2018-12

FREQUENCY-BASED FEATURE EXTRACTION FOR MALWARE CLASSIFICATION

Erwert, Jonathan P.

Monterey, CA; Naval Postgraduate School

http://hdl.handle.net/10945/61360

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

FREQUENCY-BASED FEATURE EXTRACTION FOR MALWARE CLASSIFICATION

by

Jonathan P. Erwert

December 2018

Thesis Advisor:	Neil C. Rowe
Second Reader:	Mikhail Auguston

Approved for public release. Distribution is unlimited.

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** December 2018	**3. REPORT TYPE AND DATES COVERED** Master's thesis

4. TITLE AND SUBTITLE FREQUENCY-BASED FEATURE EXTRACTION FOR MALWARE CLASSIFICATION	**5. FUNDING NUMBERS**
6. AUTHOR(S) Jonathan P. Erwert	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release. Distribution is unlimited.	**12b. DISTRIBUTION CODE** A

13. ABSTRACT (maximum 200 words)

Traditional signature-based malware detection is effective, but it can only identify known malicious programs. This thesis attempts to use machine-learning techniques to successfully identify previously unknown malware from a set of Windows executable programs. We analyzed the histogram of 4-, 8-, and 16-bit-sequence values contained in each program. We then analyzed the effectiveness of using these histograms in part or in full as feature vectors for machine learning experiments. We also explored the effect of an offset at the beginning of each program and its impact on classifier performance. We successfully show that a machine learning classifier can be learned from these features, with an f-measure in excess of 90% attained in one of our experiments. Using a part of the histogram as the feature vector did not significantly affect classifier performance up to a point, nor did including an offset. Our results also suggest that features derived from histograms are better suited to tree-based algorithms compared to Bayesian methods.

14. SUBJECT TERMS machine learning, malware analysis, static analysis	**15. NUMBER OF PAGES** 57
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

THIS PAGE INTENTIONALLY LEFT BLANK

FREQUENCY-BASED FEATURE EXTRACTION FOR MALWARE CLASSIFICATION

Jonathan P. Erwert
Lieutenant, United States Navy
BS, U.S. Naval Academy, 2011

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 2018

Approved by: Neil C. Rowe
Advisor

Mikhail Auguston
Second Reader

Peter J. Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Traditional signature-based malware detection is effective, but it can only identify known malicious programs. This thesis attempts to use machine-learning techniques to successfully identify previously unknown malware from a set of Windows executable programs. We analyzed the histogram of 4-, 8-, and 16-bit-sequence values contained in each program. We then analyzed the effectiveness of using these histograms in part or in full as feature vectors for machine learning experiments. We also explored the effect of an offset at the beginning of each program and its impact on classifier performance. We successfully show that a machine learning classifier can be learned from these features, with an f-measure in excess of 90% attained in one of our experiments. Using a part of the histogram as the feature vector did not significantly affect classifier performance up to a point, nor did including an offset. Our results also suggest that features derived from histograms are better suited to tree-based algorithms compared to Bayesian methods.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Neil Rowe, and my wife, Caitlin, for their infinite patience throughout this entire process.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

Traditional malware-detection software relies upon signatures derived from static analysis of the file or dynamic analysis of the program's behavior. Though effective on known malware, signature-based analysis cannot identify previously unknown malicious programs, and when one considers that millions of new malicious programs are discovered every year, it becomes evident that better tools are needed. Part of United States Cyber Command's most recent vision statement is hardening U.S. government networks (United States Cyber Command [USCYBERCOM], 2018). Critical to hardening of networks is speeding up the ability to detect malware on a host. This thesis builds on recent work in the fields of computer forensics and machine learning to detect malware, and explores the viability of using the frequencies of bit-sequence values occurring in a program to identify malware.

Binary strings can be used to represent base-10 numbers, where the length of the string determines the range of values that can be represented. For example, a sequence of 4 bits can represent base-10 numbers between 0 and fifteen. Our method examines how often each possible value of a bit sequence appears in a given file's raw binary. We then used this data as input to several machine-learning experiments to answer whether it possible to distinguish between malicious and benign programs by looking at their histograms (frequency distribution) of values for bit sequences.

The remainder of this thesis is organized in the following way. Chapter II is an overview of current malware-analysis techniques, previous research attempts to increase the speed of malware analysis and identification, and attempts to automate malware classification and identification. It also includes a discussion of previous uses of histograms or frequency-based methods for malware classification. Chapter III describes our methodology, including our process for obtaining malware and non-malicious programs for testing, our method for generating histograms, and a description of the machine-learning algorithms used. Chapter IV describes results of our experiments. Chapter V provides our major findings and makes recommendations for future work.

THIS PAGE INTENTIONALLY LEFT BLANK

II. BACKGROUND AND RELATED WORK

A. EXECUTABLES

Programs are used to make computers do things. Programs are written in human readable languages that computers cannot understand. To make these programs understandable to computers, the high-level source code is input to a compiler, which outputs a binary representation of the program known as the executable image (Anderson & Dahlin, 2011). The executable image, also known simply as the executable, is a series of machine code instructions written in binary, which can be immediately understood and executed by the computer's central processing unit. Running executable files of unknown origin is dangerous from a security perspective, because the advertised behavior and actual behavior of the program may not be the same. However, it is difficult to examine a program and determine if it is malicious.

B. MALWARE ANALYSIS

The goal of our research is to improve automated malware identification, specifically for Windows executables. This research is built upon the analytical techniques used to dissect and understand malware. Malware analysis can be static or dynamic (Sikorski & Honig, 2012). Static analysis is performed by passive examination of the file. This usually means an examination of the raw binary machine language using a disassembler program or, if it is available, the source code of a malicious program. However, less technical observations such as the file name and file path can also provide insight or clues to identify malicious files (Rowe, 2015b). Dynamic analysis is conducted in a safe forensic environment known as a sandbox and looks for adverse behavior as the suspected malicious program executes.

Static analysis can develop signatures such as hashes of known malicious programs or parts of programs, which can be used to identify previously analyzed examples of malware. Hashes are generated using a mathematical function that takes input data, such as a program or data file, and generates a unique alphanumeric output. Other things that can be extracted from programs are signatures, sequences of distinctive bytes appearing in the program. Antivirus programs rely heavily on these signatures to identify files of concern. Unfortunately, malware developers have adapted by manipulating or obfuscating their code so that signatures are often changing.

Signatures generated from dynamic analysis are sequences of system calls or other significant events observed in a malicious program. Dynamic analysis should be superior to static analysis because it allows the analyst to directly observe the behavior of the suspected malware. However, countermeasures for dynamic analysis can check to see if the program is being run in a forensic environment. This can be done by looking for common plugins associated with virtual machines, or checking if the execution environment has limited resources, which would indicate the use of a virtual machine as a sandbox (Sikorski & Honig, 2012). Another potential problem with dynamic analysis is that it relies on actually running the program. This is a time- and resource-intensive process that also risks infecting the machine with malware.

C. CODE REUSE AND PROGRAM COMPARISON

Understanding how similar two programs are is important for malware classification. Similarities between examples of malware can occur because the malware exploits the same vulnerability, is intended to perform a similar function, or displays the habits and idiosyncrasies of the individual malware developer. Similarities in malware also can occur due to the sharing and reuse of source code (Benjamin & Chen, 2013). Source code is shared and reused among authors to save effort and to expand the lifespan of particular malware. Malware authors will also often generate permutations of the same piece of malware to evade signature detection. All of these factors contribute to the rapid production rate of malware.

Several studies have examined how different variants of the same malware can be compared (Adkins, Jones, Carlisle, & Upchurch, 2013; Hongyuan & Osorio, 2013; Seideman, Vargas, & Khan, 2014; Casey & Shelmire, 2014; Sarntinos, Benzaid, Arabiat, & Al-Nemrat, 2016)). These studies differ in what they are specifically comparing and how the two samples are compared. Adkins et al. (2014) and Sarantinos et al. (2016) generated composite hashes of programs and compared files based on the differences between their composite hashes. Adkins et al. (2014) used a technique called basic block comparison which hashes parts of the file and then creates a composite hash-digest, while Sarantinos (2016) et al. used the ssdeep implementation of the fuzzy hashing algorithm to create a similar composite hash. Seideman et al. (2014) took a different approach, generating a system call trace and comparing programs based on the similarities of the programs' traces. Once the original program was summarized, statistical methodologies were used to compare the two files. One such method of comparison is the Jacquard index, which compares the intersection and the union of two sets and gives the fraction or percentage of the elements the two sets have in common. The Jacquard index can be used to determine how many common elements two files have (Adkins et al., 2013). It can also be used to identify code reuse in malware (Casey & Shelmire, 2014). A similar technique, the Jensen-Shannon distance, has been used to determine similarity in system call traces to classify malware in categories similar to biological genera (Seideman et al., 2014). Entropy can also be used as a measure of comparison (Sarantinos et al., 2016).

There are limits to these kinds of pattern matching because two programs with the same purpose can be represented many ways depending on the author, language, or compiler used. If pattern matching cannot be used to compare two programs, dynamic analysis is necessary to determine semantic equivalence. A program is said to be semantically equivalent to another if they execute identically. This is determined by observing the execution of each program to see if they both start and stop in the same state. This technique can be used on sections of programs to identify common functions or methods that are commonly reused (Hongyuan & Osorio, 2013). However, it is very difficult to apply.

D. DISTRIBUTION OF MALWARE

Looking at the extent of code reuse and the rapid rates of malware production might lead one to believe that our systems are inundated with malware. This is not necessarily the case. The Naval Postgraduate School (NPS) maintains a large corpus of drives for the purpose of research in digital forensics. The corpus consists of 4000 drives containing 290 million files. Out of the files only about 0.11 percent of the files were identified as malicious by one of five representative malware detection methods (Rowe, 2015b). Several projects have used machine learning to recognize characteristics of malware executables (Choi, J., Kim, H., Choi, C., & Kim, H., 2011; Mira, Huang, & Brown, 2017).

E. ATTRIBUTE EXTRACTION

Feature (or attribute) extraction is the process of finding and defining the interesting phenomena contained in each element of a data set. For malware, this is the process of taking a file's binary and finding patterns by which a machine-learning algorithm can reach useful classifications or conclusions. These patterns are usually represented as ordered lists or vectors. Just as malware analysis can be static or dynamic, so too can the process of attribute extraction. Dynamic analysis observes the execution of the malicious program. This generates an attribute vector of system calls, libraries loaded, and other observable events. For example, one study extracted file-system writes, register operations, and network-access operations as its attributes (Cabau, Buhu, & Oprisa, 2016). However, most malware studies use static analysis for attribute extraction due to its significant speed advantage. The most common static analysis technique is n-gram extraction, a technique from natural-language processing research which seeks to capture the dependencies between items in succession in a sequence (Russell & Norvig, 2010). When applied to malware detection, n-gram analysis can be performed using either the raw binary or the assembly language instructions. In their study, Zak, Raff, and Nichols (2017) found that byte code n-grams generalized better than n-grams using assembly language. Despite these findings, assembly language n-gram analysis remains popular with recent studies using “shingling” to increase performance (Hassen, Carvalho, & Chan, 2017). Shingling improves on n-gram analysis by introducing the concept of a break point in the dependency chain. It can be thought of like a paragraph break in writing where the last word of one paragraph and the first word of the next paragraph may or may not be related. Shingling reduces the length of the attribute vector.

Another method of attribute extraction uses frequency of byte values. Two studies which used this method focused on sorting malware into families. Singh and Khurmi

(2016) analyzed portions of files, focusing on unique and likely repeated sequences of code, to create the attribute vectors while Yu et al. (2010) analyzed the entire file. The first compared the sum of Euclidian distances between attributes in two malicious programs to create a baseline for triaging malware into clusters, while the second leveraged a Symbolic Aggregation Approximation (SAX) to determine similarity between programs.

F. MACHINE-LEARNING APPROACHES IN MALWARE ANALYSIS

Machine learning is the process of programming computers to improve their performance without human assistance. Given the frequency of code reuse in malware, it is unsurprising that many studies have used machine learning to create classifiers that can differentiate between types of malware. Machine learning can also be used to distinguish malware from benign programs. Both of these problems require learning a classifier from a set of training data which includes examples of malicious and benign programs or examples of all of the types of malware being classified.

Most machine learning is supervised, meaning that the training data are labeled with the correct classifications. Popular supervised learning techniques include support-vector machines, tree-based classifiers, and Bayesian networks. Support-vector machines try to find an optimal decision boundary between two classes. Tree-based classifiers derive a series of yes-or-no questions based on the values of the attributes in the training data. Bayesian networks use compounding probabilities to predict the correct classification (Russell & Norvig, 2010). All these techniques have been used to derive classifiers for malware identification, and are often used as baselines when evaluating new techniques (Fuyong & Tiezhu, 2017). N-gram attributes derived from dynamic analysis have been used to train support-vector machines to distinguish between malware and benign software (Okane, Sezer, & McLaughlin, 2014). Similarly, n-gram attributes derived from static analysis machine code have been used to train the decision-tree random-forest algorithm to create a malware classifier (Usaphapanus & Piromsopa, 2017).

Unsupervised learning differs from supervised learning in that the training data is not labeled with the correct classifications. Instead, unsupervised learning seeks to find patterns that exist in the data naturally. Unsupervised techniques include clustering and unsupervised neural networks (Kalash et al., 2018; Kargaard, Drange, Kor, Twafik, & Butterfield, 2018). In both of these studies, malicious binaries were converted to picture files prior to classification because there is much software available for picture processing by neural networks.

III. METHODOLOGY

The code we wrote appears in the Appendix.

A. THE DATA SET STUDIED

We derived our data set from the NPS “Real Drive Corpus.” The corpus consists of 4000 images of drives storing 290 million files. The corpus was compiled by purchasing secondhand drives from around the world. The primary purpose of this corpus is to facilitate research in computer forensics, with identification of malware being a significant focus (Garfinkel, Farrell, Roussev, & Dinolt, 2009). It enabled us to compile both our data set of malicious and benign executable files from a single source.

The primary tool we used for working with the forensic corpus was the SleuthKit program developed by Brian Carrier (2018). SleuthKit includes a tool developed at NPS called Fiwalk which automatically extracts file directory information and associated file metadata (Garfinkel, 2009). We used Fiwalk to compile a list of executable files in the corpus with DLL and EXE extensions. After selecting a random subset from this list, we retrieved the actual files from the drive images in Expert Witness Disk Image Format (EWF) in the corpus using Sleuthkit's “icat” command. EWF is a popular file format for forensic images which meets the legal requirements of evidence preservation (Library of Congress, 2017). EWF files accurately preserve the total content of a drive, including deleted files and data stored in slack space. The icat command retrieves one file at a time, so we automated the collection process by writing a Bash shell script with a separate icat command for every file sought. Our final data set included 4835 Windows executables, of which 4436 were classified as benign and 399 were classified as malicious.

Previous research on the corpus revealed that roughly 0.11 percent of the files in the corpus were malware, with a bit higher percentage for executables (Rowe, 2015b). We deliberately selected malware randomly with a higher percentage, so our random sample included roughly 8 percent malware. This is significantly different from earlier research which used approximately equal numbers of benign and malicious files in their data sets or used a majority of malicious files (Choi et al., 2011; Mira et al., 2017).

Having an equal number of malicious and benign programs in the data set reduces the chances the classifier would be biased toward malicious or benign programs. However, given the real-world origins of the NPS corpus, we thought it better to use a data set which reflected the corpus' ratio of malicious to benign programs.

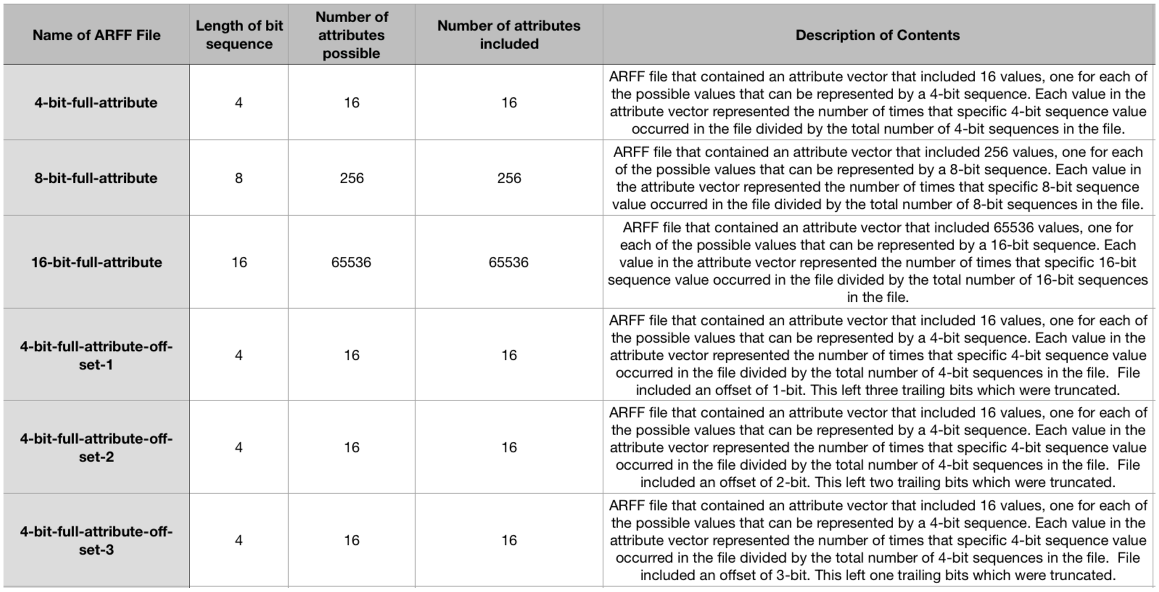
B. ATTRIBUTE EXTRACTION AND ARFF FILE GENERATION

We needed to extract a set of attributes from each executable file in our data set so that it could be input to our machine-learning platform Weka. Weka is a machine-learning workbench developed by the University of Waikato in New Zealand. It uses an input format known as Attribute Relation File Format or ARFF (Witten, Frank, & Hall, 2011). An ARFF file has two parts. The first part is a header which includes the name of the data set, the attributes extracted from each executable in the data set, and the type of the data for the attribute (integer, real, string, or list of values). The second part is the data values of the attributes extracted from each executable. This set of values is called an attribute vector and is represented as an ordered list. To generate our ARFF files we wrote programs in Python 3 and ran them on a MacBook Pro running the MacOS High Sierra Operating system.

A histogram is a set of counts on elements of a set. We generated histograms for bit sequences of 4, 8, and 16 bits within each executable file, and used those for the data in our ARFF files. To generate these histograms for 8-bit and 16-bit sequences, we used a program written by Prof. Rowe (2015a). To generate the histogram of 4-bit sequences, we wrote our own program using the Python Bitstring library. This library was necessary because Python will only work with byte-aligned data and will only process binary data at the byte level. The Bitstring library allowed us to create histograms of 4-bit values at offsets of 0-bit, 1-bit, 2-bits, and 3-bits. Because our data was byte-aligned, this left us with nonzero offsets with a final set of bits less than four which we ignored. The reason for generating multiple 4-bit histograms with varying offsets was that we hypothesized that including an offset would create a different distribution of counts, and wanted to see if including a given offset would provide significantly better or worse classifier performance. We only generated offset ARFF files for the 4-bit sequence values as a proof of concept.

Once we had generated our histograms, we generated our attribute vectors using two different methods. The first method normalized the counts in the histogram by dividing by the total count of the executable. We used this first method to generate six ARFF files: one for the 8-bit sequence, one for the 16-bit sequence, and one for each of the four 4-bit sequence offsets. We will refer to ARFF files generated using this first method as full-attribute because they contain an attribute vector which includes a value for each value represented in the executables' full histogram. Table 1 provides specific details on the six full-attribute ARFF files created using this first method. As the length of the bit-sequence, we were examining increased, so too did the length of the attribute vector, so that our full-attribute ARFF file contained an attribute vector of 65536 values. Though Weka was generally capable of handling this large number of attributes, the performance was slow and often required multiple tries to get Weka to successfully run the machine-learning algorithms using such a large input file. For this reason, we also experimented with a second method of attribute selection that explicitly tried to reduce the size of the attribute vector.

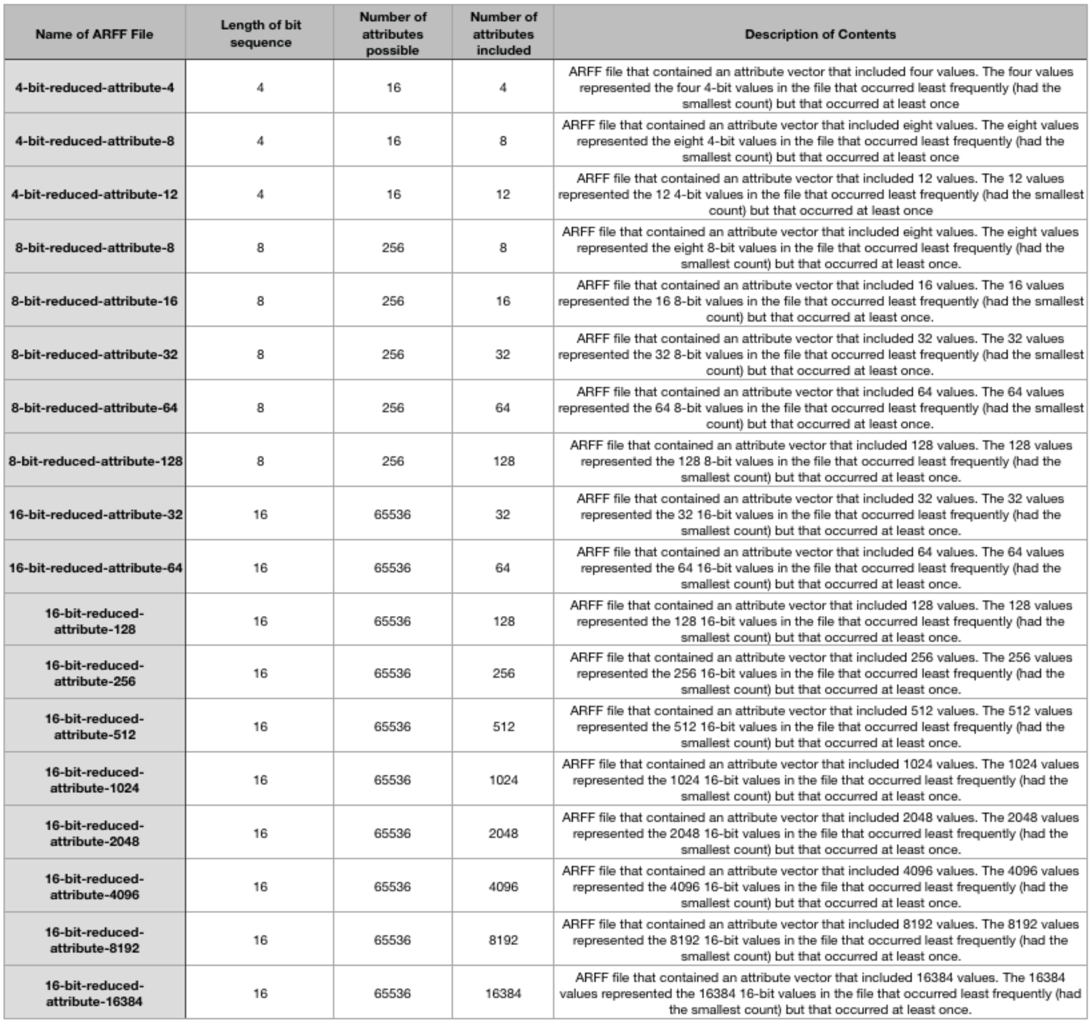
Table 1. List of Full-Attribute ARFF Files



Name of ARFF File	Length of bit sequence	Number of attributes possible	Number of attributes included	Description of Contents
4-bit-full-attribute	4	16	16	ARFF file that contained an attribute vector that included 16 values, one for each of the possible values that can be represented by a 4-bit sequence. Each value in the attribute vector represented the number of times that specific 4-bit sequence value occurred in the file divided by the total number of 4-bit sequences in the file.
8-bit-full-attribute	8	256	256	ARFF file that contained an attribute vector that included 256 values, one for each of the possible values that can be represented by a 8-bit sequence. Each value in the attribute vector represented the number of times that specific 8-bit sequence value occurred in the file divided by the total number of 8-bit sequences in the file.
16-bit-full-attribute	16	65536	65536	ARFF file that contained an attribute vector that included 65536 values, one for each of the possible values that can be represented by a 16-bit sequence. Each value in the attribute vector represented the number of times that specific 16-bit sequence value occurred in the file divided by the total number of 16-bit sequences in the file.
4-bit-full-attribute-off-set-1	4	16	16	ARFF file that contained an attribute vector that included 16 values, one for each of the possible values that can be represented by a 4-bit sequence. Each value in the attribute vector represented the number of times that specific 4-bit sequence value occurred in the file divided by the total number of 4-bit sequences in the file. File included an offset of 1-bit. This left three trailing bits which were truncated.
4-bit-full-attribute-off-set-2	4	16	16	ARFF file that contained an attribute vector that included 16 values, one for each of the possible values that can be represented by a 4-bit sequence. Each value in the attribute vector represented the number of times that specific 4-bit sequence value occurred in the file divided by the total number of 4-bit sequences in the file. File included an offset of 2-bit. This left two trailing bits which were truncated.
4-bit-full-attribute-off-set-3	4	16	16	ARFF file that contained an attribute vector that included 16 values, one for each of the possible values that can be represented by a 4-bit sequence. Each value in the attribute vector represented the number of times that specific 4-bit sequence value occurred in the file divided by the total number of 4-bit sequences in the file. File included an offset of 3-bit. This left one trailing bits which were truncated.

The second method used the bit sequences with the lowest nonzero counts as our attributes since these more likely indicate unique properties of an executable. To select the values for our attribute vector, we sorted the histograms by count from lowest to highest. We then selected increasingly large, but overlapping, subsets of the sorted histogram. Our selection criteria were simple in that we selected the value in the histogram with the smallest count greater than zero and then took the next specified number of values. This gave us a subset of the least frequently occurring values from the histogram which occurred at least one time. From the histogram of 4-bit sequence values, we selected three subsets. These subsets consisted of 4, 8 and 12 values. As previously stated these subsets were overlapping, so all of the attributes included in a smaller subset were included in any larger subsets. Each of these subsets was used as the attribute vector for its own ARFF file. We repeated this process using the histogram of 8-bit sequence values selecting subsets of 8, 16, 32, 64, and 128 values, and for the histogram of 16-bit sequence values selecting subsets of 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, and 16384. We will refer to ARFF files generated using our second method as reduced-attribute because they contain a subset of the executables' histogram. Table 2 details the 18 reduced-attribute ARFF files generated using this second method. We did not generate any reduced-attribute ARFF files from the histograms which included an offset.

Table 2. List of Reduced-Attribute ARFF Files Generated



Name of ARFF File	Length of bit sequence	Number of attributes possible	Number of attributes included	Description of Contents
4-bit-reduced-attribute-4	4	16	4	ARFF file that contained an attribute vector that included four values. The four values represented the four 4-bit values in the file that occurred least frequently (had the smallest count) but that occurred at least once
4-bit-reduced-attribute-8	4	16	8	ARFF file that contained an attribute vector that included eight values. The eight values represented the eight 4-bit values in the file that occurred least frequently (had the smallest count) but that occurred at least once
4-bit-reduced-attribute-12	4	16	12	ARFF file that contained an attribute vector that included 12 values. The 12 values represented the 12 4-bit values in the file that occurred least frequently (had the smallest count) but that occurred at least once
8-bit-reduced-attribute-8	8	256	8	ARFF file that contained an attribute vector that included eight values. The eight values represented the eight 8-bit values in the file that occurred least frequently (had the smallest count) but that occurred at least once.
8-bit-reduced-attribute-16	8	256	16	ARFF file that contained an attribute vector that included 16 values. The 16 values represented the 16 8-bit values in the file that occurred least frequently (had the smallest count) but that occurred at least once.
8-bit-reduced-attribute-32	8	256	32	ARFF file that contained an attribute vector that included 32 values. The 32 values represented the 32 8-bit values in the file that occurred least frequently (had the smallest count) but that occurred at least once.
8-bit-reduced-attribute-64	8	256	64	ARFF file that contained an attribute vector that included 64 values. The 64 values represented the 64 8-bit values in the file that occurred least frequently (had the smallest count) but that occurred at least once.
8-bit-reduced-attribute-128	8	256	128	ARFF file that contained an attribute vector that included 128 values. The 128 values represented the 128 8-bit values in the file that occurred least frequently (had the smallest count) but that occurred at least once.
16-bit-reduced-attribute-32	16	65536	32	ARFF file that contained an attribute vector that included 32 values. The 32 values represented the 32 16-bit values in the file that occurred least frequently (had the smallest count) but that occurred at least once.
16-bit-reduced-attribute-64	16	65536	64	ARFF file that contained an attribute vector that included 64 values. The 64 values represented the 64 16-bit values in the file that occurred least frequently (had the smallest count) but that occurred at least once.
16-bit-reduced-attribute-128	16	65536	128	ARFF file that contained an attribute vector that included 128 values. The 128 values represented the 128 16-bit values in the file that occurred least frequently (had the smallest count) but that occurred at least once.
16-bit-reduced-attribute-256	16	65536	256	ARFF file that contained an attribute vector that included 256 values. The 256 values represented the 256 16-bit values in the file that occurred least frequently (had the smallest count) but that occurred at least once.
16-bit-reduced-attribute-512	16	65536	512	ARFF file that contained an attribute vector that included 512 values. The 512 values represented the 512 16-bit values in the file that occurred least frequently (had the smallest count) but that occurred at least once.
16-bit-reduced-attribute-1024	16	65536	1024	ARFF file that contained an attribute vector that included 1024 values. The 1024 values represented the 1024 16-bit values in the file that occurred least frequently (had the smallest count) but that occurred at least once.
16-bit-reduced-attribute-2048	16	65536	2048	ARFF file that contained an attribute vector that included 2048 values. The 2048 values represented the 2048 16-bit values in the file that occurred least frequently (had the smallest count) but that occurred at least once.
16-bit-reduced-attribute-4096	16	65536	4096	ARFF file that contained an attribute vector that included 4096 values. The 4096 values represented the 4096 16-bit values in the file that occurred least frequently (had the smallest count) but that occurred at least once.
16-bit-reduced-attribute-8192	16	65536	8192	ARFF file that contained an attribute vector that included 8192 values. The 8192 values represented the 8192 16-bit values in the file that occurred least frequently (had the smallest count) but that occurred at least once.
16-bit-reduced-attribute-16384	16	65536	16384	ARFF file that contained an attribute vector that included 16384 values. The 16384 values represented the 16384 16-bit values in the file that occurred least frequently (had the smallest count) but that occurred at least once.

After the attribute vectors were generated, the associated executables needed to be tagged as malicious or benign. This was done by comparing an executable's hashcode to those in the malware libraries of Bit9, Open Malware, Virus Share, Symantec, and Clam AV using the data on from the NPS Real Drive Corpus (Rowe, 2015b).

C. MACHINE-LEARNING METHODS

Weka implements a wide range of well-known machine-learning algorithms. Our experiments primarily used tree-based classifiers. We used the J48, logistic-model-tree (LMT), random-tree, random-forest, and reduced-error-pruning (REP) tree classifiers.

We also experimented with the Naïve-Bayes and Bayesian-network classifiers. All classifiers were trained using 66 percent of the data set and evaluated on the remaining 34 percent; this approach could be easily extended to do cross-validation to obtain more accurate performance measures. We use the term "experiment" to describe a pairing of an input ARFF file and a machine-learning algorithm. We paired each of our 24 ARFF files with each of the 7 classifiers, for a total of 168 experiments. Five of these experiments did not return results, either due to raising an error or failing to fully execute.

D. PRECISION, RECALL, AND F-SCORE

Precision is the proportion of correctly classified instances of a class compared to the total number of instances classified as the class (Witten et al., 2016). In our experiments, this meant the number of malicious files identified divided by the number of total files classified as malicious. Recall is the proportion of correctly classified instances of a class compared to the total number of instances of that class in the data set (Witten et al., 2016). In our experiments, this meant the number of malicious files correctly identified as malicious divided by the total number of malicious files in the data set. Precision and recall are generally viewed as competing statistics, where maximizing one minimizes the other (Witten et al., 2016). To get a kind of average of these competing metrics, we also used the F-score which is their harmonic mean. It is calculated as two times the product of the recall and precision divided by their sum.

IV. DISCUSSION OF RESULTS

A. CLASSIFIER PERFORMANCE USING FULL-ATTRIBUTE ARFF FILES

Figures 1, 2, and 3 display the precision, recall, and F-score for experiments we conducted using the histogram of values generated for 4, 8, and 16-bit values. The Naïve Bayes classifier provided the highest recall but the lowest precision. Across all bit-sequence lengths, the five tree classifiers of J48, random-tree, random-forest, LMT, and REP outperformed the two Bayesian classifiers, with the random-forest classifier performing best across all bit-sequence lengths in both precision and F-score. This is likely due to random-forest being the only classifier which uses bagging, or training using multiple independent data sets, to boost performance (Russell & Norvig, 2010). Of note, the best overall performance for F-score was from the 4-bit ARFF file using the random-forest classifier. This was the best F-score noted for all experiments performed and the only F-score that exceeded 90 percent. With this exception, the overall performance of all classifiers remained consistent across bit-sequence length.

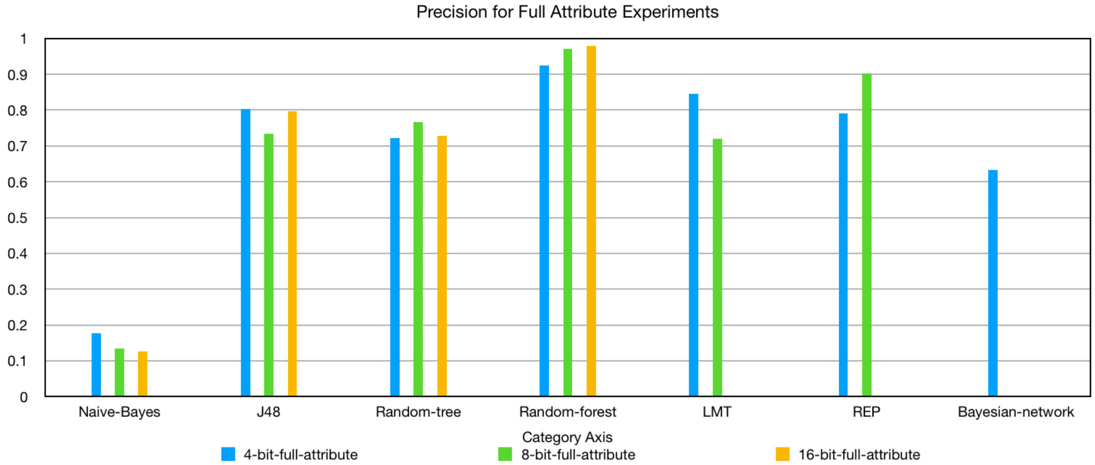


Figure 1. Precision Performance of Full-Attribute Classifiers

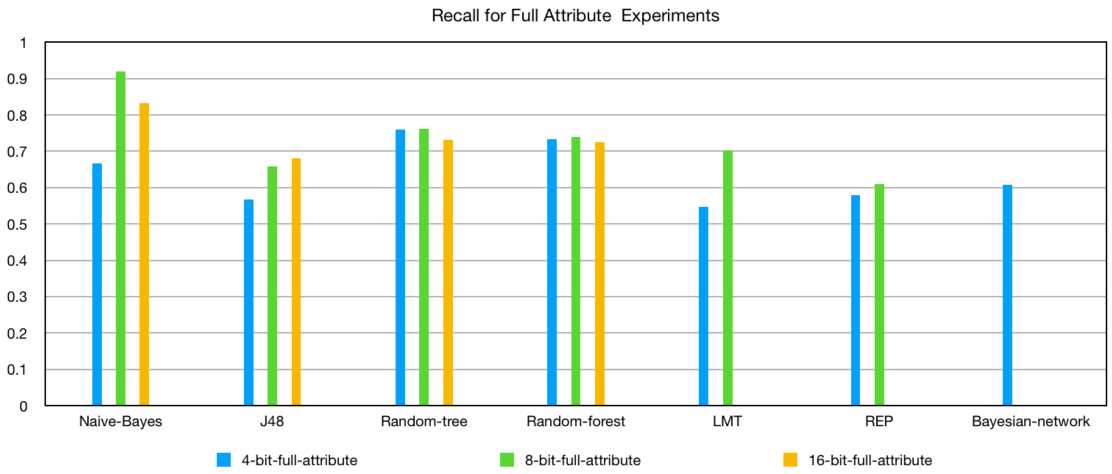


Figure 2. Recall Performance of Full-Attribute Classifiers

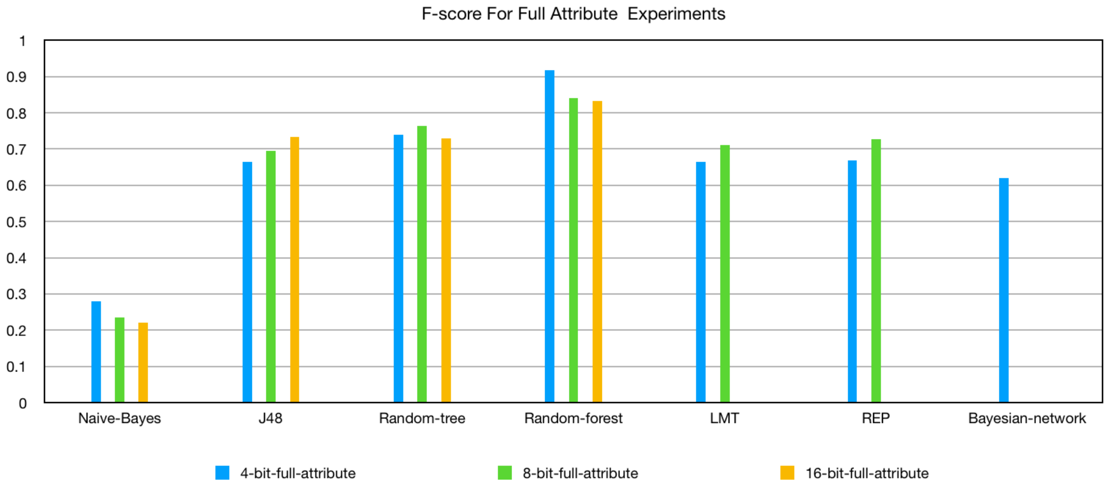


Figure 3. F-score Performance of Full-Attribute Classifiers

We were not able to get results for all seven of the classifiers for all three full-attribute ARFF files that did not include an offset. When we tested both the 16-bit and 8-bit full-attribute ARFF files using the Bayesian-network classifier, Weka raised an error for overlapping bin ranges, which is strange because Weka automatically selects bin ranges based on the range of values observed in the attribute vector. When we tested the 16-bit full-attribute ARFF file using both the LMT and REP classifiers, Weka interrupted program execution without raising an exception or outputting any results.

B. THE IMPACT OF OFFSETS ON CLASSIFIER PERFORMANCE

Figures 4, 5, and 6 show the precision, recall, and F-score statistics for using the four 4-bit-full-attribute ARFF files that were generated with the varying 0-bit, 1-bit, 2-bit, or 3-bit offsets. Overall, the results of the offset experiments follow the same trend noted in the other full-attribute experiments. Tree classifiers outperformed Bayesian classifiers, with random-forest providing the best results for both precision and F-score. No clear benefit from offsetting the start point of the binary string is evident from the data, except that the Bayesian-network classifier saw an increase in precision as the offset was increased. This did not generate a corresponding increase in F-score.

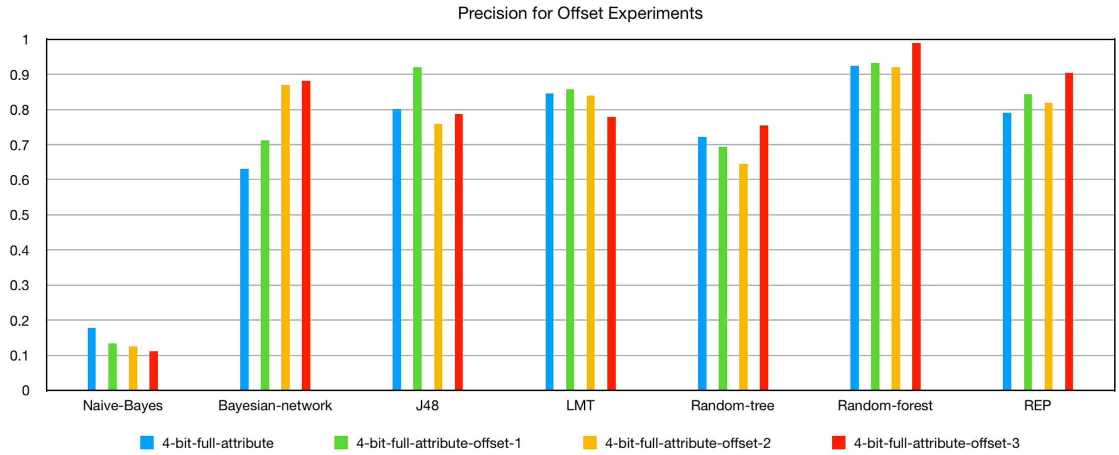


Figure 4. Classifiers' Precision Performance for 4-bit Full-Attribute-Offset Experiments

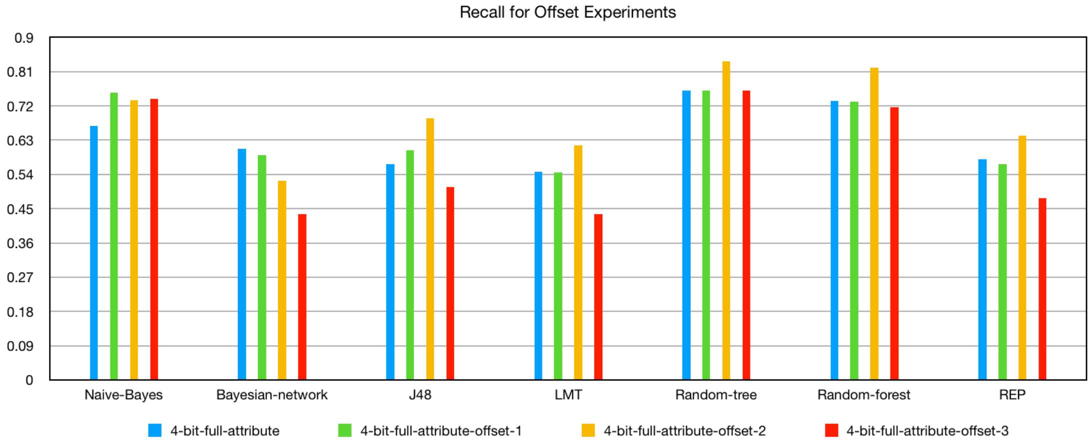


Figure 5. Classifiers' Recall Performance for 4-bit Full-Attribute-Offset Experiments

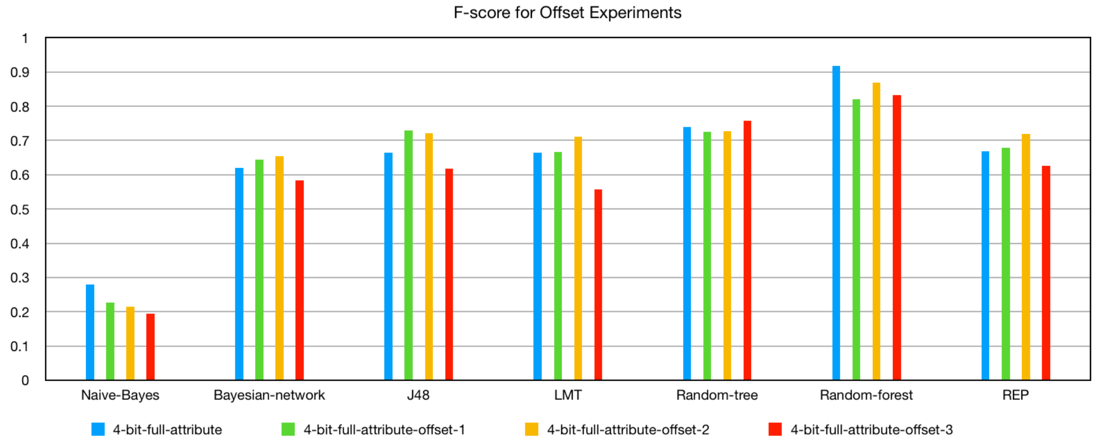


Figure 6. Classifiers' F-score Performance for 4-bit Full-Attribute-Offset Experiments

C. REDUCING ATTRIBUTE VECTOR LENGTH

To test whether a reduced number of attributes changed the performance of a classifier, we used the 18 ARFF files generated using our second method of attribute selection. When we examined those files, we noticed, particularly in ARFF files that had more than 512 values in their attribute vector, that the number of executables represented in the ARFF file did match with the total number of executables in our data set. To investigate this phenomenon further, we generated a bar chart, Figure 7, displaying the number of executables included in each ARFF file.

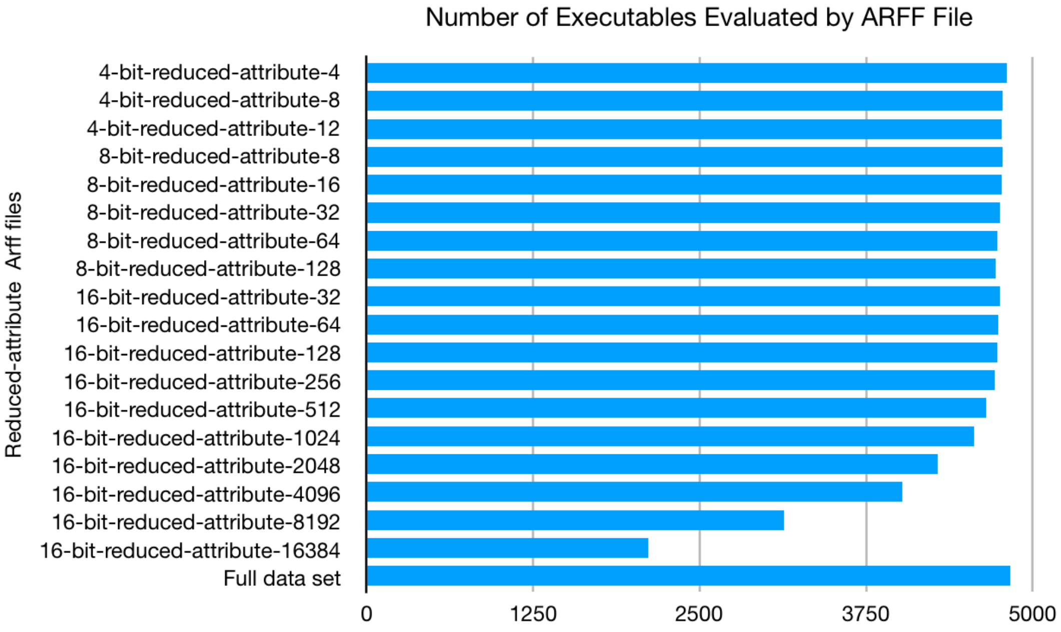


Figure 7. Comparison of the Number of Executables Captured in Our Reduced-Attribute Data Sets

A sharp drop-off in the number of executables included in each ARFF file is evident in the 16-bit-reduced-attribute ARFF files with an attribute vector containing greater than 512 values. This drop-off is concerning because it is indicative of a logical flaw in our attribute extraction methodology. None of the reduced-attribute ARFF files contains the full set of executables contained in our data set. The likely cause of this is that our program included a check to see if each attribute vector generated by our program contained a sufficient number of values to be included in an ARFF file. If an attribute vector failed that check, it was not included in the ARFF file. For example, if we wanted to generate an ARFF file containing an attribute vector with 32 values, but only 31 specific bit-sequences had a count greater than zero, that executable would not be included in the ARFF file. This casts significant doubts on how effective our attribute reduction method is. If every executable cannot be represented using our attribute selection method then not every executable can be classified using our methodology. Practically this means that some malicious files may not even make it to the classification stage and thus have no chance of being detected by our methodology.

1. Classifier Performance on 4-Bit Reduced-Attribute ARFF Files

Figures 8, 9, and 10 display the results for our 4-bit reduced-attribute experiments. The overall trend of tree classifiers outperforming Bayesian classifiers is evident in this data and consistent with the previous experiments using the full-attribute vector length. Random-forest continued to provide the best results in precision and F-score, while random-tree provided the best results for recall. By contrast, in the full-attribute experiments the Naïve-Bayes classifier provided the best results for recall. Reducing the attribute vector length from 12 to 8 did not cause a decline in performance across all classifiers and metrics, but reducing the attribute vector length from 8 to 4 caused a greater than 50 percent drop in F-score across all classifiers.

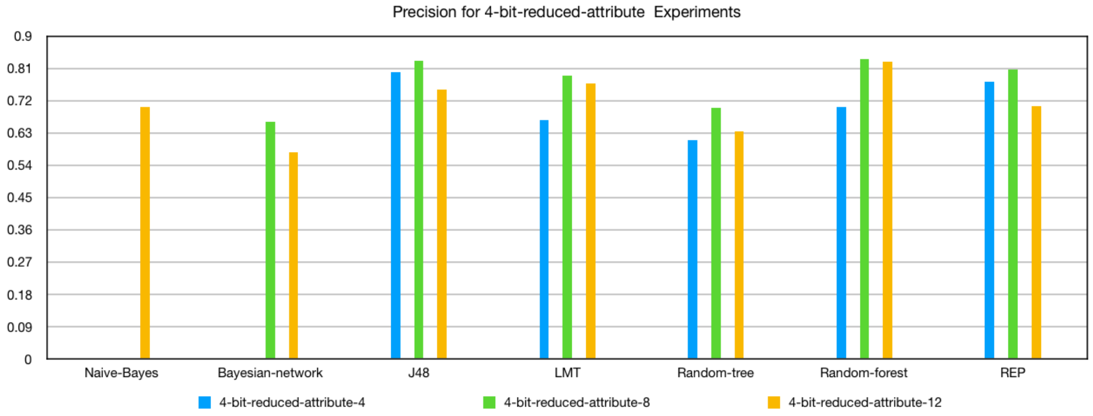


Figure 8. Classifiers' Precision Performance for 4-bit Reduced-Attribute Experiments

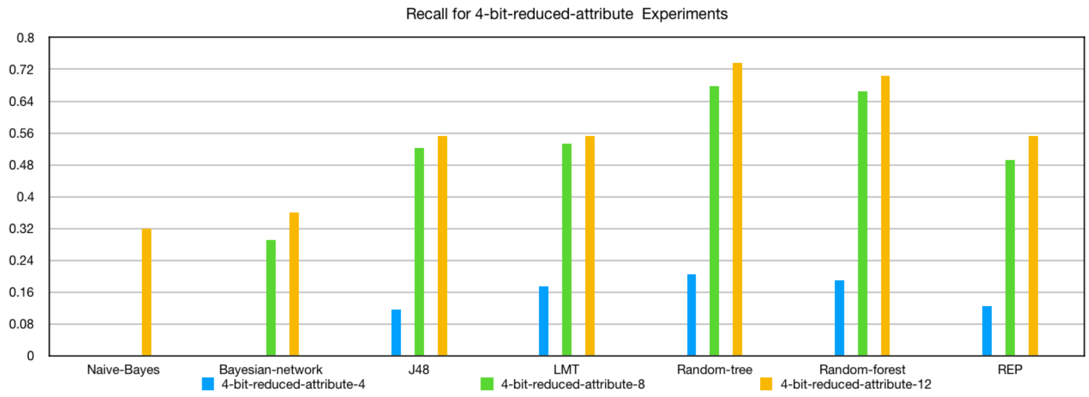


Figure 9. Classifiers' Recall Performance for 4-bit Reduced-Attribute Experiments

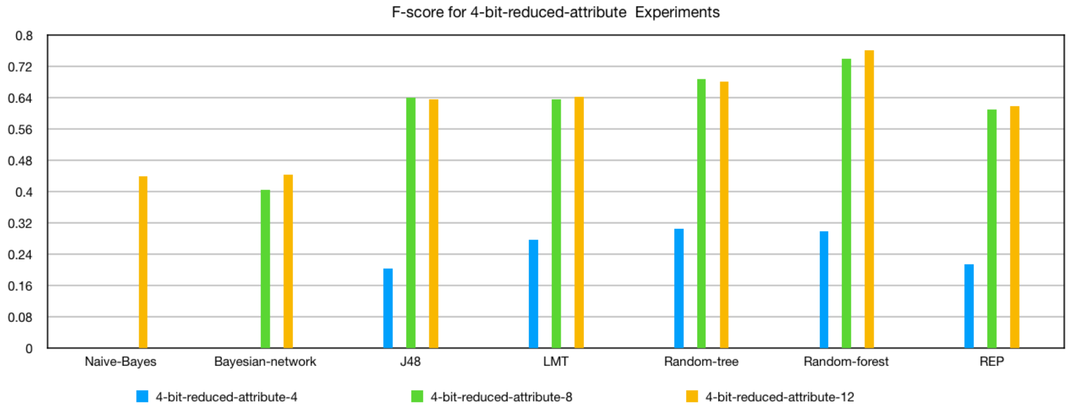


Figure 10. Classifiers' F-score Performance for 4-bit Reduced-Attribute Experiments

Precision and F-score could not be calculated for the 4-bit-reduced-attribute-4 and 4-bit-reduced-attribute-8 Naïve-Bayes experiments and the 4-bit-reduced-attribute-4 Bayesian-network experiments because in all three of these experiments the classifier identified all executables as benign resulting in a divide by zero error.

2. 8-Bit Reduced-Attribute Experiments

Figures 11, 12, and 13 show the results for our 8-bit reduced-attribute experiments. The tree classifiers outperformed Bayesian classifiers in these experiments as well. Random-forest continued to provide the best performance in precision and F-score, while random-tree provided the best results for recall. In both random-tree and random-forest, reducing the attribute vector length had very limited effects on the classifier's performance.

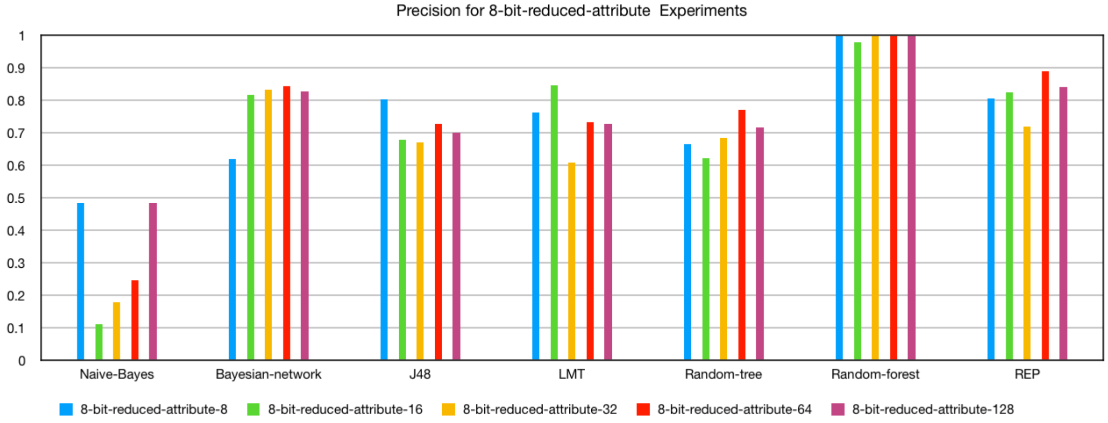


Figure 11. Classifiers' Precision Performance for 8-bit Reduced-Attribute Experiments

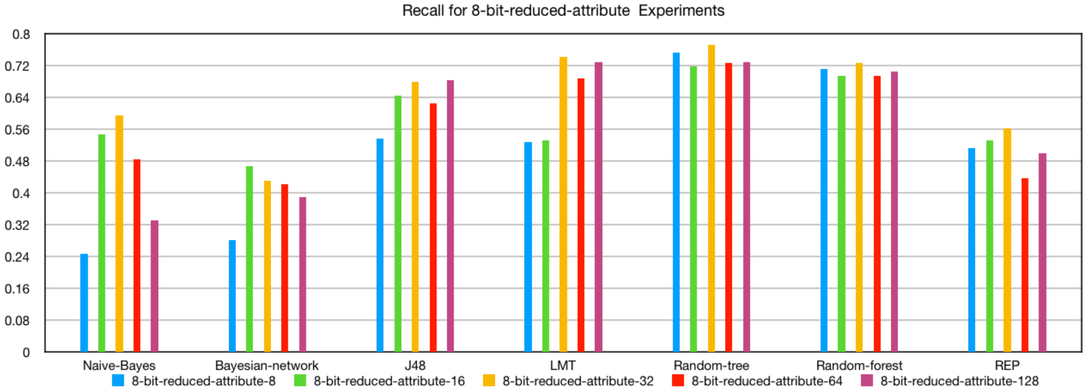


Figure 12. Classifiers' Recall Performance for 8-bit Reduced-Attribute Experiments

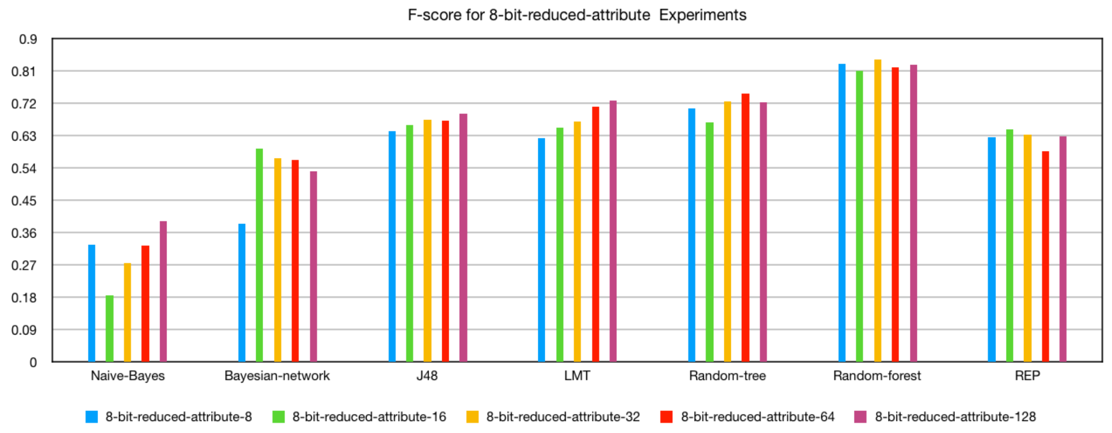


Figure 13. Classifiers' F-score Performance for 8-bit Reduced-Attribute Experiments

3. 16-Bit Reduced-Attribute Experiments

Figures 14, 15, and 16 show the results for precision, recall, and F-score for our 16-bit reduced-attribute experiments. There was considerable variability in the precision and recall performance for all the classifiers as the attribute vector length increased from 32 to 16384. Despite this variability, the results for F-score were relatively constant and consistent with the results from the full-attribute and 8-bit and 16-bit reduced-attribute experiments. Tree-based classifiers continued to outperform Bayesian classifiers and random-forest provided the best performance in terms of F-score and precision. The Naïve-Bayes classifier provided the best results in terms of recall.

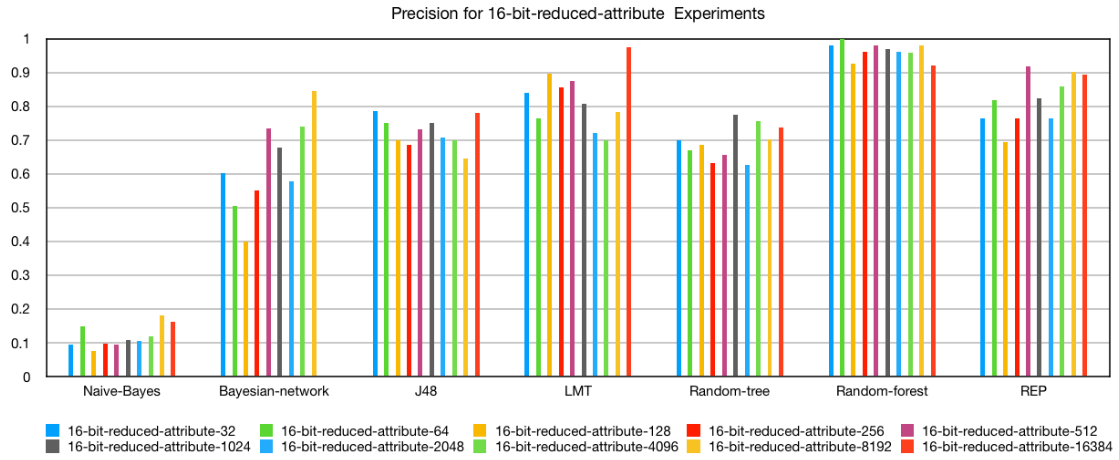


Figure 14. Classifiers' Precision Performance for 16-bit Reduced-Attribute Experiments

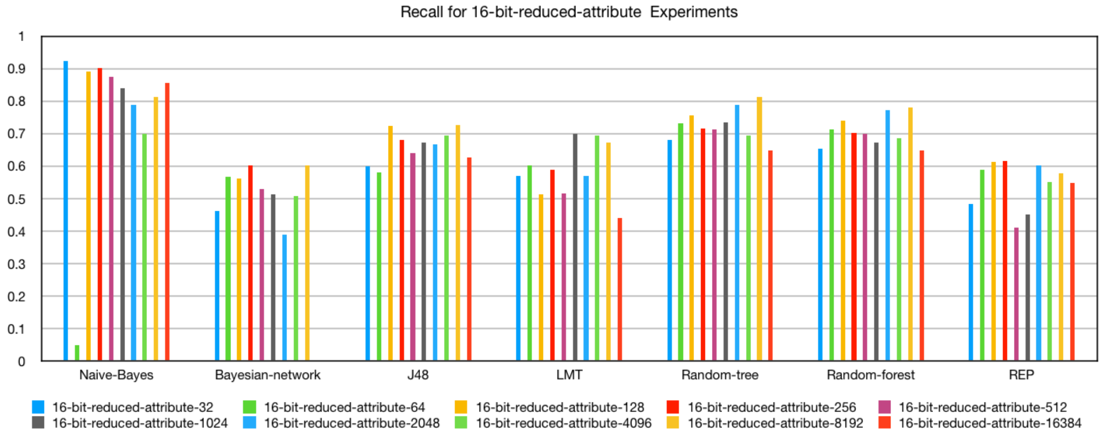


Figure 15. Classifiers' Recall Performance for 16-bit Reduced-Attribute Experiments

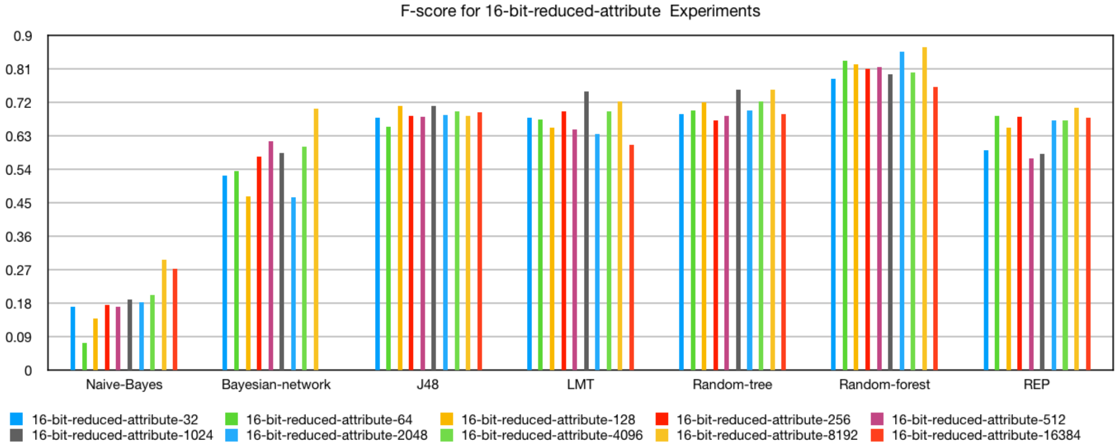


Figure 16. Classifiers' F-score Performance for 16-bit Reduced-Attribute Experiments

THIS PAGE INTENTIONALLY LEFT BLANK

V. CONCLUSIONS

The data from our experiments suggests that histograms of bit-sequence values can distinguish between malware and benign programs. However, classifier performance must be improved if it is to be a useful tool in the fight against malware. One way could be to expand or balance the data set used to train our data. In our experiments, we had 92 percent benign programs and 8 percent malware.

Our data suggests that bit-sequence length is a relatively unimportant factor. Reducing the attribute space, to a point, did not seem to degrade classifier performance. Future work to identify the best way to reduce the attribute vector length and file size would be useful. Our experiments only looked at two types of machine-learning algorithms, tree and Bayesian classifiers. Our data suggests that tree algorithms learn from frequency-based attributes better than Bayesian algorithms do. Our experiments also used only Windows-executable files. Future work should use a more diverse data set.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX. PROGRAM CODE

A. PROGRAM TO GENERATE BASH FILE TO RETRIEVE FILES FROM NPS REAL DRIVE CORPUS

```
def BashGenerator(inputFile, outputFile):
   offsets = open(inputFile, "r", encoding = "UTF-8")
   out = open(outputFile, "w", encoding = "UTF-8")
   out.write("echo OFF\n")
   out.write("echo Retrieving Files\n")
   for line in offsets:
      a = line.split("|")
      out.write('icat -r -i ewf -o '+ a[3][:-1] + ' "/corp/nus/
drives/'+a[1][0:2]+"/"+a[1]+"/"+a[1]+'.E01" ' + a[2] +' >"erwert/'+a[0]+'.exe"\n')
   out.write("echo Finished")
   offsets.close()
   out.close()
   return print("Done")


def MergeFunction(input1, input2, output1):
   a = open(input1, "r", encoding = "UTF-8")
   b = open(input2, "r", encoding = "UTF-8")
   c = open(output1, "w", encoding = "UTF-8")
   a1 = a.readlines()
   a.close()
   b1 = b.readlines()
   b.close()
   for line1 in a1:
      for line2 in b1:
         aa = line1.split("|")
         bb = line2.split("|")
         if bb[1] == aa[0]:
           c.write(bb[0] + "|" + bb[1] + "|" + bb[3] + "|" + aa[1])
   c.close()
   return print("Done")

print("Running Function 3")
function3("best_offsets_for_drives.txt",                    "exe_inode_data_rdc.txt",
"ReadyForFunction1.txt")
print("Done Running Function 3")
print("Running Function 1")
function1("ReadyForFunction1.txt", "erwert_long_test.txt")
print("Done!@#$")
```

B. PROGRAM TO GENERATE 8-BIT FULL-ATTRIBUTE DATA

```
import sys, math, hashlib, os

def endclean(OS):
    S = OS
    M = len(S)
    while ((M > 0) and \
           (S[M-1] in ['\n','\r','\t','\f','\a','\b','\v',' ']) ):
        S = S[0:M-1]
        M = M-1
    return S

def bytedistrib(filename):
    bytedist = [0 for i in range(256)]
    count = 0
    fid = open(filename, 'rb')
    byte = fid.read(1)
    while byte:
        k = int(ord(byte))
        bytedist[k] = bytedist[k] + 1
        byte = fid.read(1)
        count = count + 1
    fid.close()
    line = ""
    for x in bytedist:
        line += str(x/count) + ", "
    return line


if __name__=="__main__":
    benign_dir = sys.argv[1]
    mal_list = sys.argv[2]
    out = sys.argv[3]
    outputdir = {}
    outfile = open(out, 'w')
    outfile.write("@RELATION Single_Byte\n")
    mal1 = open(mal_list, "r", encoding = "UTF-8")
    mal2 = mal1.readlines()
    mal_files = set()

    for line in mal2:
        mal3 = line.split("|")
        mal_files.add(mal3[1][:-1])

    for i in range(256):
        outfile.write("@ATTRIBUTE " + str(i) + " NUMERIC\n")
    outfile.write("@ATTRIBUTE class {benign, malware}\n")
    outfile.write("@data\n")
    for filename in os.listdir(benign_dir):
        full_path = str(benign_dir) +"/"+ str(filename)
        if os.path.isfile(full_path):
            outputdir[full_path] = bytedistrib(full_path)

        if full_path in mal_files:
            outputdir[full_path] += "malware\n"

        else:
            outputdir[full_path] += "benign\n"

    for key in outputdir:
        outfile.write(outputdir[key])
```

C. PROGRAM TO GENERATE 16-BIT REDUCED-ATTRIBUTE DATA

```
import sys, math, hashlib, os

def endclean(OS):
    S = OS
    M = len(S)
    while ((M > 0) and \
           (S[M-1] in ['\n','\r','\t','\f','\a','\b','\v',' ']) ):
        S = S[0:M-1]
        M = M-1
    return S

def bytedistrib(filename):
    bytedist = [0 for i in range(65536)]
    bigramLib = {}
    count = 0
    features_List = []
    fid = open(filename, 'rb')
    byte1 = fid.read(1)
    while byte1:
        byte2 = fid.read(1)
        if byte2:
            k = (int(ord(byte1))*256)+int(ord(byte2))
            bytedist[k] = bytedist[k] + 1
            byte1 = fid.read(1)
        else:
            byte1 = byte2
        count = count + 1
    fid.close()

    for i in list(range(65536)):
        a = bytedist[i]
        if a not in bigramLib:
            bigramLib[a] = [i]
        else:
            bigramLib[a] += [i]

    for i in range(1,count):
        if i in bigramLib:
            features_List += bigramLib[i]
    #features_list returns a list of bigrams which
    # had the lowest non 0 count in the file (smallest keys)
    return features_List


if __name__=="__main__":
    benign_dir = sys.argv[1]
    mal_list = sys.argv[2]
    out = sys.argv[3]
    outputdir = {}
    mal1 = open(mal_list, "r", encoding = "UTF-8")
    mal2 = mal1.readlines()
    mal_files = set()
    Feature_count = int(sys.argv[4])
    outfile = open(out, 'w')
    outfile.write("@RELATION BIGRAMS "+str(Feature_count)+"\n")

    for line in mal2:
        mal3 = line.split("|")
        mal_files.add(mal3[1][:-1])

    for i in range(Feature_count):
        outfile.write("@ATTRIBUTE " + str(i) + " NUMERIC\n")
    outfile.write("@ATTRIBUTE class {benign, malware}\n")
    outfile.write("@data\n")
    for filename in os.listdir(benign_dir):
        full_path = str(benign_dir) +"/"+ str(filename)
        if os.path.isfile(full_path):
            write = False
            a = bytedistrib(full_path)
            b = []
            if len(a) > Feature_count:
                write = True
                for i in range(Feature_count):
                    b += [a[i]]
                line = ""
                for x in b:
                    line += str(x) + ", "

                outputdir[full_path] = line

        if write == True:
            if full_path in mal_files:
                outputdir[full_path] += "malware\n"

            else:
                outputdir[full_path] += "benign\n"

    for key in outputdir:
        outfile.write(outputdir[key])
```

D. PROGRAM TO GENERATE REDUCED-ATTRIBUTE 8-BIT DATA

```
import sys, math, hashlib, os

def endclean(OS):
    S = OS
    M = len(S)
    while ((M > 0) and \
           (S[M-1] in ['\n','\r','\t','\f','\a','\b','\v',' ']) ):
        S = S[0:M-1]
        M = M-1
    return S

def bytedistrib(filename):
    bytedist = [0 for i in range(256)]
    count = 0
    fid = open(filename, 'rb')
    byte = fid.read(1)
    ByteLib = {}
    features_List = []
    while byte:
        k = int(ord(byte))
        bytedist[k] = bytedist[k] + 1
        byte = fid.read(1)
        count = count + 1
    fid.close()
    #print('Analyzing file',filename,'pf',count,'bytes')
    #print('Byte distribution:',bytedist)
    for i in list(range(256)):
        a = bytedist[i]
        if a not in ByteLib:
            ByteLib[a] = [i]
        else:
            ByteLib[a] += [i]

    for i in range(1,count):
        if i in ByteLib:
            features_List += ByteLib[i]
    #features_list returns a list of bigrams which
    # had the lowest non 0 count in the file (smallest keys)
    return features_List


if __name__=="__main__":
    benign_dir = sys.argv[1]
    mal_list = sys.argv[2]
    out = sys.argv[3]
    outputdir = {}
    mal1 = open(mal_list, "r", encoding = "UTF-8")
    mal2 = mal1.readlines()
    mal_files = set()
    Feature_count = int(sys.argv[4])
    outfile = open(out, 'w')
    outfile.write("@RELATION SingleByte "+str(Feature_count)+"Feature\n")

    for line in mal2:
        mal3 = line.split("|")
        mal_files.add(mal3[1][:-1])

    for i in range(Feature_count):
        outfile.write("@ATTRIBUTE " + str(i) + " NUMERIC\n")
    outfile.write("@ATTRIBUTE class {benign, malware}\n")
    outfile.write("@data\n")
    for filename in os.listdir(benign_dir):
        full_path = str(benign_dir) +"/"+ str(filename)
        if os.path.isfile(full_path):
            write = False
            a = bytedistrib(full_path)
            b = []
            if len(a) > Feature_count:
                write = True
                for i in range(Feature_count):
                    b += [a[i]]
                line = ""
                for x in b:
                    line += str(x) + ", "

                outputdir[full_path] = line

        if write == True:
            if full_path in mal_files:
                outputdir[full_path] += "malware\n"
            else:
                outputdir[full_path] += "benign\n"

    for key in outputdir:
        outfile.write(outputdir[key])
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

Adkins, F., Jones, L., Carlisle, M., & Upchurch, J. (2013). Heuristic malware detection via basic block comparison. *2013 8th International Conference on Malicious and Unwanted Software: The Americas (MALWARE),*11–18. https://doi.org/10.1109/MALWARE.2013.6703680

Anderson, T., & Dahlin M. (2011). *Operating systems principles and practice* (2nd ed.). Charleston, SC: Recursive Books, Ltd.

Benjamin, V., & H. Chen. (2013). Machine learning for attack vector identification in malicious source code. *2013 IEEE International Conference on Intelligence and Security Informatics (ISI),* 21–23. https://doi.org/10.1109/ISI.2013.6578779

Cabau, G., Buhu, M., & Oprisa, C. (2016). Malware classification based on dynamic behavior. *2016 18th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC),* 315–318. https://doi.org/10.1109/SYNASC.2016.057

Carrier, B. (2018). The Sleuth Kit (Version 4.6.4) [Computer Program]. Retrieved from https://www.sleuthkit.org

Casey, W., & Shelmire, A. (2014.). *Signature limits: An entire map of clone features and their discovery in nearly linear time* (Report Num. arXiv:1407.2877). Retrieved from arXiv website: https://arxiv.org/abs/1407.2877

Choi, J., Kim, H., Choi, C., & Kim, P. (2011). Efficient malicious code detection using n-gram analysis and SVM. *2011 14th International Conference on Network-Based Information Systems*, 618–621. https://doi.org/10.1109/NBiS.2011.104

Fuyong, Z., & Tiezhu, Z. (2017.). Malware detection and classification based on n-grams attribute similarity. *2017 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC), 1*, 793–796. https://doi.org/10.1109/CSE-EUC.2017.157

Garfinkel, S. (2009). Automating Disk Forensic Processing with SleuthKit, XML and Python. *2009 Fourth International IEEE Workshop on Systematic Approaches to Digital Forensic Engineering*, 73–84. https://doi.org/10.1109/SADFE.2009.12

Garfinkel, S., Farrell, P., Roussev, V., & Dinolt, G. (2009). Bringing science to digital forensics with standardized forensic corpora. *Digital Investigation*, *6*(S), S2–S11. https://doi.org/10.1016/j.diin.2009.06.016

Hassen, M., Carvalho, M., & Chan, P. (2017). Malware classification using static analysis based features. *2017 IEEE Symposium Series on Computational Intelligence (SSCI),* 1–7. https://doi.org/10.1109/SSCI.2017.8285426

Hongyuan Qiu, F., & Osorio, F. (2013). Static malware detection with segmented sandboxing. *2013 8th International Conference on Malicious and Unwanted Software: The Americas (MALWARE),* 132–141. https://doi.org/10.1109/MALWARE.2013.6703695

Kalash, M., Rochan, M., Mohammed, N., Bruce, N., Wang, Y., & Iqbal, F. (2018). Malware classification with deep convolutional neural networks. *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS),* 1–5). https://doi.org/10.1109/NTMS.2018.8328749

Kargaard, J., Drange, T., Kor, A., Twafik, H., & Butterfield, E. (2018.). Defending IT systems against intelligent malware. *2018 IEEE 9th International Conference on Dependable Systems, Services and Technologies (DESSERT)*, 411–417. https://doi.org/10.1109/DESSERT.2018.8409169

Library of Congress. (2017, February 27). Expert witness disk image format (EWF) family. Retrieved from https://www.loc.gov/preservation/digital/formats/fdd/fdd000406.shtml

Mira, F., Huang, W., & Brown, A. (2017). Improving malware detection time by using RLE and N-gram. *2017 23rd International Conference on Automation and Computing (ICAC),* 1–5. https://doi.org/10.23919/IConAC.2017.8082001

O'Kane, P., Sezer, S., & Mclaughlin, K. (2015). N-gram density based malware detection. *2014 World Symposium on Computer Applications and Research (WSCAR),* 1–6. https://doi.org/10.1109/WSCAR.2014.6916806

Rowe, N. (2015a). Bytedistrib.py [Computer Program] Retrieved from http://faculty.nps.edu/ncrowe/coursematerials/bytedistrib.py.txt

Rowe, N. (2015b). Finding contextual clues to malware using a large corpus. *2015 IEEE Symposium on Computers and Communication (ISCC)*, 229–236. https://doi.org/10.1109/ISCC.2015.7405521

Rowe, N. (2016). Identifying forensically uninteresting files in a large corpus. *EAI Endorsed Transactions on Security and Safety*, *3*(7), 1-15. 10.4108/eai.8-12-2016.151725

Russell, S., & Norvig, P. (2010). *Artificial intelligence: a modern approach* (3rd ed.). Upper Saddle River, NJ: Prentice Hall.

Sarantinos, N., Benzaid, C., Arabiat, O., & Al - Nemrat, A. (2016). Forensic malware analysis: The value of fuzzy hashing algorithms in identifying similarities. *2016 IEEE Trustcom/BigdataSE/ISPA, 1782-1787.* http://roar.uel.ac.uk/5710/1/Forensic%20Malware%20Analysis.pdf

Seideman, J., Khan, B., & Vargas, A. (2014). Identifying malware genera using the Jensen-Shannon distance between system call traces. *2014 9th International Conference on Malicious and Unwanted Software: The Americas (MALWARE)*, 1–7. https://doi.org/10.1109/MALWARE.2014.6999409

Sikorski, M., & Honig, A. (2012). *Practical malware analysis the hands-on guide to dissecting malicious software.* San Francisco, CA: No Starch Press.

Singh, N., & Khurmi, S. (2016.). ByteFreq: Malware clustering using byte frequency. *2016 5th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO),* 333–337. https://doi.org/10.1109/ICRITO.2016.7784976

United States Cyber Command. (2018). Achieve and Maintain Cyberspace Superiority: Command Vision for US Cyber Command USCYBERCOM Fort Meade, MD. Retrieved from https://www.cybercom.mil/Portals/56/Documents/USCYBERCOM%20Vision%20April%202018.pdf?ver=2018-06-14-152556-010

Usaphapanus, P., & Piromsopa, K. (2017). Classification of computer viruses from binary code using ensemble classifier and recursive feature elimination. *2017 Twelfth International Conference on Digital Information Management (ICDIM)*, 27–31. https://doi.org/10.1109/ICDIM.2017.8244670

Witten, I., Frank, E., & Hall, M. (2011). *Data mining practical machine learning tools and techniques* (3rd ed.). Amsterdam: Elsevier/Morgan Kaufmann.

Yu, S., Zhou, S., Liu, L., Yang, R., & Luo, J. (n.d.). Malware variants identification based on byte frequency. *2010 Second International Conference on Networks Security, Wireless Communications and Trusted Computing*, *2*, 32–35. https://doi.org/10.1109/NSWCTC.2010.145

Zak, R., Raff, E., & Nicholas, C. (n.d.). What can N-grams learn for malware detection?. *2017 12th International Conference on Malicious and Unwanted Software (MALWARE*), 109–118. https://doi.org/10.1109/MALWARE.2017.8323963

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California