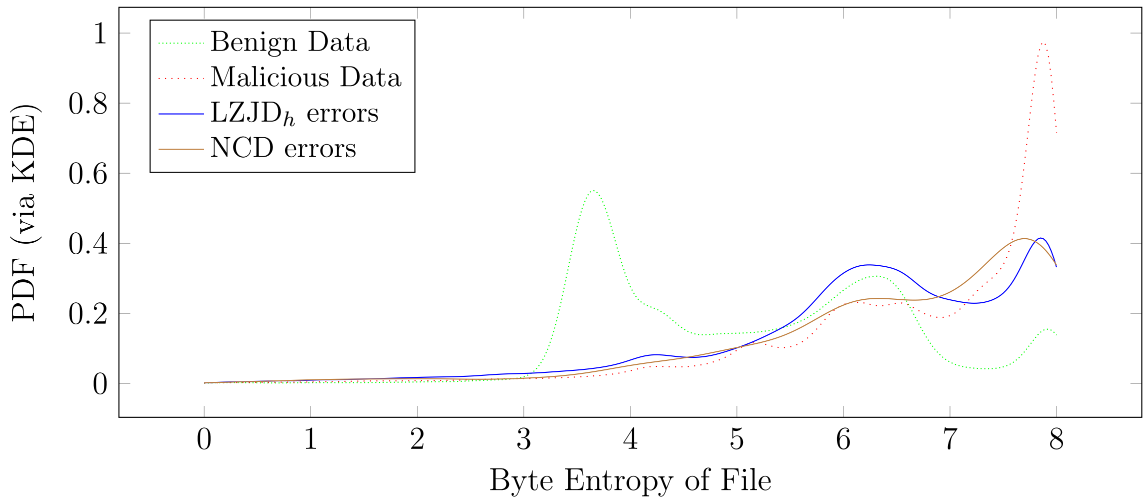


Figure 5.4: Distribution of file entropy for datapoints that were misclassified (solid lines) in the test set, and benign vs malicious sub-sets (dotted lines). Results with respect to all three test sets from subsection 5.3.1.

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 5.4 for the malware detection results from subsection 5.3.1. The Probability Density Function (PDF)[3] 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 (≥ 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 5.4. 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

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

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.

5.4.2 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 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)[4]. Then applying this to NCD we expect to get

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

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

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

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 [37]. 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

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

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.

5.5 Conclusions

In this chapter 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.

Chapter 6: Malware Classification and Class Imbalance via Stochastic Hashed LZJD

Given our new LZJD metric, we now have am alternative to NCD that is faster and more accurate for malware classification. However, it is not yet practical for use with large training sets due to the nearest-neighbor search. This issue will be remedied in this chapter, providing a LZJD based classifier with inference time independent from the training set size. Simultaneously, we will also improve LZJD to handle class imbalance directly, a common problem in the malware space that will make LZJD applicable to a wider array of sub-problems in this space.

6.1 Introduction

In chapter 5 we developed the new Lempel-Ziv Jaccard Distance. LZJD avoids the challenges of extracting features by ignoring any domain knowledge, and instead measures the similarity between arbitrary byte sequences (such as the raw binaries themselves). LZJD improved upon the Normalized Compression Distance (NCD) [35], which has been used for a number of malware related tasks [e.g., 37]–[40] by improving on the runtime and accuracy of NCD. LZJD was shown to be more effective in a nearest-neighbor classifier for both types of malware classification problems:

malware detection (is this file benign or malicious?) and malware family classification (given a malicious binary, which family is it from?). LZJD is also a true distance metric, unlike its predecessor NCD, allowing it to be used for similarity search, clustering, and other applications. However, LZJD does not lend itself to building a compact malware classifier. Nearest neighbor search time will increase linearly with the training set size, and the use of SVMs only delays the growth in inference time.

In this chapter, we extend LZJD to further increase its accuracy for malware classification and improve inference time to be invariant to training set size. In addition, we will also address the class-imbalance problem. Class imbalance occurs when there is significantly more training data from one class compared to the others. This problem occurs naturally in the malware domain [66], and thus is important to resolve. We do all of this in a single framework. Our new algorithm will work by altering LZJD to produce a fixed length, sparse feature vector that can be used effectively with algorithms like Logistic Regression. We call this algorithm SHWeL, for Stochastic Hashed Weighted Lempel-Ziv. Its derivation will be given in section 6.3, where we will also explain how we leverage SHWeL to tackle the class imbalance problem via a novel over-sampling strategy. In section 6.4 we will evaluate several datasets to show that SHWeL vectors improve upon LZJD when used for nearest-neighbor classification, work well with Logistic Regression as a classifier, and obtain further improved accuracy via our new over-sampling technique. In section 6.5 we will discuss how our SHWeL algorithm is able to outperform existing approaches, followed by our conclusions in section 6.6. But first, we will review the related work in this area in section 6.2.

6.2 Related Work

We take a moment to discuss the important prior work related to class imbalance. While the primary Industry training data used throughout this thesis has the benefit of being balanced, this is not an appropriate assumption in the general case. Indeed, our malware family datasets have high levels of class imbalance as well.

Class imbalance issues have been noted for both malware family problems [113], and malware detection [41], [130]. Despite this problem, it has been noted that there is surprisingly little work in the malware space on tackling the issue at training time [131]. The Machine Learning community has studied this problem on a more general level, and developed a number of methods for tackling this problem — primarily through the application of over- or under-sampling the training data [132], [133]. In particular, the SMOTE algorithm [134] has become one of the most popular and successful methods in this space. SMOTE works by over-sampling the minority class to improve the training balance, and interpolating the oversampled points with their neighbors to increase diversity. Many variants of SMOTE have been proposed, and we also consider one of the more popular of those variants, namely Borderline-SMOTE [135] (B-SMOTE). B-SMOTE improves upon SMOTE by restricting the set of points used to over-sample to data points near the border of the classes.

One weakness of the many Machine Learning based solutions to class imbalance is that they have been developed for use with small datasets. Let N_+ be the size of the majority dataset, N_- be the size of the minority class, and $N = N_+ + N_-$ be the size of the whole training set. The SMOTE algorithm then takes $O({N_-}^2)$, which is acceptable assuming that the minority class is small. But for our domain, N_- could be in the range of a million samples, with N_+ in the tens to hundreds of millions. This situation gets worse with B-SMOTE, which increases the complexity to $O(N \cdot N_-)$. In contrast, the approach presented here will require only $O(N)$ time, while simultaneously providing better results than either version of SMOTE.

While we focus on class imbalance at training time, since obtaining diverse benign training data is difficult, this space also exhibits class imbalance at inference time. Individual computers may be likely to see more benign files than malicious, while externally facing networking equipment may see more malicious files than benign. Moskovitch, Stopel, Feher, *et al.* [24] looked at the problem of determining what ratio of training data, r_t would provide the best performance when encountering a different ratio, $r_i \neq r_t$, at inference time. Others have taken an approach with balanced training sets, and built classification systems focused toward the imbalanced test environment [136]. Another less satisfying approach is to restrict the training set to have the same class proportions as one expects at test time [137]. All of these approaches are predicated on having a specific amount of training data, and could benefit from the methods we develop here.

6.3 Stochastic Hashed Weighted LZJD

While LZJD has been shown to be an effective distance metric and has good accuracy in a nearest-neighbor style classifier, such classifiers are not always optimal for practical use. Of particular importance is the use in scenarios with large datasets, as the amount of malware is growing at an exponential rate. For this reason we desire a method that can exploit LZJD's accuracy but provides faster inference than a nearest neighbor search.

Our second goal is a classifier that is robust to class imbalance. By the nature of the problems present in the cyber security domain, extreme class imbalance is a common scenario. For malware detection, it is easy to obtain terabytes of malware, but good and representative benign data is difficult to come by. For malware family classification, different malware families naturally have differing number of variants, with some malware intentionally designed to be prolific and others designed to be subtle. We develop a strategy for vectorizing data items with LZJD that addresses both of these challenges.

Our strategy for achieving these goals can be broken down into three steps of modifications to the original LZJD algorithm. We will review these three steps in more detail below.

1. Incorporate subsequence length into the similarity measure to capture additional information and improve accuracy.

2. Convert byte sequences to vectors so that we can exploit faster algorithms,

and efficiently incorporate the weights discussed in step 1.

3. Introduce a stochastic component to the vectorization, so that we can over-sample a byte sequence to tackle class imbalance.

6.3.1 Incorporating Weights into LZJD

We first note that we can potentially improve the effectiveness of LZJD by incorporating the concept of sequence length into the algorithm. To see how weights can be incorporated into LZJD, we note that LZJD breaks a larger byte sequence A into a set of m sub-sequences $\alpha_{1\ldots m}$, such that $\forall i \in [1, m], \alpha_i \in A$. In LZJD, the set of subsequences α is an unweighted set. However, each sub-sequence α_i may have a different length, and intuition says that it is more interesting if two binaries share a subsequence that was 100 bytes long than if they share a subsequence that was only 3 bytes long.

Thus we propose to improve LZJD into a weighted LZJD by giving each subsequence α_i a weight $w_{\alpha_i} = \log(|\alpha_i| + 1)$. Using a log term for the length's weight has the same inspiration as TF-IDF weighting schemes, in that as the length of the sequence increases, the importance of its increased length begins to diminish. This is a practical assumption for the malware domain due to issues like padding, which can intentionally place thousands of simple byte patterns into a binary. Ideally, we would use these weights with the Weighted Jaccard Similarity (6.1) instead of the standard Jaccard currently used.

$$WJS(a,b) = \frac{\sum_{\forall i \in a \cup b} \min(a_i, b_i)}{\sum_{\forall i \in a \cup b} \max(a_i, b_i)} \tag{6.1}$$

The problem with using the Weighted Jaccard approach is that we lose the fast min-hashing afforded by the Jaccard similarity. If a byte sequence A produces m subsequences and we want a digest with k hashes, LZJD takes only $O(m + k \log k)$ time to create the min-hash used. There exist digest strategies for the weighted scenario, but all take $O(km)$ time in order to produce a hash of size k [138]–[140]. Because m is in the millions to tens of millions for many binaries, these strategies are not practical even for our smallest tests.[1]

6.3.2 Incorporating Hashing

Instead we look at linearizing the LZJD feature vector, a strategy that has been used to improve the runtime performance of both the Weighted Jaccard similarity [141] and the normal Jaccard similarity [142] used by LZJD. Because building a weighted hash is too computationally demanding, we instead use the *hashing trick* [143].

The hashing trick works by indexing into a vector $x \in \mathbb{R}^d$, where d is a hyper-parameter we control. The vector starts out as all zeros, and for each feature α_i we want to consider, we index into x based on some hash function $h(\cdot)$. Normally this would be done with all features, and thus would produce a vector with m non-zero

[1]Initial testing of ICWS hashing algorithm took hours for just a few thousand files.

values. Instead, we continue to restrict ourselves to only the k smallest hash values as they would have been computed in normal LZJD. This gives us a vector x which will have exactly k non-zero elements, the values of which are determined by the weights w_i specified above.

Our choice to continue using the minimum k entries from the Lempel-Ziv dictionary allow us a principled way of ensuring that the subset of features selected will also be selected when processing new files. While the selected features are intrinsically biased toward maintaining the Jaccard similarity between two sequences, using the weighted values allow us to incorporate a behavior closer to that of the Weighted Jaccard case. This heuristic interpolation between methods does not have the same theoretical backing as Confidence Weighted Sampling algorithms for weighted min-hashes, but we find it allows us to uniformly improve upon the accuracy of LZJD while avoiding the computational shortcomings of weighted min-hashing approaches.

6.3.3 Incorporating Stochasticity

The final step to our new approach is to incorporate a stochastic behavior into the Lempel-Ziv process. When constructing the LZ dictionary [124], [125], the entires are added in a sequential manner, and items can only be added to the set if they have not been seen previously. This means the zth entry added into the dictionary is dependent on all previously seen dictionary members. Thus a single byte change early in the dictionary creation process has the potential to propagate forward, causing significant changes in the content of the dictionary itself, and thus the final set of k subsequences used in our hash. To give a concrete example of this impact, consider the bit-string *01010110111*, which will generate the LZ set {*0,1,01,011,0111*}. If we remove the first “0” from the original string, the generated LZ set will become {*1,0,10,11,01*}. Thus a one bit change in the input string has removed the two longest sub-strings and replaced them with two new (and in this case shorter) sub-strings.

This sensitivity can be seen as a weakness of the LZJD approach. In this work, instead of attempting to correct this weakness, we instead turn it into a strength by intentionally perturbing the dictionary produced by the LZ process. Using these perturbations we can obtain multiple distinct vectors or “realizations” for a single byte sequence. These multiple different samplings can then allow us to over-sample the minority classes until each class has an equal number of training vectors. This is similar in spirit to the seminal SMOTE algorithm, which produces new synthetic examples from the minority class by linearly interpolating between points and their nearest neighbors [134]. In our case, the over-sampled vectors are as real and valid as any other feature vector, where the SMOTE over-sampling can produce unrealistic or improbable feature vectors.

To perturb our LZ process, we add a stochastic “false-seen” chance p to the construction process. This is the probability that we will falsely indicate that we have previously seen the given subsequence. When we check the current set for the current subsequence, then with probability p we will behave as if the subsequence was seen before, even if it was not. Regardless of the result of this step, we still add the subsequence to the current set. The whole procedure is shown in Algorithm 2.

Algorithm 2 Stochastic Hashed Weighted Lempel-Ziv (SHWeL)

```
Require: False-Seen probability $p$, target dimension $d$, hash size $k$
 1: procedure SHWeL(Byte sequence $b$)
 2:     $x \leftarrow \vec{0} \in \mathbb{R}^d$
 3:     $s \leftarrow \emptyset$
 4:     $start \leftarrow 0$
 5:     $end \leftarrow 1$
 6:     while $end < |b|$ do                                    ▷perform Lempel-Ziv set creation
 7:         $b_s \leftarrow b[start : end]$
 8:         Sample $f \sim \mathcal{U}(0,1)$                          ▷false-seen chance
 9:         if $b_s \notin s \wedge f > p$ then
10:             $start \leftarrow end$
11:         end if
12:         $s \leftarrow s \cup \{b_s\}$                               ▷has no effect if seen before
13:         $end \leftarrow end + 1$
14:     end while
15:     $s_k \leftarrow k$ entires of $s$ with the smallest hash values
16:     for all $b_s \in s_k$ do                                   ▷set non-zero values
17:         $x\left[h\left(b_s\right) \bmod d\right] \leftarrow x\left[h\left(b_s\right) \bmod d\right] + \log(|b_s| + 1)$
18:     end for
19:     return $x$
20: end procedure
```

This new approach allows us a direct method to tackle class imbalance by over-sampling. Instead of over-sampling the feature vectors, we generate multiple feature vectors from each raw file. Because of our stochastic SHWeL algorithm, these feature vectors will contain some variable number of differences. To create a balanced dataset, we determine the ratio r between the majority class and each minority class, and then sample each minority file r times. This allows us a more effective method of oversampling the minority classes than offered by SMOTE or naive oversampling or re-weighting. We call this new strategy for oversampling with SHWeL vectors **O**ver-sampling via **S**HWe**L** (OSL).

By incorporating a stochastic decision at this step, we also enforce two important properties. First, that the LZ dictionary contains only subsequences α_i that can be found in the source sequence A. Second, that every byte of A can be found in some subsequence α_i. In this way we can be confident that, despite producing multiple vectors for a single byte sequence, each vector does represent that byte sequence. If we were to randomly inject or alter bytes within the sequence A, it is possible that we would produce false and spurious correlations to other files, or waste one of our k non-zero entries on representing a subsequence that never appeared in *any* file. Our strategy avoids such undesirable properties.

6.4 Experiments

Having developed our new method for vectorizing byte sequences, we will evaluate our new approach on multiple datasets for malware family classification in subsection 6.4.1, and malware detection in subsection 6.4.2. We will use the balanced accuracy as our target metric on all datasets. Balanced accuracy re-weights the contributions of each data point toward the final score based on the class label, such that the score produced gives equal total weight to each class. This allows us to compare results in a more meaningful way across datasets, and emphasizes the importance of learning low-frequency classes [103]. For the binary malware detection problem, we will also consider the Area Under the ROC Curve (AUC) [104], as it relates well to the triage scenario.

For each method we will show the original LZJD's performance as a nearest-neighbor classifier, using SHWeL vectors in a k-nearest neighbor (kNN) classifier with cosine distance, and using SHWeL vectors with Logistic Regression (LR). We will also test the kNN and LR classifiers when using our OSL approach, and when doing so, will denote them as “kNN-OSL” and “LR-OSL” to avoid confusion. As a baseline for handling class imbalance, we compare against the SMOTE algorithm [134], and the extension Borderline-SMOTE (B-SMOTE) [135] that attempts to improve performance by only oversampling minority samples near the decision border. In each case we will use the Logistic Regression classifier when using either version of SMOTE,[2] with our SHWeL vectors as the feature representation.

All code for the below experiments was written in Java using the JSAT library [144]. Tests were run on a workstation with 128 GB of RAM, 4 TB of SSD storage, and an Intel Xeon E5-2650 CPU at 2.30 GHz. For both LZJD and SHWeL we use $k = 1024$. For SHWeL we use a false-seen probability $p = 1\%$ and target dimension $d = 2^{20}$. SHWeL's performance was insensitive to all of these parameters in extended testing. We will see that for malware classification, our new SHWeL vectors outperform LZJD universally, outperform SMOTE for dealing with the class imbalance problem, and allow us to produce a higher accuracy classifier that can be used with large data sets with lower inference time.

6.4.1 Malware Family Classification

Our evaluation starts with malware family classification, and so we continue to use the Android Drebin and Microsoft Kaggle data. These corpora have class imbalance issues, and so provide a direct test case to show the benefit of our new

[2]While SMOTE was originally defined for binary classification problems, we have extended both to the multi-class case in such a way that they are equivalent to the original algorithms when run on binary class problems.

SHWeL algorithm's ability to tackle class imbalance.

Dataset	LZJD	1NN-SHWeL	LR-SHWeL
Kaggle Raw	97.6 (1.5)	**97.6 (1.38)**	96.7 (2.07)
Kaggle ASM	97.1 (6.1)	**97.3 (1.93)**	96.9 (2.08)
Drebin APK	80.8 (2.6)	**83.6 (1.94)**	78.4 (2.26)
Drebin TAR	81.0 (6.5)	87.9 (1.84)	**89.1 (2.29)**

Table 6.1: Balanced accuracy results from 10-fold cross validation (standard deviation in parentheses) on malware family problems, when trained on the unbalanced training data. Best results shown in **bold**, ties broken by selecting model with lowest variance.

We begin by looking at the performance of LZJD, kNN-SHWeL, and LR-SHWeL on the naturally unbalanced Kaggle and Drebin datasets. The Kaggle dataset has a mean ratio between minority and majority class ratio of 9.5:1, and a maximum ratio of 59:1. The Drebin dataset has a mean ratio between minority and majority class ratio of 9:1, and a maximum ratio of 18.6:1. The results can be seen in Table 6.1, where 1NN-SHWeL uniformly outperforms LZJD on every dataset. On both Kaggle datasets LJZD and SHWeL have similar performance, with SHWeL being superior and having reduced variance in results. We see a bigger performance difference to 1NN-SHWeL's advantage on both Drebin datasets. On the APK version 1NN-SHWeL improves performance by 2.8 percentage points, and by 6.9 whole points on the TAR version.

While LR-SHWeL does not uniformly improve upon LZJD, we will see this is because of the negative impact of class imbalance in the training data. We re-evaluate 1NN and LR using our Over-sampling scheme OSL in Table 6.2, where we compare against LR-SHWeL run with SMOTE and B-SMOTE. In this case we see that LR-OSL dominates all results for handling class imbalance, and improves

Dataset	SMOTE	B-SMOTE	1-NN-OSL	LR-OSL
Kaggle Bytes	97.3 (1.85)	97.3 (1.81)	97.8 (1.34)	**97.9 (1.86)**
Kaggle ASM	96.9 (2.09)	96.9 (2.11)	97.3 (1.93)	**97.8 (1.91)**
Drebin APK	79.1 (2.02)	78.9 (1.66)	83.6 (2.05)	**84.0 (1.75)**
Drebin TAR	90.0 (2.21)	89.7 (1.92)	88.4 (1.61)	**93.5 (1.64)**

Table 6.2: Balanced accuracy results from 10-fold cross validation (standard deviation in parentheses) on malware family problems, when trained on balanced data. Best results shown in **bold**.

upon all methods tested in Table 6.1. In all cases OSL has little impact (but still positive) upon the performance of 1NN-SHWeL, which is a reasonable expectation. While SMOTE and B-SMOTE both manage to slightly improve upon the performance of LR-SHWeL, they do not provide the same dramatic improvements that OSL is capable of. We see LR-OSL improve the accuracy of LR-SHWeL by 0.9-1.2 percentage points for the Kaggle datasets, and 4.4-5.6 points on the Drebin dataset.

6.4.2 Malware Detection

For malware detection, we continue to use the Industry dataset and testing set. In this chapter we introduce an updated version of the Industry corpus, which contains 2 million binaries: split evenly between benign and malicious. This will be the Industry 2 million corpus, and the version used in prior chapters will be denoted as Industry 400k. We first use the 400k version of the corpus to compare with our prior results, and show improved scaling using the new 2 million binaries corpus.

The results of running our new methods on this malware detection dataset are presented in Table 6.3. On the Industry and Public test sets, we can see that SHWeL vectors outperformed LZJD both when using nearest neighbors, and when

	LZJD		9NN-SHWeL		LR-SHWeL	
Test Set	Acc	AUC	Acc	AUC	Acc	AUC
Industry	85.9	91.1	**87.5**	**96.2**	*87.2*	*94.5*
Public	77.4	86.7	*78.3*	*88.6*	**83.2**	**96.9**
Open Malware	**67.8**	—	61.0	—	*64.2*	—

Table 6.3: Balanced accuracy and AUC for original LZJD (using 9-NN), and new SHWeL vectors with both 9-NN and Logistic Regression algorithms as the classifiers. Training on the Industry 400k training set, best result is shown in **bold** for each row, second best shown in *italics*.

using logistic regression, with the two approaches each performing best on one test set. While accuracy has improved from 1.6 to 5.8 points, depending on dataset, the AUC has had the most substantial improvement: By 5.1 points on the Industry test set, and 10.2 whole points on the Public dataset. This is of particular relevance for the triage application scenario, where we will be inspecting most files anyway, but having a high AUC allows us to effectively prioritize the work queue of an analyst. While LZJD did have the best performance on the Open Malware test set, this result is biased due to the fact that it contains only malware — and is thus measuring recall rate. In both cases above, models trained off SHWeL vectors had a preference toward labeling files as benign. Because the Open Malware corpus has only malicious files, the true-positives that would have been obtained by marking more benign files as benign are not reflected in the score.

A benefit of the SHWeL vector approach in particular is that it allows us to scale to even larger datasets, especially when using Logistic Regression as our classifier. We demonstrate this using the Industry 2 Million corpus. Specifically, it contains 2,011,786 binaries, with 1,000,020 benign and 1,011,766 malicious. We will refer to this larger corpus as Industry 2M. Training LR-SHWeL on this corpus of 1.8 TB of compressed binaries took under 15 hours, including the feature extraction and decompression of the original files,[3] using a server with 16 CPU cores. In contrast, the byte n-gram approach on the smaller original 400k training set took multiple days using a server with 64 CPU cores [41], [43]. Training several times faster while simultaneously using a dataset five times larger demonstrates the increased scalability of our approach. Extrapolating from running LZJD on the 400k dataset, just classifying the test data using this newer and larger corpus would have taken over 23 days on the same 64-core server.

	Byte 6-gram		LR-SHWeL	
Test Set	Acc	AUC	Acc	AUC
Industry	**91.6**	97.0	91.3	**97.3**
Public	82.6	93.4	**89.0**	**98.2**
Open Malware	79.3	—	**81.9**	—

Table 6.4: Balanced Accuracy and AUC on the three malware detection test sets using a new training corpus of 2 million binaries. Best results shown in **bold**.

The final test set accuracies for LR-SHWeL using this new dataset is given in Table 6.4. Because the byte n-gram approach is the only other domain-knowledge free method that obtains fast inference time, we have replicated the work of Raff, Zak, Cox, *et al.* [41] on this larger corpus. Doing so required a cluster of 12 machines (each with 8 CPU cores) and two weeks of time to compute the 6-grams. This makes the n-gram approach over 22 times slower to build than our new LR-SHWeL approach, even when given 6 times the compute resources.

We can see that using more data has dramatically improved the performance

[3]Each file was gzipped independently given the size of the corpus.

of LR-SHWeL on every test set, sometimes by significant margins. The smallest improvement in accuracy was the industry test set improved by 4.1 whole points, and the largest on Open Malware by 17.7. In contrast, our byte 6-grams had a reduction in accuracy on the Public test corpus, going from 87.3 (reported in [41]) down to 82.6. This result is consistent with the hypotheses proposed in Raff, Zak, Cox, *et al.* [41], which posited that the byte n-gram approach encouraged overfitting and would have diminishing returns with additional data. This shows that our new SHWeL vectors provide better results, while also being able to scale to larger corpora.

6.4.2.1 Evaluating a Spectrum of Class Imbalance

For the malware detection problem, our training sets have been equally balanced between benign and malicious samples on a large corpus. This has allowed us to show SHWeL's effectiveness, but does not exercise our ability to handle class imbalance via over-sampling SHWeL vectors. Because it is most difficult to obtain benign training data, we will under-sample the benign files from the whole training set. Synthetically restricting ourselves to a fraction of the training goodware, and over-sample the benign files to mimics a scenario where we are without such a large corpus of goodware data. This allows us to look at the performance in an unbalanced scenario, and the performance using the whole training set gives us a realistic target score.

The Industry test accuracy when trained with imbalanced data on the Industry 400k training set can be seen in Figure 6.1. The dashed black line shows the

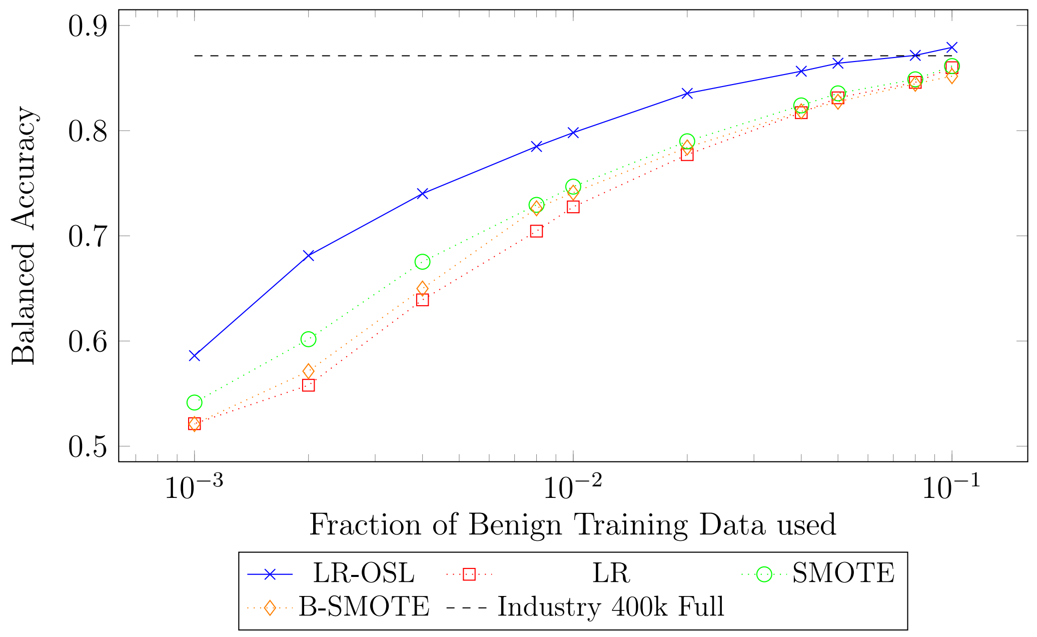


Figure 6.1: Balanced accuracy on the Industry test set. The x-axis indicates the fraction of benign data used from the training set during training. The left-most portion of the graph shows the results when training under extreme class imbalance, which is progressively lessened as you move to the right of the graph.

performance of Logistic Regression when trained on the whole dataset, with no subsampling. We evaluate over a range of 10:1 in favor of malware, down to 1000:1. As is expected, the test performance decreases as the ratio becomes further unbalanced (toward the left of the figure) for all methods. However, our new SHWeL Over-Sampling (OSL) strategy dominates SMOTE and an unbalanced Logistic Regression at all ratios. On average OSL has an additional 4.4 whole percentage points of accuracy, with a minimum of 1.7 and a maximum of 8.0. This is on top of the more minor gains SMOTE has over a naive application of Logistic Regression to an unbalanced training set.

6.4.3 Quantification of Efficiency

At this point we have shown that our SHWeL vectors provide better accuracy than LZJD, and even the byte 6-gram model previously used. We now discuss in more detail the training and inference runtime of our new approach, showing that it meets the goal set out in this work: to have efficient inference time that is not dependent on training set size (when used with Logistic Regression), and better scalability when addressing class imbalance. Because the time needed to featurize a specimen is a important part of this process, we include this cost when discussing runtime results.

6.4.3.1 Training Efficiency Under Class Imbalance

As mentioned in section 6.2, SMOTE and B-SMOTE have complexities of $O({N_-}^2)$ and $O(N \cdot N_-)$ respectively, where our new OSL approach has complexity of $O(N)$. So long as $N_- > \sqrt{N}$, which is reasonable, our new approach has superior asymptotic complexity. However, the overhead of creating SHWeL vectors is not trivial, so the benefit is not fully realized on smaller datasets. This can be seen in Table 6.5, where the total training time in seconds is presented for each of the malware family datasets. For each of these problems, $N_- \leq 4034$, so the quadratic complexity of SMOTE is not as noticeable, and furthermore, the vectorization overhead of SHWeL dominates the runtime, accounting for $\geq 85\%$ of the training time.

The algorithmic benefit of our approach becomes more obvious when we look

Training Set	LR-SHWeL	LR-OSL	SMOTE	B-SMOTE
Kaggle Raw	6810.4	9136.7	7091.6	8478.9
Kaggle ASM	32087.4	41712.8	32077.3	33740.6
Drebin APK	2590.7	7218.1	2820.4	3067.1
Drebin TAR	2988.4	8471.8	3189.3	3448.6

Table 6.5: Training time of base Logistic Regression and different over-sampling strategies on malware family problems. Time measured in seconds.

at a larger dataset, like the Industry 400k data. We plot the training time taken as a function of the amount of benign data used in Figure 6.2. Here the positive impact of $O(N)$ complexity is obvious, and we see an almost constant training time for LR-OSL as the amount of benign data (the minority class) increases. The slight increase in training time comes from the Logistic Regression solver taking slightly more time to converge as the training distribution's diversity increases from having more real samples to learn from, as opposed to the over-sampled vectors used to balance the dataset. As the fraction of benign files becomes closer to using all 200,000 training points, both LR and LR-OSL will approach the same training time of just using Logistic Regression on the full dataset (dashed black line).

We can also see that while both SMOTE and B-SMOTE are faster when there is very little benign data, their quadratic complexities quickly begin to explode the training time. Even at a ratio of just 10:1, the SMOTE algorithms are taking as long or longer to train than Logistic Regression on the whole corpus. If we had only 100,000 benign training samples compared to the 200,000 malware samples, we would be unable to use either SMOTE algorithm, despite having a minor imbalance ratio of 2:1 to correct for. This shows how our approach will scale better to large datasets. Indeed, we could not afford to produce the same learning curves with

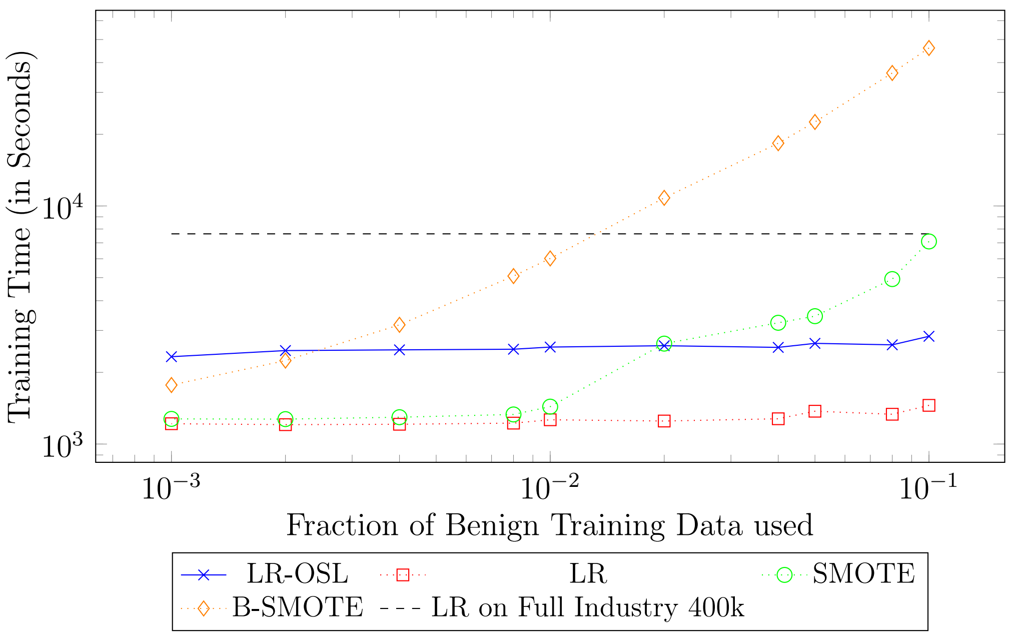


Figure 6.2: Training time on the Industry 400k training set. The x-axis indicates the fraction of benign data used from the training set during training. The left-most portion of the graph shows the results when training under extreme class imbalance, which is progressively lessened as you move to the right of the graph.

SMOTE on the larger Industry 2M corpus.

6.4.3.2 Inference Efficiency

Inference time is a simpler problem to look at, as SMOTE, B-SMOTE, LR-OSL, and LR-SHWeL will all have the same inference time since they all use the SHWeL vectors with Logistic Regression. For this reason we summarize all of them using LR-SHWeL. It is easy to predict that they will have the best runtime performance, as the alternative k-NN classification time grows linearly with the training set size. While we still see that the feature-vectorization process dominates for smaller datasets (which in our case, have larger average file size, increasing SHWeL's vectorization time), our new LR-SHWeL approach continues to have its best performance on the largest corpora. The average time to classify a datum can be found in Table 6.6 for every dataset. We see that the savings of using LR-SHWeL are muted for the malware family problems, as predicted, because they have fewer training points and larger file size.

Training Set	LZJD	9NN-SHWeL	LR-SHWeL	Avg. Size
Kaggle Raw	1580.81	189.01	170.40	4.67
Kaggle ASM	4439.53	822.72	804.65	13.5
Drebin APK	1507.19	155.79	148.00	1.27
Drebin TAR	1607.69	179.14	171.38	1.73
Industry 400k	25113.00	13677.11	44.85	0.38
Industry 2M	125563.33	68385.57	44.90	0.38

Table 6.6: Average time in milliseconds to classify a datum at test time, organized by the training set used. Underlined values indicate estimated run-times. Fifth column indicates average file size, in megabytes, of the test sets.

When training on the Industry 400k dataset, and evaluating on the associated test sets, 99.95% of time spent was on creating the feature vectors (with an average file size of around 0.38 MB). In this case the LR-SHWeL classifier presents a respectable and fast 45 ms inference time, making it easily deployable for most binaries. We note that while fast, the SHWeL vectorization is a bottleneck that is easy to parallelize. No communication is ever needed for creating the vectors at any stage of the process, allowing us to work around this issue in practice. We do note that this does still present a potential future research goal, as reducing the time for vectorization may allow deployment in scenarios with more stringent runtime requirements (such as mobile devices).

Our SHWeL approach is clearly the most scalable solution compared to existing tools for training and testing, as it handles the Industry 2M training set with ease while still having fast execution time. Nearest Neighbor classification with LZJD had to be estimated due to performance constraints, as we do not have the compute resources to run the task within a reasonable time-frame. We note again that byte 6-grams built on the Industry 2M dataset took over 20 times longer, despite using six times as much compute power.

6.5 Discussion

Now that we have shown that SHWeL's improved classification accuracy and inference time allows us to tackle the class imbalance problem, we discus two details of our new approach. In particular, we analyze why over-sampling with SHWeL provides superior improvement over existing SMOTE based approaches to class imbalance. These insights are applicable in comparing our OSL to any oversampling approach similar to SMOTE. We also note that there exists an interpretation of SHWeL and LZJD that connects them back to the n-gram based approach, which we find informative to potential research directions.

6.5.1 How Swell SHWeL Smites SMOTE

We take a moment to discuss why Oversampling SHWeL (OSL) vectors provides better performance than the seminal SMOTE algorithm in our use case. One may suppose that SMOTE should perform better, as it interpolates the space between points in the minority class. In contrast, the OSL strategy does not interpolate between points, but instead produces alternate realizations of the same subset of points.

The critical component of SHWeL's success is that the alternate realizations are equally valid points, as mentioned in subsection 6.3.3 and can produce greater diversity in features. Consider that SMOTE works by first finding the z nearest neighbors of a point, and randomly selecting one of those nearest neighbors with which to perform a linear interpolation. If these two points are a and b, then it must be the case that a and b currently share a considerable overlap in their non-zero values, otherwise they would never have become nearest neighbors. Thus for points that are shared between both a and b, the interpolated point $c = \gamma a + (1-\gamma) b$ will have the same features but with slightly altered weights *only if a feature collision has occurred*. It is likely, because a and b are nearest neighbors, that it was the exact same sub-sequences that were selected as the k items in the digest, and thus have the exact same length, and thus the features for each feature i, it is probable that $c_i = a_i = b_i$. Thus the interpolated features have provided no additional diversity.

SMOTE also has a poor interpretation in the case of features that occur in only a or b. Recall each point will have exactly k non zero values, but the interpolated point c can have $[k, 2k]$ non-zero values. For each feature i in a but not in b, that feature will occur in the interpolated point c with a discount of $c = \gamma a_i$ (or $c = (1-\gamma) b_i$ if the feature occurred in only b). This interpretation becomes confusing, as the weights are derived from the length of the sub-sequence. The interpolated feature then produces an occurrence of the same feature, that surprisingly has a weight smaller than what its length would determine. Further, since the weight is based on the log-length, we can see that the interpretation yields $\exp(\log(|\alpha_i|)\gamma) =$

$|\alpha_i|^\gamma$, meaning we are dramatically "changing" the length of the sub-sequence that provided the feature value. This is an impossibility in the actual feature construction process, making the interpolated points less meaningful and less likely to match a new point.

We now compare this behavior to OSL. Let us denote a as the original data point, and $\tilde{a}$ as an OSL oversampling produced from the same underlying byte sequence. It is likely (to a degree depending on the value of p) that a and $\tilde{a}$ will share a number of non-zero values. In this case, the behavior is similar to the value of c produced by SMOTE, as intrinsically $a_i = \tilde{a}_i$. The more important case is that we will have new non-zero values in $\tilde{a}$ that do not occur in a. These new values will have reasonable lengths, and are intrinsically increasing the diversity of features used as they account for features not previously seen, where c is bound to the features that occurred in either a or b. This property helps to improve generalization, and will have the correct weight in the feature vector based on $\log(|\alpha_i|)$, rather than the discounted weighted $|\alpha_i|^\gamma$. Further, as we are forced to oversample points many times, the interpolated points c will be constrained to a set of at most zk possible non-zero values (k features for each of the z neighbors). If we have to oversample points hundreds of times, this will quickly exhaust what little diversity can be extrapolated from the SMOTE vectors. Because SHWeL is sampling k sub-sequences from a total of m possible sub-sequences, a value which is often in the millions, there is significantly greater potential diversity to be extracted — thus extending SHWeL's utility as the ratio between majority and minority class increases. This hypothesis is supported by Figure 6.1, where we can see OSL's advantage over SMOTE increase as the amount of benign data decreases, thus increasing the number of times each point must be over-sampled.

6.5.2 A Connection Between SHWeL and N-Grams

While the origins of our new SHWeL vectorization approach come from compression, we note that there is an interpretation relating them back to the n-gram approach. Because each feature in a SHWeL vector corresponds to a byte sub-sequence α, SHWeL can be thought of as a type of n-gramming for multiple values of n, for which we do not need to perform any expensive feature selection.

In SHWeL, the values of n are determined dynamically from each individual datum. Because of the LZ process, we can expect many smaller values of n to occur when processing files of high entropy, as this corresponds to the worse-case compression scenario and thus builds a dictionary of the shortest sub-sequences. When presented with a file of no entropy (i.e., the same byte repeated over and over), SHWeL will obtain progressively larger n-grams until reaching the end of the file. These two extremes map to our *a priori* belief of what the correct behavior should be. When given high entropy inputs, it will be more difficult to find matches to longer n-grams, and so we should avoid producing them. Similarly, when a file is of low entropy, the ability to match long sub-sequences becomes more likely, and SHWeL will adapt to produce such sequences.

The feature selection for SHWeL, which is the min-hashing selection as if we were to perform Jaccard similarity computations, allows us to circumvent the need to process a much larger set of features that could be produced from any given input. While its form gives us the ability to confidently get the same features selected for multiple files without any coordination, it also leaves open the possibility of increased performance if we were to select the features more intelligently. One question would be if the treatment of the α_i selected by SHWeL/LZJD as classical features, as if we had obtained them from n-gramming, could significantly improve the performance of our approach. This would be in place of performing feature selection by selecting the k features with minimum hash values, and so would also necessitate increased training time and coordination. This presents a possible spectrum between the scalability of SHWeL (which we were able to run in under a day on 2 million binaries), and the computational burden the n-gram approach presents (which took weeks with a cluster of computers). Determining this, and any potential balances between the approaches or alternatives, is a question for future work.

6.6 Conclusions

In this chapter we have proposed the new SHWeL algorithm for vectorizing arbitrary byte sequences, and shown its applicability to malware classification and superiority to LZJD. By exploiting the sensitivity of LZJD to byte perturbations, we are able to over-sample raw binaries to produce multiple realistic feature vectors for a single byte sequence. This allows us to tackle the class imbalance problem directly, providing better accuracy and scalability compared to the seminal SMOTE algorithm. Our new approach also improves upon the accuracy of the byte n-gram model for malware detection by allowing it to easily scale to larger corpora, and avoid over-fitting to the training distribution.

Chapter 7: Lempel-Ziv Jaccard Distance, an Effective Alternative to Ssdeep and Sdhash

We have now built a classification system out of LZJD for Windows and Android malware. While this does exercise one of the primary purposes of being domain-knowledge free, being usable on multiple file types, it is important to show such ability in a broad spectrum. To emphasize LZJD's flexibility, we apply it to many new file types simultaneously in a new problem domain of digital forensics.

7.1 Introduction

In forensic investigations of IT environments, there has been a long recognized and ever increasing need to find similar files for a number of scenarios, including file clustering, detecting blacklisted material, and finding embedded objects [145]. Initial triage and screening of data can easily enter terabytes of data, collected from email archives, hard drives, USB peripherals, and network traffic[146]. Such needs occur in many other areas as well, such as firmware analysis [147] and malware triage[148], [149].

Finding similar files is often a daunting task, since manual inspection can take hours per file, if possible at all. The need to automate this task has led to the development of many similarity digests or "hashes" [30], [32], [150]–[152]. Similar to hash functions like MD5 or SHA1, these digests convert an arbitrary string of bytes into a shorter identifying byte string. However, whereas a hash function like MD5 is designed to produce dramatically different output for even a one byte change in the input, these similarity digests are designed to produce little if any change in output given a small change in input. By making the similarity hash insensitive to changes in the input, we can compare the hashes themselves as a method of comparing the similarity of two files.

The two most popular and well known similarity hashes [145] are ssdeep [30] and sdhash [31], which have become the standard benchmarks in the field. While ssdeep is often ineffective for many data types, it is readily available and one of the fastest hashing methods in use. In particular, ssdeep is sensitive to byte ordering, which is a weakness for formats that support arbitrary re-ordering of contents (such as binary executable files). While sdhash is slower than ssdeep, it makes up for runtime performance loss with significantly improved matching and detection ability and is considered state-of-the-art in this regard[145]. The sdhash program's improved matching and detection is the result of resolving the byte reordering weakness of ssdeep.

While ssdeep and sdhash are popular fuzzy hashing techniques, they have made a number of design or implementation choices that may not be suitable for all the files types we may consider now or in the future. The ssdeep algorithm uses a context triggered approach, and the context itself is dependent both on file length and a minimum block-size b_min, and a signature length S. Both of these are set to constants without explanation on the determination of these constants, or exploration of their impact. Ssdeep also uses a weighted edit-distance to determine final match scores, without explaining the determination or intuition for the values of the weights[30]. The sdhash algorithm similarly has a number of parameters which must be set, and states they are determined empirically from some set of data[31]. However, this data may not accurately reflect the content of interest for practitioners at large, yet the same parameters are now used — and no tool is provided to re-calibrate such parameters to a desired data type of interest. The scoring method used by sdhash also results in the undesirable property that $sim(A, B) \neq sim(B, A)$ [34]. It can also be difficult to interpret the exact score returned by these methods. For example, Roussev and Quates [146] recommends treating any score in the range of $[21, 100]$ as "Strong" in terms of correlation. This covers 77% of all possible values returned by sdhash. This may be in part because the digests have been designed to detect near-identicalness, rather than measuring a continuous degree of similarity[153].

In this chapter, we propose our Lempel-Ziv Jaccard Distance (LZJD) [43] as an alternative similarity digest. The lack of domain knowledge present in LZJD should allow it to work for a wider class of potential file types compare to previous approaches like sdhash. To make LZJD practical for this setting, we will also introduce a new and faster approximation of LZJD that continues to maintain all metric and kernel properties, but can be another order of magnitude faster than its original form.

We will show four primary benefits of using LZJD as a similarity digest. First, the time it takes to compare two hashes is orders of magnitude faster with LZJD compared to sdhash, which is critical when dealing with large signature indexes. Second, the LZJD score can in practice be interpreted as a lower bound on how similar the binary contents of two files are. This interoperability is not present in current digest methods. Third, LZJD is better at matching a file fragment with its source file (i.e., the source file receives the highest matching score compared to all other files) compared to both ssdeep and sdhash. We suspect that LZJD sets a new state-of-the-art in this regard. Fourth, the digest size of LZJD is fixed, making the determination of index size trivial.

The rest of this chapter is organized as follows. We briefly cover our motivation for investigating LZJD as a similarity digest in subsection 7.1.1. In doing so we will give our interpretation of the LZJD approach that leads us to believe it will make an effective similarity digest. Since efficient execution time is critical to tool adoption and use, we detail how we develop a faster version of LZJD in section 7.2, and compare results to the original LZJD work to confirm that our approach has no loss in accuracy while obtaining higher throughput. These tests will also include ssdeep and sdhash to show LZJD's superiority in a related domain, and a significant failure case for sdhash. Given our new efficient LZJD, we evaluate its abilities as a similarity digest in section 7.3 using the FRASH framework[33]. We will discuss the meaning and importance of our results in section 7.4, followed by our conclusion in section 7.5.

7.1.1 Motivating LZJD as a Similarity Digest

As introduced in chapter 5, LZJD is a distance metric between arbitrary byte sequences. To be an effective similarity digest, we need two things: compact digests of arbitrary files, and accurate matching in multiple scenarios. It is the min-hashing that accelerates LZJD that inspires us to investigate its potential as a similarity digest. The application of min-hashing gives us a theoretical bound on the error when computing similarity between files [126], [127]. Given this, it is reasonable to assume we can store a compact digest/hash of the file (the min-hash set). LZJD's accuracy in malware classification gives us confidence that the approach will also perform well in the matching part of the task. These two assumptions form the inspiration for this chapter.

While the LZJD digest will be of a number of elements k, the size of the digest on disk may be variable since each item in the LZSet may be a variable number of bytes in length. One might desire a constant digest storage size to make storage planning simpler, and it can also aid in efficient implementations by reducing degrees of freedom (which will allow for more performance optimization). We achieve this in this work with our design of a faster implementation of LZJD, which we will detail in section 7.2, and show that we are able to obtain a digest with fixed storage size and considerable performance improvements without compromising on the accuracy of LZJD.

We also argue that LZJD will provide a potential novel benefit for the forensic investigator. Specifically, that the grounding in Jaccard similarity approximations is means LZJD's score will be more interpretable than the scores produced by ssdeep and sdhash. For a direct interpretation of the math behind the LZJD score, consider two inputs A and B. A score of 0.75 means that, for all sub-strings shared between the LZSet(A) and LZSet(B), 75% of them could be found in both files. This can be loosely interpreted as saying that A and B share 75% of their byte strings. This is not an exact measure of byte content similarity, and will be impacted by two primary factors. First, that the hashing of sub-strings does not attempt to maintain information about string length. We expect this to be approximate to the average string length over many hashes, but this will introduce variability in the scores. Second, that the LZ set creation can be impacted by the contents of the binary, so it is possible to produce different sets for similar inputs. We will see that this issue does impact the score returned, but does not seem to reduce the matching ability of LZJD. We also note that sdhash has a similar issue where inputs can be modified by an adversary to reduce the matching score[154], but has found widespread use regardless. So we do not believe this potential shortcoming would be a hindrance in practice.

To ground this "loose interpretation" in a more concrete measure, we note a unique property of the tests in the FRASH framework [33]. For each test, we can analytically determine what the Levenshtein distance [155], or edit-distance, between files would have been in each test. The edit-distance being the minimum number of operations needed to transform one string into another, where an edit can either replace, remove, or add a byte to the string. The edit distance between two binary files would not normally be computationally feasible, as it is an $O(n^2)$

cost to determine this value for two strings of length n.

With this insight, we find that LZJD tends to act as a lower bound of (7.1)

$$J(\text{LZSet}(A), \text{LZSet}(B)) \lesssim \frac{\text{edit-distance}(A, B)}{\max(|A|, |B|)} \tag{7.1}$$

We use the approximately less-than symbol $\lesssim$ because this is not a proven bound, and does not hold for every experiment. For the majority of tests in the FRASH framework, this bound does hold. The results of individual tests behaving in this way, and the one exception, will be discussed throughout the paper as we review the FRASH test results.

Ultimately, the LZJD$_h$ similarity/distance performed orders of magnitude faster than NCD, with equal or better accuracy, on several malware datasets for both malware detection (correctly labeling a binary as benign or malicious) and malware family detection (finding the correct malware family for a known malicious binary). This success, combined with its use of a fixed-length digest for faster distance computations, inspires our hypothesis that it could be successfully used for the same kind of digital forensic scenarios as ssdeep and sdhash. We evaluate this feasibility in section 7.3. But first, we must further improve the runtime efficiency of LZJD to make it practical for this application.

7.2 A Faster LZJD Implementation

We now review the high level details of the original LZJD implementation, and discuss our modifications that result in a faster variant appropriate, which we denote as LZJD$_f$, for the forensic use case. This implementation is in Java, and we note that both ssdeep and sdhash are written in C/C++. This may mean that there is still room for improved performance of our new LZJD implementation. We have made a Java implementation [1] of this faster LZJD available to the public. The program has the same command line arguments as sdhash in order to facilitate integration with existing work flows. We are also working on a C++ version [2] , though performance optimization is not yet complete

The original version of LZJD was a rather naive Java implementation. The set s in Algorithm 1 was a simple HashSet of ByteBuffers. A ByteBuffer object represents a byte string. This choice meant that equality comparisons had to compare each byte in each buffer, which would take time linear with respect to the current sub-string under consideration. Furthermore, and to the detriment of performance, these comparisons force the hash of the string to be re-computed at every step, resulting in redundant work.

Once the set of ByteBuffers was obtained, the MD5 hash of each member in the set was computed and the lower 32 bits used for the min-hashing. This set of integers was then sorted, and the minimum k integers created the final set used for this faster variant of LZJD, which we will denote as LZJD$_h$. The MD5 function was chosen to ensure even distribution of hash values, which are the result of its original design as a cryptographic hash function.

We will present tests in subsection 7.2.1 that show these modifications do not

[1] https://github.com/EdwardRaff/jLZJD
[2] https://github.com/EdwardRaff/LZJD

degrade the accuracy of LZJD but do significantly reduce the runtime cost. We do this by performing hashing continuously as data is read in, and representing every sub-string by the hashed integer counterpart. By using a hash that we update with one byte at a time, we no longer need to read the entire file into memory for LZJD$_f$ to work. This may result in collisions during the LZ set construction as two hashes may collide to the same integer, but we believe the cost of such collisions to be minimal. The LZ algorithm will simply continue processing the next byte, which is now a new sub-string that is one byte longer. It is necessary that this new sub-string does not currently exist in the set, because the previous set did not contain the true prerequisite sub-string either. For the new sub-string to also have a collision becomes astronomically unlikely, assuming the hashes are uniformly distributed. Even if several collisions occurred, the impact on the output similarity should be minimal, as the sub-strings of each sub-string are also in the sets and included in the comparison. That is to say, if the sub-string "abcdefg" is not included in the set due to a hash collision, the contributions of "abcdef", "abcde", etc., are still present.

To make sure these hash values are of a high quality, but avoiding the unnecessary quality of a hash function like MD5, we use the MurmurHash3[3] function. This hash function is designed to have an even distribution of hashes and require minimal CPU time for computation. While not originally designed for it, we re-implement this algorithm so that the hash can be updated one byte at a time. This requires keeping a four-byte memory that is updated and used to compute the running hash output, in addition to the internal state of the MurmurHash3 algorithm.

[3] https://github.com/aappleby/smhasher

We also optimize the integer set object to take advantage of the two unique artifacts of the situation. First, it only needs to support the insertion of integers, so no removals are needed. Second, since the integer values are hashes, there is no need to apply any kind of hash function to them, as they will already be evenly distributed (i.e., our hash set can use the identity function as its “hash” function). We thus adapt an open addressing scheme with double hashing [156, p. 528–529] that is normally used for a hash table. We can reduce memory use by ignoring the “value” part, and using a boolean array to indicate if an entry is free or filled, and remove logic normally needed to handle the removal of entries. The “key” alone will then act as the set entry, with an implicit null “value”. This reduces memory use and execution time.

Once the entire file is processed, we will have a set of integers, which we will then convert to a list of integers. Rather than naively sorting the list, which is $O(n \log n)$, we instead apply one of many algorithms that returns us the k smallest items in $O(n)$ time [157]. Beyond optimizing how the set of k values is obtained, we can further improve how they are stored and compared.

The original LZJD would store the set of k integers in a set object, and to compute the size of the intersection of two sets, would iterate over one set and query for its entries in the other. This results in $O^*(k)$ time complexity, but is both memory inefficient and results in random memory access that negatively impact cache and pre-fetching performance. Instead we store the k items in a sorted array, which is $O(k \log k)$, but $k << n$, so this sort is of minor impact. The benefit is that we can compute the intersection by doing a merge-sort like comparison of the values in each array, incrementally stepping forward in one list when its value is less than another. This well-known approach is given in Algorithm 3, and results in a non-amortized $O(k)$ runtime for digest comparisons. Further, the dense arrays are more memory efficient, and the incremental walk through the sorted arrays will work *with* the hardware pre-fetching for improved performance.

Algorithm 3 Set Intersection Size via Sorted Lists

```
 1: procedure INTERSECTION(Integer arrays $a$ and $b$)
 2:     $\text{pos}_a \leftarrow 0, \text{pos}_b \leftarrow 0$
 3:     $size \leftarrow 0$
 4:     while $\text{pos}_a < |a|$ and $\text{pos}_b < |b|$ do
 5:         if $a[\text{pos}_a] < b[\text{pos}_b]$ then
 6:             $\text{pos}_a \leftarrow \text{pos}_a + 1$
 7:         else if $a[\text{pos}_a] > b[\text{pos}_b]$ then
 8:             $\text{pos}_b \leftarrow \text{pos}_b + 1$
 9:         else                                  ▷Equal values, means item was in both
10:             $\text{pos}_b \leftarrow \text{pos}_b + 1$
11:             $\text{pos}_a \leftarrow \text{pos}_a + 1$
12:             $size \leftarrow size + 1$
13:         end if
14:     end while
15:     return $size$
16: end procedure
```

7.2.1 LZJD Speedup Results

Having specified the modifications that produce the faster LZJD$_f$, it is important to validate that the hashing approach does not meaningfully degrade accuracy compared to the original LZJD$_h$. To do so, we will repeat the malware family classification experiments used in [43]. The malware classification problem has been previously identified as an area where similarity digests could be useful[145], making this test of particular relevance in this context of similarity digest comparisons. For this reason we will also include ssdeep and sdhash in this comparison, and see that LZJD$_f$ outperforms them both.

Malware family classification can be seen as a close corollary to the digital forensics problem of finding a related file. For each malware sample, we wish to identify the family it belongs to by comparing the sample to a database of known malware. Each specimen in the same malware family is intrinsically similar, and can be seen as one unit of "sameness" for which the inter-family similarity should be higher than the similarity to any other arbitrary sample. This task is strongly correlated with matching a modified file to its original file, but can be seen as a more challenging scenario. This is because malware is often written by an active adversary which attempts to avoid detection. Metamorphic malware, which changes itself upon propagation, makes this a common and difficult scenario [117], [158].

The two malware datasets used each have two variants of the experiment. The Microsoft malware comes from a 2015 Kaggle competition, and the data is provided and labeled by Microsoft [64]. There are 9 malware families in 10,868 files. The first variant of this dataset uses only the raw bytes of the original files, with the PE-header removed[4]. The raw binaries take 50.8 GB of storage space, and we will refer to this dataset as "Kaggle Bytes". The second variant is the disassembly produced by IDA-Pro, which is a more human-readable version of the files. This variant takes up 147GB of space, and we will refer to this dataset as "Kaggle ASM".

The second dataset is Android malware from the Drebin corpus [65]. Following [43], we remove any malware family that had less than 40 samples. This results

[4]The PE header info was removed by Microsoft to avoid accidental infection, and cannot be reversed.

in a dataset with 20 malware families and 4664 samples. Android applications are normally distirbuted as APKs, which are simply zip-files. Because the compression applied by zipping the contents can impact the effectiveness of our hashes, we evaluate the dataset in two ways. One using the raw APKs ("Drebin APK"), and the other using an uncompressed tar of the APK contents ("Drebin TAR"). These variants take 6.4GB and 8.6GB respectively. Differences in performance between these two datasets can be wholly attributed to the impact of compression[5], since it is the only source of variation between the two sets.

We also note the importance of these tests in regards to the performance of LZJD and other tools in high-entropy situations. LZJD was analytically predicted to experience sub-optimal behavior when encountering high entropy data, yet empirically performed well when given such data [43]. The impact of high entropy is discussed further in the FRASH tests in subsubsection 7.3.2.1, which use random bytes as part of the test to increase the matching challenge. The Kaggle and Drebin datasets help to validate that LZJD works even when high entropy is present, with the Android APK corpus having a median byte entropy of 7.96. Thus the performance of LZJD, ssdeep, and sdhash in this task can be seen as a test of all three approaches when dealing with higher entropy content.

To evaluate all of our hashing options on this dataset, we will use 10-fold cross validation. We will use the 1-nearest neighbor algorithm to classify each sample against the other folds. If the matching algorithm returns the highest similarity score

[5]We note that the amount of compression applied to the APKs is generally light, as a trade-off is being made between storage size and power consumption, both limited resources on mobile phones

for a member of the same malware family, then the algorithm correctly classified that point. For each fold we will measure the balanced accuracy [103]. The balanced accuracy gives equal total weight to each class. This is useful since the malware families are not evenly distributed, and results would be skewed upward by the most populous families. The accuracy for each method on each dataset is presented in Table 7.1.

Dataset	ssdeep (%)	sdhash (%)	LZJD$_f$ (%)	LZJD$_h$ (%)
Kaggle Bytes	38.4 (1.4)	60.2 (2.3)	98.0 (1.2)	97.6 (1.5)
Kaggle ASM	26.6 (2.2)	28.8 (1.3)	96.7 (1.9)	97.1 (2.0)
Drebin APK	13.6 (1.6)	5.8 (0.5)	81.3 (4.6)	80.8 (2.6)
Drebin TAR	24.2 (2.9)	8.3 (1.2)	87.5 (2.0)	87.2 (2.8)

Table 7.1: Balanced accuracy results on each data and feature set. Evaluated with 10-fold CV, standard deviation in parenthesis.

Here it is easy to see that our new LZJD$_f$ does not meaningfully change the performance on these datasets compared to the original LZJD$_h$. The largest change is an increase in standard deviation on the most difficult dataset (Drebin APK). However LZJD$_f$ has slightly higher mean accuracy and lower standard deviation on most of the datasets. This closeness in results indicates the high fidelity of our new approach, and that the simplifications in LZSet implementation do not meaningfully impact the quality of results. This gives us confidence that our changes to LZJD$_f$ will generally perform well.

Comparing both LZJD implementations to ssdeep and sdhash, we can see far superior classification accuracy. The closest either ssdeep or sdhash come to matching LZJD's performance is on the Kaggle Bytes dataset, where sdhash still trails by over 37 whole percentage points. We will see this trend of LZJD having superior matching ability repeated in section 7.3.

While sdhash performs better than ssdeep on the Kaggle datasets, we also see sdhash produce degraded results on the Drebin datasets. Its scores of 5.8% and 8.3% accuracy are barely better than the 5% threshold for random guessing. When inspecting these results manually, we discovered that the root cause is related to the nature of sdhash's scoring algorithm. Sdhash ends up keying off features generally common to all of the Android samples in our corpus, producing average nearest neighbor scores of 99.7 and 99.9 for Drebin APK and Drebin TAR respectively. This use case provides credence to the desire for a more principled and interpretable score function.

	Difference		Absolute Difference		Relative Difference	
Dataset	Avg.	Stnd. Dev.	Avg.	Stnd. Dev.	Avg. (%)	Stnd. Dev. (%)
Kaggle Bytes	0.231	0.871	0.647	0.627	0.755	0.864
Kaggle ASM	0.010	0.793	0.531	0.588	0.601	0.783
Drebin APK	0.010	0.691	0.489	0.489	0.539	0.660
Drebin TAR	-0.056	0.623	0.450	0.434	0.491	0.624
t5	0.112	0.505	0.332	0.397	0.351	0.445

Table 7.2: Statistics on the direct, absolute, and relative differences between LZJD$_h$ and LZJD$_f$ similarities for all pairwise distances. Scores for the first four columns are out of a maximum score of 100 for the difference. The last two columns are shown in percentage points.

To further confirm the high fidelity of LZJD$_f$'s approximation of LZJD$_h$, we also look at the statistics of all pair-wise distance computations in each dataset, and include the t5 corpus that will be further discussed and used in the next section of this work. We will look at three sets of statistics, where $d_h = \text{LZJD}_h(A, B)$ and $d_f = \text{LZJD}_f(A, B)$: first, the average difference, $d_h - d_f$, which we want to see centered around zero (indicating the approximation is unbiased in practice).

Second, the average absolute difference, $|d_h - d_f|$, which we wish to see being as small as possible (indicating the approximation is accurate). Last, we will consider the relative difference, $|d_h - d_f| / \max(d_h, d_f, 0.01)$, which helps us further consider changes based on their relative magnitudes. We add the 0.01 term to the relative difference computation to avoid division by zero, which occurred when there was no error in the approximation of files with no similarity.

The average and standard deviations for these three statistics are presented in Table 7.2, where the maximum possible difference would be 100. We can see that the average relative difference is less than a percentage point. Going out by three standard deviations from the mean is still less than a 4% error. Similarly the worst average absolute difference indicates that the majority of scores will differ by no more than 4 points out of 100. We also see that the average total difference is centered around zero. These results give us clear validation that our LZJD$_f$ approximation is not only faithful to the true nearest-neighbor ordering provided by LZJD$_h$, but also accurately reproduces the same score values. That is to say, we have empirically observed that $|d_f - d_h| < \epsilon$.

Dataset	ssdeep	sdhash	LZJD$_f$	LZJD$_h$
Kaggle Bytes	3.02×10^3	8.64×10^5	3.17×10^3	1.73×10^4
Kaggle ASM	3.25×10^3	4.74×10^6	1.44×10^4	4.85×10^4
Drebin APK	2.21×10^2	1.30×10^4	5.56×10^2	7.17×10^3
Drebin TAR	2.76×10^2	2.04×10^4	6.46×10^2	7.65×10^3

Table 7.3: Total evaluation time for each method in performing 10-fold CV. Time presented in seconds.

To evaluate the runtime of our new LZJD$_f$, we can see the total time taken for both hashing the files and performing the nearest neighbor searches in Table 7.3.

Dataset	ssdeep	sdhash	LZJD$_f$	LZJD$_h$
Kaggle Bytes	5.42×10^2	1.72×10^3	1.92×10^3	1.22×10^4
Kaggle ASM	2.26×10^3	5.34×10^3	7.44×10^3	4.11×10^4
Drebin APK	1.85×10^1	2.35×10^2	3.74×10^2	4.99×10^3
Drebin TAR	2.51×10^1	3.52×10^2	4.36×10^2	5.47×10^3

Table 7.4: Time spent hashing for each method in performing 10-fold CV. Time presented in seconds.

As desired, we can see that LZJD$_f$ is consistently faster than LZJD$_h$, by a factor of 3.4 to 12.9. We can further see that this total evaluation time is comparable to ssdeep, and generally two orders of magnitude faster than sdhash. These large speed advantages generally come from LZJD$_f$ being faster to compare. These results support our claim that our new LZJD$_f$ is fast enough to be a practical alternative to both ssdeep and sdhash.

In Table 7.4, we show the time spent creating the digests for this same task. This allows us to see that for ssdeep, sdhash, and LZJD$_f$, creating the digest itself is usually small relatively to the amount of time spent. When running theses tests the data was read from hard disk, and we see that LZJD$_f$ takes 12% to 60% more time to create the digest compared to sdhash. Given that the total time for LZJD$_f$ is orders of magnitude faster than sdhash, this allows us to confirm that the fast comparison time is the source of this dramatic speed advantage. We will explore the performance differences further in subsection 7.3.1.

Looking at the hash time also demonstrates the critical importance of our optimization toward practical use. For LZJD$_h$, the hashing time is one to two orders of magnitude greater than our improved LZJD$_f$. LZJD$_h$ is the only metric which spends the majority of time in producing the digests itself for every dataset.

These optimizations were thus necessary to make the tool usable for practitioners, with digest time comparable to sdhash while providing faster digest comparison. For the remainder of this chapter, we will simply refer to LZJD$_f$ as LZJD for brevity.

7.3 Similarity Hash Comparisons using FRASH

The evaluation of similarity digests is not a trivial matter. It requires a diversity of file types (that should reflect real world content) and some level of ground-truth about which files are similar to others. Roussev [152] introduced the **t5** corpus for such evaluations[6], and a manual evaluation of sdhash was performed. The t5 corpus contains a number of different file types, summarized in Table 7.5. Roussev also proposed a number of challenges for which one would want to use a similarity hash, which Breitinger, Stivaktakis, and Baier drew from to create the automated FRASH test suite [33]. The FRASH tests combined evaluate four desirable qualities:

	html	text	pdf	doc	ppt	xls	jpg	gif
Number of Files	1093	711	1073	533	368	250	362	67
Avg. File Size (KB)	66	345	590	433	1003	1164	156	218

Table 7.5: Contents of the t5 corpus. There are 4475 files in total, totaling 1.9 GB in size.

1. *Document similarity detection*: where we wish to determine which documents are intrinsically related, such as multiple revisions of the same word document.

2. *Embedded object detection*: where the goal is to detect that one object type (such as an image) has been placed inside of another (such as a email document).

[6]available at `http://roussev.net/t5/t5.html`

3. *Fragment detection*: where we are given a sub-string of some larger file, and we wish to identify the source of this sub-string.

4. *Clustering files*: where we wish to group similar files together.

The FRASH suite is written in Ruby[7], and allows for easy integration of new similarity hashing schemes. The tests are divided into two higher level sections. The first section is *Efficiency*, which measures only runtime properties of the hash digest. This includes the digest time, comparison time, and hash size relative to the input. Our improvements to the LZJD algorithm tackle only these quantities, which are critical when up to terabytes of data may need triaging[146], [159].

The second, and more expansive, are the *Sensitivity & Robustness* tests. These evaluate the ability of the hash function to perform matching under various circumstances, and the quality of the match score returned in each scenario. These tests will show that LZJD possesses a superior ability to correctly match a fragment to the correct source file, even when presented with significant byte alterations or comparatively small fragment sizes.

Below we will present and discuss the results from each of the tests in the FRASH suite. For each result we will only present a portion of the output for brevity and readability, with the algorithm getting the most successful matches shown in **bold** for each test. More complete results can be found in A.1. All tests were run on a computer running OSX version 10.10.5. With a 2.66 GHz Intel Core i5 CPU and 16 GB of RAM. In initial testing, the FRASH code was highly sensitive

[7]FRASH is available at `http://www.fbreitinger.de/wp-content/uploads/2017/04/FRASH_1.01.zip`

to random read/write time. Initial runs on a standard HDD resulted in runtime that would take days for as few as 20 files. For this reason, all code and data used were stored in a RAM Disk. This is a method by which a virtual disk is created on the system that acts likes any other file system, but all stored files are kept only in RAM. This avoided all issues with the random access impact on test runtime.

7.3.1 Efficiency

In this section we are concerned with the computational and storage efficiency of each hashing method. This is measured by computing the hash digest for every file in the t5 corpus, and creating a digest file containing every hash. Once complete, all $n^2/2$ pairwise distance computations are done. This allows us to measure the runtime efficiency of the hashing process as well as the comparison of hashes, and the storage efficiency of the hash size itself. For only the efficiency tests, the SHA1 hash function is included by the FRASH suite as a benchmark for both time and space. The intuition for comparing with the SHA1 hash is that it serves as a useful barometer for grounding the compute efficiency of storage cost of these digests for those less familiar with them.

	Average	Total	All-pairs	SHA1^{-1}
sha1sum	0.01519	67.7	—	1.00
ssdeep	0.01223	**54.5**	*32.1*	0.81
sdhash	0.04241	189.0	496.5	2.79
LZJD	0.03159	*140.8*	**8.2**	2.08

Table 7.6: Runtime efficiency results. Time taken to compute all hashes for each method, and the time needed to perform all-pairs distance computations. All times measured in seconds using only one CPU core. Best results shown in **bold**, and second best in *italics* (ignoring SHA baseline).

The runtime results can be found in Table 7.6, where we see the average and total time taken to hash the files of the t5 corpus. The total hashing time (second column) is measured using the Unix time command when giving the t5 corpus as the only input for each hashing implementation . The rightmost column shows how many times longer each method took to compute all hashes compared to the SHA1 hash. Here we can see that sdhash is the slowest hash function by a factor of 2.8, and that our Java LZJD implementation is 34% faster than sdhash. Ssdeep is the fastest at 23% faster than the SHA1 algorithm, but lacks in its ability to perform accurate matching once the hashes are produced.

We note that while the FRASH test showed LZJD was faster in hashing time compared to sdhash, our tests in section subsection 7.2.1 showed it to be slower. The difference between these tests is the use of the RAM disk for FRASH. These results combined would indicate that sdhash and LZJD are roughly comparable in hashing time, and we may expect to see variation in which one is faster based on unique hardware combinations and situations.

LZJD's runtime performance is better still when we look at the time needed for comparing the hash outputs, and is over 60 times faster than sdhash in this respect, and still 3.9 times faster than ssdeep. This would indicate that LZJD would be preferable in a situation where we have many known objects of interest in a database, and need to process the contents of a new device against the known database. One might argue that having a faster digest comparison is more important than a faster digest calculation. Indeed, others have worked on building special indexes specifically to accelerate the bottleneck of comparing many digests[151].

The time needed to hash n files is naturally an $O(n)$ task, but comparing the n derived hashes to an existing database of m hashes is $O(nm)$ in complexity. The latter will clearly become the dominant cost as the number of objects under consideration increases, and so we would want to minimize its base time requirements as much as possible. An example of this can be found in [147], where sdhash and ssdeep were used on 1.2 million files extracted from firmware images. Due to the computational burden at comparison time, these hashes couldn't be applied to the entire corpus. Our results in subsection 7.2.1 corroborate this high comparison time cost, where we see LZJD compare favorably to both ssdeep and sdhash. LZJD's efficient digest comparison pushes back this limitation.

We observe that this issue of runtime efficiency has been noted before, and others have attempted to build more efficient indices for specific use cases. Winter, Schneider, and Yannikos [160] built an indexing scheme for the ssdeep algorithm, but ssdeep's low precision and recall limit the utility of such a tool. Breitinger, Baier, and White [161] build a more general purpose index that is compatible with sdhash, but cannot return or filter based on similarity scores or indicate which specific file as a match. This work was later extended to resolve these issues, allowing it to return exact file matches [162], [163]. While able to obtain speedups of up to a factor of 2.6, it does not guarantee all matches will be found.

LZJD provides a sound method of circumventing these issues that may be explored in future work. Since LZJD is a valid distance metric, it avails itself to more principled and existing indexing strategies that are designed for metric spaces. These indices support $O(\log n)$ query time[164]–[167] and guarantee that all neighbors will be found.

7.3.1.1 Compression

The efficiency test in FRASH also produces a set of compression results. These results are concerned with the size of the hash digest with respect to the original file sizes. All things being equal, it is preferable to have a digest that is smaller rather than larger. A smaller digest size allows for the storage and transport of larger databases, and gives some indication about the information efficiency of the digest itself.

	Avg. Length	Avg. Ratio (%)	Max Length (ratio)	Digest Size
sha1sum	20 B	0.0047	—	420 KB
ssdeep	57 B	0.0133	78 B ($\leq$0.01%)	592 KB
sdhash	10.6 KB	2.5203	409 KB (2.93%)	61.3 MB
LZJD	4.01 KB	0.9566	4.01 KB (10.1%)	23.5 MB

Table 7.7: Compression test results. First column shows the average length of the digest, followed by the average ratio between digest length and original file length. The last two columns show the maximum ratio encountered and the size of the entire digest for all files.

The compression results are shown in Table 7.7, where the first two columns present the average length of the digest, and the average percentage of the digest size with respect to the original file. Here we can see that SHA1 and ssdeep both produce very small digests. Sdhash produces the largest digests, with an average of 10.6 KB that is usually 2.5% of the original file size. LZJD falls in a middle ground, with an average digest of 4 KB, 2.65 times smaller than sdhash. By the nature of our LZJD hash, the digest size will never be more than 4 KB[8] for $k = 1000$. Smaller digests

[8]With minor overhead for the header matching sdhash's output style.

may occur for small files which can be represented with less than 1000 dictionary entries for the Lempel-Ziv process. This makes LZJD especially effective for large files, with theoretical support for its method of production. However, LZJD's fixed size can also result in an overly large digest for small files, as can be seen by the maximum digest-size to original-size ratio of 10%.

7.3.2 Sensitivity & Robustness

We will now review the Sensitivity & Robustness tests that are a part of the FRASH framework. Tests subsubsection 7.3.2.1 and subsubsection 7.3.2.2 will run a digest comparison on only two files at a time, namely a source file and a target file. The source file will be an unaltered file from the t5 corpus. The target file will be a modified version of the source file. These tests will be measuring behavior of the scoring methods used and how they change with changes to a single file. The implicit assumption of the FRASH framework is that a higher score between two matching files is *always* better, all other things being equal. As we discussed in subsection 7.1.1, the LZJD score will be based on the amount of byte similarity – and will not attempt to reflect "match or no match" as ssdeep and sdhash do. This makes comparing the results in these tests more challenging, and we will discuss the issue of score function behavior further in section 7.4.

Tests subsubsection 7.3.2.3 and subsubsection 7.3.2.4 will generate a digest database from the whole t5 corpus, and then see if a target file (still a modified version of one of the t5 source files) can be correctly matched to its source. In these tests the goal is for us to correctly match a file to its source, and can be viewed as equivalent to the nearest neighbor problem we visited in section 7.2. These tests can be thought of as a harder variant of the task from a machine learning sense, as there is only one correct neighbor for each test point (which would be the source point), where any file from the same class would be considered correct for the tests done in section 7.2. We will see that LZJD far exceeds both ssdeep and sdhash in its matching ability.

7.3.2.1 Single Common Block correlation

The Single Common Block (SCB) test is designed to determine how small a "common block" of identical content can be before a digest algorithm produces a score of zero (i.e., no commonality). This test compares only two files at a time, where each file has random byte contents. A portion of each file will be set to the same common content, and this common block will be iteratively decreased in size. This test was run 50 times with common blocks extracted from 50 different source files. In the original FRASH testing, it was found that sdhash was able to produce matches for smaller common blocks then ssdeep, but ssdeep was able to produce higher matching scores.

For the tables in this section, the *Average Block size (KB)* indicates how small the common object's size was to reach a score greater than or equal to a minimum score threshold. Similarly, *Average Block size (%)* is how small this single common object was as a percentage of the block size. The *Matches* line in each table is the number of files (out of the 50 selected) that were able to be matched and achieve a score at or above the given score. The two aforementioned averages are with respect to the files matched at that level.

This particular test puts LZJD at a disadvantage, because its score does not have the same meaning as sdhash and ssdeep, and because the files are produced with completely random byte sequences. Random bytes are a weakness of LZJD in the case of matching similarity, because the LZ algorithm will begin collecting all smallest sub-strings, which will cause a non-zero match to occur. This makes it impossible to reach the original termination case of FRASH, and we we terminate LZJD in this test after a SCB size of 16KB. Thus, when interpreting Table 7.8 and Table 7.9, the score that has an average block size of 16 KB should be treated as the same as the zero score for sdhash and ssdeep.

	Score	≥25	≥15	≥10	≥5	0
ssdeep	Avg. block size (KB)	386	—	—	—	393
	Avg. block size (%)	18.9	—	—	—	19.2
	Matches	23	—	—	—	50
sdhash	Avg. block size (KB)	730	501	383	188	17.9
	Avg. block size (%)	35.7	24.5	18.7	9.17	0.88
	Matches	34	44	50	50	50
LZJD	Avg. block size (KB)	—	868	376	16	—
	Avg. block size (%)	—	42.4	18.4	0.78	—
	Matches	—	46	50	50	—

Table 7.8: Single Common Block results for a 2MB file. Columns show the scores achieved, and rows the size of the common block of content, and the number of successful matches (max of 50).

Inspecting the results for a 2MB total block size in Table 7.8, LZJD does not do well in this particular test. LZJD is unable to produce scores in the same large ranges as sdhash and ssdeep, but LZJD is also not designed to produce such scores. The use of completely random byte strings as the filler content of the SCB test also deflates the score LZJD gives, due to the increased number of sub-strings the Lempel-Ziv algorithm will find within these high entropy regions. This is a worst-case scenario for LZJD, as was theoretically analyzed in [43].

	Score	≥25	≥20	≥15	≥10	0
ssdeep	Avg. block size (KB)	94.3	106	—	—	393
	Avg. block size (%)	18.4	20.6	—	—	19.2
	Matches	28	5	—	—	50
sdhash	Avg. block size (KB)	185	160	140	107	16
	Avg. block size (%)	36.1	31.3	27.3	20.9	3.12
	Matches	32	37	44	50	50
LZJD	Avg. block size (KB)	226	80.6	16.6	—	—
	Avg. block size (%)	44.2	15.8	3.25	—	—
	Matches	39	50	50	—	—

Table 7.9: Single Common Block results for a 512 KB file. Columns show the scores achieved, and rows the size of the common block of content, and the number of successful matches (max of 50).

The particular performance of LZJD at the lowest end of the score range is comparable to or better than sdhash, depending on which results are inspected. This can be better seen for a 512KB total block size, as shown in Table 7.9. Here we can see for a score of ≥20, sdhash requires a common block that is 31% of the total block size, where LZJD requires a common block size of only 16%. The results for the 8 MB total block size follow this pattern, and can be found in A.1.

We again note that this test is comparing the score of only two files at a time, and is not as relevant for LZJD since it does not try to produce the same types of score values as sdhash and ssdeep. LZJD's score is best interpreted as an approximate measure of the byte similarity of two files, and in practice, we will see that it is best viewed as an approximate lower bound on the percentage of similar bytes.

Despite the SCB tests being a weak area for LZJD, the use of random bytes in the test construction also make this a worst-case scenario for LZJD. In practice, few files will make use of purely random byte sequences (which would have a byte entropy near 8). One of the only scenarios where we would expect the find such high-entropy sub-strings in a file is when dealing with malware and packed or compressed binaries, which corresponds to the scenario where LZJD was originally demonstrated to perform well [43], where it was tested with Windows malware and Android APKs (which are compressed zip files). Still, removing the impact of completely random sub-strings on LZJD is an area for future research and improvement.

7.3.2.2 Random-noise-resistance

The random noise test attempts to produce false negatives by randomly altering the file one byte at a time. After modification, the test records how many matches are achieved at each score and how many edits where required to reduce the score to that level. Bytes are altered via random insertions, deletions, and substitutions, and location is selected randomly.

As noted in the original FRASH paper [33], the random noise resistance test is computationally demanding, and so we use only a random sample of 100 files from the t5 corpus. Our results find that LZJD is significantly more resistant to such alterations than either ssdeep or sdhash, which further increases the time it takes the tests to run. To reduce test runtime, after 200 edits, we begin altering the files by 10 bytes at a time. Once we reach 2000 edits, we increase to 100 edits at a time, and so on. We also add an early termination after 80% of the file is altered, due to the extreme ranges that LZJD achieves in matching.

The results of running the random noise test are shown in Table 7.10, where *Matches* indicates how many files achieved a given match score, and *Avg. changes* is the average amount of bytes that needed to be altered for this score to appear, as a percentage of that file's size. For example, ssdeep was able to get a score equal to or higher than 70 for only 88 of the 100 files tested. It only took changing 0.005% of the byte contents of a file to lower the score of ssdeep to this level. The better an algorithm's resistance to noise, the more we should be able to alter a file and still obtain a relatively high score. Because ssdeep and sdhash desire to produce a maximal score for any match, we would want to see a maximally high matching score for any percentage of edits. Under the LZJD interpretation of content similarity, we want the matching score to be similar to the percent of byte alterations performed. That is to say, if 25% of the bytes were altered in the target file, we want to see LZJD return a score of 75 (i.e., 100-25% = 75).

In examining the full results (see A.1), it is clear that sdhash performs best when we consider only the higher scores (≥ 55). It routinely obtains the lowest percentage of average changes, followed by LZJD, and then ssdeep. While ssdeep is the only method to obtain the most high scores (≥ 80), this is of little utility due to the small number of changes needed to reduce such scores.

	Score	≥70	≥50	≥40	≥25	≥10
ssdeep	Avg. changes (%)	0.0052	0.0206	0.0615	—	—
	Matches	88	18	7	—	—
sdhash	Avg. changes (%)	**0.1068**	0.2775	0.3940	0.5492	0.9739
	Matches	96	99	100	100	99
LZJD	Avg. changes (%)	0.0238	**0.6061**	**1.967**	**10.99**	**48.63**
	Matches	75	96	99	100	77

Table 7.10: Random Noise tests. Best average number of changes needed to reduce the matching score to a specific level is shown in **bold**.

The robustness of LZJD becomes more apparent when we consider a score of ≥ 50, at which point LZJD requires twice as many byte edits to produce such a score compared to sdhash. Reducing LZJD to a score of ≥ 40 required altering 1.97% of the file, where sdhash produces a score of zero (no match) after an average of only 1.56% of the file is edited. The rate at which LZJD's score is lowered decreases with each byte edit, and so its performance advantage improves dramatically relative to sdhash and ssdeep as we move down in matching score. Reducing LZJD to a score of 25 required 11.0% of the bytes to be altered, which is 20 times greater than for sdhash. At the extreme end, reducing LZJD to a score of ≥ 10 requires editing almost half the file. The 77 matches at this level is lower than 100 because the random noise test *couldn't get LZJD to produce a score that low for many files*, and the FRASH test framework didn't anticipate a scenario where a score of 0 could not be obtained. This indicates a strong matching ability beyond the expectations of the FRASH designers. The FRASH code failed to count the files which obtained a score in the $(25, 10)$ range, and could not be reduced to the $[10, 0)$ before the test was forced to finish running by our modifications.

7.3.2.3 Fragment Test

In the fragment tests of FRASH, a portion of the each file is removed, and then the remaining fragment is searched for against the database of all complete file hashes. The size of the fragment starts at 95%, nearly the whole file, and decreases down to only a 1% portion of the original file. The motivation of these tests are to determine how small a fragment can be while still being matched with the source file. This scenario may occur with any storage or transport format where a file may be broken up into chunks, such as the fragment storage in a file system or individual packets in network traffic.

FRASH runs these fragmentation tests in two modes, one where the file has data removed from the end only (*end cut*), and one where a random portion of the file is removed from both the beginning and end of the file (*random cut*). In the former case, the fragment always starts as the same string of bytes but ends prematurely. In the latter case, the fragment is essentially a random portion of the file (and most likely from near the middle of the original file). The results of the fragment tests are presented in the next two tables. In each table, the File Size (%) is the size of the file fragment as a percentage of the original file it came from.

The ssdeep algorithm is particularly vulnerable to this approach, and is significantly degraded in its ability to correctly match files by a fragment being just 50% of the original file size. Sdhash is more robust, and is not meaningfully impacted in matching ability until fragments are 5% of the original file size or less, where it starts to quickly degrade in accuracy. We also notice a confusing behavior in the

	File Size (%)	95	50	10	5	3	1
ssdeep	Matches (%)	99.9	91.3	0.65	<0.01	0	0
	Avg. Score	96.7	65.9	46.2	61.0	—	—
sdhash	Matches (%)	**100**	**100**	98.1	90.6	81.1	57.9
	Avg. Score	83.4	68.5	75.7	73.4	76.7	81.0
LZJD	Matches (%)	**100**	**100**	**>99.9**	**99.9**	**99.4**	**98.5**
	Avg. Score	72.4	24.9	6.43	3.88	2.73	1.31

Table 7.11: Fragment detection test result, random cut. Column indicates size of the fragment with respect to the source file. Rows show percent of correctly matched files and average score for correctly matched files.

	File Size (%)	95	50	10	5	3	1
ssdeep	Matches (%)	**100**	93.1	1.73	0.49	0.20	0
	Avg. Score	97.7	71.7	56.9	55.7	47.9	—
sdhash	Matches (%)	**100**	**100**	98.3	91.1	82.5	58.7
	Avg. Score	97.3	99.5	97.9	96.9	95.04	90.5
LZJD	Matches (%)	**100**	**100**	**100**	**100**	**100**	**99.7**
	Avg. Score	92.8	40.1	8.33	4.63	3.09	1.36

Table 7.12: Fragment detection test result, end cut. Column indicates size of the fragment with respect to the source file. Rows show percent of correctly matched files and average score for correctly matched files.

average matched score produced by sdhash. In Table 7.12, the sdhash score slowly decreases from high 90s to low 90s, which is a reasonable behavior to expect as the fragment size decreases. However, in Table 7.11, the sdhash score first decreases from the low 80s to the low 70s, and then begins increasing back into the low 80s.

Compared to sdhash, LZJD obtains lower average matching scores. In Table 7.12, these scores are nearly perfectly aligned with the interpretation of a similarity of X% indicating that the X% of the contents are the same. The scores returned for LZJD are a bit below this expectation in Table 7.11, but still match the general trend. This can be explained by the LZSet construction process being sensitive to changes in the byte string, causing changes in the set. In the case of Table 7.12, corresponding to the end cut version of the fragment test, the start of the byte string will remain unchanged. This means the LZ set generated will also be generated in the same order, and will simply stop early once the fragment comes to a premature end. This results in a high quality match of LZ set contents when computing the Jaccard similarity. In the random cut case of Table 7.11, the beginning of the file has been removed. This changes the set of sub-strings computed by the LZ approach, resulting in a lowered match. However the match is still robust, as evident by the high number of matches LZJD obtained.

This robustness in matching ability is emphasized in Figure 7.1, where we plot the number of correct matches in the random-cut test against the size of the fragment as a percentage of the original file. We can clearly see ssdeep requires fragments to be 60% of the original file or larger to get reliable matches. Sdhash holds for a larger range, but begins dropping once the fragments are 10% of the original file or less. LZJD performs well across all sizes, still obtaining the majority of matches even at 1% size. The end-cut version of the fragmentation tests are similar, and can be reviewed in the Appendix.

We claim that this robustness is the more important property. The fragment results support the conclusion that LZJD is more robust in its ability to match small fragments to their source files compared to ssdeep and sdhash. In all cases, LZJD is either tied with or better than sdhash at this task. Even down to 1% fragment sizes, LZJD is able to match 99% of fragments to their source file. In comparison, sdhash is only able to match just under 60% of fragments.

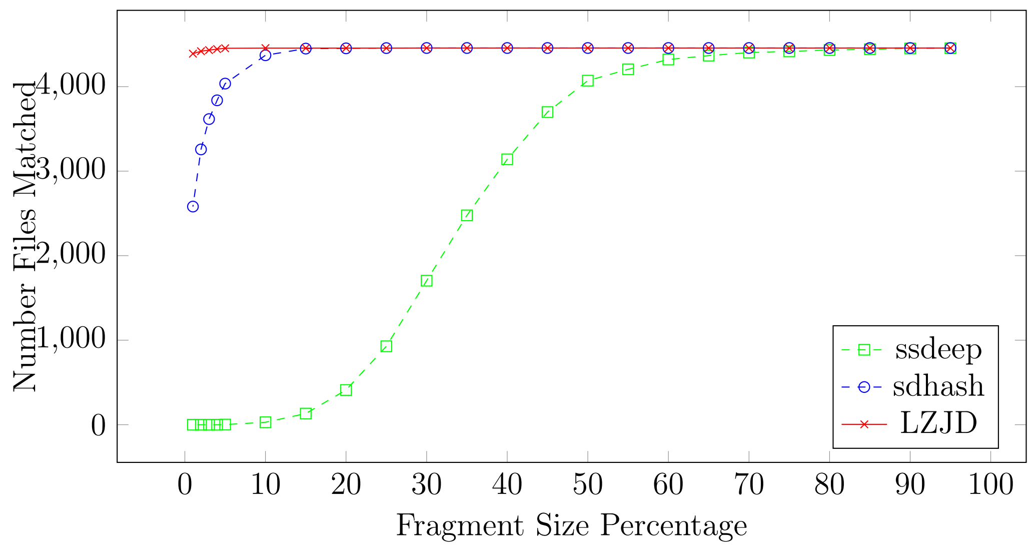


Figure 7.1: Fragment detection random-cut results, x-axis shows the fragment size as a precentage of the original file, and y-axis shows the number of files correctly matched.

7.3.2.4 Alignment Test

One area of weakness for many similarity hash functions is padding inserted at the beginning of a file. Ssdeep in particular is weak in this scenario [168]. The alignment test in FRASH is designed for this scenario, and inserts random bytes into the beginning of a file, and then attempts to match it back against the full database. An analysis of the LZSet algorithm used by LZJD may also lead one to assume that LZJD is susceptible to this same problem. Because the LZSet is built incrementally, strings seen earlier can impact the LZSet, changing what is captured in the later sections of the byte string. The results of this section will show that while this could be a problem for LZJD in the limit, the performance on the FRASH tests indicate that its matching ability is not hampered by this scenario.

The FRASH tests for matching in-spite of excess padding is run in two modes:

one where a fixed number of bytes are added to the file, and the other where a fixed percentage of the original file size is added to the front. The results for the latter scenario are presented in Table 7.13. We present only the percentage results as they are the most aggressive and challenging version of the test.

	Added (%)	10	50	100	300	500
ssdeep	Matches (%)	99.6	92.8	72.4	3.03	0
	Avg. Score	91.1	71.7	60.1	35.3	—
sdhash	Matches (%)	**100**	**100**	**100**	**100**	**100**
	Avg. Score	67.6	69.2	68.6	68.8	68.2
LZJD	Matches (%)	**100**	**100**	**100**	**100**	**100**
	Avg. Score	40.9	22.1	14.8	6.79	4.53

Table 7.13: Alignment test result. Column shows the size of the added bytes, as a percentage of original file size. Rows show percent of correctly matched files and average score for correctly matched files.

As expected, we can see that ssdeep is significantly impacted by the front-padding of the binary, and can only match 3% of files when 300% of the file size is padded to the front. Both sdhash and LZJD are able to match 100% of files in the tested range. We also see that the scores for both are negatively impacted by the addition of the bytes to the front of the file. For sdhash, the scores are in the high 60s instead of the normal 80s-90s that it is able to achieve in the other benchmarks. Because there is no particular interpretation that applies to the sdhash score, we cannot offer any analysis as to cause or reason.

For LZJD, we would expect a score in the range of $1/(1 + x/100)$, where x is the percent of the file size added as padding. In each case, the LZJD score is one third to one half of this expected value. This can be explained by the Lempel-Ziv encoding scheme, which creates a maximal number of entries in the set when presented with high entropy (i.e., random looking) data. Because the x% of bytes added by FRASH are random, this will create disproportionately more entries in the LZ set, and thus become a larger portion of the hash digest. The effect is that there will be considerably more than x% new hashes added to the set, with the amount more being dependent upon the normal entropy of the file under consideration. Because these entries in the hash are from random sub-strings, they are unlikely to appear in another file, and so they are not matched and the score is reduced.

7.4 Discussion

At this point we have performed extensive testing of LZJD compared to ssdeep and sdhash. It is faster to hash, faster at hash comparisons, produces more compact hashes, and provides higher matching accuracy for smaller files, compared to these previous tools. Only ssdeep is faster at hashing and has smaller digests, but its matching ability is not sufficient for the multitude of file types in the t5 corpus. This coalesces to a strong argument for the use of LZJD as an alternative to ssdeep and sdhash for digital forensic applications. The faster comparison time and accuracy combined will allow LZJD to be used in real deployments with databases larger than what either ssdeep or sdhash can handle, while stemming a natural increase in false positives due to the use of larger datasets. This runtime advantage is critical for tool adoption, as practitioners would be unlikely to make use of a tool that did not produce timely results.

7.4.1 LZJD use Compared to Ssdeep and Sdhash

The most significant difference between LZJD and prior similarity digests is the nature of the score value produced. LZJD, like ssdeep and sdhash, will need a "significance" threshold to be determined which may change for different file types and scenarios. The difference comes in the nature of the score's value itself. For ssdeep and sdhash, the exact score x has no particular meaning, and instead a single meaning is often prescribed to only a few ranges of values. For example, the $[21-100]$, $[11-20]$, and $[1-10]$ recommended for sdhash divide up the entire positive range of values into classifications of "Strong", "Marginal", and "Weak" correlation [146]. This can be uninformative when multiple files produce high scores — an issue that occurred in our malware experiments in subsection 7.2.1.

For LZJD, we can interpret the score as a rough measure of byte similarity, or more precisely, as an approximate lower bound on a normalized edit-distance between the files. Not only does this give us an interpretation of the score returned by LZJD, but we can use it to infer what a reasonable threshold might be for many file types and scenarios. This may require more thought on the practitioner's part, that is to make an estimate of what the expected overlap between files might be, or what the maximum score one might expect. Though this requires more mental effort, it is not a requirement — and users could choose to empirically determine their desired scores just as they have done with ssdeep and sdhash. We believe that this interpretation though will ultimately aid its use not just by giving it meaning, but avoiding failure-cases that can occur without such a background (as exemplified by sdhash's failure in subsection 7.2.1).

To give concrete examples of what we mean, consider that file types such as PDFs and EXEs have some amount of boiler-plate mandated by the file format's specification, or may simply be common to most files of that type. In this scenario, one would expect LZJD to produce a minimal score dependent on how much of the boilerplate or common content is shared across files. If the practitioner knows what this level of boilerplate is, they can use that as a minimum-threshold for potential matches.

As another example, consider the results of the fragment tests in subsubsection 7.3.2.3. If an analyst were to use LZJD in this fragment scenario, where it is known that we have a α byte long file fragment that we want to compare against a known (larger) file of length β, it may then be reasonable to use an adjusted scoring of $\text{sim}(\alpha, \beta) \cdot \beta / \alpha$ to adjust for the fact that our expected similarity should not generally exceed the ratio of the differences in file length (i.e., α/β). This requires thought on the analyst's part to realize that smaller scores should be expected, but accurate matching is still possible — and thus might want to alter their interpretation of the score's significance.

7.4.2 LZJD Score Interpretation

As we have discussed throughout this work, LZJD's score is more interpretable than the ones returned by ssdeep and sdhash. We noted in subsection 7.1.1 that LZJD's score can be loosely interpreted as the percentage of shared byte contents between files, and empirically tends to act as a lower bound on a normalized edit distance (as specified in equation (7.1)). The extensive experiments provided by FRASH in section 7.3 support this conclusion. To condense these results, we plot in Figure 7.2 the relationship between LZJD's score and actual percentage of shared bytes. This percentage can be determined for all of the FRASH tests, though would not be known *a priori* in practice. These results show LZJD almost uniformly under-estimating the percentage of bytes altered. The only exception being the five points from the Single Common Block tests, three of which are from the most-extreme terminating state. This overall result leads us to recommend treating LZJD's score as a *lower bound* on the percentage of bytes altered. That is to say, if LZJD returns a score of 23, then it is relatively safe to assume that the two files share at least 23% of their byte contents with each other. While this is not a guarantee, it is empirically supported by a considerable majority of test cases (84 out of 89 data points) and we believe will be useful to the practitioner.

The immediate question would be why does LZJD tend to produce a lower bound estimate? The LZSet method that produces the initial set of sub-strings is sensitive to single byte alterations. Because the set is constructed in a sequential manner, once one byte is altered, it has the potential to propagate forward and alter the rest of the set. This byte sensitivity is what causes LZJD to act as a lower bound, and is the reason why it is often difficult for LZJD to obtain match scores above 50%. Despite this weakness LZJD operates effectively, and the few cases where LZJD over-estimated the percentage of bytes changed are cases where LZJD successfully matched 100% of the altered files to their original sources.

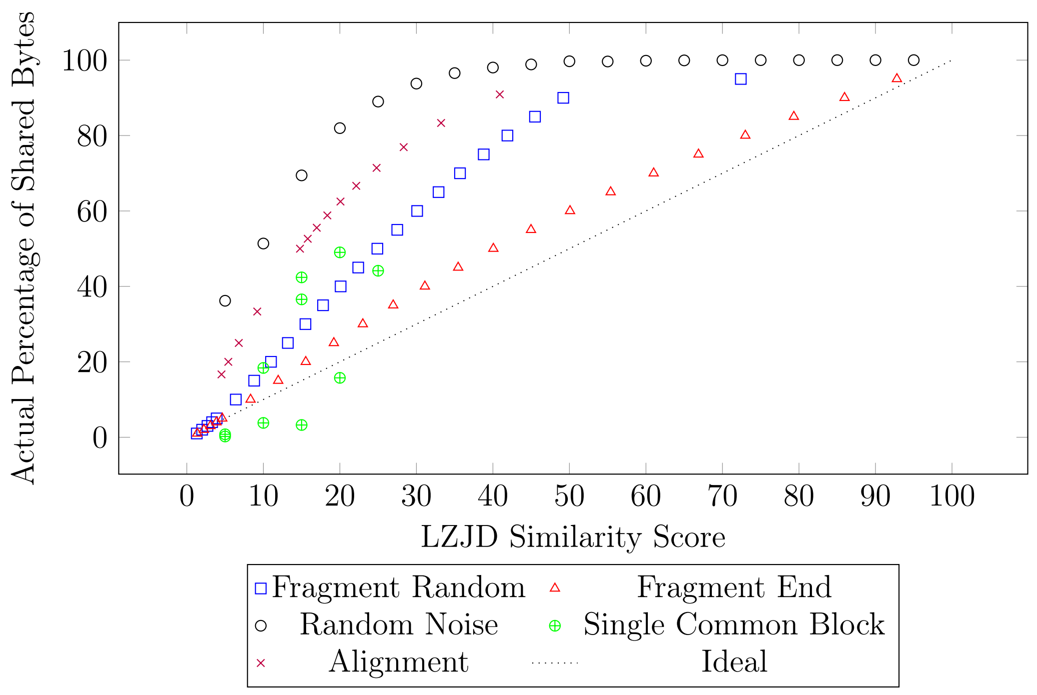


Figure 7.2: Comparing the LZJD similarity score (x-axis) with the actual percentage of altered or added bytes (y-axis) for all tests run by the FRASH suite. The ideal 1-to-1 correspondence (7.1) is shown as a dashed line. Values above this line indicate LZJD under-estimating the change in bytes.

This is also connected with the effect of random byte sequences on LZJD's similarity score, as random bytes will cause the same impact on the LZSet. The impact of random bytes is tested by the SCB, Alignment, and Random Noise tests in FRASH. All three of theses tests make use of random bytes to create the test case files. These tests show that LZJD can perform well even if random bytes are present, but does tend to impact the similarity score LZJD returns. The Malware tests in subsection 7.2.1 also test a higher average entropy file than the t5 corpus, which has become the standard benchmark corpus for similarity digests. The exact entropy statistics are shown in Table 7.14. The performance of LZJD in accurately matching nearest neighbors when the average and median file entropy is as high as 7.96 shows that this weakness does not nullify LZJD's matching ability.

	Kaggle		Android		
Entropy	Bytes	ASM	APK	TAR	t5
Average	6.73	4.48	7.96	6.68	5.88
Median	6.83	4.51	7.96	6.77	5.32
Min	1.64	3.83	4.10	2.61	0.21
Max	7.85	5.35	8.00	8.00	8.00

Table 7.14: Statistics on file entropy broken down by each corpus used in this work.

Since current methods used for digest similarity do not return interpretable similarity scores, there are no current use-cases to compare LZJD against. As analysts begin to use LZJD, we believe the interpretability will become useful to practitioners. Investigating the reality of these hypotheses is beyond the scope of this work. In particular, this new ability may have an impact on:

1. New user training. Being able to explain the results that a method produces is a natural way to help new users learn and understand their tools, and the LZJD algorithm itself can be specified with only a few lines of code. This may aid in helping in enabling tool adoption to a wider breadth of professionals and skill sets.

2. Evidence and testimony. In legal proceedings there is often a need to present evidence to support a case, either in court or in the pursuit of an arrest warrant, for example. That LZJD can be described in a less technical manner as a "conservative estimate of shared content" between two files could be useful in this regard, and is empirically supported. The exact interpretation as the intersection of compression dictionaries is available as well for more technical needs.

3. Machine Learning applications. While ssdeep and sdhash have been used with other machine learning methods before, they both lack the nice metric space and kernel properties of the Jaccard distance that LZJD inherits. We suspect LZJD will thus find wider use with machine learning methods, and its interpretable descriptions will aid in being able to explain and interpret larger models built using LZJD as a component.

To our knowledge, there has yet to be any discussion on what the ideal scoring approach would be for a similarity digest. Our results open an opportunity to discuss such potential design choices. In particular, should scores indicate a level of similarity (resemblance), or a level of commonality (containment)? By this we mean, should scores be interpretable as a measure of how much content of two files are *shared* in aggregate (as LZJD currently does)? Or should scores reflect that two byte strings share some commonality, such as being from the same file, or how much one file could be *subsumed* by another (as sdhash does)? For LZJD, we have already given one instance in which its design could be modified to reflect a preference for commonality when searching for the source of a file fragment (see the discussion near the end of subsection 7.4.1). There may also be other goals toward which one could design a similarity digest, but leave further discussion of this question for future work.

7.4.2.1 On Resemblance and Containment

We take a moment to further discuss the resemblance vs containment question with respect to the results we saw with LZJD. As mentioned in subsection 7.1.1, LZJD measures the resemblance between LZ sets A and B. That we use the Jaccard similarity, for the purpose of computing resemblance, is what allows us to develop a digest of fixed size. Another potential measure of interest is containment, which can be expressed as (7.2).

$$c(A, B) = \frac{|A \cap B|}{|A|} \tag{7.2}$$

Containment asks how much of set A is contained within set B. Sdhash's variable length digest sizes allow it to answer queries regarding either containment or resemblance fashion [146]. Answering containment queries in an unbiased manner requires such variable-length digests [126].

The FRASH Fragmentation, Alignment, and Single Common Block (SCB) tests (subsubsection 7.3.2.3, subsubsection 7.3.2.4 and subsubsection 7.3.2.1 respectively) are tests of containment. LZJD out-performs both ssdeep and sdhash in the fragmentation tests, especially for extreme cases. LZJD ties with sdhash in obtaining all matches for Alignment, and LZJD has only comparable performance to sdhash in the SCB tests. One may then wonder, if LZJD is answering resemblance, how is it able to do well at these containment tasks, and even outperform approaches that should have an advantage?

We believe the insight into understanding this approach is to recognize that $c(A, B) \geq J(A, B) \geq 0$. That is to say, the resemblance query is necessarily a lower bound on containment. If the correct containment score is zero, the resemblance score must necessarily also be zero. Thus LZJD will never over-estimate the containment case. Obtaining the same matching scores then relies on obtaining the same rank ordering between resemblance and containment. Our results with the FRASH tests would indicate that LZJD does well in this regard, as it achieves matching performance comparable to or better than sdhash in all tests.

7.4.3 Future Work

Another advantage of the LZJD approach, which we have not tested in this work, is further scaling abilities of the digest hash. Because the LZJD hash produces a valid distance metric, it is possible to use metric indexes to prune distance computations from a search[164], [166]. Further speedups can be obtained by performing partial digest comparisons. Because the LZJD hash is obtained by selecting the k smallest hash values, every LZJD digest of length k contains the k' digest $\forall k' < k$. This gives a natural way to balance between speed and accuracy. We leave exploring these options to future work.

File size is also an important consideration in digest construction and application. This has been tested to some degree by the FRASH suite and our Malware classification tests. The fragment tests in section subsubsection 7.3.2.3 are explicitly testing matching performance when file sizes differ by up to two orders of magnitude (the original file compared to a 1% fragment). The Kaggle ASM corpus has an average file size of 13.5 MB, compared to only 425 KB for the t5 corpus normally used. In both of these cases, LZJD outperforms ssdeep and sdhash by wide margins. Further exploring the impact of large file size comparisons (GB vs GB) and disparate size comparisons (GB vs KB) is an important topic. In particular, what files should be used, and what are the real-life scenarios that should be simulated?

7.5 Conclusions

The Lempel-Ziv Jaccard Distance was introduced to address problems in malware classification, but we have shown that it has significant utility as a similarity digest for digital forensic applications. Compared to existing tools, such as sdhash, LZJD offers a non-heuristic score that can be interpreted by the user as the amount of byte similarity between two files. Beyond this property, LZJD is more robust in its ability to match file fragments to their source, even when forced to match a fragment on the order of 1% of the original file's size. We have also shown that LZJD can be made practical from a speed perspective, with digest comparison over 60 times faster than sdhash's, and hashing time 34% faster. This will allow the use of larger search databases than is possible with other tools, while also being more accurate. In the interest of tool adoption, we have released an open-source implementation that mimics sdhash's command line options. This should allow LZJD to be easily integrated with existing work-flows for fast adoption by practitioners.

Chapter 8: Metric Indexes for Incremental Insertion and Querying

As LZJD was used in chapter 7, the importance of performing fast nearest-neighbor queries is tantamount for the similarity digest scenario and can't be circumvented in the same way we did for malware classification. Since LZJD is a valid distance metric, we can avail ourselves to many of the existing metric index structures that accelerate nearest-neighbor queries. Unfortunately, they do not perfectly match some of the use cases in which we would like to use LZJD, namely, the incremental insertion and querying of an index. We develop the first such methods in this chapter that would be usable with LZJD.

8.1 Introduction

Many applications are built on top of distance metrics and nearest neighbor queries, and have achieved better performance through the use of metric indexes. A metric index is a data structure used to answer neighbor queries that accelerates these queries by avoiding unnecessary distance computations. The indexes we will look at in this work require the use of a valid distance metric (i.e., obeys triangle inequality, symmetry, and indiscernibility) and returns exact results. Since LZJD is a valid distance metric that obeys these properties, it is a candidate for such acceleration.

Such indexes can be used to accelerate basic classification and similarity search, as well as many popular clustering algorithms like k-Means [169], [170], density based clustering algorithms like DBSCAN [171], [172], and visualization algorithms like t-SNE [173]–[176]. However, most works assume that the data to be indexed is static, and that there will be no need to update the index over time. Even when algorithms are developed with incremental updates, the evaluation of such methods is not done in such a context. In this chapter we seek to evaluate metric indexes for the case of incremental insertion and querying. Because these methods are not readily available, we modify three existing indexes to support incremental insertion and querying.

Our interest in this area is particularly motivated by an application in malware analysis, where we maintain a database of known malware of interest. Malware may be inserted into the database with information about malware type, method of execution, suspected origin, or suspected author. When an analyst is given new malware to dissect, the process can be made more efficient if a similar malware sample has already been processed, and so we want to efficiently query the database to retrieve potentially related binaries. This triaging task is a common problem in malware analysis, often related to malware family detection [92], [148], [177], [178]. Once done, the analyst may decide the binary should be added to the database. In this situation our index would be built once, and have insertions into the database regularly intermixed with queries. This read/write ratio may depend on workload, but is unfortunately not supported by current index structures that support arbitrary distance metrics. This scenario inspires our work to build and develop such indexes, which we test on a wider array of problems than just malware. We do this in part because the feature representations that are informative for malware analysis may change, along with the distance metrics used, and so a system that works with a wide variety of distance measures is appropriate. Our results are also relevant to the work in chapter 7, where we used LZJD for digital forensic purposes. Prior methods did not maintain the distance metric properties, can so can't be accelerated using the methods we discuss in this chapter.

To emphasize the importance of such malware triage, we note it is critical from a time saving perspective. Such analysis requires extensive expertise, and it may take an expert analyst upward of 10 hours to dissect a single binary [55]. Being able to identify a related binary that has been previously analyzed may yield significant time savings. The scale of this problem is also significant. A recent study of 100 million computers found that 94% of files were unique [179], meaning exact hashing approaches such as MD5 sums will not help, and similarity measures between files are necessary. In terms of incremental addition of files, in 2014 most anti-virus vendors were adding 2 to 3 million new binaries each month[180].

Given our motivation, we will review the related metric index work in section 8.2. We will review and modify three algorithms for incremental insertion and querying in section 8.3, followed by the evaluation details, datasets and distance metrics in section 8.4. Evaluations of our modifications and their impact will be done in section 8.5, followed by an evaluation of the incremental insertion and querying scenario in section 8.6. Finally, we will present our conclusions in section 8.7.

8.2 Related Work

There has been considerable work in general for retrieval methods based on k nearest neighbor queries, and many of the earlier works in this area did support incremental insertion and querying, but did not support arbitrary distance metrics. One of the earliest methods was the Quad-Tree[181], which was limited to two-dimensional data. This was quickly extended with the kd-tree, which also supported insertions, but additionally supported arbitrary dimensions and deletions as well[182]. However, the kd-tree did not support arbitrary metrics, and was limited to the euclidean and similar distances. Similar work was done for the creation of R-trees, which supported the insertion and querying of shapes, and updating the index should an entry's shape change[183]. However improving the query performance of R-trees involved inserting points in a specific order, which requires having the whole dataset available from the onset[184], and still did not support arbitrary metrics.

The popular ball-tree algorithm was one of the first efforts to devise and evaluate multiple construction schemes, some which required all the data to be available at the onset, while others which could be done incrementally as data became available [185]. This is similar to our work in that we devise new incremental insertion strategies for two algorithms, though Omohundro [185] do not evaluate incremental insertions and querying. This ball-tree approach was limited to the euclidean distance primarily from the use of a mean data-point computed at every node. Other early work that used the triangle inequality to avoid distance computations had this same limitation [186].

While almost all of these early works in metric indexes supported incremental insertion, none contain evaluation of the indexes under the assumption of interleaved insertions and queries. These works also do not support arbitrary distance metrics.

The first algorithm for arbitrary metrics was the metric-tree structure [165], [187], which used the distance to a randomly selected point to create a binary tree. This was independently developed, slightly extended, and more throughly evaluated to become the Vantage-Point tree we explore in this work[164]. However, these methods did not support incremental insertion. We will modify and further improve the Vantage-Point tree in section 8.3.

Toward the creation of provable bounds for arbitrary distance metrics, the concept of the expansion constant c was introduced by Karger and Ruhl [188]. The expansion constant is a property of the current dataset under a given metric, and describes a linear relationship between the radius around a point, and the number of points contained within that radius. That is to say, if the radius from any arbitrary point doubles, the number of points contained within that radius should increase by at most a constant factor. Two of the algorithms we look at in this work, as discussed in section 8.3, make use of this property.

The first practical algorithm to make use of the expansion constant was the Cover-tree [166], which showed practical speed-ups across multiple datasets and values of $k \in [1, 10]$. Their results were generally shown under L_p norm distances, but also included an experiment using the string edit distance. Later work then simplified the Cover-tree algorithm and improved performance, demonstrating its benefit on a wider variety of dataset and distance metrics [167]. Of the algorithms for metric indexes, the Cover-tree is the only one we are aware of with an incremental construction approach, and so we consider it one of our metrics of interest in section 8.3. While the Cover-tree construction algorithm is described as an incremental insertion process, the more efficient variant proposed by Izbicki and Shelton [167] includes a bound which requires the whole dataset in advance to calculate bounds, preventing the efficient interleaving of insertions and queries[1].

Another algorithm we consider is the Random Ball Cover (RBC), which was designed for making effective use of GPUs with the euclidean distance [189]. Despite testing on only the euclidean distance, the algorithm and proof does not rely on this assumption – and will work with any arbitrary distance metric. We consider the RBC in this work due to its random construction, which allows us to devise an incremental construction procedure that closely matches the original design and maintains the same performance characteristics. While the Random Ball Cover has inspired a number of GPU based follow ups [190]–[192], we do not assume that a GPU will be used in our work.

Li and Malik [193] develop an indexing scheme that supports incremental updates, but only works for the euclidean distance. They also do not evaluate the performance as insertions and queries are interleaved.

[1]The original Cover-tree did not have this issue, and so would meet our requirements for incremental insertion. We consider the newer variant since it is the most efficient.

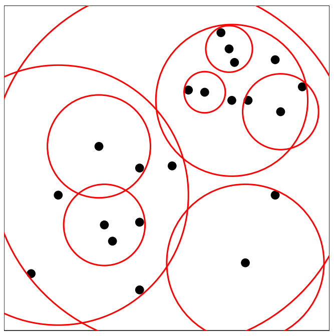
8.3 Metric Indexes Used

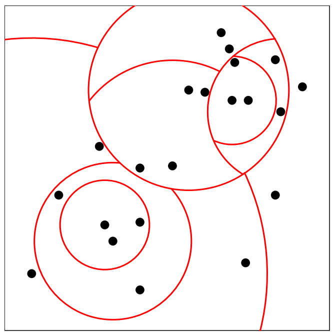
Given the existing literature on metric indexes there appear to be no readily available methods that suit our needs. For this reason we take three algorithms and modify them for incremental index construction and querying. In particular, we adapt the Random Ball Cover, Vantage Point tree, and Cover-tree algorithms for incremental insertion. As classically presented, the first two methods methods are not designed for this use case. While the original cover tree algorithm did support incremental insertions, its improved variants do not. More importantly, as we will show in section 8.5, the Cover-tree has worse than brute-force performance with one of our distance metrics. With our modifications we satisfy three goals that have not yet been achieved in a single data structure:

1. New datapoints can be added to the index at any point
2. We can efficiently query the index after every insertion
3. The index can be efficiently used with any distance metric

While the latter point would seem satisfied by the original Cover-tree algorithm, our results indicate a degenerate case where the Cover-tree performs significantly worse than a brute force search. For this reason we consider it to have not satisfied our goals.

We also contribute improvements to both the Random Ball Cover and Vantage Point Tree structures that further reduce the number of distance computations needed by improving the rate at which points are pruned out. These improvements


(a) Cover-trees produce a hierarchy of circles, but each node may have a variable number of children. Each node has a radius that upper bounds the distance to all of its children, and nodes may partially overlap.


(b) Vantage-Point trees divide the space using a hierarchy of circles. The in/outside of each space acts as a hard boundary when subdividing.

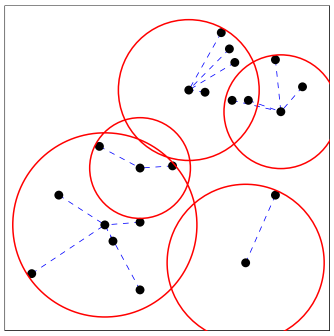

(c) RBC selects a subset of representatives, and each point is assigned to its nearest representative (relationships marked with dashed blue line).

Figure 8.1: Example partitionings for all three algorithms. Red circles indicate the radius from which one node covers out in the space.

can dramatically increase their effective pruning rate, which leads us to alter our conclusions about which method should be used in the general case.

In the below descriptions, we will use S to refer to the set of points currently in the index, and $n = |S|$ as the number of such points. A full review of all details related to the three methods is beyond this scope of this work, but we will provide the details necessary to understand what our contributions are to each approach.

8.3.1 Cover Tree

The Cover-tree [166] is a popular method for accelerating nearest neighbor queries, and one of the first practical metric indexes to have a provable bound using the expansion constant c [188]. The Cover-tree can be constructed in $O(c^6 n \log n)$ time, and answer queries in $O(c^{12} \log n)$ time. Izbicki and Shelton [167] developed the Simplified Cover Tree, which reduces the practical implementation details and increases efficiency in both runtime and avoiding distance computations.[2] To reproduce the Simplified Cover Tree algorithm without any nearest-neighbor errors, we had to make two slight modifications to the algorithm as originally presented. These adjustments are detailed in Appendix section B.1.

The Cover-tree algorithm, as its name suggests, stores the data as a tree structure where each node represents only one data point and may have any number of children nodes[3]. The tree is constructed via incremental insertions, which means we require no modifications to the construction algorithm to support our use case. However, at query time it is necessary for each node p in the tree to compute a *maxdist*, which is the maximum distance from the point represented by node p to any of its descendant nodes. This maxdist value is used at every level of the tree to prune children nodes from the search path. Insertions can cause re-organizations of the tree, resulting in the need to re-compute maxdist bounds. For this reason the Simplified Cover-tree cannot be used to efficiently query the index between consecutive insertions.

Because of the re-balancing and re-organization that occurs during tree construction, it is not trivial to selectively update the maxdist value based on the changes that have occurred. Instead we will use an upper bound on the value of maxdist. Each node in the tree maintains a maximum child radius of the form 2^l, where l is an integer. This also upper bounds the maxdist value of any node by 2^{l+1} [167]. This will allow us to answer queries without having to update maxdist, but

[2]Izbicki and Shelton also introduced a Nearest Ancestor Cover Tree, but we were unable to replicate these results. The reported performance difference between these two variants was not generally large, and so we use only the simplified variant.

[3]The maximum number of children is actually bounded by the expansion constant c.

results in a loosening of the bound. The performance of this upper bounded version of the Cover-tree we will refer to as Cover_{B}, and is more naturally suited to the use case of interleaved insertions and queries.

We note as well that this relaxation on the maxdist based bound represents a compromise between the simplified approach proposed by Izbicki and Shelton and the original formulation by Beygelzimer, Kakade, and Langford. In the later case, the 2^{l+1} bound is used to prune branches, but all branches are traversed simultaneously. In the former, the maxdist bound is used to descend the tree one branch at a time, and the nearest neighbor found so far is used to prune out new branches. By replacing maxdist with 2^{l+1}, we fall somewhere in-between the approaches. Using a looser bound to prune, but still avoiding traversing all branches. In our extensive tests of these algorithms, we discovered two issues with the original specification of the simplified Cover-tree. These are detailed in Appendix section B.1, along with our modifications that restore the Cover-tree's intended behavior.

8.3.2 Vantage Point Tree

The Vantage Point tree [164], [165] (VP-tree) is one of the first data structures proposed for accelerating neighbor searches using an arbitrary distance metric. The construction of the VP-tree results in a binary tree, where each node p represents one point from the dataset, the "vantage point". The vantage point splits its descendant into a low and high range based on their distance from the aforementioned vantage point, with half of the child vectors in each range. For each range, we also have a nearest and farthest value, and an example of how these are used is given in Figure 8.2.

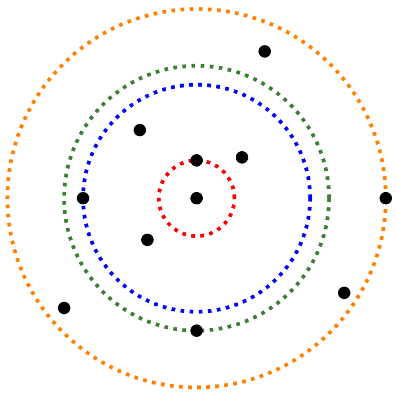


Figure 8.2: Example of a node in a vp-tree, with the vantage point in the center. The low-near bound is in red, the distance to the point closest to the center. The low-far (blue) and high-near (green) braket the boundry of the median. No points can fall between these bounds. The farthest away point provides the high-far bound in orange.

This tree structure is built top-down, and iteratively splits the remaining points into two groups at each node in the tree. Rather than continue splitting until each node has no children, there is instead a minimum split size b. This is because there are likely too few points for which we can obtain good low/high bounds. Instead, once the number of datapoints is $\leq b$, we create a “bucket” leaf node that stores the points together and uses the distance from each point to its parent node to do additional pruning.

At construction time, since each split is done by breaking the tree in half, the maximum depth of the tree is $O(\log n)$ and construction takes $O(n \log n)$ time. Assuming the bounds are successful in pruning most branches, the VP-tree then answers queries in $O(\log n)$ time.

The bucketing behavior can provide practical runtime performance improvements as well. Some of this comes from better caching behavior, as bucket values will be accessed in a sequential pattern, and avoids search branches that can be more difficult to accurately predict for hardware with speculative execution. This can be done for the VP-tree because its structure is static as it is created, where the Cover-tree cannot create bucket nodes due to the re-balancing done during construction.

8.3.2.1 Incremental Construction

While the Cover-tree required minimal changes since its construction is already incremental, we must define a new method to support such a style for the VP-tree. To support incremental insertions into a VP-tree, we must first find a location with which to store the new datapoint x. This can be done quite easily by descending the tree via the low/high bounds stored for each point, and updating the bounds as we make the traversal. One we reach a leaf node, x is simply inserted into the bucket list. However, we do not expand the leaf node when its sizes exceeds b.

Ideally, these bounds will be changed infrequently as we insert new points. Getting a better estimate of the initial bound values should minimize this occurrence. For this reason we expand a bucket b once it reaches a size of b^2. This gives us a larger sample size with which to estimate the four bound values. We use the value b^2 as a simple heuristic that follows our intuition that a larger sample is needed for better estimates, allows us to maintain the fast construction time of the VP algorithm, and results in an easy to implement and replicate procedure.

Algorithm 4 Insert into VP-tree

```
Require: vp-tree root node $p$, and new datapoint $x$ to insert into tree.
 1: while $p$ is not a leaf node do
 2:     $dist \leftarrow d(x, p.vp)$
 3:     if $dist < (p.\text{low}_{\text{far}} + p.\text{high}_{\text{near}})/2$ then
 4:         $p.\text{low}_{\text{far}} \leftarrow \max(dist, p.\text{low}_{\text{far}})$
 5:         $p.\text{low}_{\text{near}} \leftarrow \min(dist, p.\text{low}_{\text{near}})$
 6:         $p \leftarrow p.\text{lowChild}$
 7:     else
 8:         $p.\text{high}_{\text{far}} \leftarrow \max(dist, p.\text{high}_{\text{far}})$
 9:         $p.\text{high}_{\text{near}} \leftarrow \min(dist, p.\text{high}_{\text{near}})$
10:         $p \leftarrow p.\text{highChild}$
11:     end if
12: end while
13: Add $x$ to bucket leaf node $p$
14: if $|p.\text{bucket}| > b^2$ then
15:     Select vantage point from $p$.bucket and create a new split, adding two children
    nodes to $p$.
16: end if
17: return
```

Thus our insertion procedure is given in Algorithm 4, and is relatively simple. Assuming the tree remains relatively balanced, we will have an insertion time of $O(\log n)$. This will also maintain the query time of $O(\log n)$.

8.3.2.2 Faster Search

We also introduce a new modification to the VP-tree construction procedure that reduces search time by enhancing the ability of the standard VP-tree search procedure to prune out branches of the tree. This is done by using an extension of the insight from subsubsection 8.3.2.1, that we want to make our splits only when we have enough information to do so. That is, once we have enough data to make a split, choosing the median distance from the vantage point may not be the smartest split.

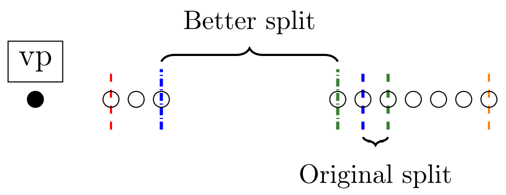


Figure 8.3: Example on how the split can be improved, with vantage point in black and other points sorted by distance to it. Colors correspond to Figure 8.2.

Instead, we can use the distribution of points from the vantage point to choose a split that better bifurcates the data based on the distribution. An example of this is given in Figure 8.3, where the data may naturally form a binary split. This increases the gap between the low_{far} and $\text{high}_{\text{near}}$ bounds, which then allows the search procedure to more easily prune one of the branches.

To do this quickly, so to minimize any increase in construction time, we borrow from the CART algorithm used to construct a regression tree[194]. Given a set of n distances to the vantage-point, we find the split that minimizes the weighted variance of each split

$$\arg\min_s s \cdot \sigma^2_{1:s} + (n - s) \cdot \sigma^2_{s:n} \tag{8.1}$$

Where $\sigma^2_{s:n}$ indicates the variance of the points in the range of $[s, n)$ when sorted by distance to the vantage point. Because (8.1) can be solved with just two passes over the n points [195], [196], we can solve this quickly with only an incremental increase in runtime.

8.3.3 Random Ball Cover

The Random Ball Cover [189] (RBC) algorithm was originally proposed as an accelerating index that would make efficient use of many-core systems, such as GPUs. This was motivated by the euclidean distance metric, which can be computed with high efficiency when computing multiple distances simultaneously. This can be done by exploiting a decomposition of the euclidean distance into matrix operations, for which optimized BLAS routines are readily available. To exploit batch processing while also pruning distances, the RBC approach organizes data into large groups and uses the triangle inequality sparingly to prune out whole groups at a time. Compared to the VP and Cover Tree, the RBC algorithm is unique in that it aims to answer queries in $O(\sqrt{n})$ time and perform construction in $O(n\sqrt{n})$ time.

The training procedure of the RBC algorithm is to randomly select $O(\sqrt{n})$ centers from the dataset, and denote that set of points as R. These are the R *random balls* of the algorithm. Each representative $r_i \in R$ will own, or *cover*, all the datapoints for which it is the nearest neighbor, $\arg\min_x d(x, r_i) \forall x \in S \setminus R$, which is denoted as L_{r_i}. It is expected that each r_i will then own $O(\sqrt{n})$ datapoints. Querying is done first against the subset of points R, from which many of the representatives are pruned. Then a second query is done against the points owned by the non-pruned representatives. To do this pruning, we need the representatives to be sorted by their distance to the query point q. We will denote this as $r_i^{(q)}$, which would be the i'th nearest representative to q. Pruning for k nearest neighbor queries is then done using two bounds,

$$d(q, r_i) < d(q, r_k^{(q)}) + \psi_{r_i} \tag{8.2}$$

$$d(q, r_i) < 3 \cdot d(q, r_k^{(q)}) \tag{8.3}$$

Where $\psi_{r_i} = \max_{x \in L_{r_i}} d(r_i, x)$ is the radius of each representative, such that all datapoints fall within that radius. Each bound must be true for any r_i to have the k'th nearest neighbor to query q, and the overall procedure is given in Algorithm 5. Theoretically the RBC bounds are interesting in that they provide a small dependency on the expansion constant c of the data, where queries can be answered in $O(c^{3/2}\sqrt{n})$ time. This is considerably smaller than the c^{12} term in cover trees, but has the larger $\sqrt{n}$ dependence on n instead of logarithmic. However, the RBC proof depends on setting the number of representatives $|R| = O(c^{3/2}\sqrt{n})$ as well, which we would not know in advance in practice. Instead we will use $|R| = \sqrt{n}$ in all experiments.

Algorithm 5 Original RBC Search Procedure

```
Require: Query $q$, desired number of neighbors $k$
1: Compute sorted order $r_i^{(q)}$ $\forall r \in R$ by $d(r, q)$
2: FinalList $\leftarrow \emptyset$
3: for all $r_i \in R$ do
4:     if Bounds (8.2) and (8.3) are True then
5:         FinalList $\leftarrow$ FinalList $\cup L_{r_i}$
6:     end if
7: end for
8: $k$-NN $\leftarrow$ BruteForceSearch($q$, $R \cup$ FinalList) ▷distances for $R$ do not need to be
   re-computed
9: return $k$-NN
```

8.3.3.1 Incremental Construction

If our goal was to build a static index, the random selection of R may lead to a sub-optimal selection. It is possible that different representatives will have widely varying numbers of members. For our goal of incrementally adding to an index, this stochastic construction becomes a benefit. Because the representatives are selected randomly without replacement, it is possible to incrementally add to the RBC index while maintaining the same quality of results.

Algorithm 6 Insert into RBC Index

```
Require: RBC representatives $R$, associated lists $L_r, \forall r \in R$, and new datapoint $x$ to add to RBC.
 1: Compute sorted order $r_i^{(x)}$ $\forall r \in R$ by $d(r,x)$
 2: $L_{r_1^{(x)}} \leftarrow L_{r_1^{(x)}} \cup x$
 3: $\psi_{r_1^{(x)}} \leftarrow \max\left(d(r_1^{(x)},x), \psi_{r_1^{(x)}}\right)$          ▷keep radius information correct
 4: if $\text{ceil}\left(\sqrt{n}\right)^2 \neq n$ then
 5:     return                                                                  ▷else, expand R set
 6: end if
 7: select randomly a datapoint $l_{\text{new}}$ from $\bigcup_{\forall r \in R} L_r$
 8: let $r_{old}$ be the representative that owns $l_{\text{new}}$, i.e., $l_{\text{new}} \in L_{r_{old}}$
 9: $L_{r_{old}} \leftarrow L_{r_{old}} \setminus l_{\text{new}}$
10: $r_{\text{new}} \leftarrow l_{\text{new}}$
11: potentialChildren $\leftarrow$ RadiusSearchRBC($r_{\text{new}}$, $\arg\max_{r,\forall r \in R} \psi_r$)
12: $L_{r_{\text{new}}} \leftarrow \emptyset$
13: $R \leftarrow R \cup r_{\text{new}}$
14: $\psi_{r_{\text{new}}} \leftarrow 0$
15: for all $y \in$ potentialChildren do
16:     Let $r_y$ be the representative that owns $y$
17:     if $d(y, r_y) > d(y, r_{\text{new}})$ then                              ▷change ownership
18:         $L_{r_y} \leftarrow L_{r_y} \setminus y$
19:         $L_{r_{new}} \leftarrow L_{r_{new}} \cup y$
20:         $\psi_{r_y} \leftarrow \arg\max_{\forall z \in L_{r_y}} d(r_y, z)$               ▷update radius info
21:         $\psi_{r_{\text{new}}} \leftarrow \max\left(\psi_{r_{\text{new}}}, d(y, r_{\text{new}})\right)$
22:     end if
23: end for
```

The details of our approach are given in Algorithm 6. Whenever we add a new datapoint to the index, we find its representative and add it to the appropriate list L. This can be done in $O(\sqrt{n})$ time, consistent with the query time of RBC. Once the closest representative is found, the radius to the farthest point may need to be updated, which is trivial. For the majority $(n - \sqrt{n})$ of insertions, this is all the work that needs to be done.

For the remaining $\sqrt{n}$ insertions, the total number of datapoints will reach a size such that we should have a new representative. The new representative will be selected randomly from all the points in $S \setminus R$. We can find all the datapoints that may belong to this new representative using a "range" or "radius" search. A radius search is given a query and radius, and returns all datapoints within the specified radius of the query. In this case we give the new representative as the query and specify the range as the maximum ψ_r in the RBC so far. This is by definition the maximum distance of any point to its representative, so any point that will be owned by the new representative must have a smaller distance. In the worst case scenario, we cannot prune any points using a radius search. This means at most n other points must be considered. But since this scenario can only occur $\sqrt{n}$ times, we maintain the same construction time complexity of $O(n\sqrt{n})$ in all cases. We can also state that this approach yields an amortized $O^*(\sqrt{n})$ insertion time.

8.3.3.2 Faster Search

While the original RBC search is fast and efficient on GPUs and similar many-core machines, it is not as efficient for our use case. Our scenario of interleaved insertions and queries means we will be querying over only a few datapoints at a time. This means we will not obtain a large enough group of queries points to obtain the batch and SIMD efficiencies that were the original goal of Cayton [189]. Further, when we consider arbitrary distance metrics, we can not expect the same efficient method of grouping calculations as can be done with the euclidean distance. Thus we have developed an improved querying method for the RBC search to make it more efficient in our incremental insertion and querying scenario. Our improvements to the RBC search procedure can be broken down into three steps.

First, we modify the search to create the k-NN list incrementally as we visit each representative $r \in R$. In particular we can improve the application of bound (8.2) by doing this. First, we note that in (8.2), the $d(q, r_k^{(q)})$ term serves as an upper bound on the distance to the k'th nearest neighbor. By building the k-NN list incrementally, we can instead use the current best candidate for k'th nearest neighbor as a bound on the distance to the k'th nearest neighbor. This works intuitively, as the true k'th neighbor, if not yet found, must by definition have a smaller distance than our current candidate.

Second, when visiting the points owned by each representative, $l \in L_r$, we can apply this bound again and tighten the bound further. This is done by replacing the ψ_{r_i} term of (8.2) by the distance of l to its representative r. Since this distance $d(l, r)$ had to be computed when building the RBC in the first place, these distances can simply be cached at construction — avoiding any additional overhead.

Third, to increase the likelihood of finding the k'th neighbor earlier in the process, we visit the representatives in sorted order by their distance to the query.

Because our first modification tightens the bound as we find better k'th candidates, this will accelerate the rate at which we tighten the bound.

The complete updated procedure is given in Algorithm 7. A similar treatment can improve the RBC search procedure for range queries. We note that on lines 2 through 4, we add all the children points of the closest representative $L_{r_1^{(q)}}$ unconditionally. This satisfies requirements of the RBC search algorithm's correctness in the k nearest neighbor case, rather than just one nearest neighbor. We refer the reader to Cayton [189] for details. This step's purpose is to pre-populate the k-NN list with values for the bounds checks done in lines 8 and 10.

Algorithm 7 New RBC Search Procedure

```
Require: Query $q$, desired number of neighbors $k$
 1: Compute sorted order $r_i^{(q)}$ $\forall r \in R$ by $d(r, q)$
 2: $k$-NN ← $\{r_1^{(q)}\}$                ▷sorted list implicitly maintains max size of k
 3: for all $l \in L_{r_1^{(q)}}$ do        ▷Add the children of the nearest representative
 4:     $k$-NN ← $k$-NN $\cup\, l$
 5: end for
 6: for $i \in 2 \ldots |R|$ do              ▷visit representatives in sorted order
 7:     $qr \leftarrow d(q, r_i^{(q)})$
 8:     Add tuple $r_i^{(q)}$, $d(r_i^{(q)}, q)$ to $k$-NN
 9:     if $qr <$ $k$-NN[k].dist $+ \psi_{r_i}$ and (8.3) are True then
10:         for all $l \in L_{r_i^{(q)}}$ do
11:             if $qr <$ $k$-NN[k].dist $+ d(l, r_i^{(q)})$ then      ▷$d(l, r_i^{(q)})$ is pre-computed
12:                 Add tuple $l$, $d(l, q)$ to $k$-NN
13:             end if
14:         end for
15:     end if
16: end for
17: return $k$-NN
```

The first step of our new algorithm must still compute the distances for each r_i, and $|R| = \sqrt{n}$. In addition, we add all the children of the closest represent $r_1^{(q)}$, which is expected to own $O(\sqrt{n})$ points. Thus this modified RBC search is still an

$O(\sqrt{n})$ search algorithm. Our work does not improve the algorithmic complexity but does improve its effectiveness at pruning.

8.4 Datasets and Methodology

We use a number of datasets and distance metrics to evaluate our changes and the efficiency of our incremental addition strategies. For all methods we have confirmed that the correct nearest neighbors are returned compared to a naive brute-force search. Our evaluation will cover multiple aspects of performance, such as construction time, query time, and the impact of incremental insertions of index efficiency. We will use multiple values of k in the nearest neighbor search so that our results are relevant to multiple use-cases. Toward this end we will also use multiple datasets and distance metrics to further validate our findings.

8.4.1 Evaluation Procedure

The approach used in most prior works to evaluate metric indexes is to create the index from all of the data, and then query each datapoint in the index search for the single nearest neighbor [167]. For consistency we replicate this experiment style, but do not use every datapoint as a query point. This results in worst case $O(n^2)$ runtime for some of our tests, preventing us from comparing on our larger datasets. Since our interest is in whether the index allows for faster queries, we can instead determine the average pruning efficiency with extreme accuracy by using only a small sample of query points. In tests using a sample of 1000 points for testing, versus using all data points, we found no difference in conclusions or results[4]. Thus we will use 1000 test points in all experiments. This will allow us to run any individual test in under a week, and evaluate the insertion-query performance in a more timely manner.

When using various datasets, if the dataset has a standard validation set, it will not be used. Instead points from the training set will be used for querying. This is done for consistency since not every dataset has a standard validation or testing set. Our experiments will be performed searching for the k nearest neighbors with $k \in \{1, 5, 25, 100\}$. Evaluating for multiple values of k is often ignored in most works, which focus on the $k = 1$ case in their experiments [e.g. 164], [167], [189], or will test on only a few small value of $k \leq 10$ [166]. This is despite many applications, such as embeddings for visualization [173]–[175], [197], using values of k as large as 100. By testing a range of values for k we can determine if one algorithm is uniformly better for all values of k, or if different algorithms have an advantage in one regime over the others.

To evaluate the impact of incremental index construction on the quality of the final index, each index will be constructed in three different ways. Differences in performance between these three versions of the index will indicate the relative impact that incremental insertions have.

1. Using the whole dataset and performing the classic batch construction method, by which we mean the original index construction process for each algorithm (referred to as batch construction)

[4] The largest observed discrepancy was of 0.3 percentage points

2. Using half the dataset to construct an initial index using the classic batch method, and incrementally inserting the second half of the data (referred to as half-batch)

3. Constructing the entire dataset incrementally (referred to as incremental).

For these experiments, the Cover-tree is excluded — as its original batch construction is already incremental (though does not support efficient queries between insertions). In our results we will expect the RBC algorithm to have minimal change in performance, due to the stochastic nature of representative selection. The expected performance impact of the VP-tree was unknown, though we would expect the tree to perform best in batch construction, second best when using half-batch construction, and worst when fully incremental. Results will consider both the number of distance computations when including and excluding distanced performed during index construction. We note that runtime of all methods and tests correlates directly with number of distance computations done for our code. Comparing distance computations is preferred so that we observe the true impact of pruning, rather than efficiency of micro optimizations, and is thus comparable to implementations written in other languages.

We will also test the effectiveness of each method when interleaving queries and insertions. This will be evaluated in a manner analogous to common data structures, where we have different number of possible read (query) and write (insert) ratios.

8.4.2 Data and Distances Used

Now that we have reviewed how we will evaluate our methods, we will list the datasets and distance metrics used in such evaluations, a summary of which is presented in Table 8.1. Datasets and distance metrics were selected to cover a wide range of data and metric types, include common baselines, and so that experiments would finish within a one-week execution window.

Dataset	Samples	Distance Metric
MNIST	60,000	Euclidean
MNIST8m	8,000,000	Euclidean
Covtype	581,012	Euclidean
VxHeaven	271,095	LZJD
VirusShare5m	5,000,000	LZJD
ILSVRC 2012 Validation	50,000	EMD
IMDB Movie Titles	143,337	Levenshtein

Table 8.1: Datasets used in experiments, including the number of points in each dataset and the distance metric used.

Our first three datasets will all use the familiar euclidean distance ($d(x, y) = \|x - y\|$). The first of which is the well known MNIST dataset [198], which is a commonly used benchmark for machine learning in general. Due to its small size we also include a larger version of the dataset, MNIST8m, which contains 8 million points produced by random transformations to the original dataset [199]. We also evaluate the Forest Cover Type (Covtype) datasets [200], which has historically been used for metric indexes.

Finding nearest neighbors and similar examples is important for malware analysis [148], [201]. The VxHeaven corpus has been widely used for research in malware analysis [202], and so we use it in our work for measuring the similarity of binaries. VxHeaven contains 271k binaries, but malware datasets are routinely reaching the hundreds of millions to billions of samples. For this reason we also select a random 5 million element set from the VirusShare corpus [49], which shares real malware with interested researchers. As the distance metric for these datasets, we will use the Lempel-Ziv Jaccard Distance (LZJD) [43].

One of the metrics measured in the original Cover-tree paper was the a string edit distance [166]. They compared to the dataset and methods used in Clarkson [203], however the available data contains only 200 test strings. Instead we use the Levenshtein edit distance on IMDB movie titles [204], which contains both longer strings and is three orders of magnitude larger.

The simplified Cover-tree paper evaluated a larger range of distance metrics [167], including the Earth Mover's Distance (EMD) [205]. The EMD provides a distance measure between histograms, and was originally proposed for measuring the similarity of images. We follow the same procedure as Izbicki and Shelton [167] for using the "thresholded" EMD [206], except we use the RGB color space[5]. We use the 2012 validation set of the ImageNet challenge [207] for this distance metric, as it is the most computationally demanding metric of the ones we evaluate in this work.

[5]Our software did not support the LabCIE color space previously used, and we did not notice any significant difference in results for other color spaces.

8.5 Evaluation of Construction Time and Pruning Improvements

We first evaluate the impact of our changes to each of the three algorithms. For RBC and VP-trees, we have made alterations that aim to improve the ability of these algorithms to avoid unnecessary distance computations at query time. For the Cover-tree, we have made a modification that will negatively impact its ability to perform pruning, but will make it viable for interleaved insertions and queries. We will evaluate the impact of our changes on construction time, query efficiency under normal construction, and the impact incremental construction has on the efficiency of the complete index.

8.5.1 Impact on Construction Time

To determine the impact of the incremental construction and our modifications, we will compare each algorithm in terms of the number of distance computations needed to construct the index. We will do this for all three construction options, batch, half-batch, and incremental, as discussed in section 8.4. The time for only constructing the indices in these three ways are shown in Figure 8.4. We note that there is no distinction between the Cover and Cover$_\mathrm{B}$ construction times, and that the cover-tree is always incremental in construction. For this reason we only show one bar to represent Cover and Cover$_\mathrm{B}$ across all three construction scenarios to avoid graph clutter.

Here we see the two performance characteristics observed. On datasets like MNIST, where we use the euclidean distance, RBC is the slowest to construct.

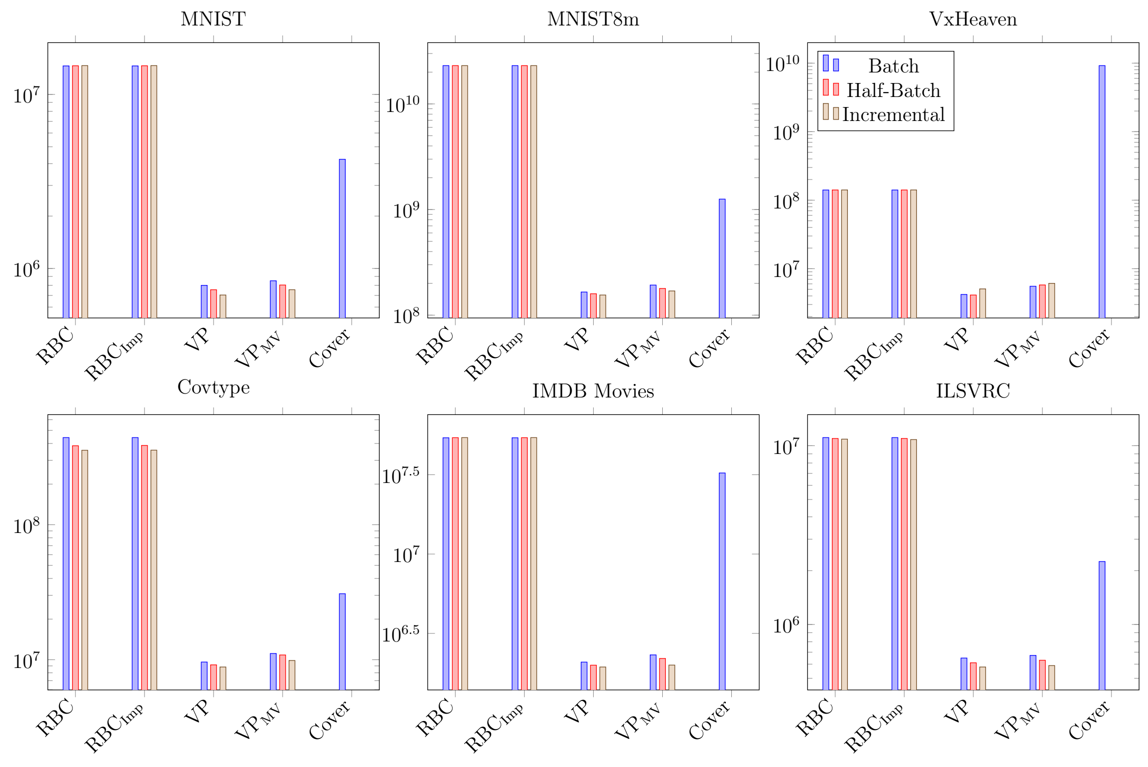


Figure 8.4: Construction performance for each algorithm on each dataset. The y-axis represents the number of distance computations performed to build each index. Each algorithm is plotted three times, once using classic batch construction, half-batch, and incremental. The Cover-tree's construction algorithm is equivalent in all scenarios, so only one bar is shown.

This is expected, as it also has the highest complexity at $O(n\sqrt{n})$ time. We also note that the RBC radius search is not as efficient at pruning, and fails to do so on most datasets. Only on datasets that are most accelerated, such as the Covtype dataset, does the RBC incremental construction avoid distance computations during construction. This empirically supports the theoretical justification that we maintain the same construction time for the RBC algorithm, as discussed in subsubsection 8.3.3.1.

The second slowest to construct is the Cover-tree, followed by the VP-trees which is fastest. On the VxHeaven dataset, with the LZJD metric, the construction time performance of the Cover-tree degrades dramatically, using two orders of magnitude more distance computations than the RBC. We believe this performance degradation is an artifact of the expansion constant c that occurs when using the LZJD metric. The VP tree has no construction time impact with c, and the RBC algorithm has a small $O(c^{3/2})$ dependency compared to the Cover-tree's $O(c^6)$ dependence. On the VirusShare5m dataset, the Cover-tree couldn't be constructed given over a month of compute time. We also note that the Cover-tree had degraded construction performance on the IMDB Movies dataset using the Levenshtein distance. These results present a potential weakness in the Cover-tree algorithm.

Barring the performance behavior of the Cover-tree, both the RBC and VP-tree have more consistent performance on various datasets. We note of particular interest that the incremental construction procedure for the RBC results in almost no change in the number of distance computations needed to build the index[6]. The radius search is rarely able to do any pruning for the RBC algorithm, and so the brute force degrades to the same number of distance computations as the batch insertion. The Covtype dataset is the one for which each algorithm was able to do the most pruning, and thus has the most pronounced effect of this.

The VP_{MV} variant of the VP-tree also matches the construction profile of the standard VP-tree on each dataset, with slightly increased or decreased computations depending on the dataset. This is to be expected, as the standard VP-tree always produces balanced splits during batch construction. The incremental construction can also cause lopsided splits for both the VP and VP_{MV}-tree, which results in a

[6]The same cannot be said for wall clock time, which is expected.

longer recurrence during construction, and thus increased construction time and distances. The VP_{MV}-tree may also encourage such lopsided splits, increasing the occurrence of this behavior. Simultaneously, the incremental construction requires fewer distance computations to determine early splits, and so can result in fewer overall computations if the splits happen to come out near balanced. The data and metric dependent properties will determine which impact is stronger for a given case. The impact of incremental construction on the VP-trees is also variable, and can increase or decrease construction time. In either direction, the change in VP construction time is minor relative to the costs for Cover-trees and the RBC algorithm.

Overall we can draw the following conclusions about construction time efficiency. 1) that the VP-trees are fastest in all cases, and the proposed VP_{MV} variant has no detrimental impact. 2) the RBC algorithms are the most consistent, but often slowest, and that the RBC_{Imp} has no detrimental impact. 3) the Cover-tree is not consistent in its performance relative to the other two algorithms, but when it works well, is in the middle of the road.

8.5.2 Impact on Batch Query Efficiency

We now look at the impact of our changes to the three search procedures on querying the index, when the index is built in the standard batch manner. This isolates the change in performance to only our modifications of the three algorithms. Our goal here is to show that RBC_{Imp} and VP_{MV} are improvements over the standard

RBC and VP-tree methods. We also want to quantify the negative impact of using the looser bounds in Cover$_B$ that will allow for incremental insertion and querying, which is not easy with the standard simplified Cover-tree due to its use of the *maxdist* bound and potential restructuring on insertions [167].

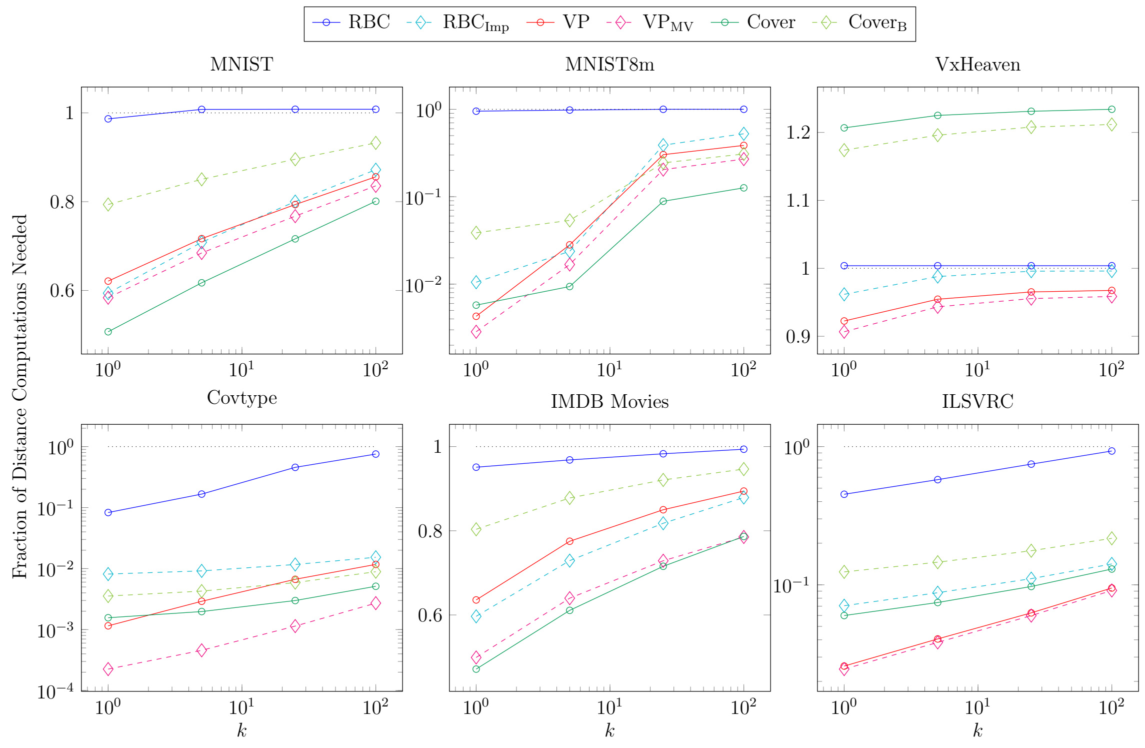


Figure 8.5: Number of distance computations needed as a function of the desired number of neighbors k. The y-axis is the ratio of distance computations compared to a brute-force search (shown at 1.0 as a dotted black line).

Considering only batch construction, we can see the query efficiency of these methods in Figure 8.5, where we look at the fraction of distance computations needed compared to a brute-force search. This figure factors in the distance computations needed during construction time, so the query efficiency is with respect to the whole process[7].

[7]The batch construction is scaled to have the same impact as if we used every dataset instead of a random sample. In extended testing, just 100 samples reliably estimated the percentage. Using 1000 samples gives even higher confidence and makes large tests tractable.

We can see that for the RBC and VP-tree algorithms, our enhancements to the search procedure are effective. For the RBC algorithm in particular, more distance computations were done than the brute force search in most cases, but RBC_{Imp} dramatically improves the competitiveness of the approach. This comes at a loss of compute efficiency when using the euclidean metric, which is where the RBC obtains its original speed improvements. But our work is looking at the general efficiencies of the RBC for arbitrary distance metrics, which may not have the same efficiency advantages when answering queries in batches. In this respect the pruning improvements of RBC_{Imp} are dramatic and important if the RBC algorithm is to be used.

The VP_{MV} reduces the number of computations needed compared to the standard VP-tree in all cases. The amount of improvement varies by dataset, ranging from almost no improvement, to nearly an order of magnitude less distance computations for the Covtype dataset. Given these results our choice to produce unbalanced splits during construction is empirically validated.

As expected, the Cover_{B} variant of the simplified Cover-tree had a detrimental impact on efficiency, as it is relaxing the bound to the same one used in the original Cover-tree work[166]. Among all tests, the Cover_{B}-tree required 1.6 to 6.7 times as many distance computations as the standard Cover-tree, with the exact values given in Table 8.2 for all tested values of k. The few distance computations avoided for determining the tighter bound clearly make up for a considerable portion of the simplified Cover-tree's improved performance.

While the Cover-tree was the most efficient at avoiding distance computations

Table 8.2: For each dataset, the this table shows the multipler on the number of distance computations Cover_B had to perform compared to a normal Cover-tree.

	Dataset					
k	MNIST	MNIST8m	ILSVRC	Covtype	IMDB	VxHeaven
1	1.57	6.73	2.07	2.27	1.70	0.97
5	1.38	5.71	1.96	2.16	1.44	0.98
25	1.25	2.75	1.81	1.97	1.29	0.98
100	1.16	2.44	1.67	1.73	1.20	0.98

on the MNIST dataset, the Cover-tree is the worst performer by far on the VxHeaven dataset. The increased construction time results in the Cover-tree performing 20% more distance computations than would be necessary with the brute force approach. We also see an interesting artifact that more distance computations were done on VxHeaven when using the tighter maxdist bound than the looser Cover_B approach. This comes from the extra computations needed to obtain the maxdist bound in the first place, and indicates that more distances computations are being done to obtain that bound then are saved in more efficient pruning.

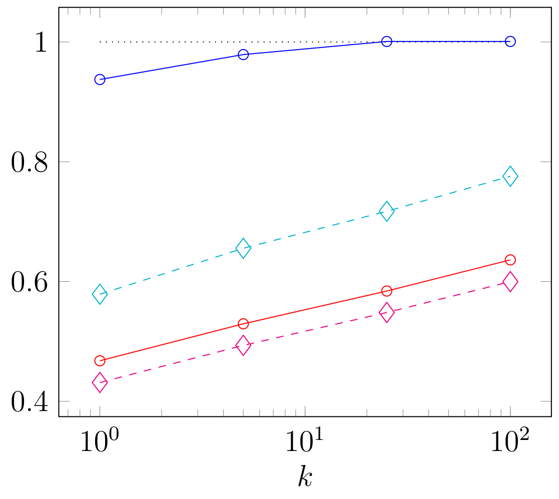


Figure 8.6: Query performance on the VirusShare5m dataset.

We also note that the VxHeaven dataset, using the LZJD distance, had the worst query performance amongst all datasets, with LZJD barely managing to avoid 5% of the distance computations compared to a brute-force search. By testing this on the larger VirusShare5m dataset, as seen in Figure 8.6, we can see that increasing the corpus size does lead to pruning efficiencies. While the Cover-tree couldn't be built on this corpus, both the RBC and VP algorithms are able to perform reasonably well. The VP_{MV} did best, avoiding between 57% and 40% of the distance computations a brute-force search would require.

Viewing these results as a whole, we would have to recommend the VP_{MV} algorithm as the best choice in terms of query efficiency. In all cases it either prunes the most distances for all values of k, or is a close second to the Cover-tree (which has an extreme failure case with LZJD).

8.5.3 Impact of Incremental Construction on Query Efficiency

For the last part of this section, we examine the impact on query pruning based on how the index was constructed. That is to say, does half-batch or incremental construction of the index negatively impact the ability to prune distance computations, and if so, by how much? Such evaluation will be shown for only the more efficient RBC_{Imp} and VP_{MV} algorithms that we will further evaluate in section 8.6. We do not consider the Cover-tree variants in this portion. As noted in subsection 8.3.1, the Cover-tree's construction is already incremental. Thus these indexes will be equivalent when given the same insertion ordering. The only change in Cover-tree efficiency would be from random variance caused by changes in insertion order.

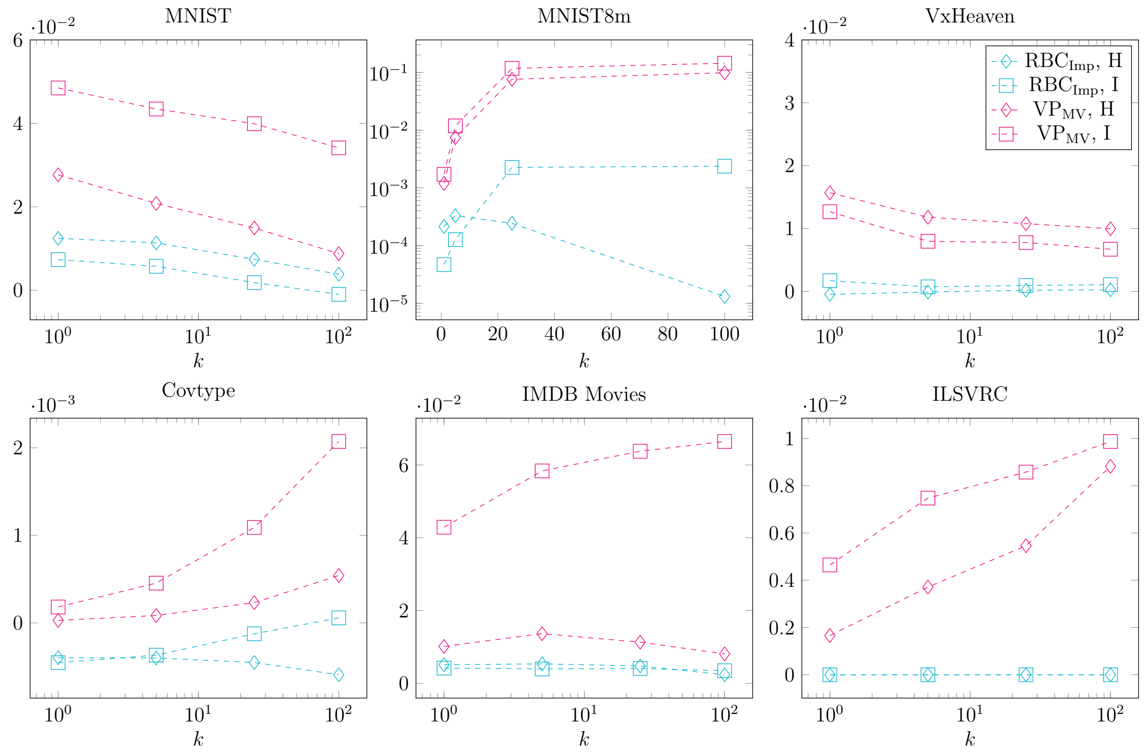


Figure 8.7: Difference in the number of distance computations needed as a function of the desired number of neighbors k. The y-axis is the difference in the ratio of distance computations compared to a brute-force search. We note that the scale on the y-axis is different for various figures, and the small scale indicates that incremental construction has little impact on query efficiency.

The difference between the ratio of distance computations done for Half-Batch (H) and Incremental (I) index construction is shown in Figure 8.7. That is to say, if $r_H = \frac{\text{Distance Computations with Half-Batch}}{\text{Distance Computations Brute Force}}$, and r_B has the same definition but for the Batch construction, then the y-axis of the figure shows $r_B - r_H$. When this value is near zero, it means that both the Batch and Half-Batch or Incremental construction approaches have avoided a similar number of distance computations.

We remind the reader that Half-Batch is where the dataset is constructed using the standard batch construction approach for the first $n/2$ data-points, and the remaining $n/2$ are inserted incrementally. Incremental construction builds the index from empty to full using only the incremental insertions.

Positive values indicate an increase in the number of distance queries needed. Negative values indicate a reduction in the number of distance queries needed, and are generally an indication of problem variance. That is to say, when the difference in ratios can go negative, it's because the natural variance (caused by insertion order randomness) is greater than the impact of the incremental construction. Such scenarios would generally be considered favorable, as it would indicate that our modifications have no particular positive or negative impact.

We first note a general pattern in that the difference in query efficiency can go up or down with changes in the desired number of neighbors k. This will be an artifact of both the dataset and distance metric used, and highlights the importance of testing metric structures over a large range of k. Testing over a wide range of k has not been historically done in previous works, usually performing only the $1-nn$ search.

In our results we can see that the RBC algorithm performs best in these tests. The RBC_{Imp} approach's pruning ability is minimally impacted by changes in construction for all datasets and values of k. The largest increase is on MNIST for $k = 1$, where the Half-Batch insertion scenario increases from 59.4% to 60.6%, an increase of only 1.2 percentage points. It makes sense that the RBC_{Imp} approach would have a consistent minimal degradation in query efficiency, as the structure of the RBC is coarse, and our incremental insertion strategy closely matches the behavior of the batch creation strategy.

The VP_{MV}-tree does not perform as well as the RBC_{Imp}, and we can see that incremental construction always has a more larger, but still small, impact on its performance for all datasets. The only case where this exceeds a two percentage point difference is on the MNIST8m dataset, where a $\approx 7.6\%$ point gap occurs for incremental and half-batch construction. The larger impact on the VP_{MV}'s performance is understandable given that our insertion procedure does not have the same information available for choosing splits, which may cause sub-optimal choices.

Our expectation would be that the VP_{MV}'s performance would degrade more when using incremental (I) insertion rather than half-batch (H), as the half-batch insertion will get to use more datapoints to estimate the split point for nodes higher up in the tree. Our results generally support this hypothesis, with VP_{MV} (I) causing more distance queries to be performance than the (H) case. However, for MNIST8m, VxHeaven, and ILSVRC, the performance gap is not that large across the tested values of k. This suggests that the loosened bounds during insertion may also be an issue impacting the efficiency after insertions. One possible way to reduce this impact would be to add multiple vantage points dynamically during insertion, to avoid impacting the existing low/high bounds of the VP-tree. Such Multi-Vantage-Point (MVP) trees have been explored previously[208] in a batch construction context. We leave research in exploiting such extensions to to future work.

Regarding the impact on query efficiency given incremental insertions, we can confidently state that the RBC approach is well poised to this part of the problem, with almost no negative impact to efficiency. The VP-tree does not fare quite as well, but is still more efficient than the RBC_{Imp} algorithm in all of these cases after construction from only incremental insertions.

8.6 Evaluation of Incremental Insertion-Query Efficiency

At this point we have shown that RBC_{Imp} and VP_{MV} are improvements over the original RBC and VP-tree algorithms in terms of query efficiency, with no significant impact on the construction time. We have also shown that the indexes constructed by them are still effective are pruning distance computations, which encourages their use. We can now evaluate their overall effectiveness when we interleave insertions and queries in a single system.

In this section we now consider the case of evaluating each index from the context of incremental insertion and querying. Contrasting with the standard scenario, where we build an index and immediately query it (usually for k-nearest neighbor classification, or some similar purpose), we will be building an index and evaluating the number of distance computations performed after construction. This scenario corresponds to many realistic use cases, where a large training set is deployed for use, and new data added to the index over time.

Given a dataset with n items in it, our evaluation procedure will consider r queries (or "reads") and w insertions (or "writes") to the index. The naive case, where we perform brute force search, there is no cost to writing to the index, only when we perform a query. This brute force approach also represents our baseline for the maximum number of distance computations needed to answer the queries.

Similar to data structures for storing and accessing data and concurrency tools, we may also explore differing ratios of reads to writes. In our experiments we evaluated insert/query ratios from 100:1 to 1:100. In all cases, we found that the most challenging scenario was when we had 100 insertions for each query. This is not surprising, as all of our data structures have a non-zero cost for insertions, and in the case of RBC and Cover-trees, can be quite significant. Thus, below we will only present results for the case where we have 100 insertions for each query, and our tests will be limited to 1000 insertions due to runtime constraints[8]. We construct each initial index on half of the data points, using the batch construction method.

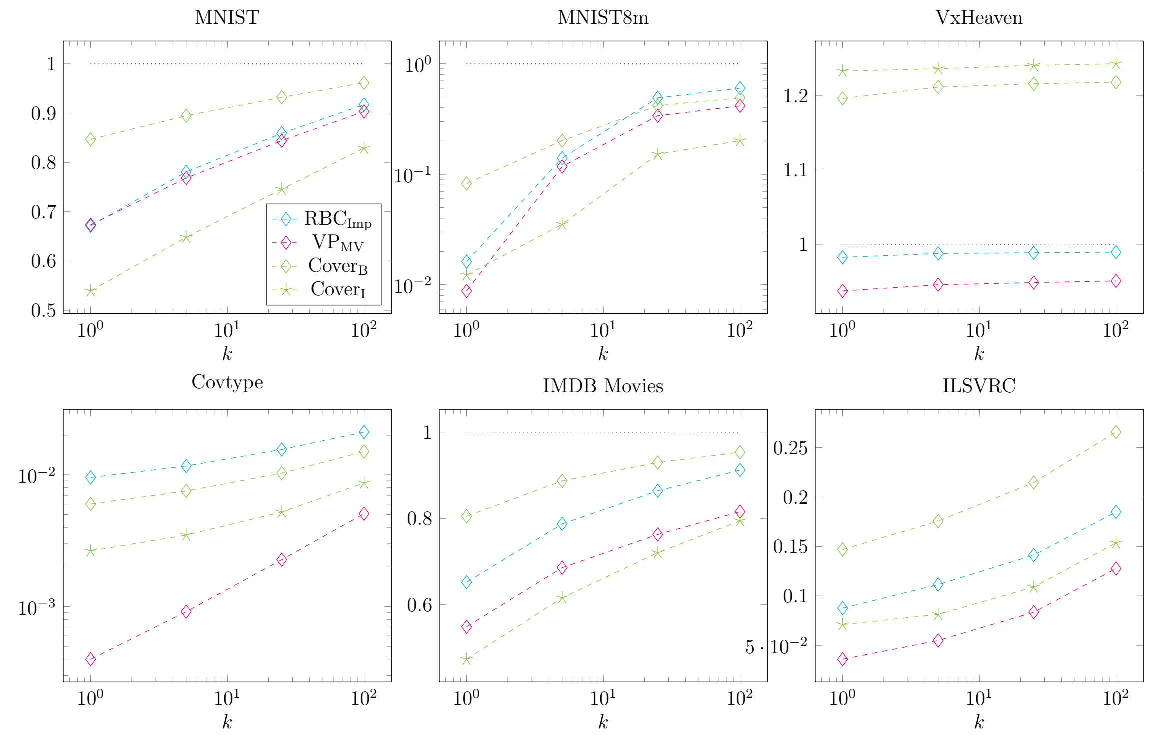


Figure 8.8: Fraction of distance computations needed (relative to naive approach) in incremental scenario, with 100 insertions for every query. Does not include *initial* construction costs, only subsequent insertion costs.

For the Cover-tree, only Cover$_{\text{B}}$ produces reasonable insertion/query perfor-

[8]We allowed a maximum of one week runtime for tests to complete in this scenario.

mance, as the *maxdist* bound can't be maintained when re-balancing occurs. Using the original loose bound causes a considerable reducing in efficiency at query time. By recording the multiplicative difference between the tighter bound Cover-tree and the original looser bound in Cover_B in Table 8.2, we can plot the performance of the *ideal* Cover-tree as a function of Cover_B. This gives us a measure of what the best possible performance of the Cover-tree would be in this scenario, as it ignores all overheads in any potential scheme for selectively updating the Cover-tree bound as items are inserted that would cause re-balancing. We will indicate this ideal Cover-tree as Cover_I.

The results of our methods are presented in Figure 8.8. Amongst the RBC_Imp, VP_MV, and Cover_B algorithms, the VP_MV dominates all other approaches. It successfully avoids the most total distance computations to answer nearest neighbor queries for all values of k on all datasets. This is not surprising given the cumulative results of section 8.5, which found the VP_MV to require the fewest distance computations during construction time and was always either the most efficient at avoiding distance computations, or nearly behind the Cover-tree approach.

If we had an ability to obtain the *maxdist* bound for free, we can also see that the Cover_I approach is still not very competitive with the VP_MV-tree. While Cover_I does have better performance than VP_MV on some datasets, it often trails behind on the Covtype by nearly an order of magnitude. Especially when we consider the failure of the Cover-trees to perform with the LZJD distance on VxHeaven and VirusShare5m. This variability in performance makes the Cover-tree less desirable to use for arbitrary distance metrics.

While the VP_{MV} appear to be the best overall fit to our task, we note that our RBC_{Imp} also makes a strong showing despite the $O(\sqrt{n})$ complexity target instead of $O(\log(n))$. RBC_{Imp} consistently performs better than random guessing, which can't be said for the Cover-tree. On the more difficult datasets, it is often not far behind the VP_{MV}-tree in performance, though it is an order of magnitude less efficient on the Covtype and ILSVRC datasets. The biggest weakness of the RBC approach is that the incremental insertions will have an amortized cost, with the insertion time increasing dramatically every $\sqrt{n}$ insertions to expand the representative set. If the number of insertions is known to be bounded, this may be an avoidable cost – thus increasing the RBC's practicality. We note as well that in the case of datasets stored in a distributed index across multiple server's, the RBC's coarse structure may allow for more efficient parallelization. This may be an important factor in future work when we consider datasets larger than what can be stored on a single machine.

8.6.1 Discussion

While we have modified three algorithms for our scenario of incremental querying and insertion, we note that there is a further unexplored area for improvement in the "Read write" ratio. In our case it was most challenging for all algorithms to handle more "Writes" per "read", as each insertion required multiple distance computations and the insertions did not dramatically change the performance at query time. This is in part because we have modified existing algorithms to support this scenario, and so the performance interleaving insertions and queries closely follows the performance when we evaluate query by including the construction cost, as we did in section 8.5.

Of the algorithms we have tested the $\mathrm{VP}_{\mathrm{MV}}$ performs best with the lowest construction time, and is almost always the fastest at query time. This is also in the context of evaluation in a single-threaded scenario. When we consider a multi-threaded scenario, the $\mathrm{VP}_{\mathrm{MV}}$ can utilize multiple threads for index construction using the batch-construction approach. However, insertion of a single data-point cannot easily be parallelized. The Cover-tree also has this challenge.

Our $\mathrm{RBC}_{\mathrm{Imp}}$ approach presents a potential advantage over both of these algorithms when we consider the multi-thread or distributed scenario. As a consequence of how the RBC algorithm achieves its $O(\sqrt{n})$ insertion and query time, we can readily parallelize line 1 of Algorithm 6 on up to $\sqrt{p}$ processors, requiring only a reduce operation to determine which processor had the closest representative. It may then be more practical than the $\mathrm{VP}_{\mathrm{MV}}$ approach for extremely large indexes if sufficient compute resources are available. The downside to the RBC algorithm comes when the representative set must be increased, requiring more work and presenting a insertion cost that will periodically spike. This could be remedied by amortizing the cost of increasing the representative set across the preceding insertions, but we leave this to future work as we must consider the real-world efficiency of an implementation to determine how practical a solution it would be.

In future work we hope to develop new algorithms that are specifically designed for incremental insertion and querying. We note two potential high level strategies in which one may develop methods that perform better for read and write heavy use-cases. We consider these beyond the scope of our current work, which looks at modifying existing algorithms, but may be fruitful inspiration for specialized methods.

8.6.1.1 Write and Insert Heavy

When we have multiple datapoints inserted before each query, it may become possible to use the index itself to accelerate the insertion process. Say that there will be a set of Z points inserted into the index at a time. We can cluster the members of Z by their density/closeness, and insert each cluster together as a group. One option may be to find the medoid of the group and its radius, which can then be used as a proxy point that represents the group as a whole. One could then insert the sub-groups into the index with a reduced number of distance computations if the triangle inequality can be used to determine that all members of the group belong in the same region of the index. The group may then be dissolved as such macro level pruning becomes impossible, or reduced into smaller sub-groups to continue the process. The dual-tree query approach [209], at a high level, presents a similar strategy for efficiently answering multiple queries at a time.

8.6.1.2 Read and Query Heavy

Another scenario is that insertions into the index will be relatively rare, compared to the amount of nearest neighbor queries given to the index. In this case it may be desired to have the query process itself build and restructure the tree. This notion is in a similar spirit to splay trees and the union-find algorithm [210]–[212]. Insertions to the dataset would be placed in a convenient location, and their first distances computed when a new query is given. Say that x_i was a previously inserted point. Once we have a new query x_q, the distance to the query is obtained for x_i and for x_q's nearest neighbors. If $d(x_i, x_q) \approx c \cdot d(x_q, x^{(k)})$, where $x^{(k)}$ is x_q's k'th nearest neighbor and c is some constant, we can then infer that x_i should be placed in a similar location in the index. As multiple insertions are performed, we can use these distances with respect to the query to determine which points are related and should be kept close in the index.

8.7 Conclusions

We have now evaluated and improved three different algorithms, Cover Trees, Vantage-Point Trees, Random Ball Covers, for the use case of incremental insertions and querying. We have significantly improved the query efficiency of the later two with our new RBC_{Imp} and VP_{MV} variants, and introduced schemes to incrementally add to these collections. Evaluation of all these methods was done with a number of datasets with varying sizes using four different distance metrics. In doing so, we can conclude that the VP_{MV} tree provides the best overall performance for our task. It requires the fewest distance computations during construction, is consistently one of the fastest at query time, and this balance produces the best overall results when interleaving insertions and queries.

While already successful, the VP_{MV} tree still has room for improvement. It has the highest degradation to performance from insertions, which could perhaps be remedied by a smarter update algorithm or the use of multiple vantage points. While the Cover_{B} algorithm could be improved by obtaining a better alternative bound than *maxdist*, it appears obtaining a computational cheaper version *maxdist* bound itself is not sufficient to remedy the performance gap when using the LZJD distance.

Chapter 9: Conclusions and Future Work

Through this thesis we have now developed the Lempel-Ziv Jaccard Distance. We've used it to develop a fast and efficient malware classifier that outperforms byte n-grams, and can directly tackle the class imbalance problem. We've used it on Windows and Android malware, as well as a variety of common file types. In doing so LZJD has also outperformed pre-existing tools for similarity search used in digital forensic investigations. To further support LZJD's use in such scenarios, we have developed enhancements to the Vantage-Point tree enabling incremental index construction and querying.

While we have developed a tool that is orders of magnitude faster than what was previously available, we must emphasize a critical communal item of future work: improved sharing. Datasets in the malware space are valuable and often highly guarded due to the collection and labeling costs associated with them. Advanced domain knowledge tools for feature extraction or processing of executables often involve years of development, making it impractical for independent groups to re-implement, yet are also kept from the public. Combined, these make it nearly impossible for researchers to effectively compare with prior works, and will prevent consensuses from forming on many issues. Solving these problems, even in part, will likely take the action of larger corporations that own the kind of data researchers like ourselves must ask for access to.

Until such issues are solved, we note some general items of future work, beyond the chapter-specific future works that have been discussed periodically.

New Compression Method Specific Algorithms

The NCD measure can be thought of as a compression-method agnostic way of measuring file similarity. LZJD in turn is compression method *specific*, in that is based specifically upon the Lempel-Ziv scheme. This opens up a natural question: what other specific compression methods can have new distance measured developed from them, and what advantages might they have? Given the long history of compression based research, this may be an especially fertile research area in the future.

New Domain-Knowledge Free Methods

Currently, the primary methods that can be used today with copora like our Industry 2 Million set are byte n-grams and LZJD. The n-gram approach will likely need modification to scale to even larger corpora, as it is already pushing the computational limits of our resources. Developing new orthogonal approaches is important, whether that be increasing the effectiveness of entropy based methods or other approaches. One approach that has made progress in recent years is the use of neural networks trained on raw bytes [213], [214]. This too has its own complications and limitations, but may prove valuable in the long term.

Appendix A: Effective Alternative to Ssdeep and Sdhash Appendix

A.1 Full FRASH Results

Here in the appendix we provide more complete results from the FRASH tests for those who are interested. In these full tables, *Score* is the average score for each match, and *Matches* or *Match* it the absolute number of matches at that size.

A.1.1 Single Common Block Tables

In these tables, we show the average block size percentage as the *Size* column. The associated average block size can be computed from these tables by multiplying the total block size of the table, with the percentage given in each column. The SCB tests were run for 50 trials each. This covers the results in Table A.1, Table A.2, and Table A.3.

	ssdeep		sdhash		LZJD	
Score	Size (%)	Matches	Size (%)	Matches	Size (%)	Matches
≥65	48.44	2	—	—	—	—
≥60	47.54	14	—	—	—	—
≥55	43.59	20	—	—	—	—
≥50	42.63	39	—	—	—	—
≥45	38.78	44	48.96	6	—	—
≥40	32.42	48	46.61	12	—	—
≥35	26.45	43	42.38	16	—	—
≥30	22.94	41	40.85	28	—	—
≥25	18.42	28	36.13	32	44.15	39
≥20	20.62	5	31.33	37	15.75	50
≥15	—	—	27.27	44	3.25	50
≥10	—	—	20.88	50	—	—
≥5	—	—	10.88	50	—	—
0	15.5	50	3.12	50	—	—

Table A.1: Complete Single Common Block results for a total block size of 512 KB.

	ssdeep		sdhash		LZJD	
Score	Size (%)	Matches	Size (%)	Matches	Size (%)	Matches
≥70	44.92	2	—	—	—	—
≥65	44.82	8	—	—	—	—
≥60	43.36	12	—	—	—	—
≥55	42.86	28	—	—	—	—
≥50	40.28	41	—	—	—	—
≥45	37.27	48	47.27	4	—	—
≥40	30.05	49	46.03	12	—	—
≥35	25.98	48	42.68	19	—	—
≥30	21.54	46	39.84	27	—	—
≥25	18.85	23	35.66	34	—	—
≥20	19.98	7	31.38	42	—	—
≥15	—	—	24.47	44	42.39	46
≥10	—	—	18.72	50	18.38	50
≥5	—	—	9.17	50	0.78	50
0	19.17	50	0.88	50	—	—

Table A.2: Complete Single Common Block results for a total block size of 2 MB.

	ssdeep		sdhash		LZJD	
Score	Size (%)	Matches	Size (%)	Matches	Size (%)	Matches
≥70	44.92	1	—	—	—	—
≥65	46.7	8	—	—	—	—
≥60	45.81	20	—	—	—	—
≥55	42.55	31	—	—	—	—
≥50	39.05	40	49.9	2	—	—
≥45	35.83	48	47.01	7	—	—
≥40	28.52	50	43.86	13	—	—
≥35	23.74	49	41.94	22	—	—
≥30	20.07	50	37.3	25	—	—
≥25	18.46	36	33.41	31	—	—
≥20	16.41	9	28.36	35	49.02	1
≥15	12.89	1	26.47	44	36.56	50
≥10	—	—	19.75	50	6.8	50
≥5	—	—	9.51	50	0.2	50
0	17.61	50	0.99	50	—	—

Table A.3: Complete Single Common Block results for a total block size of 8 MB.

A.1.2 Random Noise Table

The full results from the random noise test are given in Table A.4. The *Change* column is the average percent of bytes in the filed that needed to be edited for a score of that value to be obtained, and *Match* is the number of files that FRASH was able to successfully reduce to the given score range. The most robust method for each score is shown in **bold**. The default spacing used in FRASH is 10, but we reduced the spacing to 5 to take advantage of LZJD's performance of LZJD. The high resitance of LZJD meant that a zero value was never produced, which did not interact well with FRASH's execution. The second to last row shows that for 7 of the 100 files, a match score in the range of $[1, 5)$ was produced by modifying an average of 32% of the file. This value is artificially low, as almost all tests were stopped prematurely before LZJD even reached a score of 15.

	ssdeep		sdhash		LZJD	
Score	Change (%)	Match	Change (%)	Match	Change (%)	Match
≥95	0.00058	100	**0.01293**	84	0.00151	60
≥90	0.00136	99	**0.02655**	90	0.00317	46
≥85	0.00211	99	**0.04710**	91	0.00490	55
≥80	0.00291	99	**0.06560**	93	0.00777	59
≥75	0.00398	93	**0.09060**	95	0.01243	65
≥70	0.00524	88	**0.10677**	96	0.02378	75
≥65	0.00814	73	**0.13483**	97	0.07260	86
≥60	0.01214	51	**0.17455**	97	0.16215	95
≥55	0.01676	30	0.22571	96	**0.36321**	95
≥50	0.02057	18	0.27750	99	**0.60610**	96
≥45	0.04409	10	0.32868	100	**1.14643**	99
≥40	0.06148	7	0.39398	100	**1.96722**	99
≥35	0.04466	3	0.45643	100	**3.44188**	100
≥30	0.05740	2	0.54923	100	**6.22476**	100
≥25	—	—	0.62586	99	**10.99275**	100
≥20	—	—	0.69417	99	**18.03654**	100
≥15	—	—	0.84723	98	**30.56353**	90
≥10	—	—	0.97390	99	**48.63043**	77
≥5	—	—	1.14414	100	**63.80973**	57
$[1, 5)$	—	—	—	—	32.03773	7
0	0.01283	100	1.55763	100	—	—

Table A.4: Random Noise Test.

The change and match values in this table are also shown in Figure A.1, which plots the number of files matched against the the percentage of the file changed. Note that the x-axis is on a log-scale. This score can go up and down because it is based on the number of files matched receiving a minimum score (i.e., a score ≥ 90). Because of this interpretation of the figure must be done carefully, and emphasize that this figure is to demonstrate the range of byte alterations each method can withstand. In this light is becomes clear that ssdeep is only able to produce matches when very little of the file has been altered, less than 0.1%. Sdhash is able to perform matches

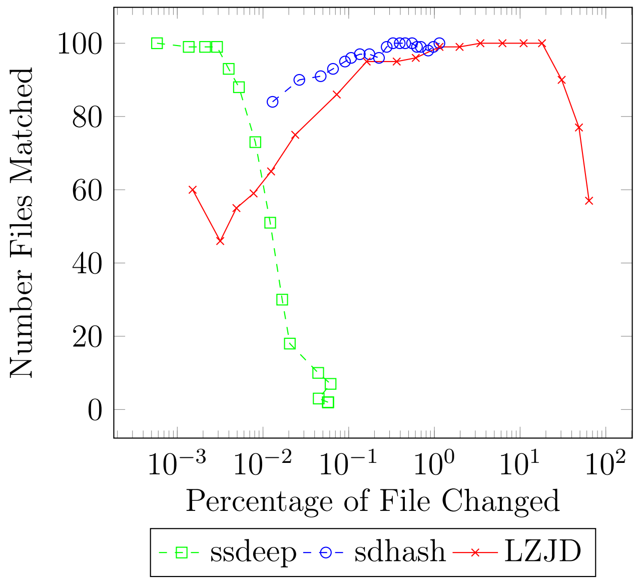


Figure A.1: Random Noise results, plotted for each method showing how much of the file can be changed while still obtaining a correct match.

in a range up to 1.1% of the file being randomly altered, but fails to produce any matches past this point. LZJD in contrast is able to suffer from as much as 63% of the file being randomly altered, and still over half the files. It is the only method to cover this large of a range in the amount of bytes that can be altered. Again, we note that the lower number of matches obtained by LZJD and sdhash in the left-most portion of the plot are because the associated minimum score is not factored in. For example, the 60 matches of LZJD at 0.002% is not indicating that only 60 of the files could be matched after that percentage of files changed. It's value is 60 because only 60 files could be matched *and* obtain a similarity score ≥ 95.

A.1.3 Fragment Test Tables

Tables Table A.5 and Table A.6 are the complete version of Table 7.11 and Table 7.12 respectively. The *Size* column is the percent file size. We can see that when LZJD and Sdhash don't get every match, LZJD always has more matches. It is also more clear that Sdhash's performance degrades at around 5% and drops quickly, where LZJD is more robust in being able to still hit matches.

	ssdeep		sdhash		LZJD	
Size	Score	Matches	Score	Matches	Score	Matches
95	96.7	4454	83.4	**4457**	72.4	**4457**
90	92.3	4452	70.1	**4457**	49.2	**4457**
85	89.5	4442	69.9	**4457**	45.5	**4457**
80	86.5	4433	69.1	**4457**	41.9	**4457**
75	83.5	4417	68.3	**4457**	38.8	**4457**
70	80.1	4403	68.0	**4457**	35.7	**4457**
65	76.7	4367	68.4	**4457**	32.9	**4457**
60	73.2	4321	68.4	**4457**	30.1	**4457**
55	69.7	4205	68.0	**4457**	27.5	**4457**
50	65.9	4071	68.5	**4457**	24.9	**4457**
45	62.4	3699	69.2	**4457**	22.4	**4457**
40	58.6	3140	69.8	**4457**	20.1	**4457**
35	54.7	2477	71.0	**4457**	17.8	**4457**
30	51.2	1704	71.4	**4457**	15.5	**4457**
25	47.9	928	72.2	**4456**	13.2	**4456**
20	45.7	411	73.1	4453	11.0	**4456**
15	43.7	132	73.9	4450	8.8	**4456**
10	46.2	29	75.7	4371	6.4	**4456**
5	61.0	2	77.4	4036	3.9	**4454**
4	—	—	78.4	3838	3.3	**4444**
3	—	—	78.7	3616	2.7	**4432**
2	—	—	79.1	3257	2.0	**4419**
1	—	—	81.0	2581	1.3	**4390**

Table A.5: Fragment detection test result (cut side: random start, then alternating). Matches gives the

	ssdeep		sdhash		LZJD	
Size	Score	Matches	Score	Matches	Score	Matches
95	97.70	**4457**	97.28	**4457**	92.80	**4457**
90	95.90	4456	98.17	**4457**	85.96	**4457**
85	93.81	4453	98.86	**4457**	79.33	**4457**
80	91.42	4444	99.32	**4457**	72.99	**4457**
75	88.85	4440	99.37	**4457**	66.87	**4457**
70	85.92	4429	99.44	**4457**	61.01	**4457**
65	82.79	4414	99.49	**4457**	55.39	**4457**
60	79.36	4378	99.46	**4457**	50.08	**4457**
55	75.68	4307	99.49	**4457**	44.98	**4457**
50	71.73	4148	99.51	**4457**	40.06	**4457**
45	68.01	3815	99.41	**4457**	35.46	**4457**
40	64.39	3326	99.37	**4457**	31.11	**4457**
35	61.15	2697	99.28	**4457**	26.97	**4457**
30	58.26	1968	99.20	**4457**	23.00	**4457**
25	56.76	1191	99.07	**4457**	19.21	**4457**
20	55.35	632	98.83	**4457**	15.54	**4457**
15	53.21	289	98.46	**4457**	11.94	**4457**
10	56.90	77	97.88	4380	8.33	**4457**
5	55.68	22	96.93	4061	4.63	**4457**
4	59.00	12	96.40	3896	3.88	**4457**
3	47.89	9	95.04	3678	3.09	**4457**
2	66.00	2	93.09	3304	2.23	**4456**
1	—	—	90.50	2617	1.36	**4442**

Table A.6: Fragment detection test result Fragment detection test result (cut side: right (end only), 5 %). Matches gives the

A.1.4 Alignment Table Results

In Table A.7 we provide the alignment results for adding ≤64 KB to the query file. This is the easier range of the test, and we can see that both sdhash and LZJD successfully match all files regardless of added bytes.

	ssdeep		sdhash		LZJD	
Added (KB)	Score	Matches	Score	Matches	Score	Matches
4	91.31	4439	51.30	**4457**	42.38	**4457**
8	87.07	4279	78.51	**4457**	36.28	**4457**
12	84.39	4120	65.57	**4457**	32.60	**4457**
16	82.76	3901	64.22	**4457**	30.01	**4457**
20	82.19	3690	80.50	**4457**	28.04	**4457**
24	81.21	3580	51.58	**4457**	26.50	**4457**
28	79.98	3465	90.36	**4457**	25.20	**4457**
32	79.41	3314	52.38	**4457**	24.13	**4457**
36	79.49	3154	78.37	**4457**	23.21	**4457**
40	79.15	3059	65.83	**4457**	22.39	**4457**
44	79.34	2949	64.25	**4457**	21.65	**4457**
48	78.67	2895	80.62	**4457**	21.02	**4457**
52	78.03	2839	52.48	**4457**	20.40	**4457**
56	77.41	2775	88.19	**4457**	19.88	**4457**
60	76.65	2721	53.60	**4457**	19.39	**4457**
64	76.27	2645	78.05	**4457**	18.95	**4457**

Table A.7: Alignment test result (fixed size, step size = 4 KB, max size = 64 KB)

Appendix B: Metric Index Appendix

B.1 Corrections to Simplified Cover Tree

We encountered two difficulties in replicating the simplified cover tree results of Izbicki and Shelton [167]. We detail these two issues and their remediations in this section for completeness and reproducibility. In the below algorithm descriptions we will use the same terminology and description as the algorithm's original paper, but note our changes in green.

We now review some of the properties needed to understand our corrections. The simplistic such property is that each node p in the Cover tree has an associated level l, which we can obtain as $l = \text{level}(p)$. Each child c_p of p must also satisfy the property that $\text{level}(p) = \text{level}(c_p) + 1$.

Using a node's level, we can define its *coverdist* as $\text{coverdist(p)} = 2^{\text{level}(p)}$. Each child c_p of p will satisfy the covering invariant property, $d(c_p, p) \leq \text{coverdist}(p), \forall c_p \in \text{children}(p)$.

We also must make use of the *maxdist* bound discussed in subsection 8.3.1, which we make more explicit as: $\text{maxdist}(p) = \arg\max_{d_p \in \text{descendants}(p)} d(d_p, p)$. This is the maximum distance from one node p to *any* descendant note of p. If p is a leaf node, meaning it has no children, then $\text{maxdist}(p) = 0$.

B.1.1 Nearest Neighbor Correction

We present the revised nearest neighbor search procedure for the simplified Cover-tree in Algorithm 8. The green $d(x, p)$ term was originally presented to be $d(y, q)$. We show that this is not correct using a simple counter example using scalar node values and the euclidean distance.

Algorithm 8 Cover Tree Find Nearest Neighbor

```
Require: cover tree $p$, query point $x$, nearest neighbor so far $y$
1: if $d(p, x) < d(y, x)$ then
2:     $y \leftarrow p$
3: end if
4: for each child $q$ of $p$ sorted by distance to $x$ do
5:     if $d(y, x) > d(x, q) - \text{maxdist}(q)$ then        ▷Original paper used $d(y, q)$
6:         $y \leftarrow \text{findNearestNeighbor}(q, x, y)$
7:     end if
8: end for
9: return $y$
```

Consider the Cover-tree with root α, that stores value 5. α has one child, β, which has the value -2. This is the whole tree.

We would begin on line one of the algorithm, with $p \leftarrow \alpha$ and we will use our query point x to have a value of 0. $d(p, x)$ is 5, and we have no nearest neighbor so far, so $y \leftarrow p$ (which is α) becomes the nearest neighbor so far.

We will obtain $q \leftarrow \beta$ as it is the only child of α, which leads us to evaluate the original expression

$$\underbrace{d(y, x)}_{=5-0=5} > \underbrace{d(y, q)}_{=5-(-2)=7} - \underbrace{\text{maxdist}(q)}_{=0}$$

Because $5 > 7$ is false, the if statement fails, and we then break from the loop, returning y as the nearest neighbor to x with a distance of 5. But x's value is 0, and β's is -2, which is a distance of only two away.

B.1.2 Insertion Correction

We also provide a correction to the insertion procedure of the simplified Cover-tree. Our fixed version is presented in Algorithm 9, with the green text indicating only added statements to the algorithm.

The issue with the original procedure occurs when an outlier x is inserted into the index, the distance from which to any point in the dataset is larger than the largest pairwise distance of any two points in the existing Cover-tree. This is because the $2\text{coverdist}(p) \geq maxdist(p)$ in all cases. If x is farther than the maximum pairwise distance, then the simple bound on line four may be true for a all points in a valid cover tree. This means the loop will never exit, and will simply continue re-structuring the tree in search of a non-existing node that can satisfy the loop condition.

We fix this by keeping track of the points visited in the tree, and only loop while there is a potential candidate remaining. If no such candidate occurs because we have visited all possible leaf nodes, the loop must exit so that the outlier may be inserted as the new root of the tree.

From a practical implementation perspective, we note two additional choices. First, rather than attempt to remove leaf nodes in the specified form above, it is easier to define a specific leaf removal order and leaf insertion order. For example,

Algorithm 9 Simplified Cover Tree Insertion

```
Require: Query $q$, desired number of neighbors $k$
 1: procedure INSERT(cover tree $p$, data point $x$)
 2:     if $d(p, x) > \text{covdist}(p)$ then
 3:         $z \leftarrow \emptyset$
 4:         while $d(p, x) > 2\text{covdist}(p)$ and $|\text{descendants}(p)| > |z|$ do
 5:             Remove any leaf $q$ from $p \backslash z$
 6:             $p' \leftarrow$ tree with root $q$ and $p$ as only child
 7:             $p \leftarrow p'$
 8:         end while
 9:         return tree with $x$ as root and $p$ as only child
10:     end if
11:     return INSERT_$(p, x)$
12: end procedure
13: procedure INSERT_(cover tree $p$, data point $x$)
14:     for all $q \in \text{children}(p)$ do
15:         if $d(q, x) \leq \text{covdist}(q)$ then
16:             $q' \leftarrow$ INSERT_$(q, x)$
17:             $p' \leftarrow p$ with child $q$ replaced with $q'$
18:             return $p'$
19:         end if
20:     end for
21:     return $p$ with $x$ added as a child
22: end procedure
```

if one always removes the least recently added leaf node, we will obtain a consistent ordering of the leaf nodes as we iterate line four of the algorithm. This makes it easy to use simple cycle detection to determine that the all possible children have been visited, and then escape the loop when this occurs.

To speed up insertion of outlier points, we also note that the covering invariant can be used to catch extreme outliers. If $d(p, x) > 4\text{covdist}(p)$, then we can skip the loop entirely and proceed directly to line eight of the algorithm. This bound is easy to see, as $2\text{covdist}(p) \geq \text{maxdist}(p)$. Assuming that there exists a descendant point γ that is maximally far from p. Let ζ be the point maximally far from γ, and let $d(\gamma, \zeta)$ be the maximal pairwise distance for all points in the Cover-tree. Direct application of the triangle inequality gives us

$$d(\gamma, \zeta) \leq d(\gamma, p) + d(p, \zeta)$$

This bounds the distance between these points by their distance to the root. The covering invariant tells us that $2\text{coverdist}(p) \geq \mathit{maxdist}(p)$. Therefore it must be the case that

$$d(\gamma, \zeta) \leq 2\text{coverdist}(p) + 2\text{coverdist}(p)$$

Which reduces to the bound $d(\gamma, \zeta) \leq 4\text{coverdist}(p)$. Thus if a new query violates this bound, we know that no point in the whole tree can satisfy the loop on line 4.

B.2 Query Efficiency Plots

Below we provide the plots of query efficiency as a function of k for half-batch and incremental construction. These plots are the counterparts to Figure 8.5, and are visually very similar. The visual similarity is because of the minimal impact incremental construction has on our algorithms, as discussed in subsection 8.5.3. We hope these tables provide a more intuitive visual proof for those who prefer looking at the results rather than a plot of the difference in ratio.

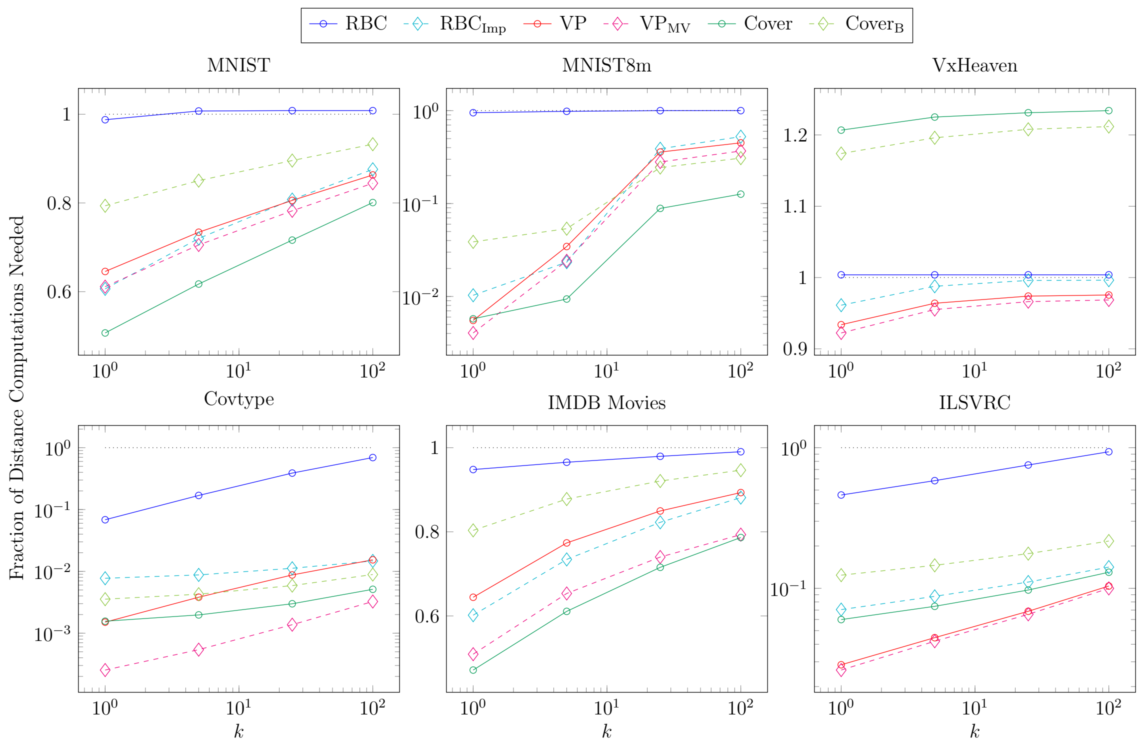


Figure B.1: Number of distance computations needed as a function of the desired number of neighbors k, each index is constructed in the half-batch manner. The y-axis is the ratio of distance computations compared to a brute-force search (shown at 1.0 as a dotted black line).

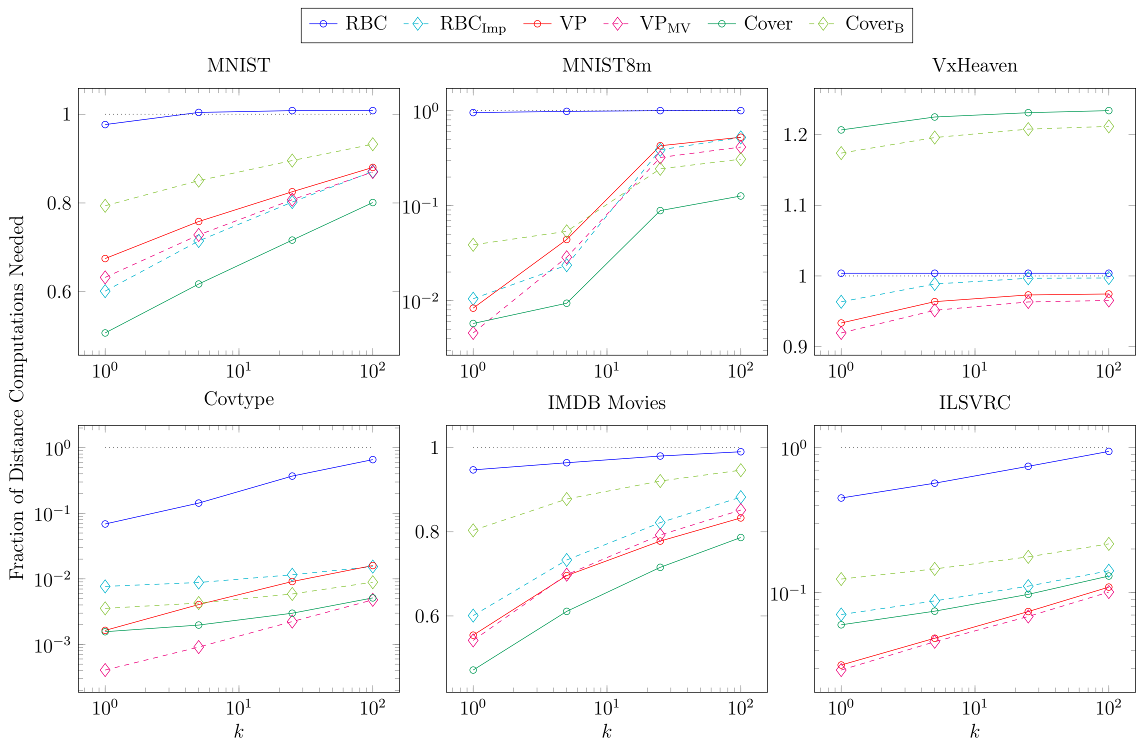


Figure B.2: Number of distance computations needed as a function of the desired number of neighbors k, each index is constructed completely incrementally. The y-axis is the ratio of distance computations compared to a brute-force search (shown at 1.0 as a dotted black line).

Bibliography

[1] M. Schultz, E. Eskin, F. Zadok, and S. Stolfo, “Data Mining Methods for Detection of New Malicious Executables,” in *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*, IEEE Comput. Soc, 2001, pp. 38–49, ISBN: 0-7695-1046-9. DOI: 10.1109/SECPRI.2001.924286.

[2] R. Islam, R. Tian, L. Batten, and S. Versteeg, “Classification of Malware Based on String and Function Feature Selection,” in *Cybercrime and Trustworthy Computing Workshop (CTC), 2010 Second*, Jul. 2010, pp. 9–17. DOI: 10.1109/CTC.2010.11.

[3] J. Saxe and K. Berlin, “Deep neural network based malware detection using two dimensional binary program features,” in *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*, IEEE, Oct. 2015, pp. 11–20, ISBN: 978-1-5090-0317-4. DOI: 10.1109/MALWARE.2015.7413680.

[4] R. Lyda and J. Hamrock, “Using Entropy Analysis to Find Encrypted and Packed Malware,” *IEEE Security and Privacy Magazine*, vol. 5, no. 2, pp. 40–45, Mar. 2007, ISSN: 1540-7993. DOI: 10.1109/MSP.2007.48.

[5] K. S. Han, J. H. Lim, B. Kang, and E. G. Im, “Malware analysis using visualized images and entropy graphs,” *International Journal of Information Security*, vol. 14, no. 1, pp. 1–14, 2015, ISSN: 1615-5270. DOI: 10.1007/s10207-014-0242-0.

[6] D. Baysa, R. M. Low, and M. Stamp, “Structural entropy and metamorphic malware,” *Journal of Computer Virology and Hacking Techniques*, vol. 9, no. 4, pp. 179–192, 2013, ISSN: 2263-8733. DOI: 10.1007/s11416-013-0185-4.

[7] I. Sorokin, “Comparing files using structural entropy,” *Journal in Computer Virology*, vol. 7, no. 4, pp. 259–265, 2011, ISSN: 1772-9904. DOI: 10.1007/s11416-011-0153-9.

[8] G. Shanmugam, R. M. Low, and M. Stamp, “Simple Substitution Distance and Metamorphic Detection,” *J. Comput. Virol.*, vol. 9, no. 3, pp. 159–170, Aug. 2013, ISSN: 1772-9890. DOI: 10.1007/s11416-013-0184-5.

[9] S. Naval, V. Laxmi, N. Gupta, M. S. Gaur, and M. Rajarajan, “Exploring Worm Behaviors Using DTW,” in *Proceedings of the 7th International Conference on Security of Information and Networks*, ser. SIN ’14, New York, NY, USA: ACM, 2014, 379:379–379:384, ISBN: 978-1-4503-3033-6. DOI: 10.1145/2659651.2659737.

[10] M. Wojnowicz, G. Chisholm, M. Wolff, and V. K. Ave, “Suspiciously Structured Entropy : Wavelet Decomposition of Software Entropy Reveals Symptoms of Malware in the Energy Spectrum,” in *Proceedings of the Twenty-Ninth International Florida Artificial Intelligence Research Society Conference*, Z. Markov and I. Russell, Eds., Key Largo, Florida: {AAAI} Press, 2016, pp. 294–298, ISBN: 978-1-57735-756-8.

[11] T. Abou-Assaleh, N. Cercone, V. Keselj, and R. Sweidan, “N-gram-based detection of new malicious code,” in *Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004.*, vol. 2, IEEE, 2004, pp. 41–42, ISBN: 0-7695-2209-2. DOI: 10.1109/CMPSAC.2004.1342667.

[12] G. Dahl, J. Stokes, L. Deng, and D. Yu, “Large-Scale Malware Classification Using Random Projections and Neural Networks,” in *Proceedings IEEE Conference on Acoustics, Speech, and Signal Processing*, IEEE SPS, May 2013.

[13] S. Robertson and H. Zaragoza, “The Probabilistic Relevance Framework: BM25 and Beyond,” *Foundations and Trends in Information Retrieval*, vol. 3, no. 4, pp. 333–389, Apr. 2009, ISSN: 1554-0669. DOI: 10.1561/1500000019.

[14] S. E. Robertson and S. Walker, “Some Simple Effective Approximations to the 2-Poisson Model for Probabilistic Weighted Retrieval,” in *Proceedings of the 17th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR ’94, New York, NY, USA: Springer-Verlag New York, Inc., 1994, pp. 232–241, ISBN: 0-387-19889-X.

[15] J. O. Kephart, G. B. Sorkin, W. C. Arnold, D. M. Chess, G. J. Tesauro, and S. R. White, “Biologically Inspired Defenses Against Computer Viruses,” in *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 1*, ser. IJCAI’95, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995, pp. 985–996, ISBN: 1-55860-363-8, 978-1-558-60363-9.

[16] J. Z. Kolter and M. A. Maloof, “Learning to Detect and Classify Malicious Executables in the Wild,” *Journal of Machine Learning Research*, vol. 7, pp. 2721–2744, Dec. 2006, ISSN: 1532-4435.

[17] J. Z. Kolter and M. A. Maloof, “Learning to detect malicious executables in the wild,” in *Proceedings of the 2004 ACM SIGKDD international conference on Knowledge discovery and data mining - KDD ’04*, New York, New York, USA: ACM Press, 2004, pp. 470–478, ISBN: 1581138889. DOI: 10.1145/1014052.1014105.

[18] Y. Freund and R. Schapire, “Experiments with a new boosting algorithm,” in *Proceedings of the Thirteenth International Conference on Machine Learning (ICML 1996)*, L. Saitta, Ed., Morgan Kaufmann, 1996, pp. 148–156.

[19] J. R. Quinlan, *C4.5: Programs for Machine Learning*, M. Kaufmann, Ed., ser. Morgan Kaufmann series in {M}achine {L}earning 3. Morgan Kaufmann, 1993, vol. 1, p. 302, ISBN: 1558602380.

[20] S. Jain and Y. K. Meena, "Computer Networks and Intelligent Computing: 5th International Conference on Information Processing, ICIP 2011, Bangalore, India, August 5-7, 2011. Proceedings," in *Computer Networks and Intelligent Computing*, K. R. Venugopal and L. M. Patnaik, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, ch. Byte Level, pp. 51–59, ISBN: 978-3-642-22786-8. DOI: 10.1007/978-3-642-22786-8{_}6.

[21] Y. Elovici, A. Shabtai, R. Moskovitch, G. Tahan, and C. Glezer, "Applying Machine Learning Techniques for Detection of Malicious Code in Network Traffic," in *Proceedings of the 30th Annual German Conference on Advances in Artificial Intelligence*, ser. KI '07, Berlin, Heidelberg: Springer-Verlag, 2007, pp. 44–50, ISBN: 978-3-540-74564-8. DOI: 10.1007/978-3-540-74565-5{_}5.

[22] E. Menahem, A. Shabtai, L. Rokach, and Y. Elovici, "Improving Malware Detection by Applying Multi-inducer Ensemble," *Comput. Stat. Data Anal.*, vol. 53, no. 4, pp. 1483–1494, Feb. 2009, ISSN: 0167-9473. DOI: 10.1016/j.csda.2008.10.015.

[23] M. M. Masud, L. Khan, and B. Thuraisingham, "A scalable multi-level feature extraction technique to detect malicious executables," *Information Systems Frontiers*, vol. 10, no. 1, pp. 33–45, Mar. 2008, ISSN: 1387-3326. DOI: 10.1007/s10796-007-9054-3.

[24] R. Moskovitch, D. Stopel, C. Feher, N. Nissim, N. Japkowicz, and Y. Elovici, "Unknown malcode detection and the imbalance problem," *Journal in Computer Virology*, vol. 5, no. 4, pp. 295–308, Nov. 2009, ISSN: 1772-9890. DOI: 10.1007/s11416-009-0122-8.

[25] M. M. Masud, T. M. Al-Khateeb, K. W. Hamlen, J. Gao, L. Khan, J. Han, and B. Thuraisingham, "Cloud-based malware detection for evolving data streams," *ACM Transactions on Management Information Systems*, vol. 2, no. 3, pp. 1–27, Oct. 2011, ISSN: 2158656X. DOI: 10.1145/2019618.2019622.

[26] R. Perdisci, A. Lanzi, and W. Lee, "McBoost: Boosting Scalability in Malware Collection and Analysis Using Statistical Classification of Executables," in *2008 Annual Computer Security Applications Conference (ACSAC)*, IEEE, Dec. 2008, pp. 301–310, ISBN: 978-0-7695-3447-3. DOI: 10.1109/ACSAC.2008.22.

[27] G. Tahan, L. Rokach, and Y. Shahar, "Mal-ID: Automatic Malware Detection Using Common Segment Analysis and Meta-features," *Journal of Machine Learning Research*, vol. 13, pp. 949–979, Apr. 2012, ISSN: 1532-4435.

[28] B. Zhang, J. Yin, J. Hao, D. Zhang, and S. Wang, "Malicious Codes Detection Based on Ensemble Learning," in *Proceedings of the 4th International Conference on Autonomic and Trusted Computing*, ser. ATC'07, Berlin, Heidelberg: Springer-Verlag, 2007, pp. 468–477, ISBN: 3-540-73546-1, 978-3-540-73546-5.

[29] S. J. Stolfo, K. Wang, and W.-J. Li, "Towards Stealthy Malware Detection," in *Malware Detection*, M. Christodorescu, S. Jha, D. Maughan, D. Song, and C. Wang, Eds., Boston, MA: Springer US, 2007, pp. 231–249, ISBN: 978-0-387-44599-1. DOI: 10.1007/978-0-387-44599-1{_}11.

[30] J. Kornblum, "Identifying almost identical files using context triggered piecewise hashing," *Digital Investigation*, vol. 3, pp. 91–97, 2006, ISSN: 17422876. DOI: 10.1016/j.diin.2006.06.015.

[31] V. Roussev, "Data Fingerprinting with Similarity Digests," in *Advances in Digital Forensics VI: Sixth IFIP WG 11.9 International Conference on Digital Forensics, Hong Kong, China, January 4-6, 2010, Revised Selected Papers*, K.-P. Chow and S. Shenoi, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 207–226, ISBN: 978-3-642-15506-2. DOI: 10.1007/978-3-642-15506-2{_}15.

[32] F. Breitinger, K. P. Astebol, H. Baier, and C. Busch, "mvHash-B - A New Approach for Similarity Preserving Hashing," in *Proceedings of the 2013 Seventh International Conference on IT Security Incident Management and IT Forensics*, ser. IMF '13, Washington, DC, USA: IEEE Computer Society, 2013, pp. 33–44, ISBN: 978-0-7695-4955-2. DOI: 10.1109/IMF.2013.18.

[33] F. Breitinger, G. Stivaktakis, and H. Baier, "FRASH: A framework to test algorithms of similarity hashing," *Digital Investigation*, vol. 10, S50–S58, Aug. 2013, ISSN: 17422876. DOI: 10.1016/j.diin.2013.06.006.

[34] Y. Li, S. C. Sundaramurthy, A. G. Bardas, X. Ou, D. Caragea, X. Hu, and J. Jang, "Experimental Study of Fuzzy Hashing in Malware Clustering Analysis," in *8th Workshop on Cyber Security Experimentation and Test (CSET 15)*, Washington, D.C.: USENIX Association, 2015.

[35] M. Li, X. Chen, X. Li, B. Ma, and P. M. Vitanyi, "The Similarity Metric," *IEEE Transactions on Information Theory*, vol. 50, no. 12, pp. 3250–3264, 2004, ISSN: 0018-9448. DOI: 10.1109/TIT.2004.838101.

[36] N. Alshahwan, E. T. Barr, D. Clark, and G. Danezis, "Detecting Malware with Information Complexity," Feb. 2015.

[37] R. S. Borbely, "On normalized compression distance and large malware," *Journal of Computer Virology and Hacking Techniques*, pp. 1–8, 2015, ISSN: 2263-8733. DOI: 10.1007/s11416-015-0260-0.

[38] M. Hayes, A. Walenstein, and A. Lakhotia, "Evaluation of malware phylogeny modelling systems using automated variant generation," *Journal in Computer Virology*, vol. 5, no. 4, pp. 335–343, 2008, ISSN: 1772-9904. DOI: 10.1007/s11416-008-0100-6.

[39] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario, "Automated Classification and Analysis of Internet Malware," in *Proceedings of the 10th International Conference on Recent Advances in Intrusion Detection*, ser. RAID'07, Berlin, Heidelberg: Springer-Verlag, 2007, pp. 178–197, ISBN: 3-540-74319-7, 978-3-540-74319-4.

[40] S. Wehner, "Analyzing Worms and Network Traffic Using Compression," *Journal of Computer Security*, vol. 15, no. 3, pp. 303–320, Aug. 2007, ISSN: 0926-227X.

[41] E. Raff, R. Zak, R. Cox, J. Sylvester, P. Yacci, R. Ward, A. Tracy, M. McLean, and C. Nicholas, "An investigation of byte n-gram features for malware classification," *Journal of Computer Virology and Hacking Techniques*, Sep. 2016, ISSN: 2263-8733. DOI: 10.1007/s11416-016-0283-1.

[42] R. Zak, E. Raff, and C. Nicholas, "What can N-grams learn for malware detection?" In *2017 12th International Conference on Malicious and Unwanted Software (MALWARE)*, IEEE, Oct. 2017, pp. 109–118, ISBN: 978-1-5386-1436-5. DOI: 10.1109/MALWARE.2017.8323963.

[43] E. Raff and C. Nicholas, "An Alternative to NCD for Large Sequences, Lempel-Ziv Jaccard Distance," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '17*, New York, New York, USA: ACM Press, 2017, pp. 1007–1015, ISBN: 9781450348874. DOI: 10.1145/3097983.3098111.

[44] ——, "Malware Classification and Class Imbalance via Stochastic Hashed LZJD," in *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, ser. AISec '17, New York, NY, USA: ACM, 2017, pp. 111–120, ISBN: 978-1-4503-5202-4. DOI: 10.1145/3128572.3140446.

[45] E. Raff and C. K. Nicholas, "Lempel-Ziv Jaccard Distance, an effective alternative to ssdeep and sdhash," *Digital Investigation*, Feb. 2018, ISSN: 17422876. DOI: 10.1016/j.diin.2017.12.004.

[46] E. Raff and C. Nicholas, "Toward Metric Indexes for Incremental Insertion and Querying," *ArXiv*, 2018.

[47] P. Domingos, "A Few Useful Things to Know About Machine Learning," *Commun. ACM*, vol. 55, no. 10, pp. 78–87, Oct. 2012, ISSN: 0001-0782. DOI: 10.1145/2347736.2347755.

[48] A. Halevy, P. Norvig, and F. Pereira, "The unreasonable effectiveness of data," *Intelligent Systems, IEEE*, vol. 24, no. 2, pp. 8–12, 2009.

[49] J.-M. Roberts, *Virus Share*, 2011. [Online]. Available: https://virusshare.com/.

[50] D. Quist, *Open Malware*, 2009. [Online]. Available: http://openmalware.org/.

[51] P. Baecher, M. Koetter, T. Holz, M. Dornseif, and F. Freiling, "The Nepenthes Platform: An Efficient Approach to Collect Malware," in *Recent Advances in Intrusion Detection: 9th International Symposium, RAID 2006 Hamburg, Germany, September 20-22, 2006 Proceedings*, D. Zamboni and C. Kruegel, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 165–184, ISBN: 978-3-540-39725-0. DOI: 10.1007/11856214{_}9.

[52] J. Zhuge, T. Holz, X. Han, C. Song, and W. Zou, "Collecting Autonomous Spreading Malware Using High-Interaction Honeypots," in *Information and Communications Security: 9th International Conference, ICICS 2007, Zhengzhou, China, December 12-15, 2007. Proceedings*, S. Qing, H. Imai, and G. Wang, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 438–451, ISBN: 978-3-540-77048-0. DOI: 10.1007/978-3-540-77048-0{_}34.

[53] N. Krawetz, "Anti-honeypot technology," *IEEE Security & Privacy Magazine*, vol. 2, no. 1, pp. 76–79, Jan. 2004, ISSN: 1540-7993. DOI: 10.1109/MSECP.2004.1264861.

[54] E. Raff, J. Sylvester, and C. Nicholas, "Learning the PE Header, Malware Classification with Minimal Domain Knowledge," *Journal of Computer Virology and Hacking Techniques*, 2016.

[55] A. Mohaisen and O. Alrawi, "Unveiling Zeus: Automated Classification of Malware Samples," in *Proceedings of the 22Nd International Conference on World Wide Web*, ser. WWW '13 Companion, New York, NY, USA: ACM, 2013, pp. 829–832, ISBN: 978-1-4503-2038-2. DOI: 10.1145/2487788.2488056.

[56] K. Berlin, D. Slater, and J. Saxe, "Malicious Behavior Detection Using Windows Audit Logs," in *Proceedings of the 8th ACM Workshop on Artificial Intelligence and Security*, ser. AISec '15, New York, NY, USA: ACM, 2015, pp. 35–44, ISBN: 978-1-4503-3826-4. DOI: 10.1145/2808769.2808773.

[57] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario, "Automated Classification and Analysis of Internet Malware," *Analysis*, vol. 4637, no. 1, pp. 178–197, 2007, ISSN: 03029743.

[58] A. Kantchelian, M. C. Tschantz, S. Afroz, B. Miller, V. Shankar, R. Bachwani, A. D. Joseph, and J. D. Tygar, "Better Malware Ground Truth: Techniques for Weighting Anti-Virus Vendor Labels," in *Proceedings of the 8th ACM Workshop on Artificial Intelligence and Security*, ser. AISec '15, New York, NY, USA: ACM, 2015, pp. 45–56, ISBN: 978-1-4503-3826-4. DOI: 10.1145/2808769.2808780.

[59] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero, "AVclass: A Tool for Massive Malware Labeling," in *Research in Attacks, Intrusions, and Defenses: 19th International Symposium, RAID 2016, Paris, France, September 19-21, 2016, Proceedings*, F. Monrose, M. Dacier, G. Blanc, and J. Garcia-Alfaro, Eds., Cham: Springer International Publishing, 2016, pp. 230–253, ISBN: 978-3-319-45719-2. DOI: 10.1007/978-3-319-45719-2{_}11.

[60] A. Kantchelian, S. Afroz, L. Huang, A. C. Islam, B. Miller, M. C. Tschantz, R. Greenstadt, A. D. Joseph, and J. D. Tygar, "Approaches to Adversarial Drift," in *Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security*, ser. AISec '13, New York, NY, USA: ACM, 2013, pp. 99–110, ISBN: 978-1-4503-2488-5. DOI: 10.1145/2517312.2517320.

[61] A. Singh, A. Walenstein, and A. Lakhotia, "Tracking Concept Drift in Malware Families," in *Proceedings of the 5th ACM Workshop on Security and Artificial Intelligence*, ser. AISec '12, New York, NY, USA: ACM, 2012, pp. 81–92, ISBN: 978-1-4503-1664-4. DOI: 10.1145/2381896.2381910.

[62] O. Henchiri and N. Japkowicz, "A Feature Selection and Evaluation Scheme for Computer Virus Detection," in *Proceedings of the Sixth International Conference on Data Mining*, ser. ICDM '06, Washington, DC, USA: IEEE Computer Society, 2006, pp. 891–895, ISBN: 0-7695-2701-9. DOI: 10.1109/ICDM.2006.4.

[63] I. Santos, Y. K. Penya, J. Devesa, and P. G. Bringas, "N-grams-based File Signatures for Malware Detection.," in *Proceedings of the 11th International Conference on Enterprise Information Systems*, 2009, pp. 317–320.

[64] *Microsoft Malware Classification Challenge (BIG 2015)*, 2015. [Online]. Available: https://www.kaggle.com/c/malware-classification/.

[65] D. Arp, M. Spreitzenbarth, H. Malte, H. Gascon, and K. Rieck, "Drebin: Effective and Explainable Detection of Android Malware in Your Pocket," *Symposium on Network and Distributed System Security (NDSS)*, no. February, pp. 23–26, 2014. DOI: 10.14722/ndss.2014.23247.

[66] A. Shabtai, R. Moskovitch, Y. Elovici, and C. Glezer, "Detection of malicious code by applying machine learning classifiers on static features: A state-of-the-art survey," *Information Security Technical Report*, vol. 14, no. 1, pp. 16–29, 2009, ISSN: 1363-4127. DOI: http://dx.doi.org/10.1016/j.istr.2009.03.003.

[67] "Microsoft Portable Executable and Common Object File Format Specification Version 8.3," Microsoft, Tech. Rep., 2013, p. 98.

[68] M. Z. Shafiq, S. M. Tabish, F. Mirza, and M. Farooq, "PE-Miner: Mining Structural Information to Detect Malicious Executables in Realtime," in *Recent Advances in Intrusion Detection*, 2009, pp. 121–141, ISBN: 978-3-642-04342-0. DOI: 10.1007/978-3-642-04342-0{_}7.

[69] M. Banko and E. Brill, "Scaling to Very Very Large Corpora for Natural Language Disambiguation," in *Proceedings of the 39th Annual Meeting on Association for Computational Linguistics*, ser. ACL '01, Stroudsburg, PA, USA: Association for Computational Linguistics, 2001, pp. 26–33. DOI: 10.3115/1073012.1073017.

[70] R. Tibshirani, "Regression Shrinkage and Selection Via the Lasso," *Journal of the Royal Statistical Society, Series B*, vol. 58, no. 1, pp. 267–288, 1994.

[71] H. Zou and T. Hastie, "Regularization and variable selection via the elastic net," *Journal of the Royal Statistical Society, Series B*, vol. 67, no. 2, pp. 301–320, Apr. 2005, ISSN: 1369-7412. DOI: 10.1111/j.1467-9868.2005.00503.x.

[72] A. Y. Ng, "Feature selection, L1 vs. L2 regularization, and rotational invariance," *Twenty-first international conference on Machine learning - ICML '04*, p. 78, 2004. DOI: 10.1145/1015330.1015435.

[73] G.-X. Yuan, K.-W. Chang, C.-J. Hsieh, and C.-J. Lin, "A Comparison of Optimization Methods and Software for Large-scale L1-regularized Linear Classification," *The Journal of Machine Learning Research*, vol. 11, pp. 3183–3234, 2010.

[74] J. Friedman, T. Hastie, and R. Tibshirani, "Regularization Paths for Generalized Linear Models via Coordinate Descent," *Journal of Statistical Software*, vol. 33, no. 1, pp. 1–22, 2010, ISSN: 19390068.

[75] G.-x. Yuan, C.-H. Ho, and C.-j. Lin, "An improved GLMNET for L1-regularized logistic regression," *Journal of Machine Learning Research*, vol. 13, S. S. Keerthi, Ed., pp. 1999–2030, 2012. DOI: 10.1145/2020408.2020421.

[76] P. Gong and J. Ye, "A Modified Orthant-Wise Limited Memory Quasi-Newton Method with Convergence Analysis," in *The 32nd International Conference on Machine Learning*, vol. 37, 2015.

[77] J. Seymour, "How to build a malware classifier [that doesn't suck on real-world data]," in *SecTor*, Toronto, Ontario, 2016.

[78] C. Team, *Exploit writing tutorial part 11 : Heap Spraying Demystified*, 2011. [Online]. Available: https://www.corelan.be/index.php/2011/12/31/exploit-writing-tutorial-part-11-heap-spraying-demystified/.

[79] K. Griffin, S. Schneider, X. Hu, and T.-c. Chiueh, "Automatic Generation of String Signatures for Malware Detection," in, 2009, pp. 101–120. DOI: 10.1007/978-3-642-04342-0{_}6.

[80] A. H. Ibrahim, M. B. Abdelhalim, H. Hussein, and A. Fahmy, "Analysis of x86 instruction set usage for Windows 7 applications," in *Computer Technology and Development (ICCTD), 2010 2nd International Conference on*, Nov. 2010, pp. 511–516. DOI: 10.1109/ICCTD.2010.5645851.

[81] R. Bellman, *Dynamic Programming*. Princeton, NJ: Princeton University Press, 1957.

[82] C. C. Aggarwal, A. Hinneburg, and D. A. Keim, "On the Surprising Behavior of Distance Metrics in High Dimensional Spaces," in *Proceedings of the 8th International Conference on Database Theory*, ser. ICDT '01, London, UK, UK: Springer-Verlag, 2001, pp. 420–434, ISBN: 3-540-41456-8.

[83] M. Verleysen and D. François, "The Curse of Dimensionality in Data Mining and Time Series Prediction," in *Proceedings of the 8th International Conference on Artificial Neural Networks: Computational Intelligence and Bioinspired Systems*, ser. IWANN'05, Berlin, Heidelberg: Springer-Verlag, 2005, pp. 758–770, ISBN: 3-540-26208-3, 978-3-540-26208-4. DOI: 10.1007/11494669{_}93.

[84] A. Shabtai, R. Moskovitch, C. Feher, S. Dolev, and Y. Elovici, "Detecting unknown malicious code by applying classification techniques on OpCode patterns," *Security Informatics*, vol. 1, no. 1, pp. 1–22, 2012, ISSN: 2190-8532. DOI: 10.1186/2190-8532-1-1.

[85] N. Runwal, R. M. Low, and M. Stamp, "Opcode Graph Similarity and Metamorphic Detection," *J. Comput. Virol.*, vol. 8, no. 1-2, pp. 37–52, May 2012, ISSN: 1772-9890. DOI: 10.1007/s11416-012-0160-5.

[86] A. Damodaran, F. D. Troia, C. A. Visaggio, T. H. Austin, and M. Stamp, "A comparison of static, dynamic, and hybrid analysis for malware detection," *Journal of Computer Virology and Hacking Techniques*, pp. 1–12, 2015, ISSN: 2263-8733. DOI: 10.1007/s11416-015-0261-z.

[87] G. Yan, N. Brown, and D. Kong, "Exploring Discriminatory Features for Automated Malware Classification," in *Proceedings of the 10th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. DIMVA'13, Berlin, Heidelberg: Springer-Verlag, 2013, pp. 41–61, ISBN: 978-3-642-39234-4. DOI: 10.1007/978-3-642-39235-1{_}3.

[88] W. Mazurczyk and L. Caviglione, "Information Hiding as a Challenge for Malware Detection," *IEEE Security & Privacy*, vol. 13, no. 2, pp. 89–93, Mar. 2015, ISSN: 1540-7993. DOI: 10.1109/MSP.2015.33.

[89] N. Karampatziakis, J. W. Stokes, A. Thomas, and M. Marinescu, "Using File Relationships in Malware Classification," in *Proceedings of Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, Springer, Sep. 2012.

[90] R. Rosenthal, "The file drawer problem and tolerance for null results.," *Psychological Bulletin*, vol. 86, no. 3, pp. 638–641, 1979, ISSN: 0033-2909. DOI: 10.1037/0033-2909.86.3.638.

[91] D. L. Sackett, "Bias in analytic research," *Journal of Chronic Diseases*, vol. 32, no. 1-2, pp. 51–63, 1979, ISSN: 0021-9681. DOI: 10.1016/0021-9681(79)90012-2.

[92] X. Hu, K. G. Shin, S. Bhatkar, and K. Griffin, "MutantX-S: Scalable Malware Clustering Based on Static Features," in *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, San Jose, CA: USENIX, 2013, pp. 187–198, ISBN: 978-1-931971-01-0.

[93] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "BYTEWEIGHT: Learning to Recognize Functions in Binary Code," in *23rd USENIX Security Symposium (USENIX Security 14)*, San Diego, CA: USENIX Association, 2014, pp. 845–860, ISBN: 978-1-931971-15-7.

[94] I. Santos, F. Brezo, J. Nieves, Y. K. Penya, B. Sanz, C. Laorden, and P. G. Bringas, "Idea: Opcode-sequence-based Malware Detection," in *Proceedings of the Second International Conference on Engineering Secure Software and Systems*, ser. ESSoS'10, Berlin, Heidelberg: Springer-Verlag, 2010, pp. 35–43, ISBN: 3-642-11746-5, 978-3-642-11746-6. DOI: 10.1007/978-3-642-11747-3{_}3.

[95] D. Bilar, "Opcodes As Predictor for Malware," *Int. J. Electron. Secur. Digit. Forensic*, vol. 1, no. 2, pp. 156–168, Jan. 2007, ISSN: 1751-911X. DOI: 10.1504/IJESDF.2007.016865.

[96] M. K. Shankarapani, S. Ramamoorthy, R. S. Movva, and S. Mukkamala, "Malware Detection Using Assembly and API Call Sequences," *J. Comput. Virol.*, vol. 7, no. 2, pp. 107–119, May 2011, ISSN: 1772-9890. DOI: 10.1007/s11416-010-0141-5.

[97] L. Breiman, "Bagging predictors," *Machine Learning*, vol. 24, no. 2, R. Quinlan, Ed., pp. 123–140, 1996, ISSN: 08856125. DOI: 10.1007/BF00058655.

[98] ——, "Arcing Classifiers," *The Annals of Statistics*, vol. 26, no. 3, pp. 801–824, 1998.

[99] D. H. Wolpert, "Stacked generalization," *Neural networks*, vol. 5, pp. 241–259, 1992.

[100] T. Singh, F. Di Troia, V. A. Corrado, T. H. Austin, and M. Stamp, "Support vector machines and malware detection," *Journal of Computer Virology and Hacking Techniques*, pp. 1–10, 2015, ISSN: 2263-8733. DOI: 10.1007/s11416-015-0252-0.

[101] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[102] C.-J. Hsieh, K.-W. Chang, C.-J. Lin, S. S. Keerthi, and S. Sundararajan, "A Dual Coordinate Descent Method for Large-scale Linear SVM," in *Proceedings of the 25th international conference on Machine learning - ICML '08*, New York, New York, USA: ACM Press, 2008, pp. 408–415, ISBN: 9781605582054. DOI: 10.1145/1390156.1390208.

[103] K. H. Brodersen, C. S. Ong, K. E. Stephan, and J. M. Buhmann, "The Balanced Accuracy and Its Posterior Distribution," in *Proceedings of the 2010 20th International Conference on Pattern Recognition*, ser. ICPR '10, Washington, DC, USA: IEEE Computer Society, 2010, pp. 3121–3124, ISBN: 978-0-7695-4109-9. DOI: 10.1109/ICPR.2010.764.

[104] A. P. Bradley, "The use of the area under the ROC curve in the evaluation of machine learning algorithms," *Pattern Recognition*, vol. 30, no. 7, pp. 1145–1159, 1997, ISSN: 00313203. DOI: 10.1016/S0031-3203(96)00142-2.

[105] C. Cortes and M. Mohri, "AUC Optimization vs. Error Rate Minimization," in *Advances in Neural Information Processing Systems 16*, S. Thrun, L. K. Saul, and B. Schölkopf, Eds., MIT Press, 2004, pp. 313–320.

[106] R. Moskovitch, C. Feher, N. Tzachar, E. Berger, M. Gitelman, S. Dolev, and Y. Elovici, “Unknown Malcode Detection Using OPCODE Representation,” in *Proceedings of the 1st European Conference on Intelligence and Security Informatics*, ser. EuroISI ’08, Berlin, Heidelberg: Springer-Verlag, 2008, pp. 204–215, ISBN: 978-3-540-89899-3. DOI: 10.1007/978-3-540-89900-6{_}21.

[107] C. Linn and S. Debray, “Obfuscation of executable code to improve resistance to static disassembly,” in *Proceedings of the 10th ACM conference on Computer and communication security - CCS ’03*, New York, New York, USA: ACM Press, 2003, p. 290, ISBN: 1581137389. DOI: 10.1145/948148.948149.

[108] C. Collberg and C. Thomborson, “Watermarking, tamper-proofing, and obfuscation - tools for software protection,” *IEEE Transactions on Software Engineering*, vol. 28, no. 8, pp. 735–746, Aug. 2002, ISSN: 0098-5589. DOI: 10.1109/TSE.2002.1027797.

[109] C. Collberg, C. Thomborson, and D. Low, “A Taxonomy of Obfuscating Transformations,” The University of Auckland, Auckland, Tech. Rep., 1997.

[110] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, “Static Disassembly of Obfuscated Binaries,” in *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, ser. SSYM’04, Berkeley, CA, USA: USENIX Association, 2004, p. 18.

[111] N. Karampatziakis, “Static Analysis of Binary Executables Using Structural SVMs,” in *Proceedings of the 23rd International Conference on Neural Information Processing Systems*, ser. NIPS’10, USA: Curran Associates Inc., 2010, pp. 1063–1071.

[112] Y. Ye, T. Li, Y. Chen, and Q. Jiang, “Automatic Malware Categorization Using Cluster Ensemble,” in *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD ’10, New York, NY, USA: ACM, 2010, pp. 95–104, ISBN: 978-1-4503-0055-1. DOI: 10.1145/1835804.1835820.

[113] C. Rossow, C. J. Dietrich, C. Grier, C. Kreibich, V. Paxson, N. Pohlmann, H. Bos, and M. v. Steen, “Prudent Practices for Designing Malware Experiments: Status Quo and Outlook,” in *2012 IEEE Symposium on Security and Privacy*, IEEE, May 2012, pp. 65–79, ISBN: 978-1-4673-1244-8. DOI: 10.1109/SP.2012.14.

[114] N. Tran, “The normalized compression distance and image distinguishability,” in *Proc. SPIE 6492, Human Vision and Electronic Imaging XII*, B. E. Rogowitz, T. N. Pappas, and S. J. Daly, Eds., vol. 64921D, Feb. 2007. DOI: 10.1117/12.704334.

[115] E. Keogh, S. Lonardi, C. A. Ratanamahatana, L. Wei, S.-H. Lee, and J. Handley, “Compression-based data mining of sequential data,” *Data Mining and Knowledge Discovery*, vol. 14, no. 1, pp. 99–129, 2007, ISSN: 1573-756X. DOI: 10.1007/s10618-006-0049-3.

[116] M. Cebrián, M. Alfonseca, A. Ortega, *et al.*, "Common pitfalls using the normalized compression distance: What to watch out for in a compressor," *Communications in Information & Systems*, vol. 5, no. 4, pp. 367–384, 2005.

[117] W. Wong and M. Stamp, "Hunting for metamorphic engines," *Journal in Computer Virology*, vol. 2, no. 3, pp. 211–229, 2006, ISSN: 1772-9904. DOI: 10.1007/s11416-006-0028-7.

[118] I. Pavlov, "LZMA SDK (Software Development Kit)," http://www.7-zip.org/sdk.html, Jul. 2007.

[119] M. Cebrin, M. Alfonseca, and A. Ortega, "The Normalized Compression Distance Is Resistant to Noise," *IEEE Transactions on Information Theory*, vol. 53, no. 5, pp. 1895–1900, May 2007, ISSN: 0018-9448. DOI: 10.1109/TIT.2007.894669.

[120] E. Keogh, S. Lonardi, and C. A. Ratanamahatana, "Towards Parameter-free Data Mining," in *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '04, New York, NY, USA: ACM, 2004, pp. 206–215, ISBN: 1-58113-888-1. DOI: 10.1145/1014052.1014077.

[121] D. Sculley and C. E. Brodley, "Compression and Machine Learning: A New Perspective on Feature Space Vectors," in *Proceedings of the Data Compression Conference*, ser. DCC '06, Washington, DC, USA: IEEE Computer Society, 2006, p. 332, ISBN: 0-7695-2545-8. DOI: 10.1109/DCC.2006.13.

[122] T. Graepel, R. Herbrich, and J. Shawe-Taylor, "PAC-Bayesian Compression Bounds on the Prediction Error of Learning Algorithms for Classification," *Machine Learning*, vol. 59, no. 1, pp. 55–76, 2005, ISSN: 1573-0565. DOI: 10.1007/s10994-005-0462-7.

[123] O. David, S. Moran, and A. Yehudayoff, "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., 2016, pp. 2784–2792.

[124] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, May 1977, ISSN: 0018-9448. DOI: 10.1109/TIT.1977.1055714.

[125] ——, "Compression of individual sequences via variable-rate coding," *IEEE Transactions on Information Theory*, vol. 24, no. 5, pp. 530–536, Sep. 1978, ISSN: 0018-9448. DOI: 10.1109/TIT.1978.1055934.

[126] A. Z. Broder, "On the Resemblance and Containment of Documents," in *Proceedings of the Compression and Complexity of Sequences 1997*, ser. SEQUENCES '97, Washington, DC, USA: IEEE Computer Society, 1997, pp. 21–29, ISBN: 0-8186-8132-2.

[127] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher, "Min-wise Independent Permutations (Extended Abstract)," in *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, ser. STOC '98, New York, NY, USA: ACM, 1998, pp. 327–336, ISBN: 0-89791-962-9. DOI: 10.1145/276698.276781.

[128] T. Cover and P. Hart, "Nearest Neighbor Pattern Classification," *IEEE Trans. Inf. Theor.*, vol. 13, no. 1, pp. 21–27, Sep. 2006, ISSN: 0018-9448. DOI: 10.1109/TIT.1967.1053964.

[129] W. Yan, Z. Zhang, and N. Ansari, "Revealing Packed Malware," *IEEE Security and Privacy*, vol. 6, no. 5, pp. 65–69, Sep. 2008, ISSN: 1540-7993. DOI: 10.1109/MSP.2008.126.

[130] O. Patri, M. Wojnowicz, and M. Wolff, "Discovering Malware with Time Series Shapelets," in *Proceedings of the 50th Hawaii International Conference on System Sciences*, 2017.

[131] Z. Markel and M. Bilzor, "Building a machine learning classifier for malware detection," in *2014 Second Workshop on Anti-malware Testing Research (WATeR)*, IEEE, Oct. 2014, pp. 1–4, ISBN: 978-1-4799-6070-5. DOI: 10.1109/WATeR.2014.7015757.

[132] G. E. a. P. a. Batista, a. L. C. Bazzan, and M. C. Monard, "Balancing Training Data for Automated Annotation of Keywords: a Case Study," *Revista Tecnologia da Informação*, vol. 3, no. 2, pp. 15–20, 2004.

[133] G. Lemaître, F. Nogueira, and C. K. Aridas, "Imbalanced-learn: A Python Toolbox to Tackle the Curse of Imbalanced Datasets in Machine Learning," *Journal of Machine Learning Research*, vol. 18, no. 17, pp. 1–5, 2017.

[134] N. Chawla, K. Bowyer, L. Hall, and P. Kegelmeyer, "SMOTE: synthetic minority over-sampling technique," *Artificial Intelligence Research*, vol. 16, pp. 321–357, 2002.

[135] H. Han, W.-Y. Wang, and B.-H. Mao, "Borderline-SMOTE: A New Over-sampling Method in Imbalanced Data Sets Learning," in *Proceedings of the 2005 International Conference on Advances in Intelligent Computing - Volume Part I*, ser. ICIC'05, Berlin, Heidelberg: Springer-Verlag, 2005, pp. 878–887, ISBN: 3-540-28226-2, 978-3-540-28226-6. DOI: 10.1007/11538059{_}91.

[136] Y. Ye, T. Li, K. Huang, Q. Jiang, and Y. Chen, "Hierarchical Associative Classifier (HAC) for Malware Detection from the Large and Imbalanced Gray List," *Journal of Intelligent Information Systems*, vol. 35, no. 1, pp. 1–20, Aug. 2010, ISSN: 0925-9902. DOI: 10.1007/s10844-009-0086-7.

[137] R. Moskovitch, N. Nissim, and Y. Elovici, "Malicious Code Detection Using Active Learning," in *Privacy, Security, and Trust in KDD*, 2009, pp. 74–91. DOI: 10.1007/978-3-642-01718-6{_}6.

[138] M. Manasse, F. McSherry, and K. Talwar, "Consistent Weighted Sampling," Tech. Rep., Jun. 2008.

[139] S. Ioffe, "Improved Consistent Sampling, Weighted Minhash and L1 Sketching," in *Proceedings of the 2010 IEEE International Conference on Data Mining*, ser. ICDM '10, Washington, DC, USA: IEEE Computer Society, Dec. 2010, pp. 246–255, ISBN: 978-0-7695-4256-0. DOI: 10.1109/ICDM.2010.80.

[140] W. Wu, B. Li, L. Chen, and C. Zhang, "Consistent Weighted Sampling Made More Practical," in *Proceedings of the 26th International Conference on World Wide Web*, ser. WWW '17, Republic and Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee, 2017, pp. 1035–1043, ISBN: 978-1-4503-4913-0. DOI: 10.1145/3038912.3052598.

[141] P. Li, "0-Bit Consistent Weighted Sampling," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '15, New York, NY, USA: ACM, 2015, pp. 665–674, ISBN: 978-1-4503-3664-2. DOI: 10.1145/2783258.2783406.

[142] P. Li, A. Shrivastava, J. L. Moore, and A. C. König, "Hashing Algorithms for Large-Scale Learning," in *Advances in Neural Information Processing Systems 24*, J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, Eds., Curran Associates, Inc., 2011, pp. 2672–2680.

[143] K. Weinberger, A. Dasgupta, J. Langford, A. Smola, and J. Attenberg, "Feature hashing for large scale multitask learning," in *Proceedings of the 26th Annual International Conference on Machine Learning - ICML '09*, New York, New York, USA: ACM Press, 2009, pp. 1113–1120, ISBN: 9781605585161. DOI: 10.1145/1553374.1553516.

[144] E. Raff, "JSAT: Java Statistical Analysis Tool, a Library for Machine Learning," *Journal of Machine Learning Research*, vol. 18, no. 23, pp. 1–5, 2017.

[145] V. Harichandran, F. Breitinger, and I. Baggili, "Bytewise Approximate Matching: The Good, The Bad, and The Unknown," *Journal of Digital Forensics, Security and Law*, vol. 11, no. 2, pp. 59–78, 2016, ISSN: 15587223. DOI: 10.15394/jdfsl.2016.1379.

[146] V. Roussev and C. Quates, "Content triage with similarity digests: The M57 case study," *Digital Investigation*, vol. 9, S60–S68, 2012, ISSN: 17422876. DOI: 10.1016/j.diin.2012.05.012.

[147] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, "A Large-scale Analysis of the Security of Embedded Firmwares," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC'14, Berkeley, CA, USA: USENIX Association, 2014, pp. 95–110, ISBN: 978-1-931971-15-7.

[148] J. Jang, D. Brumley, and S. Venkataraman, "BitShred: Feature Hashing Malware for Scalable Triage and Semantic Analysis," in *Proceedings of the 18th ACM conference on Computer and communications security - CCS*, New York, New York, USA: ACM Press, 2011, pp. 309–320, ISBN: 9781450309486. DOI: 10.1145/2046707.2046742.

[149] A. Lakhotia, A. Walenstein, C. Miles, and A. Singh, "VILO: A Rapid Learning Nearest-neighbor Classifier for Malware Triage," *Journal in Computer Virology*, vol. 9, no. 3, pp. 109–123, Aug. 2013, ISSN: 1772-9890. DOI: 10.1007/s11416-013-0178-3.

[150] F. Breitinger and H. Baier, "Similarity Preserving Hashing: Eligible Properties and a New Algorithm MRSH-v2," in *Digital Forensics and Cyber Crime*, 2013, pp. 167–182. DOI: 10.1007/978-3-642-39891-9{_}11.

[151] C. Winter, M. Schneider, and Y. Yannikos, "F2S2: Fast forensic similarity search through indexing piecewise hash signatures," *Digital Investigation*, vol. 10, no. 4, pp. 361–371, Dec. 2013, ISSN: 17422876. DOI: 10.1016/j.diin.2013.08.003.

[152] V. Roussev, "An evaluation of forensic similarity hashes," *Digital Investigation*, vol. 8, S34–S41, 2011, ISSN: 17422876. DOI: 10.1016/j.diin.2011.05.005.

[153] N. Beebe, "Digital Forensic Research: The Good, the Bad and the Unaddressed," in *Advances in Digital Forensics V*, Springer, Berlin, Heidelberg, 2009, pp. 17–36, ISBN: 978-3-642-04154-9, 978-3-642-04155-6. DOI: 10.1007/978-3-642-04155-6{_}2.

[154] F. Breitinger, H. Baier, and J. Beckingham, "Security and implementation analysis of the similarity digest sdhash," in *First International Baltic Conference on Network Security & Forensics (NeSeFo)*, 2012.

[155] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals," in *Soviet physics doklady*, vol. 10, 1966, p. 707.

[156] D. E. Knuth, *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998, p. 780, ISBN: 0-201-89685-0.

[157] D. Dor and U. Zwick, "Median Selection Requires $(2+\epsilon)$N Comparisons," *SIAM J. Discret. Math.*, vol. 14, no. 3, pp. 312–325, Mar. 2001, ISSN: 0895-4801. DOI: 10.1137/S0895480199353895.

[158] E. Konstantinou, "Metamorphic Virus: Analysis and Detection," Royal Holloway University of London, Tech. Rep., 2008.

[159] V. Roussev, "Building a Better Similarity Trap with Statistically Improbable Features," in *Proceedings of the 42Nd Hawaii International Conference on System Sciences*, ser. HICSS '09, Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–10, ISBN: 978-0-7695-3450-3. DOI: 10.1109/HICSS.2009.97.

[160] C. Winter, M. Schneider, and Y. Yannikos, "F2S2: Fast forensic similarity search through indexing piecewise hash signatures," *Digital Investigation*, vol. 10, no. 4, pp. 361–371, Dec. 2013, ISSN: 17422876. DOI: 10.1016/j.diin.2013.08.003.

[161] F. Breitinger, H. Baier, and D. White, “On the database lookup problem of approximate matching,” *Digital Investigation*, vol. 11, S1–S9, May 2014, ISSN: 17422876. DOI: 10.1016/j.diin.2014.03.001.

[162] F. Breitinger, C. Rathgeb, and H. Baier, “An Efficient Similarity Digests Database Lookup - A Logarithmic Divide & Conquer Approach,” *The Journal of Digital Forensics, Security and Law (JDFSL)*, vol. 9, no. 2, pp. 155–166, 2014.

[163] D. Lillis, F. Breitinger, and M. Scanlon, “Expediting MRSH-v2 Approximate Matching with Hierarchical Bloom Filter Trees,” in *9th EAI International Conference on Digital Forensics and Cyber Crime (ICDF2C 2017)*, Prague, Czechia: Springer, 2017.

[164] P. Yianilos, “Data structures and algorithms for nearest neighbor search in general metric spaces,” in *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, Society for Industrial and Applied Mathematics, 1993, pp. 311–321.

[165] J. K. Uhlmann, “Satisfying general proximity / similarity queries with metric trees,” *Information Processing Letters*, vol. 40, no. 4, pp. 175–179, Nov. 1991, ISSN: 00200190. DOI: 10.1016/0020-0190(91)90074-R.

[166] A. Beygelzimer, S. Kakade, and J. Langford, “Cover trees for nearest neighbor,” in *International Conference on Machine Learning*, New York: ACM, 2006, pp. 97–104.

[167] M. Izbicki and C. R. Shelton, “Faster Cover Trees,” in *Proceedings of the Thirty-Second International Conference on Machine Learning*, vol. 37, 2015.

[168] H. Baier and F. Breitinger, “Security Aspects of Piecewise Hashing in Computer Forensics,” in *2011 Sixth International Conference on IT Security Incident Management and IT Forensics*, IEEE, May 2011, pp. 21–36, ISBN: 978-1-4577-0146-7. DOI: 10.1109/IMF.2011.16.

[169] S. P. Lloyd, “Least squares quantization in PCM,” *IEEE Transactions on Information Theory*, vol. 28, no. 2, pp. 129–137, Mar. 1982, ISSN: 0018-9448. DOI: 10.1109/TIT.1982.1056489.

[170] T. Kanungo, D. Mount, N. Netanyahu, C. Piatko, R. Silverman, and A. Wu, “An efficient k-means clustering algorithm: analysis and implementation,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 24, no. 7, pp. 881–892, 2002, ISSN: 0162-8828. DOI: 10.1109/TPAMI.2002.1017616.

[171] E. Biçici and D. Yuret, “Locally scaled density based clustering,” in *Adaptive and Natural Computing Algorithms*, B. Beliczynski, A. Dzielinski, M. Iwanowski, and B. Ribeiro, Eds., Warsaw, Poland: Springer-Verlag, 2007, pp. 739–748.

[172] R. J. G. B. Campello, D. Moulavi, and J. Sander, "Density-Based Clustering Based on Hierarchical Density Estimates," in *Advances in Knowledge Discovery and Data Mining*, J. Pei, V. Tseng, L. Cao, H. Motoda, and G. Xu, Eds., Springer Berlin Heidelberg, 2013, pp. 160–172, ISBN: 978-3-642-37455-5. DOI: 10.1007/978-3-642-37456-2{_}14.

[173] L. van der Maaten, "Accelerating t-SNE using Tree-Based Algorithms," *Journal of Machine Learning Research*, vol. 15, pp. 3221–3245, 2014.

[174] L. V. D. Maaten and G. Hinton, "Visualizing Data using t-SNE," *Journal of Machine Learning Research*, vol. 9, pp. 2579–2605, 2008.

[175] J. Tang, J. Liu, M. Zhang, and Q. Mei, "Visualizing Large-scale and High-dimensional Data," in *Proceedings of the 25th International Conference on World Wide Web*, ser. WWW '16, Republic and Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee, 2016, pp. 287–297, ISBN: 978-1-4503-4143-1. DOI: 10.1145/2872427.2883041.

[176] K. Narayan, A. Punjani, and P. Abbeel, "Alpha-Beta Divergences Discover Micro and Macro Structures in Data," in *Proceedings of The 32nd International Conference on Machine Learning*, 2015, pp. 796–804.

[177] R. Gove, J. Saxe, S. Gold, A. Long, and G. Bergamo, "SEEM: A Scalable Visualization for Comparing Multiple Large Sets of Attributes for Malware Analysis," in *Proceedings of the Eleventh Workshop on Visualization for Cyber Security*, ser. VizSec '14, New York, NY, USA: ACM, 2014, pp. 72–79, ISBN: 978-1-4503-2826-5. DOI: 10.1145/2671491.2671496.

[178] A. Walenstein, M. Venable, M. Hayes, C. Thompson, and A. Lakhotia, "Exploiting similarity between variants to defeat malware," in *Proc. BlackHat DC Conf*, 2007.

[179] B. Li, K. Roundy, C. Gates, and Y. Vorobeychik, "Large-Scale Identification of Malicious Singleton Files," in *7TH ACM Conference on Data and Application Security and Privacy*, 2017.

[180] E. C. Spafford, "Is Anti-virus Really Dead?" *Computers & Security*, vol. 44, p. iv, 2014, ISSN: 0167-4048. DOI: http://dx.doi.org/10.1016/S0167-4048(14)00082-0.

[181] R. A. Finkel and J. L. Bentley, "Quad Trees a Data Structure for Retrieval on Composite Keys," *Acta Informatica*, vol. 4, no. 1, pp. 1–9, Mar. 1974, ISSN: 0001-5903. DOI: 10.1007/BF00288933.

[182] J. L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, Sep. 1975, ISSN: 0001-0782. DOI: 10.1145/361002.361007.

[183] A. Guttman, "R-trees: a dynamic index structure for spatial searching," in *1984 ACM SIGMOD international conference on Management of data*, New York, NY: ACM, 1984, pp. 47–57, ISBN: 0897911288.

[184] I. Kamel and C. Faloutsos, "Hilbert R-tree: An Improved R-tree Using Fractals," in *Proceedings of the 20th International Conference on Very Large Data Bases*, ser. VLDB '94, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994, pp. 500–509, ISBN: 1-55860-153-8.

[185] S. M. Omohundro, "Five Balltree Construction Algorithms," International Computer Science Institute, Tech. Rep., 1989.

[186] K. Fukunage and P. Narendra, "A Branch and Bound Algorithm for Computing k-Nearest Neighbors," *IEEE Transactions on Computers*, vol. C-24, no. 7, pp. 750–753, 1975, ISSN: 0018-9340. DOI: 10.1109/T-C.1975.224297.

[187] J. K. Uhlmann, "Implementing Metric Trees to Satisfy General Proximity / Similarity Queries," Naval Research Laboratory, Washington, D.C., Tech. Rep., 1991.

[188] D. R. Karger and M. Ruhl, "Finding Nearest Neighbors in Growth-restricted Metrics," in *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing*, ser. STOC '02, New York, NY, USA: ACM, 2002, pp. 741–750, ISBN: 1-58113-495-9. DOI: 10.1145/509907.510013.

[189] L. Cayton, "Accelerating Nearest Neighbor Search on Manycore Systems," *IEEE 26th International Parallel and Distributed Processing Symposium*, pp. 402–413, May 2012. DOI: 10.1109/IPDPS.2012.45.

[190] S. Li and N. Amenta, "Brute-Force k-Nearest Neighbors Search on the GPU," in *Proceedings of the 8th International Conference on Similarity Search and Applications - Volume 9371*, ser. SISAP 2015, New York, NY, USA: Springer-Verlag New York, Inc., 2015, pp. 259–270, ISBN: 978-3-319-25086-1. DOI: 10.1007/978-3-319-25087-8{_}25.

[191] J. Kim, S.-G. Kim, and B. Nam, "Parallel multi-dimensional range query processing with R-trees on GPU," *Journal of Parallel and Distributed Computing*, vol. 73, no. 8, pp. 1195–1207, 2013, ISSN: 07437315. DOI: 10.1016/j.jpdc.2013.03.015.

[192] F. Gieseke, J. Heinermann, C. Oancea, and C. Igel, "Buffer K-d Trees: Processing Massive Nearest Neighbor Queries on GPUs," in *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ser. ICML'14, JMLR.org, 2014, pp. I-172–I-180.

[193] K. Li and J. Malik, "Fast k-Nearest Neighbour Search via Dynamic Continuous Indexing," in *Proceedings of The 33rd International Conference on Machine Learning*, 2016, pp. 671–679.

[194] L. Breiman, J. Friedman, C. J. Stone, and R. Olshen, *Classification and Regression Trees*. CRC press, 1984.

[195] B. P. Welford, "Note on a Method for Calculating Corrected Sums of Squares and Products," *Technometrics*, vol. 4, no. 3, p. 419, Aug. 1962, ISSN: 00401706. DOI: 10.2307/1266577.

[196] T. F. Chan, G. H. Golub, and R. J. LeVeque, "Algorithms for Computing the Sample Variance: Analysis and Recommendations," *The American Statistician*, vol. 37, no. 3, p. 242, Aug. 1983, ISSN: 00031305. DOI: 10.2307/2683386.

[197] D. Tarlow, K. Swersky, L. Charlin, I. Sutskever, and R. Zemel, "Stochastic k-Neighborhood Selection for Supervised and Unsupervised Learning," in *Proceedings of the 30th International Conference on Machine Learning*, S. Dasgupta and D. McAllester, Eds., ser. Proceedings of Machine Learning Research, vol. 28, Atlanta, Georgia, USA: PMLR, 2013, pp. 199–207.

[198] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998, ISSN: 0018-9219. DOI: 10.1109/5.726791.

[199] G. Loosli, S. Canu, and L. Bottou, "Training Invariant Support Vector Machines using Selective Sampling," in *Large Scale Kernel Machines*, L. Bottou, O. Chapelle, D. DeCoste, and J. Weston, Eds., Cambridge, MA.: MIT Press, 2007, pp. 301–320.

[200] J. A. Blackard and D. J. Dean, "Comparative accuracies of artificial neural networks and discriminant analysis in predicting forest cover types from cartographic variables," *Computers and Electronics in Agriculture*, vol. 24, no. 3, pp. 131–151, 1999, ISSN: 01681699. DOI: 10.1016/S0168-1699(99)00046-0.

[201] X. Hu, T.-c. Chiueh, and K. G. Shin, "Large-scale Malware Indexing Using Function-call Graphs," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ser. CCS '09, New York, NY, USA: ACM, 2009, pp. 611–620, ISBN: 978-1-60558-894-0. DOI: 10.1145/1653662.1653736.

[202] *VX Heaven*. [Online]. Available: https://vxheaven.org/.

[203] K. L. Clarkson, "Nearest neighbor searching in metric spaces: Experimental Results for sb(S)," Bell Laboratories, Lucent Technologies, New Jersey, Tech. Rep., 2002.

[204] A. Behm, C. Li, and M. J. Carey, "Answering Approximate String Queries on Large Data Sets Using External Memory," in *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ser. ICDE '11, Washington, DC, USA: IEEE Computer Society, 2011, pp. 888–899, ISBN: 978-1-4244-8959-6. DOI: 10.1109/ICDE.2011.5767856.

[205] Y. Rubner, C. Tomasi, and L. J. Guibas, "The Earth Mover's Distance as a Metric for Image Retrieval," *International Journal of Computer Vision*, vol. 40, no. 2, pp. 99–121, 2000, ISSN: 09205691. DOI: 10.1023/A:1026543900054.

[206] O. Pele and M. Werman, "Fast and robust Earth Mover's Distances," in *2009 IEEE 12th International Conference on Computer Vision*, IEEE, Sep. 2009, pp. 460–467, ISBN: 978-1-4244-4420-5. DOI: 10.1109/ICCV.2009.5459199.

[207] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015. DOI: 10.1007/s11263-015-0816-y.

[208] T. Bozkaya and M. Ozsoyoglu, "Indexing large metric spaces for similarity search queries," *ACM Transactions on Database Systems (TODS)*, vol. 24, no. 3, pp. 361–404, 1999.

[209] R. R. Curtin and P. Ram, "Dual-tree Fast Exact Max-kernel Search," *Statistical Analysis and Data Mining*, vol. 7, no. 4, pp. 229–253, Aug. 2014, ISSN: 1932-1864. DOI: 10.1002/sam.11218.

[210] R. E. Tarjan and J. van Leeuwen, "Worst-case Analysis of Set Union Algorithms," *Journal of the ACM (JACM)*, vol. 31, no. 2, pp. 245–281, Mar. 1984, ISSN: 0004-5411. DOI: 10.1145/62.2160.

[211] R. E. Tarjan, "Efficiency of a Good But Not Linear Set Union Algorithm," *Journal of the ACM (JACM)*, vol. 22, no. 2, pp. 215–225, Apr. 1975, ISSN: 0004-5411. DOI: 10.1145/321879.321884.

[212] J. E. Hopcroft and J. D. Ullman, "Set Merging Algorithms," *SIAM Journal on Computing*, vol. 2, no. 4, pp. 294–303, Dec. 1973, ISSN: 0097-5397. DOI: 10.1137/0202024.

[213] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. Nicholas, "Malware Detection by Eating a Whole EXE," *ArXiv preprint arXiv:1710.09435*, Oct. 2017.

[214] E. Raff, J. Sylvester, and C. Nicholas, "Learning the PE Header, Malware Detection with Minimal Domain Knowledge," in *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, ser. AISec '17, New York, NY, USA: ACM, 2017, pp. 121–132, ISBN: 978-1-4503-5202-4. DOI: 10.1145/3128572.3140442.