Authorship Attribution of Compiled Malware

John Montgomery

A paper submitted to the

Computer Science & Electrical Engineering Department

in partial fulfillment of the requirements for the M.S. degree at

University of Maryland Baltimore County

**May 5, 2014**

Certified by: Charles Nicholas

Advisor’s Signature:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Date\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## Abstract

In the mostly anonymous world of the Internet, hackers and cyber criminals may often perpetrate crimes with little to no traces leading back to them. Often this is done using malware that the criminal has created. However, researchers can often obtain copies of this malware, and identifying information may be present within these binaries, even if only unintentionally by the authors. Within this project, I analyze various malware binaries using authorship attribution techniques to determine if there are features that can set malicious binaries apart from benign binaries, or families of malware from other malware. We use n-gram analysis of the binaries’ bytes, as well as frequency distribution of opcodes and n-gram analysis of opcodes to show that there is an identifying feature set that can be taken advantage of to show distinct authorship between files.

Table of Contents

Abstract 2

Introduction 4

Background 4

Authorship Attribution 4

N-gram Analysis 5

Source Code Analysis 6

Methodology 7

Datasets Used 7

N-gram Analysis of Binaries 7

Opcode Frequency Distribution 8

N-gram Analysis of Opcodes 8

Results 9

N-gram Analysis of Binaries 9

Opcode Frequency Distribution 10

N-gram Analysis of Opcodes 10

Conclusions 11

Future Work 11

Acknowledgments 12

References 13

Appendix A: Code for Byte-wise N-grams 15

Appendix B: Code for Opcode N-grams 17

Appendix C: Code for Cosine Scoring Comparison of Distributions 19

Appendix D: Driver for Executing Experiment Workflow 21

## Introduction

Since the advent of the Internet, humanity has found many ways to use it to increase their productivity, comfort, and enjoyment. However, with all of these benefits we've reaped, some have dedicated their efforts towards undermining others by creating and distributing malware. Anti-virus company Symantec has claimed to have detected over a million unique pieces of malware as of 2008, and the amount of malware has only increased since then [Symantec, 2008].

The fact that these digital attacks can originate from any computer on the globe, with plausible deniability for the owner of that computer, being able to attribute these attacks back to the people who created them would be a useful technique for law enforcement officers and security researchers, on par with fingerprinting and handwriting analysis for attributing physical crimes to criminals. The volume of these digital attacks has steadily increased each year, and thus, automated methods for associating these tools to authors must be created, as attribution of all of these attacks by individual malware researchers is infeasible.

## Background

### Authorship Attribution

The quantitative study of authorship is known as stylometrics. This field has been used in literary fields since 1887 beginning with an analysis of the works of Charles Dickens and William Thackeray, two contemporary authors [Mendenhall, 1887]. This experiment used word length analysis of Dicken’s and Thackeray’s work to illustrate that a simple measure is enough to distinguish two similar authors. While the semantic aspect of these documents was never analyzed in this study, the lexical information alone proved to be enough evidence to support the theory that word length alone could disambiguate one other from another.

This counter-intuitive notion that the meaning of words is of little consequence appears frequently in authorship attribution. One of the key problems that causes this is that if all of a document’s data is used in analysis, much of the document’s domain-specific data will cause a natural skew even when the author is the same. For example, if one author’s story contains several characters, and the author writes a different story with an entirely different set of characters, the disjoint of names might cause the analysis to conclude that there are two authors, or that one of the stories is written by another author who has a story that coincidentally has the same named characters [Binongo, 2003]. Thus, while the addition of more test data is often useful, we must also exercise feature selection to attempt to condense our data to the most useful subset of data.

The field of stylometrics gained more traction with an analysis of the disputed articles in The Federalist Papers [Mosteller & Wallace, 1963]. James Madison, Alexander Hamilton, and John Jay were all believed to be the authors of some of these papers published in The Federalist. Researchers used Bayesian analysis to create models of what each of the three authors undisputed works looked like, then applied these models to the disputed papers to assert which were written by each man, and even which were collaborations between each man. Their analysis again eschewed semantic information because of domain issues, and instead chose to focus on what they called “filler words,” such as articles, prepositions, and conjunctions. The frequency of the usage of these words was enough information to disambiguate these three authors. Later research using other methods corroborates these results [Tweedie et al., 1996].

Another study performed more recently that analyzed the authorship of the 15th book in the Oz series of books. L. Frank Baum passed away during the writing of the 15th book, and the authorship of the subsequent books in the series fell to Ruth Thompson. Many believed that the 15th book should be credited to Thompson instead of Baum, which occurred initially. Again, researchers ignored the semantics of the text, instead focusing on what they decided to call “function words” such as auxiliary verbs, pronouns, prepositions, articles, conjunctions and degree adverbs [Binongo, 2003]. They then performed an additional step of principal component analysis (PCA) to reduce the many variables into a two-dimensional plane for simple visual analysis by humans. Their visualizations show a clear and distinct difference between Baum’s and Thompson’s works, and the 15th book of Oz falls solidly in the area of Thomspon’s works.

### N-gram Analysis

At its core, authorship attribution is a specialized area of document comparison, focusing on the source of the documents, rather than the emotion, topic, or complexity of the document. Thus, we expect that many techniques that are applicable to document comparison might be applicable in some way to binary authorship attribution.

One simple, yet effective, technique that is often used is known as n-gram analysis. The document is parsed into overlapping token groups of size n. The tokens are usually words or characters, but other tokens have been used in some areas of research. Frequency counts of the n-grams of a document are kept and normalized to the unit vector for comparison to other documents’ sets of n-grams. The similarity comparison is also applicable in cases when there are small changes in words or phrases between the two documents. For example, consider the words “described” and “describes” and their character 3-grams. All of their 3-grams will be the same until the last 3-gram, when one will be “bed” and the other will be “bes.” Thus, even though they are different words, there is still a high degree of similarity between the two words that is indicative of their shared root.

One issue with n-grams is in choosing an acceptable value for n. For very small values of n, n-grams approach frequency analysis. For large values of n, the distributions will consist entirely of hapax legomena, n-grams that appear only once, and the distribution will be overfit to the data. For most topics, a value of n in the range of two to six will often be acceptable.

N-gram analysis has been used in authorship attribution with great success [Kešelj et al., 2003], and has been used in other similar fields such as text categorization with comparable success [Cavnar & Trenkle, 1994]. It is an effective and efficient tool that has proven useful in numerous comparative cases [Houvardas & Stamatatos, 2006].

### Source Code Analysis

Source code contains all of the working parts of the program as the programmer intended it, complete with comments describing parts of the code in plain language. Data of this type is the gold standard for analyzing programs. However, in an environment where the program in question is malicious, many times we will have only the source code as reverse engineered from the compiled malware, missing its comments and descriptive variable names and optimized by the compiler. In the worst case, we have only fragments of the machine code that have been recovered from the memory of the system it resided upon.

Human analysis of source code can yield interesting insights that might not be discovered by a computer, but it also requires the time and effort of a human that might be better used elsewhere. Many methods and techniques have been created to assist and automate the analysis of source code.

Methods span numerous disciplines, including statistics, by using metric histograms, artificial intelligence, by using genetic algorithms to enhance the searching methods of relevant statistics, and natural language processing techniques [Lange & Mancoridis, 2007]. One method that has frequently had results is using n-grams. N-grams have been used to detect new malware [Abou-Assaleh et al., 2004], for classifying malware into different types [Pektas et al., 2011], and for analyzing source code for authorship attribution [Burrows et al., 2009].

While many of these experiments seem similar to our work, they all differ in key ways. Our experiment will be working with files that are already known to be malicious. We will not be concerned with what type of malware we have, but who the author behind the malware is. Our authorship attribution will be using only the binaries from the compiled source code, without decompiling the binary to recover a set of source code that could create the binary.

## Methodology

### Datasets Used

For the data in this project, a collection of viruses from the Zeus family of malware was obtained. We reduced this dataset to only those that were uncompressed, post-installation binaries compiled for an 8086 architecture on Windows. From this set, we randomly chose twenty viruses. Next, we added in 16 Windows executables and DLLs ranging across a wide variety of different functions. Finally, we added in three viruses all from a similar source, all compiled for an 8086 architecture on Windows. We feel that this is a dataset of an acceptable size in light of the fact that many digital attacks are very targeted and the most successful pieces of malware are used infrequently and with great precision, so many instances of an author’s work would be difficult to procure.

### N-gram Analysis of Binaries

From a general human perspective, a compiled binary (which is how most malware is seen) is essentially ciphertext. However, this is not the case, the method to change source code into a compiled binary is essentially a lossy encoding scheme. While a good encryption algorithm would create a different cipher for a program’s function in different places in a program, a compiler would create the same machine code for that function regardless of where it is placed in the source code. However, information like variable names and comments would be lost in the transition to machine code.

There is an underlying structure to the machine code that the computer can read, but it is not readily apparent to the naked eye. Most code attribution studies have used uncompiled source code [Frantzeskou et al., 2006; Spafford & Weeber, 1992; Krsul & Spafford, 1997; Lange & Mancoridis, 2007] as their dataset, since it is the actual text that the author created. Compiling creates a layer of obfuscation between the author's original text and the data at hand. However, unlike a cipher text, this compiled data has a roughly deterministic relationship to the source code, and thus, can be used as if it were the author's original text with only a small loss in accuracy.

This compiled data is in the form of machine code, which lacks much of the stylistic choices such as names of functions and variables (though some of this data can still be recovered, as described later in the future work section), but still retains the functional choices such as program organization and algorithm implementation.

### Opcode Frequency Distribution

Malware is the same as any other program on a computer, except for its malicious activity. This malicious activity might manifest in a different distribution of operations for the CPU to perform. For example, a piece of malware might have many calls to interrupt operations in order to disrupt program flow and obtain control of a privileged process. If it is concerned with cryptographically securing information before sending it back to some sort of controller, there might be an above-average amount of shift operations and operations to send data to external devices.

These opcodes are derived from the same dataset as above by simply taking a variable length subset of the bytes. We will use only the opcodes, not the registers and other arguments passed to the opcodes. We believe that this feature selection process will provide us with a more effective disambiguation of the data into their respective classes.

For ease of programming, we use the Unix objdump utility to obtain a textual representation of all of these opcodes from the code section of the programs. All programs are compiled for the same architecture (8086) so they all draw from the same set of CPU instructions.

### N-gram Analysis of Opcodes

While the frequency analysis of opcodes may show distinguishing characteristics between different binaries, it lacks any knowledge about how these opcodes are arranged together to create an entire binary. Thus, we will also perform an n-gram analysis of these opcodes to discover if these binaries have distinguishing characteristics that can set them apart from others. Since many viruses rely on specific techniques that are rarely if ever seen in other files, we expect we may be able to disambiguate them based on these code segments. Some malware will specifically embed machine code into their source code, and this human-optimized code may be very different from what a compiler may have created from comparable source code.

## Results

For the comparison of all of these n-gram and frequency distributions, the distributions were normalized to the unit vector, then compared using a cosine similarity score. The average of all of these cosine scores for the cross product of each of our three chosen classes was then computed to show the similarity score between the classes writ large. To illustrate this process more clearly, consider two sets of malware n-grams, A and B, each representing different classes such as Zeus, other viruses, or benign, such that A consists of a1 to ai where each element a is a vector of n-grams, and B consists of n-gram vectors b1 to bj. Thus, the average cosine similarity score between class A and class B is computed by the following formula, where cos() is the cosine similarity function.

CodeCogsEqn.gif

### N-gram Analysis of Binaries

The n-gram analysis of the bytes of these binaries showed little variation between the types of malware we had available to us. One such problem that contributed to this is the size of the search space. For each byte, there are 255 distinct values, and the range of values given an n-gram of size n is 255n. At that size, many of the programs when converted to 5-grams had a frequency distribution that consisted almost entirely of hapax legomena. As we can see from the data, this method was not particularly effective for many values of n, and since the entirety of each binary was being analyzed, was computationally expensive compared to our later techniques.

|  |  |  |  |
| --- | --- | --- | --- |
| **Average Cosine Score of 1-grams** | | | |
|  | Zeus | Other Viruses | Benign |
| Zeus | 90.87% | 43.96% | 84.72% |
| Other Viruses | 43.96% | 99.97% | 39.41% |
| Benign | 84.72% | 39.41% | 81.82% |

|  |  |  |  |
| --- | --- | --- | --- |
| **Average Cosine Score of 3-grams** | | | |
|  | Zeus | Other Viruses | Benign |
| Zeus | 90.94% | 90.77% | 87.09% |
| Other Viruses | 90.77% | 94.50% | 88.41% |
| Benign | 87.09% | 88.41% | 85.96% |

|  |  |  |  |
| --- | --- | --- | --- |
| **Average Cosine Score of 5-grams** | | | |
|  | Zeus | Other Viruses | Benign |
| Zeus | 89.01% | 89.78% | 86.94% |
| Other Viruses | 89.78% | 91.24% | 89.84% |
| Benign | 86.94% | 89.84% | 86.14% |

There was some disambiguation when we used 1-grams, but only between some of the classes, and thus, this technique does not seem to generalize well across domains.

### Opcode Frequency Distribution

The frequency of opcodes within a program showed roughly similar results as the 1-gram of bytes distribution, with some disambiguation appearing, but not a clear split across all classes.

|  |  |  |  |
| --- | --- | --- | --- |
| **Frequency Distribution of Opcodes** | | | |
|  | Zeus | Other Viruses | Benign |
| Zeus | 71.72% | 72.65% | 66.37% |
| Other Viruses | 72.65% | 92.44% | 82.35% |
| Benign | 66.37% | 82.35% | 87.69% |

### N-gram Analysis of Opcodes

As we proceed from a strict frequency distribution to an analysis that takes more of the context of the program into account, we begin to see a clearer split between each of the binary classes we are concerned with.

|  |  |  |  |
| --- | --- | --- | --- |
| **Average Cosine Score of 2-grams** | | | |
|  | Zeus | Other Viruses | Benign |
| Zeus | 51.86% | 48.49% | 42.31% |
| Other Viruses | 48.49% | 83.97% | 62.58% |
| Benign | 42.31% | 62.58% | 71.40% |

|  |  |  |  |
| --- | --- | --- | --- |
| **Average Cosine Score of 4-grams** | | | |
|  | Zeus | Other Viruses | Benign |
| Zeus | 18.36% | 14.63% | 17.08% |
| Other Viruses | 14.63% | 73.75% | 28.25% |
| Benign | 17.08% | 28.25% | 38.51% |

|  |  |  |  |
| --- | --- | --- | --- |
| **Average Cosine Score of 3-grams** | | | |
|  | Zeus | Other Viruses | Benign |
| Zeus | 34.23% | 27.88% | 26.52% |
| Other Viruses | 27.88% | 78.33% | 44.25% |
| Benign | 26.52% | 44.25% | 53.56% |

|  |  |  |  |
| --- | --- | --- | --- |
| **Average Cosine Score of 5-grams** | | | |
|  | Zeus | Other Viruses | Benign |
| Zeus | 16.62% | 7.84% | 11.37% |
| Other Viruses | 7.84% | 70.87% | 17.36% |
| Benign | 11.37% | 17.36% | 27.69% |

We can see that there is a much clearer split between the different classes for many values of n. The most consistent split seems to occur when n equals 3, which is consistent with previous research [Pektas et al., 2011].

## Conclusions

From the results that we have obtained, we can conclude that an analysis of the bytes of binaries without some sort of pre-processing step would cause us to have too much information that would skew our data to appear the same as all other data. However, by restricting ourselves to a subset of the bytes, namely, opcodes, we can begin to discover the results we desire. However, we still must have some contextual information available, and thus individual opcodes would not be sufficient for proper authorship attribution.

## Future Work

Future work in this field will become more important as digital warfare becomes more widespread and damaging. Hackers will escalate their tools and techniques to counteract whatever countermeasures security researchers come up with, so security researchers should always be creating newer and better techniques. One area that we believe has promise is in analyzing the text strings found within files that often contain constants to be used within the program and function names. While there are some counters to this analysis already, such as the munging of important strings, we believe that many malware creators will not bother to create new munging techniques or keys for each new piece of malware or version of malware. Thus, we will begin to see patterns in the ciphertext that is contained within several programs.

Methods that could assist and automate aspects of a security researcher’s analysis of malware would also be an excellent focus for future work. Though the cost of analyzing each piece of malware is prohibitively expensive, the most important and devastating pieces of malware will be analyzed by humans, and any analysis of the humans’ processes and subsequent improvement of this work would be a boon for security research.

## Acknowledgments

The author wishes to thank his advisor, Charles Nicholas, for his assistance in creating this research paper.

## References

Abou-Assaleh, Tony, et al. "N-gram-based detection of new malicious code." *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*. Vol. 2. IEEE, 2004.

Binongo, J. N. G. (2003). Who wrote the 15th book of Oz? An application of multivariate analysis to authorship attribution. Chance, 16(2), 9-17.

Burrows, S., Uitdenbogerd, A. L., & Turpin, A. (2009, January). Application of information retrieval techniques for source code authorship attribution. In Database Systems for Advanced Applications (pp. 699-713). Springer Berlin Heidelberg.

Cavnar, W. B., & Trenkle, J. M. (1994). N-gram-based text categorization. *Ann Arbor MI*, *48113*(2), 161-175.

Frantzeskou, G., MacDonell, S., Stamatatos, E., & Gritzalis, S. (2008). Examining the significance of high-level programming features in source code author classification. Journal of Systems and Software, 81(3), 447-460.

Frantzeskou, G., Stamatatos, E., Gritzalis, S., & Katsikas, S. (2006, May). Effective identification of source code authors using byte-level information. In Proceedings of the 28th international conference on Software engineering (pp. 893-896). ACM.

Houvardas, J., & Stamatatos, E. (2006). N-gram feature selection for authorship identification. In *Artificial Intelligence: Methodology, Systems, and Applications* (pp. 77-86). Springer Berlin Heidelberg.

Kešelj, V., Peng, F., Cercone, N., & Thomas, C. (2003, August). N-gram-based author profiles for authorship attribution. In *Proceedings of the conference pacific association for computational linguistics, PACLING* (Vol. 3, pp. 255-264).

Krsul, I., & Spafford, E. H. (1997). Authorship analysis: Identifying the author of a program. *Computers & Security*, *16*(3), 233-257.

Juola, P. (2006). Authorship attribution. Foundations and Trends in information Retrieval, 1(3), 233-334.

Lange, R. C., & Mancoridis, S. (2007, July). Using code metric histograms and genetic algorithms to perform author identification for software forensics. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation* (pp. 2082-2089). ACM.

Mendenhall, T. C. (1887). The characteristic curves of composition. Science, (214S), 237-246.

Mosteller, F., & Wallace, D. L. (1963). Inference in an authorship problem: A comparative study of discrimination methods applied to the authorship of the disputed Federalist Papers. *Journal of the American Statistical Association*, *58*(302), 275-309.

Pektas, A., Eris, M., & Acarman, T. (2011, August). Proposal of n-gram based algorithm for malware classification. In SECURWARE 2011, The Fifth International Conference on Emerging Security Information, Systems and Technologies (pp. 14-18).

Spafford, E. H., & Weeber, S. A. (1993). Software forensics: Can we track code to its authors?. *Computers & Security*, *12*(6), 585-595.

Stamatatos, E. (2009). A survey of modern authorship attribution methods. Journal of the American Society for information Science and Technology, 60(3), 538-556.

Tweedie, F. J., Singh, S., & Holmes, D. I. (1996). Neural network applications in stylometry: The Federalist Papers. *Computers and the Humanities*, *30*(1), 1-10.

Symantec Corporation. (2008). Trends for July-December 07. *Symantec Global Internet Security Threat Report*.

## Appendix A: Code for Byte-wise N-grams

#! /usr/bin/env python3

import argparse

import sys

import json

import math

import binascii

from collections import defaultdict

import logging

def main():

parser = argparse.ArgumentParser(description="Create byte-wise n-gram distribution",)

parser.add\_argument('-n', default=1, help="Size of n-grams")

parser.add\_argument('-o', '--output', default=None, help="File to write output to")

parser.add\_argument('input', help="The file to be n-gram analyzed")

parser.add\_argument('--normalize', default=True, help="Normalize the vector before outputting", action="store\_true")

parser.add\_argument('-d', '--debug', default=False, action="store\_true")

args = parser.parse\_args()

if args.debug:

logging.basicConfig(level=logging.DEBUG)

else:

logging.basicConfig(level=logging.ERROR)

file\_name = args.input

n = int(args.n)

output = args.output

distribution = defaultdict(int)

with open(file\_name, "rb") as f:

f.seek(0,2)

end = f.tell()

f.seek(0)

bytes = f.read(n)

while f.tell() != end:

bytestring = binascii.hexlify(bytes)

distribution[bytestring] += 1

f.seek(1-n, 1)

bytes = f.read(n)

logging.debug(distribution)

if output is None:

if args.normalize:

for key, value in sorted(normalize(distribution).items()):

print("%s %s" % (key, value))

else:

print(distribution)

#print(json.dumps(distribution))

else:

outfile = open(output, "w")

if args.normalize:

for key, value in sorted(normalize(distribution).items()):

outfile.write("%s %s\n" % (key, value))

else:

outfile.write(str(distribution))

def normalize(vector):

sum = 0.0

result = dict()

for value in vector.values():

sum += value\*\*2

sum = math.sqrt(sum)

for key in vector.keys():

result[key] = vector[key]/sum

return result

if \_\_name\_\_ == "\_\_main\_\_":

# I like defining functions at the end, so we're using this so I can do that

main()

## Appendix B: Code for Opcode N-grams

#! /usr/bin/env python3

import argparse

import sys

import json

import math

import binascii

from collections import defaultdict

import logging

def main():

parser = argparse.ArgumentParser(description="Create op-code-wise n-gram distribution",)

parser.add\_argument('-n', default=1, help="Size of n-grams")

parser.add\_argument('-o', '--output', default=None, help="File to write output to")

parser.add\_argument('input', help="The .obj file to be n-gram analyzed")

parser.add\_argument('--normalize', default=True, help="Normalize the vector before outputting", action="store\_true")

parser.add\_argument('-d', '--debug', default=False, action="store\_true")

args = parser.parse\_args()

if args.debug:

logging.basicConfig(level=logging.DEBUG)

else:

logging.basicConfig(level=logging.ERROR)

file\_name = args.input

n = int(args.n)

output = args.output

distribution = defaultdict(int)

gram = list()

with open(file\_name, "r") as f:

for line in f:

if len(gram) == n:

gram.remove(gram[0])

gram.append(line.strip())

if len(gram) < n:

continue

else:

distribution['["'+'","'.join(gram)+'"]'] += 1

logging.debug(distribution)

if output is None:

if args.normalize:

for key, value in sorted(normalize(distribution).items()):

print('%s %s' % (key, value))

else:

print(distribution)

#print(json.dumps(distribution))

else:

outfile = open(output, "w")

if args.normalize:

for key, value in sorted(normalize(distribution).items()):

outfile.write('%s %s\n' % (key, value))

else:

outfile.write(str(distribution))

def normalize(vector):

sum = 0.0

result = dict()

for value in vector.values():

sum += value\*\*2

sum = math.sqrt(sum)

for key in vector.keys():

result[key] = vector[key]/sum

return result

if \_\_name\_\_ == "\_\_main\_\_":

# I like defining functions at the end, so we're using this so I can do that

main()

## Appendix C: Code for Cosine Scoring Comparison of Distributions

#! /usr/bin/env python3

import math

import argparse

from collections import defaultdict

import logging

def main():

parser = argparse.ArgumentParser(description="Compare two clusters for similarity",)

parser.add\_argument('-f', '--first', help="First file")

parser.add\_argument('-s', '--second', help="Second file")

parser.add\_argument('-d', '--debug', action="store\_true",

help="Print debug messages")

args = parser.parse\_args()

if args.debug:

logging.basicConfig(level=logging.DEBUG)

else:

logging.basicConfig(level=logging.ERROR)

logging.debug("Retrieving arguments")

file1 = open(args.first, "r")

file2 = open(args.second, "r")

logging.debug("Start reading files")

#We could be a bit safer by parsing each argument separately with the

#proper functions, but this works as well and it's more fun to look at.

n\_gram1 = [eval(x) for x in file1.readline().strip().split()]

n\_gram2 = [eval(x) for x in file2.readline().strip().split()]

numerator = 0

denom1 = 0

denom2 = 0

while n\_gram1 or n\_gram2:

logging.debug("Processing %s and %s" % (n\_gram1, n\_gram2))

if not n\_gram1:

#process the n-gram from the second file

denom2 += n\_gram2[1]\*\*2

n\_gram2 = [eval(x) for x in file2.readline().strip().split()]

elif not n\_gram2:

#process the n-gram from the first file

denom1 += n\_gram1[1]\*\*2

n\_gram1 = [eval(x) for x in file1.readline().strip().split()]

elif n\_gram1[0] < n\_gram2[0]:

#process the n-gram from the first file

denom1 += n\_gram1[1]\*\*2

n\_gram1 = [eval(x) for x in file1.readline().strip().split()]

elif n\_gram2[0] < n\_gram1[0]:

#process the n-gram from the second file

denom2 += n\_gram2[1]\*\*2

n\_gram2 = [eval(x) for x in file2.readline().strip().split()]

elif n\_gram1[0] == n\_gram2[0]:

#process both of the n\_grams

numerator += n\_gram1[1] \* n\_gram2[1]

denom1 += n\_gram1[1]\*\*2

denom2 += n\_gram2[1]\*\*2

n\_gram1 = [eval(x) for x in file1.readline().strip().split()]

n\_gram2 = [eval(x) for x in file2.readline().strip().split()]

else:

#How did we get here? Clearly an error.

logging.error("Got to a conditional that'd probably be impossible."

" N-gram values were %s and %s" % (n\_gram1, n\_gram2))

file1.close()

file2.close()

print(numerator/(math.sqrt(denom1)\*math.sqrt(denom2)))

if \_\_name\_\_ == "\_\_main\_\_":

main()

## Appendix D: Driver for Executing Experiment Workflow

#! /usr/bin/env python3

from argparse import ArgumentParser

from subprocess import Popen, call, check\_output

from itertools import combinations

from collections import defaultdict

import csv

import logging

parser = ArgumentParser(description="Compare many binaries to each other",)

parser.add\_argument('-f', '--file', help="File listing binaries, one per line",

required=True)

parser.add\_argument('-o', '--output', help="The base filename for the outputs",

required=True)

parser.add\_argument('-n', '--number', help="The n value for n-grams",

required=True)

parser.add\_argument('--opcode', help="Use op-code n-gram parser",

action="store\_true")

parser.add\_argument('-d', '--debug', action="store\_true",

help="Print debug messages")

args = parser.parse\_args()

if args.debug:

logging.basicConfig(level=logging.DEBUG)

else:

logging.basicConfig(level=logging.ERROR)

logging.debug("Executing cross\_compare\_driver.py")

input\_file = open(args.file, "r")

n = args.number

output\_file = open(args.output+"."+n+".csv", "w")

file\_list = list()

opcode = args.opcode

grams = list()

files = list()

# Read through the input file and run each binary through the n-gram program

for line in input\_file:

line = line.strip()

# Run through n-gram parser

out\_name = line+"."+n+"gram"

print("Making n-grams for "+line)

if opcode:

cmd = ["./n\_grams\_ops.py", "-n", n, "-o", out\_name, line]

else:

cmd = ["./n\_grams\_bytes.py", "-n", n, "-o", out\_name, line]

if args.debug:

cmd.append("-d")

call(cmd)

grams.append(out\_name)

files.append(out\_name)

input\_file.close()

table = defaultdict(dict)

#Now, run each binary against every other binary

for first, second in combinations(grams, 2):

print(first+" x "+second)

cmd = ["./compare\_cluster.py", "-f", first, "-s", second]

if args.debug:

cmd.append("-d")

output = float(check\_output(cmd))

table[first][second] = output

table[second][first] = output

table[first][first] = 1.0

table[second][second] = 1.0

cw = csv.writer(output\_file)

files.insert(0, "")

#Write the header to the file

cw.writerow(files)

del files[0]

#Go through the table and print out the cosine scores to fit a table.

logging.debug(str(table))

logging.debug(str(files))

for key in files:

line = list()

line.append(key)

for item in files:

logging.debug("%s, %s" % (key, item))

line.append(table[key][item])

cw.writerow(line)

**UNIVERSITY OF MARYLAND, BALTIMORE COUNTY**

***DEPT. OF COMPUTER SCIENCE AND ELECTRICAL ENGG***

***APPROVAL OF SCHOLARLY PAPER***

The following student’s scholarly paper has been read and satisfies the

Master’s degree requirement. A copy of the scholarly paper is attached.

Date: \_\_\_May 9, 2014\_\_\_\_

\_\_Montgomery, John\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_jmont1\_\_\_\_\_\_

Student name (Last name, First name) UMBC Email ID

Title of Scholarly Paper:

\_\_\_Nicholas, Charles\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Advisor name Advisor signature and date

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Reader name Reader signature and date

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Graduate Program Director name GPD signature and date