2017

The Effect of Code Obfuscation on Authorship Attribution of Binary Computer Files

Steven Hendrikse
Nova Southeastern University, steve@hendrikse.ca

The effect of code obfuscation on authorship attribution of binary computer files

by

Steven Hendrikse

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in
Information Assurance

College of Engineering and Computing
Nova Southeastern University
2017

We hereby certify that this dissertation, submitted by Steven Hendrikse, conforms to acceptable standards and is fully adequate in scope and quality to fulfill the dissertation requirements for the degree of Doctor of Philosophy.

_____ _____

Francisco J. Mitropoulos, Ph.D. Date
Chairperson of Dissertation Committee

_____ _____

James D. Cannady, Ph.D. Date
Dissertation Committee Member

_____ _____

Matthew Tennyson, Ph.D. Date
Dissertation Committee Member

Approved:

_____ _____

Yong X. Tao, Ph.D., P.E., FASME Date
Dean, College of Engineering and Computing

College of Engineering and Computing
Nova Southeastern University

2017

An Abstract of a Dissertation Submitted to Nova Southeastern University
In Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

The effect of code obfuscation on authorship attribution of binary computer files

by
Steven Hendrikse
April 2017

In many forensic investigations, questions linger regarding the identity of the authors of the software specimen. Research has identified methods for the attribution of binary files that have not been obfuscated, but a significant percentage of malicious software has been obfuscated in an effort to hide both the details of its origin and its true intent. Little research has been done around analyzing obfuscated code for attribution.

In part, the reason for this gap in the research is that deobfuscation of an unknown program is a challenging task. Further, the additional transformation of the executable file introduced by the obfuscator modifies or removes features from the original executable that would have been used in the author attribution process.

Existing research has demonstrated good success in attributing the authorship of an executable file of unknown provenance using methods based on static analysis of the specimen file. With the addition of file obfuscation, static analysis of files becomes difficult, time consuming, and in some cases, may lead to inaccurate findings.

This paper presents a novel process for authorship attribution using dynamic analysis methods. A software emulated system was fully instrumented to become a test harness for a specimen of unknown provenance, allowing for supervised control, monitoring, and trace data collection during execution. This trace data was used as input into a supervised machine learning algorithm trained to identify stylometric differences in the specimen under test and provide predictions on who wrote the specimen.

The specimen files were also analyzed for authorship using static analysis methods to compare prediction accuracies with prediction accuracies gathered from this new, dynamic analysis based method. Experiments indicate that this new method can provide better accuracy of author attribution for files of unknown provenance, especially in the case where the specimen file has been obfuscated.

Acknowledgements

I'd like to thank my committee chair, Dr. Frank Mitropoulos, and my committee members, Dr. James Cannady and Dr. Matthew Tennyson, for their guidance, support, and for asking all their hard questions, as these have helped me become a better researcher.

I'd also like to thank my family as they have supported and encouraged me to pursue this lifelong dream.

Table of Contents

List of Tables

Tables

List of Figures

Figures

Chapter 1

Introduction

Background

In human readable files, authorship attribution relies on an analysis of stylometry, vocabulary diversity, word choice, and frequency of function words (Bozkurt, Baghoglu, & Uyar, 2007; Stamatatos, 2009). However, for computer executable files, the process of compilation removes many of the elements of style that were added by the author and replaces them with standardized machine level instructions (Rosenblum, Miller, & Zhu, 2010, 2011).

Laws against copyright infringement and plagiarism have long protected intellectual property that is represented in human readable format. However, in many settings, valuable intellectual property is contained within computer software that is not in a human readable form. The loss of this intellectual property can have significant impact on a company's financial health (Federal Bureau of Investigation, 2014; Jeremy Kirk, 2013; Lilly, 2012). Similarly, malicious executable files (malware) have caused significant damage to individuals and companies alike (Frizell, 2014; Jones, 2014; Jeremy Kirk, 2014). In both cases, without effective means of identifying the original authors of the stolen software or malware, criminal investigations may not be solved (Pierluigi, 2013).

Some research has been done in an effort to create a process or tool that can be used to identify or match the author of a binary file of unknown provenance to either a previously stored author profile or to other samples of known provenance using style

identifiers found within the binary. However, challenges remain in determining authorship in binary executable files, one of which is the difficulty in identifying style characteristics of a particular author after the binary file has been transformed through some means of obfuscation. In the context of binary executable file analysis, the process of obfuscating a file involves the transformation of that file into a file that has identical observable behavior (the behavior as experienced by the user) and that the effort required to analyze or reverse engineer the file back into a form that is readily understood by a human analyst is prohibitively high (Collberg & Thomborson, 2002; Linn & Debray, 2003).

This dissertation report presents the impact of obfuscation of binary executable files in terms of the ability to attribute the author of the file. In addition to summarizing the research specifically covering the authorship attribution of binary files and binary file obfuscations, this dissertation report will summarize the existing research in the areas that are primarily related to the authorship attribution problem in obfuscated binary executable files, including source code authorship attribution, compiler provenance, program and programmer evolution, software plagiarism, executable file obfuscation, binary file disassembly, representation and comparison, and binary file authorship attribution.

The remainder of this chapter is organized into the following sections: Problem Statement; Dissertation Goal; Research Questions; Relevance and Significance; Barriers and Issues; Assumptions, Limitations, and Delimitations; Definition of Terms; List of Acronyms; and Summary. The Problem Statement section defines the specific scope of the problem being addressed by this research. The Dissertation Goal section defines the

goal of the research being addressed in this dissertation. The Research Questions section has a list of qualitative questions that will be investigated in this research and addressed within this dissertation report. The Relevance and Significance section provides justification in support of this research as being worthy of advanced study. The Barriers and Issues section provides insight into the challenges that exist in this field of study and specifically by the research questions posed in this thesis. The Assumptions, Limitations, and Delimitations section will define the scope and context within which this research will exist. The Definition of Terms section defines the key terms that will be frequently used during the course of this research. Similarly, the List of Acronyms section provides an explanation of the frequently used acronyms in this field of research.

Problem Statement

Analysis of authorship of binary executable files having an unknown provenance has not addressed the scenario in which the binary file has been obfuscated in an effort to hide its true functionality, and as such, it is unknown to what extent the features that uniquely identify the authorship of the file are affected by the obfuscation.

This research will use the participant submissions from the Google Code Jam (GCJ) (Google, 2015) contest as the corpus of files to be analyzed for authorship attribution. Submissions from this contest have been used in previous research on this topic (Alrabaee, Saleem, Preda, Wang, & Debbabi, 2014; Caliskan-Islam, Yamaguchi, et al., 2015; Rosenblum, Zhu, & Miller, 2011), allowing for the extension of ideas, comparisons of techniques, and a measurement of any improvements in attribution. In an effort to maintain a level of similarity between this research and past research, a

measurement of attribution of authorship in files that have not been obfuscated will be undertaken in order to define a baseline accuracy of attribution within the specimen corpus.

Existing research has demonstrated that machine learning algorithms, using either supervised or unsupervised methods (or a combination of both), are capable of correctly identifying features and classifying executable file samples based on authorship. However, in all of the previous research studies referenced in this dissertation report, obfuscation of the binary files has been left for future research. Using the malware complexity classification scheme proposed by Ugarte-Pedrero, Balzarotti, Santos, and Bringas (2015), this research will determine the impact of different types of binary file obfuscation on the accuracy of attribution of authorship.

Dissertation Goal

The goal of this dissertation is to accurately describe the effects of executable file obfuscation techniques on the ability to attribute the authorship of a binary executable file of unknown provenance.

Research Questions

This research on the effects of obfuscation and on how these transformations impacts one's ability to attribute the authorship of unknown binary executable files will address the following research questions:

1. What classes of stylometric markers persist though the obfuscation of binary executable files and how do these persistent markers, and the absence of markers that do not persist after obfuscation, affect the accuracy of attribution of authorship?

2. How does the method (static or dynamic) and process of disassembly affect the accuracy of attribution of the authorship in obfuscated binary executable files?
3. What challenges exist in trying to fingerprint and attribute the obfuscation routines that are blended into the obfuscated binary file?
4. How, if at all, does the tool-chain/compiler provenance of a binary program contribute to the determination of authorship as in the case of obfuscated binaries (or binaries that have been de-obfuscated)?
5. How does program and/or programmer evolution, plagiarism, and shared access to source code repositories affect the accuracy of attribution of the authorship of a binary file that has been previously obfuscated?

Relevance and Significance

There is a problem when attributing authorship in binary files that have been obfuscated because the compiler, its optimizing methods, and the obfuscation routines actively strip away stylistic markers that existed in the original source code. These original stylistic markers, like markers found in other types of human readable documents, are typically used to attribute authorship.

There are primarily two scenarios where authorship attribution of binary files that have been obfuscated becomes very important. The first is the scenario of malicious software (malware), like a rootkit, that can cause damage to valuable intellectual property or be used to commit some other crime. Investigators may not be able to determine the identity of the cyber-criminal if the author of the malware cannot be determined. Secondly, in cases where commercial intellectual property is represented in binary files, breaches of non-disclosure agreements and other violations

of copyright may only be successfully resolved when an accurate determination of the legal owner is determined.

The scope of these two scenarios is quite large. Malware represents a global threat that does not appear to be diminishing (DarkReading, 2016; Evans & Scott, 2017; Matthews, 2017). Similarly, intellectual property theft of technology assets can have a staggering impact on the continued success of a commercial organization (Burg, McConkey, & Amra, 2016; Furchtgott-Roth, 2017; Shaw, 2016). If such a method of author attribution on files that have been transformed using some binary obfuscation routine could be realized, the conviction rates in both of these scenarios might increase, which would result in an overall positive change to the Internet environment.

Prior research (Alrabaee et al., 2014; Alrabaee, Shirani, Debbabi, & Wang, 2016; Caliskan-Islam, Yamaguchi, et al., 2015; Rosenblum, Zhu, et al., 2011) exists that has addressed the goal of identifying the authorship of machine readable executable binary files, and a future research opportunity was noted, but not addressed, with potential difficulties in identifying style characteristics of a particular author after the binary file had been transformed through some means of obfuscation. Without addressing this opportunity, the ability to reliably answer authorship questions in obfuscated binary files will remain unfulfilled.

This research will begin to address the challenge of accurately identifying the author of a binary file that has been obfuscated. As a first phase, the effects of obfuscation on binary file authorship attribution will be studied. As described in Collberg, Thomborson, and Low (1997), obfuscation techniques include (but are not limited to) inserting “junk” code (code that has no observable effect on program

execution), modifying loop conditions, removal of library call references, the addition of redundant operands, unrolling (or in-lining) loops, and interleaving multiple methods. Beyond this first phase of research, covered by this Dissertation Report, future phases of research will involve using the observations and conclusions identified herein in order to define a system whereby a binary file that has been transformed by some type of obfuscator will be readily capable of being attributed to a particular author.

Barriers and Issues

Binary exectuable files do not possess the same characteristics as humanly readable source files do. The characteristics that are of interest in this research are those stylistic identifiers that define the work product of a seasoned developer. For instance, stylistic identifiers can include individual preferences on the use of different styles of looping (for instance: do…while (condition) instead of while (condition)…) and choices of preferred combinations/orderings of function calls. As noted above, these stylistic identifiers may be modified or removed once the process of binary obfuscation has been applied to the binary executable. If a sufficient set of stylistic identifiers survive the obfuscation process, the binary program should be able to be attributed to a specific programmer with confidence, be it a piece of malicious code in the context of cyber-crime or be it a component in an enterprise software suite in dispute between commercial entities on who owns that particular piece of intellectual property.

A list of the significant challenges inherent in the attribution of authorship of obfuscated binary executable files is given below. A brief discussion of each challenge follows the list:

1. De-obfuscation may not return the binary to it's pre-obfuscated form,

2. In the face of obfuscation, the process of disassembly is difficult. Small variances in interpretation of executable code may have a significant impact on the disassembled representation of the binary.
3. Program similarity does not imply equivalence of author,
4. It is very easy to copy or share source code, and the practice of sharing is quite typical
5. It is typical in the software industry to hold source code in some type of shared repository. Given the nature of this type of programming, it cannot be definitively ascertained that a particular source code module has had only one author (or contributor), thus clouding or co-mingling the stylistic markers within the file.
6. Different levels of optimization result in markedly different executables, and some optimizations destroy some of the stylistic identifiers needed to attribute authorship.
7. The process of creating a model of an authors style based on previous work output for the purpose of comparing programming style against an unknown binary sample can be difficult, as there may not be an appropriately sized corpus of previous work that can be accurately attributed to a specific author.

De-obfuscation has been the topic of research from both the perspective of analyzing malware samples and developing a tool or procedure that can restore an obfuscated binary file to its original un-obfuscated state. However, in the case of malware analysis, “malware classification seeks to extract characteristics specific to a program or a family of programs with related behavior, while our authorship attribution

techniques must discover more general properties of author style." (Rosenblum, Zhu, et al., 2011, p. 17). Similarly, research in de-obfuscation strategies has been also focused on specific, rather than general properties of files. Experiments will be undertaken in order to identify which stylistic markers survive compilation and obfuscation such that they can be used to provide clues towards attributability of authorship.

As discussed, compiler environment can play a role in the preservation or removal by optimization of characteristics that aid in the identification of an author (H. Chen, 2013). A thorough understanding of how the compilation environment affects the binary output, including methods available to identify the level of optimization of code (Aho, Lam, Sethi, & Ullman, 2006; Haneda, Knijnenburg, & Wijshoff, 2005; Hoste & Eeckhout, 2008; Moseley, Grunwald, & Peri, 2009), will be undertaken to identify additional inputs that can be fed into the classification algorithm to either strengthen or weaken the confidence of classification.

As discussed in Horspool and Marovac (1980), returning to an accurate human readable representation of a computer program is a difficult task. It is difficult for two reasons: one, that instructions and data are stored, interspersed, in the computer's memory; and two, it is difficult to attach the correct semantics to certain classes of data or instructions.

Additional study has been done to determine the efficacy of modern disassemblers and de-compilers, given the aforementioned challenges. Multiple commercially available products have been assessed, and in every case the results were found to be lacking (Baldwin, Sinha, Salois, & Coady, 2011; Bao, Burket, Woo, Turner, & Brumley, 2014; Emmerik & Waddington, 2004; Kinder & Veith, 2008; Paleari,

Martignoni, Fresi Roglia, & Bruschi, 2010; Treude, Storey, & Salois, 2011). The overarching goal of this research, as elucidated in the Research Questions, is to characterize the effects of binary file obfuscation on the accuracy of authorship attribution. To this end, processor instruction data from the specimen executable files will be collected by two methods; one using static analysis methods and the other using a dynamic analysis process. This instruction data will then be used to determine the author of the original binary executable. The two methods of data collection will be compared and contrasted against the research questions in the analysis portion of the research.

While there exists published research in the area of authorship attribution in binary executable files, (Alrabaee et al., 2014; Caliskan-Islam, Yamaguchi, et al., 2015; Rosenblum, Zhu, et al., 2011), there continues to be many opportunities for further research and refinement. The latest of these previous studies has shown that classification can be done accurately without the need to create a distinct author profile for each contributing author, as suggested in previous research. Despite these successful demonstrations of author attribution in binary executable files, none of the existing research has taken the step to address obfuscation in binary files. To this end, an ensemble approach, made up of the best components of the aforementioned research in attribution of authorship, will be used as a baseline of potential accuracy for attribution, prior to the introduction of obfuscation to the speciment executable files.

Assumptions, Limitations, and Delimitations

In as much as the Google Code Jam regulations forbid submissions created by team effort or by plagiarism of another programmer, there is no certainty of the ground

truth assertion that each submission being studied in this research is actually written by the developer who has claimed to have written it. Even though there may be residual concern over the validity of the claim of ground truth authorship, the submissions from this contest have been used in past research with good results. Further, by using the same set of test data, comparisons and conclusions regarding performance and accuracy can be made, thus furthering the body of knowledge in authorship attribution.

A recent longitudinal study of executable packers (another name for an executable file obfuscator) identified 389 unique packers in their study of 685 executable file samples (Ugarte-Pedrero et al., 2015). Even if the full population of executable packers was limited to these 389 unique packers, it would still not be feasible to study the attribution characteristics of each. As an alternative, the packer complexity categories defined in Ugarte-Pedrero et al. (2015) will be used; and three representative samples of each complexity type will be used in this research.

As described in Roundy and Miller (2013), the majority of executable file obfuscators implement additional measures to further thwart reverse engineering beyond just the obfuscation technique itself. Throughout all the experiments, wherever possible, all user defined selections for anti-debug and/or anti-reverse engineer will be set to disabled. By setting these options to disabled, the experiments have the best chance of executing completely and without error within both the static and dynamic analysis environments.

There continues to be debate within the research community involved with attribution of authorship and the minimum size and number of samples required to provide consistently high accuracy of results. Zheng, Li, Chen, and Huang (2006)

showed that regardless of machine learning algorithm, as the number of samples per author increased (from 10 to 30 in increments of five), the accuracy of correct classification increased, however, as the number of candidate authors increased (from five to 20 in increments of five), the accuracy of correct classification decreased. Similarly, Luyckx and Daelemans (2008) demonstrated a decrease in classification accuracy from approximately 96% to 34% when the number of authors in the test went from two to 145. In addition to attribution accuracy challenges with respect to the number of candidate authors, (Koppel, Schler, & Argamon, 2009, 2013) found that the total size of each candidate's sample was highly correlated to the accuracy of future classification attempts. Finally, Argamon et al. (2007) conducted experiments which found that "different kinds of features are needed for different kinds of stylistic text classification, and that addition of irrelevant features often reduce performance" (Argamon et al., 2007, p. 816).

While the findings above are consistent with one's intuition, only a single known study has investigated the effect of sample size on authorship attribution accuracy of binary files. In their research, Caliskan-Islam, Yamaguchi, et al. (2015) performed two sets of experiments, one set mirroring the sample size of previous authorship attribution research (Alrabaee et al., 2014; Rosenblum, Zhu, et al., 2011), and the other set of experiments to determine the optimal training set sample size against the number of candidate programmers. The optimal number of training samples is dependent on the number of candidate authors, but in the case of authorship attribution classification with 20 candidate authors, Caliskan-Islam, Yamaguchi, et al. (2015) found only slight

improvements in classification accuracy with larger sets after a minimum set size of three executables.

Even though most machine learning textbooks consider the determination of optimal training set sample size to be identifiable only through experimentation (Abu-Mostafa, Magdon-Ismail, & Lin, 2012; Hastie, Tibshirani, & Friedman, 2009; Witten, Frank, & Hall, 2011), this research will follow the sample sizes defined in past research in this area (Alrabaee et al., 2014; Caliskan-Islam, Yamaguchi, et al., 2015; Rosenblum, Zhu, et al., 2011). Two different configurations will be used, the first will use eight samples from each of 20 authors to train the classifier, and the second will use 14 samples from each of the 20 authors to train the classifier. For both configurations, testing will follow the training phase and author prediction accuracies will be recorded.

Definition of Terms

1. *Authorship attribution.* The task of determining the author of a document of unknown provenance (Zhao & Zobel, 2007).
2. *Abstract Syntax Tree.* A tree representation of the structure of computer source code that has a node for each important token and uses operators as the sub-tree roots. (Parr, 2009)
3. *Bertillonage*. "A method to reduce the search space when trying to locate a software entity's origin within a corpus of possibilities" (Davies, German, Godfrey, & Hindle, 2011, p. 3). This is typically accomplished by comparing coarser details of candidate possibilities, assuming that coarser detail similarity will imply finer detail similarity.

4. *Bloom Filter* A space-efficient probabilistic data structure that is used to test whether an element is a member of a set. Its primary benefit in this context is to quickly determine whether two control flow graphs are similar. (Bloom, 1970)
5. *BinHunt.* A tool for identifying semantic differences between files based on the control flow of the program using a technique that involved a graph isomorphism technique, symbolic execution, and theorem proving (Gao, Reiter, & Song, 2008).
6. *Call Graph.* A graph representation of a program in which the nodes of the graph represent the procedures of the program and the edges represent one or more executions of the procedure (Ryder, 1979).
7. *C4.5 decision tree*. A machine learning algorithm that chooses the attribute of the data that most effectively splits the set of samples into subsets of the highest similarity. The splitting criterion is the based on the calculated information gain. The attribute with the highest information gain is chosen as the decision point for the current location in the tree. The C4.5 algorithm continues recursively to split each subset (Quinlan, 1986).
8. *Crypter*. A synonym for Obfuscator, popular in hacker cultures.
9. *Cryptic evolution.* The ability of a computer program to hide or obfuscate its file signature. Differences in a file's signature suggest (to automated detection routines) that the file is entirely unique from others that have already been encountered (Iliopoulos, Adami, & Szor, 2011).

10. *Concept drift.* The evolution within a family of malware as a response to outside pressures or the need to implement new features (Kantchelian et al., 2013; Singh, Walenstein, & Lakhotia, 2012).
11. *Control Flow Graph.* A graph representation of a program in which the nodes of the graph represent a linear sequence of instructions having one entry point and one exit point and the edges represent control flow paths (Allen, 1970).
12. *Data-flow guided Recursive Traversal disassembly.* A method of disassembly based on the Recursive Travel disassembly algorithm with an additional feature of data flow analysis, using slicing and forward substitution to increase the accuracy of disassembly during the identification and traversal of jump tables (Cifuentes & Van Emmerik, 1999; Vinciguerra, Wills, Kejriwal, Martino, & Vinciguerra, 2003).
13. *Decompilation.* A tool used to reconstruct the higher level structure of an executable program in an attempt to recover the equivalent source code (Caliskan-Islam, Yamaguchi, et al., 2015).
14. *Disassembly*. A method of extracting the set of CPU processor instructions, in the proper sequence, that make up a computer program from an existing executable file (Vigna, 2007).
15. *Dynamic taint analysis.* A method of program analysis that executes a program and observes how predefined taint sources (such as user input) are affected by computations (Schwartz, Avgerinos, & Brumley, 2010).
16. *Ether.* A tool used for binary analysis that is based on hardware level virtualization (Dinaburg, Royal, Sharif, & Lee, 2008).

17. *Eureka.* A tool used for malware analysis that dumped the memory contents of a malware specimen based on a call to programs exit system call (Sharif, Yegneswaran, Saidi, Porras, & Lee, 2008).
18. *Extended Linear Sweep disassembly.* A method of disassembly with the same characteristics of linear sweep disassembly, with the addition feature of being able to properly traverse jump tables embedded in the instruction stream. (Schwarz, Debray, & Andrews, 2002; Vinciguerra et al., 2003)
19. *Functional evolution.* The step-wise improvement and additional capability made available in successive versions of a software program in order to better adapt to its environment (Iliopoulos et al., 2011).
20. *Ground Truth.* Factual data as identified through direct observation (Kobielus, 2014; Krig, 2014; TheFreeDictionary.com, 2016).
21. *Hidden Markov Models*. "Hidden Markov models (HMM) are a general statistical modeling technique for 'linear' problems like sequences or time series. An HMM is a finite model that describes a probability distribution over an infinite number of possible sequences" (Eddy, 1996, p. 361).
22. *Hybrid Extended Linear Sweep / Recursive travel disassembly.* A method of disassembly that combines the strengths of linear sweep disassembly with the strengths of recursive travel disassembly which offers higher accuracy results than either method individually. (Schwarz et al., 2002; Vinciguerra et al., 2003).
23. *In sample error.* The rate of errors received when predicting results using the same data as used to train the classifier (Abu-Mostafa et al., 2012).
24. *Git*. An open source distributed version control system for computer software.

25. *Linear sweep disassembly.* A method of disassembly that starts at the first byte of the code section of a binary executable file and decodes each byte back to its original instruction in linear order. This method of disassembly can often produces inaccurate results as misalignment of a single instruction will impact the decoding accuracy of the following instructions (Schwarz et al., 2002; Vinciguerra et al., 2003).
26. *Machine learning.* "A set of methods that can automatically detect patterns in data, and then use the uncovered patterns to predict future data, or to perform other kinds of decision making under uncertainty" (Murphy, 2012, p. 1).
27. *Machine-morphed variant.* A version of malware that has been generated automatically with the use of a morphing engine. The capabilities of the malware are not modified, just encoded with a different obfuscation key (Chouchane, Stakhanova, Walenstein, & Lakhotia, 2013).
28. *N-fold cross validation*. An experimental design in which the entire dataset is split into *n* equal shares and each of the *n* shares is withheld from the set for testing purposes after training the classifier with the other *(n-1)* shares (Witten et al., 2011). The final result is computed as the average from the *n* iterations of the test (Witten et al., 2011).
29. *N-gram.* A sequence of contiguous characters, words, tokens, or some other representation of data of length *n*, where *n* can be any natural number, taken from a document.

30. *N-perm.* A type of n-gram that has the sequence of characters dropped, giving a set that contains of all the possible permutations of the original n-gram (Karim, Walenstein, Lakhotia, & Parida, 2005).
31. *Obfuscator.* A type of executable packer that implements one or more protections against reverse engineering and/or analysis.
32. *Out of sample error.* The rate of errors resulting from predictions made using a trained classifier on new data. Also called generalization error (Abu-Mostafa et al., 2012).
33. *Packer*. A tool that transforms a binary program into a derivative that has identical observable behavior (the behavior as experienced by the user), but has been transformed by compressing the code and data bytes and optionally applying obfuscations to its code (Roundy & Miller, 2013).
34. *Primary changes (in computer program).* Changes directly attributed to the author of a computer program, including the addition of new code, deletion or modification of existing code (Zheng Wang, Pierce, & McFarling, 1999).
35. *Phylogeny (of malware).* The evolutionary path of development of a particular family of malware as evidenced by new or updated features, the removal of unneeded code, and the remediation of coding errors (Kruegel, Kirda, Mutz, Robertson, & Vigna, 2006).
36. *Plagiarism. "*The unauthorized use or close imitation of the ideas and language/expression of someone else. It involves representing their work as your own. It is usually associated, too, with little or no acknowledgement of the borrowing and the source" (Hannabuss, 2001, p. 311).

37. *Portable Executable*. A file format used by Microsoft to represent binary executable files (Pietrek, 1994).
38. *Random Forest*. "A combination of tree predictors such that each tree depends on the values of a random vector sampled independently and with the same distribution for all trees in the forest" (Breiman, 2001, p. 1).
39. *Recursive travel disassembly.* A method of disassembly where the starting instruction is identified and disassembly begins from there, following the instructions as they are decoded, branching with the disassembly to ensure instruction alignment is maintained (Schwarz et al., 2002; Vinciguerra et al., 2003).
40. *Secondary changes (in computer program).* Changes to a program that are a result of the compilation and/or optimization steps being done by the compiler. These changes can include changes in control flow instruction targets, pointers, and register allocations (Zheng Wang et al., 1999).
41. *Simulated Annealing.* A method of simulation that takes 1) a known initial configuration, 2) a mechanism for randomly modifying the configuration, 3) a comparison function to measure the improvement (if any) in the change in configuration, and 4) a schedule to determine how long to run the simulation in order to automatically find the optimal matching solution (Kirkpatrick, Gelatt, & Vecchi, 1983).
42. *Stylometry*. The set of stylistic features, including lexical, syntactic, structural, content-specific, and idiosyncratic attributes found within a document that provides support for the determination of authorship (Abbasi & Chen, 2008).

43. *Supervised Learning.* A form of Machine Learning that uses a training set, which includes the presence of outcome variables, to guide the learning process (Hastie et al., 2009).
44. *Symbolic Execution*. A method of program analysis in which a logical formula of the program's execution path is built. This formula is then used to reason about the domain of logic with respect to the program's execution (Schwartz et al., 2010).
45. *τ-obfuscation*. A measurement that captures the difference in time required for an obfuscated executable to realize its mission compared to the amount of time required by an analyst to de-obfuscate the executable (Beaucamps & Filiol, 2007).
46. *Unsupervised learning.* A form of Machine Learning that seeks to describe how data are organized or clustered based only on the observations of features (with no known outcomes) (Hastie et al., 2009).
47. *Virtual Machine Obfuscation*. A method of program obfuscation where a Virtual Machine (VM) section is appended to the original program, the original binary code is translated into code that will run within the appended VM section, and the program's OEP is reset to the entry point of the VM section (Fang, Wu, Wang, & Huang, 2011).

List of Acronyms

1. *AST*. Abstract Syntax Tree.
2. *API*. Application Programming Interface.

3. *BIRD*. Binary Interpretation using Runtime Disassembly (Nanda, Li, Lam, & Chiueh, 2006).
4. *CAL*. Code Analysis Layer (Alrabaee et al., 2014).
5. *CFG*. Control Flow Graph.
6. *DBI*. Dynamic Binary Instrumentation.
7. *FEP*. Function Entry Point.
8. *GCJ*. Google Code Jam (Contest).
9. *IAT*. Imports Address Table (Pietrek, 1994)
10. *MAAGI*. Malware Analysis and Attribution using Genetic Information (Pfeffer et al., 2012).
11. *NUANCE*. N-gram Unsupervised Automated Natural Cluster Ensemble (Layton, Watters, & Dazeley, 2012, 2013).
12. *OEP*. Original Entry Point.
13. *PE (file)*. Portable Executable (file).
14. *RFG*. Register Flow Graph (Alrabaee et al., 2014).
15. *SCAP*. Source Code Author Profile (Frantzeskou, MacDonell, Stamatatos, & Gritzalis, 2008; Frantzeskou, Stamatatos, Gritzalis, Chaski, & Howald, 2007).
16. *STL*. Software Template Library (Alrabaee et al., 2014).
17. *SVM*. Support Vector Machine (Vapnik, 1999).

Summary

Authorship attribution in the case of human readable files relies on an analysis of stylometry, vocabulary diversity, word choice, and frequency of function words (Bozkurt et al., 2007; Stamatatos, 2009). However, in the case of executable computer

files, the process of compilation removes many of the elements of style that were added by the author and replaces them with standardized machine level instructions (Rosenblum et al., 2010; Rosenblum, Miller, et al., 2011).

The process of obfuscating an executable computer file involves the transformation of that file into a file that has identical observable behavior (the behavior as experienced by the user) and that the effort required to analyze or reverse engineer the file back into a form that is readily understood by a human analyst is prohibitively high (Collberg & Thomborson, 2002; Linn & Debray, 2003). Collberg et al. also characterized an obfuscated executable computer file as a file that "consists of two programs merged into one: a real program which performs a useful task and a bogus program which computes useless information. The sole purpose of the bogus program is to confuse the reverse engineers by hiding the real program behind the irrelevant code" (Collberg et al., 1997, p. 23).

The goal of this dissertation is to accurately describe the effects of executable file obfuscation techniques on the ability to attribute the authorship of a binary executable file of unknown provenance.

Chapter 2

Review of the Literature

Overview

As noted in Alrabaee et al. (2014) and held true in this research, there seems to be little previous research done in the field of determining authorship of binary files prior to Rosenblum, Zhu, et al. (2011). Rosenblum, Zhu, et al. (2011) discusses the utility of authorship attribution in the case of malware analysis and digital forensics, but does not specifically address the challenges that might arise from attempts to use their procedures on obfuscated malware specimens. In later research, Alrabaee et al. (2014) further improve the accuracy of author attribution in binary files, but leave the problem of obfuscated binary files to future research. In Caliskan-Islam, Yamaguchi, et al. (2015), researchers used a combination of disassembly and decompilation with machine learning classification to improve both the accuracy and the size of the author pool in attribution of authorship in binary files. The methodologies used by each of these research groups, and their applicability to attributing authorship in obfuscated samples will be discussed in considerable detail later in this chapter.

Determination of authorship in obfuscated binary files intersects with natural language and source code authorship attribution, compiler provenance of programs, program and programmer evolution, software plagiarism, binary file obfuscation, program disassembly, representation, and comparison. As such, there is a significant corpus of research to draw from and each of these topics will be discussed in the following subsections.

Natural Language and Source Code Authorship Attribution

Authorship attribution in natural language readable files has been the subject of much research and many of the research problems in this area are well understood and solutions have been proposed to address them (Burrows, Uitdenbogerd, & Turpin, 2014; Krsul & Spafford, 1997). Spafford and Weeber (1993) hypothesized that attributing authorship in software source code should follow the same process as other natural language formatted documents: identifying features in the code that can be attributed to the writing style of a particular programmer. They provided a list of both potential features for use in attributing authorship in source code, as well as some potential challenges that might arise during the attribution analysis phase. Sallis, Aakjaer, and MacDonell (1996) pointed out that even though computer programmers appear to have distinct programming styles, there weren't any techniques available to accurately identify the author of computer source code. Krsul and Spafford (1997) demonstrated in an experimental fashion that stylistic markers could be used to measure the likelihood of authorship in a sample of computer source code, but their data showed that many subjects have very similar style characteristics that led to lower accuracy of results. Gray, Sallis, and MacDonell (1997) and Frantzeskou, Gritzalis, and MacDonell (2004) concluded that based on these earlier works, authorship attribution of computer source code did belong in the field of software forensics.

As early as Kilgour, Gray, Sallis, and MacDonell (1998), there were attempts to combine the problem of authorship attribution of human readable documents with the artificial intelligence practice of fuzzy logic variables to aid in classification.

In MacDonell and Gray (2001), the researchers identified a set of 26 authorship

related metrics from 351 programs written by seven different authors and fed each of them into a feed-forward neural network, a multiple discriminant analysis model, and a case-based reasoning system and reported an accuracy of over 80% in the determination of author.

Li, Zheng, and Chen (2006) described a "writeprint", the computer software version of a fingerprint that could be used to attribute authorship of computer source code. They noted that a "writeprint" was made up of a collection of features that could be extracted from the human readable source code. Shortly thereafter, Frantzeskou and her team created the Source Code Author Profile (SCAP) method (Frantzeskou et al., 2008; Frantzeskou et al., 2007) which could be used to identify authors of computer source by comparing profiles created using byte-level n-gram sections of source code specimens. Lange and Mancoridis (2007) demonstrated good results in determining authorship of computer source code by using histogram distributions of code metrics, but their results were not as accurate as Frantzeskou's.

Instead of relying on simple n-grams or histogram distributions, Bozkurt et al. (2007) demonstrated how authorship attribution would benefit from primary component analysis and machine learning algorithms. Bozkurt posited that these advanced techniques could be adapted to automate the process of source code authorship attribution. Caliskan-Islam, Harang, et al. (2015) continued with the notion of a machine learning based problem and used natural language processing to extract necessary features (called The Code Stylometry Feature Set) from computer source code to accurately identify authors of source code.

However, Burrows, Uitdenbogerd, and Turpin (2009) studied the coding efforts of

272 students and identified that coding style evolves over time and that the domain of the coding problem had a large impact on the ability to accurately attribute authorship.

In Meng, Miller, Williams, and Bernat (2013), the researchers drew from code stored in a GIT source code repository to determine authorship. It is typical for programmers to incrementally adjust or update existing code (that could have been written by another programmer entirely). However, even with this mixing of contributions from multiple authors, Meng et al. (2013) concluded that accurate attribution of authorship in source files could be achieved with good levels of accuracy.

Juola (2006) described the challenge of authorship attribution and discussed how unsupervised machine learning algorithms might be better suited to the problem of attributing authorship due to the lack of sufficient existing material which could be used to create an author profile; a profile that would be used during the teaching phase of a supervised machine learning algorithm. Iqbal, Binsalleeh, Fung, and Debbabi (2010) applied a method using unsupervised clustering algorithms to create visualizations that assisted in the identification of authors of anonymous emails. Layton, Watters, and Dazeley (2010) extended the SCAP method of authorship attribution (Frantzeskou et al., 2007) using unsupervised machine learning algorithms to identify phishing campaigns. (Layton et al., 2012, 2013) further extended the unsupervised learning version of SCAP by creating the NUANCE (N-gram Unsupervised Automated Natural Cluster Ensemble) model, a higher accuracy model for displaying documents of unknown origin in clusters centered around commonality of features.

Arabyarmohamady, Moradi, and Asadpour (2012) incorporated style markers into the detection of plagiarism in order to identify the original author in cases where original

coding efforts had been plagiarized by another author. Like most other methods, the method described in Arabyarmohamady et al. (2012) relied on supervised machine learning algorithms which required an existing author profile against which the unknown document could be compared.

Chatzicharalampous, Frantzeskou, and Stamatatos (2012) sought to prove whether a single profile-based representation of a set of source code specimens resulted in higher accuracy classification of authorship over instance-based representations, where each member in the set of source code specimens had their own representation. The researchers found that instance-based representations resulted in higher accuracy classifications if the class of specimens was balanced.

In Matthew F Tennyson and Francisco J Mitropoulos (2014), improvements were made to two of the state-of-the-art methods for attributing authorship in source code, namely the Burrows (Burrows et al., 2009) and SCAP (Frantzeskou et al., 2007) methods. Once improved, the two methods were joined together to form an ensemble classifier based on Bayesian probability. The probability that a selected author wrote a document, given all the observations, was based on all the evidence as well as the combined probabilities from the Burrows' and SCAP's observations. By creating this ensemble classifier, the researchers were able to bridge the divide between ranking classifiers and machine learning classifiers (Burrows et al., 2014) and create an author attribution system with classification accuracy of about 98%.

Text Analysis as a Machine Learning Problem

An emerging trend in the processing and analysis of natural language has been the application of machine learning classification algorithms to identify patterns in datasets.

Based on the research described in the previous section, the use of machine learning tools and algorithms for the purpose of classifying text can be considered a well understood problem.

Researchers have examined the effect of differently sized sample sets (Koppel et al., 2009; Luyckx & Daelemans, 2008; Zheng et al., 2006) and found that larger datasets increase prediction accuracy, but increased numbers of candidate authors add considerable complexity to classification, which reduces overall accuracy of attribution. The effectiveness of different classification algorithms (Iqbal, Khan, Fung, & Debbabi, 2010; Lamirel, Cuxac, Chivukula, & Hajlaoui, 2015; Luyckx & Daelemans, 2005) has also been researched, with Support Vector Machines (SVM) (Vapnik, 1999) being shown as a leading classification algorithm. However, Random Forests (Breiman, 2001) have been shown to be effective at classifying data that has many weak inputs (Breiman, 2001). Similarly, Koppel et al. (2013) have suggested that "large data sets of very simple features are more accurate than small sets of sophisticated features for [authorship attribution]" (Koppel et al., 2013, p. 321). Maitra, Ghosh, and Das (2015) demonstrated good accuracy using Random Forests in the verification of authorship, and achieved better performance than with an SVM classifier.

In datasets having many weak inputs, with weak input described as "no single input or small group of inputs that can distinguish between the classes" (Breiman, 2001, p. 22), the selection of attributes is both important to accurate classification and difficult to determine. The goal is to select attributes from each record in the dataset such that the attribute values demonstrate author invariance (Baron, 2014; Stańczyk, 2013). These principal attributes can be identified using the calculation of information gain

contributed by each attribute (Quinlan, 1986). Information gain was used in Argamon et al. (2007) and in Caliskan and Greenstadt (2012) to identify the principal components in natural language texts and again in Caliskan-Islam, Yamaguchi, et al. (2015) for principal component analysis of authorship identifying attributes in binary executable files.

In majority of the research papers mentioned in these last two sections, the material to be classified is natural language text or human readable computer source code. Rosenblum, Zhu, et al. (2011) started down a new path of using the techniques based on natural language text classification on binary executable files. Alrabaee et al. (2014) furthered this research path by creating hashes of basic blocks, Register Flow Graphs (RFGs), and a Software Template Library (STL) that could be used to classify binary files using the same techniques as natural language text. Caliskan-Islam, Yamaguchi, et al. (2015) proceeded further by utilizing methods from both Rosenblum, Zhu, et al. (2011) and Alrabaee et al. (2014) with the addition of attributes collected during decompilation of the binary executable files. These research efforts are described in detail in the *Binary File Authorship Attribution* section ahead.

Compiler Chain Provenance

During the search to discover a method to identify Function Entry Points in unknown binary files, Rosenblum, Zhu, Miller, and Hunt (2008) recognized that different compilers generate regular patterns that could be used to identify the compiler–environment of unknown binaries. In following papers, (Rosenblum et al., 2010; Rosenblum, Miller, et al., 2011) created a system to classify these different compiler sequences allowing the researchers to identify the compiler chain used in 90% of their

test cases. Torri (2009) demonstrated how knowledge of the compiler environment of a particular binary could make the process of reverse engineering and analysis less error-prone and time consuming. Jacobson, Rosenblum, and Miller (2011) extended the UNSTRIP tool to create a new, derivative binary file with meaningful names of functions added to the symbol table in the previously stripped binary executable files.

Hosic, Tauritz, and Mulder (2014) extended the research of Rosenblum, Zhu, Miller, and Hunt (2007) by using genetic algorithms to classify compiler usage in unknown binaries. The use of genetic algorithms pushed the accuracy of classification to an average of 95.3%.

Rahimian, Shirani, Alrbaee, Wang, and Debbabi (2015) created BinComp, a tool that could identify the compiler provenance of an executable file with an accuracy of up to 90% using a combination of control flow graphs, register flow graphs, and function call graphs.

However, (G. Balakrishnan & Reps, 2010; Bao et al., 2014; Dullien, Carrera, Eppler, & Porst, 2010; Payer et al., 2014; Stojanović, Radivojević, & Cvetanović, 2014) identified that different compilers, different versions of the same compiler, and a single compiler with different optimizations enabled could produce remarkably different output, which reduced the predictive value of the identified compiler markings.

In their research, Chaki, Cohen, and Gurfinkel (2011) concluded compiler provenance to be a sub-set of the problem of representing and comparing two unknown binaries for similarity. Additional facets of binary file representation and comparison will be discussed further in the section entitled *Representation & Comparison.*

In Roy, Cordy, and Koschke (2009), code clones were defined as a section of

code that is very similar to another section of code. Four types of code clones were defined in Roy et al. (2009):

Type 1: Identical code fragments except for differences in whitespace, layout, or comments;

Type 2: fragments that are syntactically identical except for variations in identifiers, literals, types, or whitespace;

Type 3: fragments that are identical, but additional commands are interspersed into the copied material; and

Type 4: different fragments that perform the same computation but are implemented using different syntax.

Sæbjørnsen, Willcock, Panas, Quinlan, and Su (2009) proposed a method for detecting code clones in binary executable files using a combination of exact matching of normalized code regions and inexact matching of feature vectors representing important aspects of the code regions. H. Chen (2013) undertook a study wherein the four types of code clones were evaluated against compiler optimizations. He concluded that code clone types three and four were the types that most accurately capture an author's style, and it is these clone types that are affected the most by compiler optimizations. David and Yahav (2014) proposed using 'tracelets', short sequences of executable instructions for use in detecting similarity between functions.

Program & Programmer Evolution

In 1980, Meir Lehman defined the 5 laws of software evolution and described how the laws were applicable to all complex software projects. The laws of software evolution are:

1. Software undergoes continual change otherwise the software will become less useful;
2. Since software is continually being changed, its complexity continues to increase;
3. As changes are made to the software, the software continues to represent the problem and solution domains accurately;
4. While the program is actively being used, the rate of change is statistically invariant;
5. During the active life of the software, the release content of successive releases is statistically invariant (Lehman, 1980; Lehman & Ramil, 2001).

Software evolution as a mandatory part of the normal lifecycle of software development becomes important when considering authorship attribution, as changes and/or corrections to a program's source code are done incrementally, over time, and, potentially, by multiple authors. The study of malware as it relates to software evolution can be considered a special case as the time between generations is often very short and the number of evolutionary versions can be quite high (Barat, Prelipcean, & Gavriluţ, 2013; Dumitras & Neamtiu, 2011; Gupta, Kuppili, Akella, & Barford, 2009; Iliopoulos et al., 2011). In addition, research has identified that typical cybercrime (malware) development teams are, on average, sized with between four and seven members (not including Nation state hacking teams) (Broadhurst, Grabosky, Alazab, Bouhours, & Chon, 2013; Grabosky, 2014; Layton & Azab, 2014). As suggested in Alrabaee et al. (2014), with malware development teams being of a smaller size, the accuracy of authorship attribution within a single development team should increase.

Vlachos, Ilioudis, and Papanikolaou (2012) studied the nature and impact of security incidents related to malware. Even though it is reported that there are 200,000 new malware variants identified per day (Kaspersky Labs, 2012), Vlachos et al. (2012) reported that the majority of damage due to malware came from evolutionary variants of the top ten malware families.

Of the two types of evolution discussed in (Iliopoulos et al., 2011), cryptic and functional evolution, only the functional evolution of malware is applicable to the study of authorship attribution of binary files as related to malware. Cryptic evolution refers to the capability of a computer program to obfuscate its signature. This obfuscation makes different instances of the same program appear different (for example, an anti-virus program would compute different hashes of the two instances of the same file, resulting in one file being flagged for cleaning and the other not), but does not result in different behavior or functionality in the program itself. Functional evolution refers to the step-wise improvement and additional capability made available in successive versions of a software program in order to better adapt to its environment (Iliopoulos et al., 2011).

Kruegel et al. (2006) presented a technique that allowed researchers to identify structural similarities between different computer worm mutations based on the structural analysis of the worm's binary code. This technique could be used to identify the phylogeny (evolutionary path) of a particular version of a computer worm.

Yu, Zhou, Liu, Yang, and Luo (2010) proposed a novel byte frequency based identification process to identify variants of known malware. Using this method, the researchers could quickly gauge the similarity of two suspect binaries and draw

conclusions about the lineage of the binaries.

Khoo and Lió (2011) created a full execution path logging system to create traces that could be transformed into phylogenic trees and networks for topological comparisons.

Chouchane et al. (2013) presented a set of procedures for the identification of machine-morphed malware variants. Machine-morphed malware variants can be generated automatically and in quick succession with the use of a morphing engine. These morphing engines do not change the capabilities of the malware, rather they obfuscate the same version of program using different encoding keys within a compression algorithm, modifying the algorithm used for compression, modifying constant values, or randomly inserting "junk" instructions (Chouchane et al., 2013). The procedures presented in their research were presented as a sort of pre-processor to the authorship attribution challenge of malware.

Xu et al. (2013) proposed a new method of analysis to determine similarity between malware variants by extracting the function call graph of the variant using static analysis. Once the call graphs were extracted, a comparison could be done to quantify the similarity of the variants.

In contrast to the work done on machine-morphed variants, Karim et al. (2005) defined a method to determine variants of malware using "n-perms", which are similar to n-grams (an 'n' length section of a program), except that the ordering of the sequence was dropped, giving a set of all the possible permutations of the original n-gram. The researchers report that their n-perm based matching algorithm provided better accuracy in identifying variants of malware than their n-gram alternatives.

M. Hayes, Walenstein, and Lakhotia (2009) artificially created 2 separate malware evolutionary trees for the purpose of studying previous techniques in determining phylogeny of malware. The results of this study called into question the validity of malware phylogeny research that was undertaken using artificially generated files (which they report is typical). Lindorfer, Di Federico, Maggi, Comparetti, and Zanero (2012) chose instead to find a real malware example and document and study its actual evolution in an effort to better understand malware phylogeny.

Pfeffer et al. (2012) created a system called MAAGI which is currently being used to determine similarity between malware samples, identify families of malware, and identify the lineage of multiple variants of malware. The researchers report promising results when using MAAGI against known datasets.

Singh et al. (2012) discussed methods to track concept drift in malware families. Concept drift is defined as evolution within a family of malware as a response to outside pressures or the need to implement new features. Kantchelian et al. (2013) argued that concept drift in malware was a particularly challenging problem for classification, as the classification must balance between over-fitting and under-fitting.

Darmetko, Jilcott, and Everett (2013) considered the case of variants generated by the addition of supplemental or new features. As discussed above, laws of software evolution have been identified and widely agreed upon (Lehman, 1980). Darmetko et al. (2013) utilized these laws, including functional complexity, similarity of string data, and shared code subsections to identify patterns in software lineage.

Jiyong Jang, Woo, and Brumley (2013) constructed a novel tool for identifying an evolutionary hierarchy from program binaries that had a measured accuracy of 84%

in identifying the correct evolutionary hierarchy for goodware and 72% for malware.

Y. Park, Reeves, and Stamp (2013) described a system whereby behavioral graphs from multiple program variants were overlaid on a common graph in order to create a model behavior that could be used during a comparison of a binary to an unknown variant.

Anderson, Lane, and Hash (2014) described a novel approach for representing malware phylogeny as a graphical lasso, which found the weighted combination of static and dynamic views of the variants that allowed for a graph of a family of programs phylogeny to be constructed.

Not only do computer programs evolve over time, but also so do the human programmers who write them. In J. H. Hayes (2008), the researcher observed that programmers had style consistency in their work, such that classifications could accurately predict the author of a specimen of computer source code. Upon revisiting the hypothesis of the consistent programmer, J. H. Hayes and Offutt (2010) discussed the finding that as a computer programmer reaches a level of expertise, they developed their own personal style of programming and that once defined, their programming style changed little over time. Burrows et al. (2009) described their research where they were able to observe an evolution of coding skill in students that had just started demonstrating proficiency in computer programming. Their research observed that as the students became more proficient, the more accurate the authorship attribution was.

Bhattathiripad (2012) introduced the notion of programmer blunders. These blunders are errors in programming that have been introduced by a programmer but yield little to no chance of causing software failure. Often these blunders are carried

along in software and can provide clues about the author of the program.

Software Plagiarism

The opportunity to simply copy another's computer source code and present it as one's own has existed as long as there have been general purpose programming languages (Ottenstein, 1976). Early research on software engineering observed that there were "natural laws" that controlled the algorithmic structure of the solution to most software engineering problems (Halstead, 1972, 1977). As a result of these laws governing the algorithmic structure of solutions, Ottenstein (1976) was able to demonstrate that differences in the resulting solutions could be attributed to programmer style, and these differences could be used to detect plagiarism between different versions of a solution. Many early plagiarism systems were based on the idea of attribute counting and that matching scores of attributes within source code files could accurately identify examples of plagiarism (Berghel & Sallach, 1984; Donaldson, Lancaster, & Sposato, 1981; Grier, 1981; Jankowitz, 1988; Parker & Hamblen, 1989; Rees, 1982). In Verco and Wise (1996), these attribute-counting based plagiarism detection systems were compared to the structure-metric systems, which include the plagiarism detection systems discussed in (Belkhouche, Nix, & Hassell, 2004; Gitchell & Tran, 1999; Whale, 1990; Wise, 1996). Each method, attribute-counting or structure-metric, have superior capabilities for identifying some types of plagiarism in software source code and should be used, according to Verco and Wise (1996), together and in combination with a human reviewer.

Besides these two main types of plagiarism detection systems, other processes have been proposed. Freire (2008) used visualizations of binary files to help identify

program similarities. Each of (Chae, Ha, Kim, Kang, & Im, 2013; Liu, Chen, Han, & Yu, 2006) created systems that generated graphs to represent the binary under analysis. These graphs could then be compared for similarity. Chilowicz, Duris, and Roussel (2009) used abstract syntax tree fingerprints to create a type of software fingerprint that could be compared to fingerprints from other binaries, and (Chae, Kim, Ha, Lee, & Woo, 2013; McMillan, Grechanik, & Poshyvanyk, 2012; X. Wang, Jhi, Zhu, & Liu, 2009) used software birthmarks, which, like fingerprints, are unique characteristics of a program that could be used for comparing program similarity. Tian, Zheng, Liu, and Fan (2013) extended the idea of software birthmarks by incorporating dynamic data flow analysis into the birthmark. This addition of dynamic data flow information reduced the negative impact of software obfuscation techniques.

There was research that married the problem of determination of plagiarized content with the strength of classification offered in machine learning algorithms. (Engels, Lakshmanan, & Craig, 2007; Kilgour et al., 1998) fed the output of their feature selection process into a feed forward neural network for the purpose of classification and (Jadalla & Elnagar, 2008; Ohmann & Rahal, 2014; Zou, Long, & Ling, 2010) have used clustering techniques to identify likely plagiarized content.

Kučečka (2011) discussed methods that exist that are specifically designed to foil plagiarism detection systems. With knowledge of these types of deceptions, the researcher was able to modify an existing plagiarism system to improve the accuracy of detection and reduce the number of false positive findings.

Executable File Obfuscation

Cohen (1993) first described the goal of security through obscurity as making the difficulty of attack so great that it was not worth performing in practice, even if it could eventually be successful. He continued to describe a number of methods that were available for use in making a binary executable file difficult to attack. In the context of binary executable file obfuscation, attack is defined as being able to peer into the binary and identify its capabilities in order to render those capabilities useless (Cohen, 1993). The methods he described included equivalent instruction replacement, instruction reordering, and garbage instruction insertion (Cohen, 1993). Collberg et al. (1997) studied obfuscation techniques in use in malicious software and offered a classification mechanism for obfuscation methods. They also characterized an obfuscated binary executable as an executable file that "consists of two programs merged into one: a real program which performs a useful task and a bogus program which computes useless information. The sole purpose of the bogus program is to confuse the reverse engineers by hiding the real program behind the irrelevant code" (Collberg et al., 1997, p. 23). The conclusion reached in Collberg et al. (1997) was that an obfuscated program may behave differently from the original program, as long as the observable behavior was identical. This research team completed additional research into obfuscation options for executable files in (Collberg, Thomborson, & Low, 1998a, 1998b) and again in (Collberg & Thomborson, 2002).

A formal proof for program obfuscation was attempted in Barak et al. (2001), in which a program *P* is transformed into another, semantically equivalent, program, *P'* that has the added feature of being harder to understand, and this proof was shown to be

impossible to solve. However, as discussed in Mavrogiannopoulos, Kisserli, and Preneel (2011), and consistent with Cohen (1993), obfuscation is used in practice to raise the barrier against successful reverse engineering. Beaucamps and Filiol (2007) challenged the findings of Barak et al. (2001) based on the required amount of time needed to de-obfuscate a specimen of obfuscated code. They coined the term τ-obfuscation to capture the difference in time required for the obfuscated executable to realize its mission compared to the amount of time required to de-obfuscate the executable (Beaucamps & Filiol, 2007). In as much as these researchers tried to extend the work of (Barak et al., 2001) by finishing the proof for program obfuscation, they were not able to. The researchers concluded that since most binary analysis or reverse engineering efforts are constrained by the amount of time the engineer is authorized to spend doing the analysis, most obfuscation techniques would be successful in keeping the true capability of the unknown binary from the engineer, even if the obfuscation method is solvable.

Wroblewski (2002) demonstrated a technique for obfuscating binary executable files that could be applied generically on different computer architectures. Linn and Debray (2003) offered strategies for obfuscation of binaries, with the appreciation that evading disassembly was the goal of obfuscation. These strategies were applicable to both common disassembly algorithms: linear sweep and recursive traversal (definitions provided in the next section). In response to these obfuscation techniques, disassembly tools could utilize symbolic execution of the executable to extract program instructions. However, Zhi Wang, Ming, Jia, and Gao (2011) presented an obfuscation method that foiled symbolic execution.

A system that "camouflaged" program instructions by replacing them with more complex structures was presented in (Kanzaki, Monden, Nakamura, & Matsumoto, 2003; Kanzaki, Monden, Nakamura, & Matsumoto, 2006). Popov, Debray, and Andrews (2007) proposed a system where binary executable files could be obfuscated by replacing control transfer instructions with Traps that caused signals to be sent. The signal handling routine would then affect the original transfer of control, restoring the appropriate execution behavior while confusing the disassembler (Popov et al., 2007).

Anckaert et al. (2007) proposed a new method to measure the complexity influence of both obfuscators and de-obfuscators and showed how some obfuscation techniques imposed a limited effect on the difficulty of reverse engineering.

Automated malware analyzers are able to discover obfuscated sections within code using symbolic execution, forced conditional execution, and multiple path exploration, instead of waiting until program execution to observe program behavior. However, Sharif, Lanzi, Giffin, and Lee (2008) proposed an obfuscation technique that used input values as the secret phrase for the decryption of code blocks which held the trigger condition for the malware's execution. The researchers showed how effective this technique could be against current automated analysis systems. Instead of embedding encrypted sections into code, Cappaert, Preneel, Anckaert, Madou, and De Bosschere (2008) proposed a method which included the on-demand decryption of 'guards' embedded in executable files for the demonstration of code integrity. The key material for these 'guards' is the calling code itself. In this context, on-demand refers to the characteristic of having only sections of code encrypted, as opposed to other

techniques that encrypt the entire program and attach a small decryption stub to initiate the program's execution.

Z. Wu, Gianvecchio, Xie, and Wang (2010) proposed a new approach to binary code obfuscation called 'mimimorphism', which relied on transforming the original binary file to a "mimicry" executable, which had the same capabilities but presented to malware detection systems as benign.

Fang et al. (2011) proposed a novel obfuscation technique that involved creating a private and random virtual machine core, complete with a transformed instruction set. This obfuscation technique was more efficient and provided a higher level of security than previous models of obfuscation (Fang et al., 2011). This strategy of obfuscation will be discussed further in the *Binary File Disassembly* section of this Literature Review.

O'Kane, Sezer, and McLaughlin (2011) described the mechanism of applying obfuscation to malware and Roundy and Miller (2013) discussed the popular obfuscation techniques used in malware and the strategies and tools used by malware investigators. These popular obfuscation techniques can be found in Table 1.

Obfuscation Technique	Explanation
Code Packing	The main program content is compressed or encrypted and an unpacking program is added to the program. The unpacking program runs first; expanding the programs executable content into memory then passes execution control to the in-memory version of the program.
Code Overwriting	The existing code is overwritten by new instructions placed into the execution path during runtime.
Non-returning Calls	A CALL instruction is used to jump to another location in code in

	order to execute some instructions. These instructions are followed by a RET (return) which brings execution back to the call location. A CALL instruction could be substituted for a JMP (jump) instruction, so long as the destination instructions clean the extra return address from the stack. This substitution confuses many disassembly tools.
Call-Stack Tampering	Similar to non-returning calls, call-stack tampering involves the non-standard use of the ret instruction to confuse disassemblers.
Obfuscated Control-Transfer Targets	Instead of using the compiler-supplied direct control transfers, this obfuscation technique relies on using memory or register content to determine the location of the CALL and JMP functions.
Exception-Based Control Transfers	Signal- and exception-handling routines can be used for obfuscating control transfers in much the same way as discussed above. Additionally, the signal- and exception-handler routines define the location in the program of where to resume once the handler has finished, adding further to the obfuscation of the control transfer.
Ambiguous Code and Data	By introducing a conditional branch to the instruction path where one of the conditions is never met, the disassembler may erroneously interpret the "junk codes" inserted into the never-followed conditional path.
Disassembler Fuzz Testing	By adding seemingly random instructions to the program, the program can "fuzz test" the disassembler, which usually results in errors in disassembly due to incorrect interpretation or definition of the fuzzed instructions by the disassembler. This technique also works in sandbox environments and is typically used to identify that the execution of a program is indeed sandboxed.
Obfuscated Calls and Returns	The program uses a call and ret pair rather than the more appropriate JMP instruction. The addition of the code needed to undo the automatic stack modification can interfere with disassembly.
Overlapping Functions and Basic Blocks	Due to the dense nature of the variable instruction length IA-32 architecture, small functions and code blocks are often able to overlap due to the duplication in a small subset of opcodes.
Obfuscated Constants	Using a known obfuscation algorithm, any constant defined within an executable program can be replaced by the seemingly random set of characters generated by the obfuscation algorithm.

	The program, when run, will replace that set of symbols with the original value. However, an analyst will not be able to identify these constants during casual observation (as can be done with constants that haven't been obfuscated).
Calling Convention Violations	Calling conventions standardize the contract between a caller and callee function. While many calling conventions exist, most compilers default to a standard calling convention. By changing the calling conventions with a program, the difficulty of analysis will increase.
Do-Nothing Code	By adding additional instructions that have no purpose (beyond obfuscation), the real set of instructions is diluted which increases the likelihood it will be overlooked.
Embedded Virtual Machines	By translating original instructions into instructions that can be run only on a specific virtual machine, one that implements many of the above obfuscations, the true nature of the original program is effectively hidden.
Stolen Bytes	By implementing the first bytes of code from a function provided by an external library in one's own code, and then removing the reference to the library, small sections of code are missing from an executable until the library is dynamically re-referenced during execution.
Self-Checksumming	The packer embeds a checksum of its own code into its code and then regularly checks the validity of the current image against the checksum.
Anti-OEP (Original Entry Point)* Finding	Since finding the OEP is so important in unpacking, packers implement these obfuscations within the bootstrap code as well, in an effort to hide the OEP. * The OEP is the first instruction to be executed by the program once de-obfuscated. Typically the last instruction to be executed by the packer is a JMP (unconditional jump) to the OEP.
Payload-Code Modification	By linking the main program execution back into routines found in the packing stub, the packing stub becomes a critical section of code execution, rather than an artifact of analysis that can be discarded.

Table 1 - Popular obfuscation techniques (Roundy & Miller, 2013)

Ugarte-Pedrero et al. (2015) completed a longitudinal study of binary packers over seven years. In the context of their research, and typically in analysis of binary executable files, the term 'packer' is used to generically describe a tool that compresses binary data, and optionally applies the obfuscations listed in Table 1. By studying examples of packed malware over a period of seven years, the researchers were able to draw conclusions about how packer techniques and usage in malware have changed over the last number of years. In addition to the results of this study over time, the researchers proposed a taxonomy of complexity types for packers (Ugarte-Pedrero et al., 2015). Their packer complexity types are listed in Table 2 (Ugarte-Pedrero et al., 2015). The researchers continued by demonstrating that an assumption made by many reverse-engineering and analysis procedures is false in that there is a complete and de-obfuscated image of the executable in memory during some phase in the binary file's execution (Ugarte-Pedrero et al., 2015).

Complexity Type	Description
Type I	A single packing routine executes before transferring execution to the unpacked program.
Type II	Multiple unpacking layers, each of the layers executed sequentially in order to unpack the following code. The packing routine transfers execution to the unpacked program once all the packing layers have been removed.
Type III	Similar to Type II, however instead of sequential execution of the unpacking layers, the packing layers are organized in a more complex topology that includes loops. As a result, the original code may not be located in the last, or deepest, layer. In such cases, the last layer often contains integrity checks, anti-debug routines, or a portion of the packer routine (which has also been obfuscated). In Type III there is still a final execution transfer to the original code.

Type IV	Type IV packers are either single or multiple-layered and interleave portions of the packer code into the execution of the original code. The portion of the packer code responsible for unpacking is not interleaved into the main execution body. While there is still a precise moment where the entire original code is completely unpacked in memory, the transition is difficult to detect due to the jumping between packing and application layers (due to the interleaving of the two).
Type V	A packer of type V has interleaved the packer into the original program to the extent that the packer unpacks each program frame (a small set of instructions) at a time for execution. As a consequence, the transfer of execution happens more than once, and for each transfer, there is only the single frame of the original program revealed. A full image of the original application is available immediately following the termination of the program itself.
Type VI	Packers of type VI describe a packer that unpacks only a single fragment, as small as a single instruction, at a time. Virtual Machine type packers belong to this type of packer.

Table 2 - Packer complexity types (Ugarte-Pedrero et al., 2015)

Binary File Disassembly

Disassembly is defined as a method of extracting the set of CPU processor instructions, in the proper sequence, that make up a computer program from an existing executable file (Vigna, 2007). The challenge of correctly disassembling binary code is difficult "because fully correct x86 disassembly is provably undecidable: Bytes are code if and only if they are reachable at runtime – a decision that reduces to the halting problem" (Wartell, Zhou, Hamlen, Kantarcioglu, & Thuraisingham, 2011, p. 11). Disassembly of unknown binary executable files that have the x86/x64 architecture is made more difficult due to the variable length of instructions and the comingling of instructions and data (Zwanger, Gerhards-Padilla, & Meier, 2014). Wartell et al. (2011) showed that IDA-Pro (Hex-Rays SA, 2015b) can make the following mistakes in disassembly: misclassifying data as returns, misaligned code bytes that are interpreted

as 16-bit legacy instructions, mislabeled padding bytes, and flows from code to data. In response, these researchers developed an automated disassembler that more accurately separated code and data segments.

There are utilities that "strip" binary files, in much the same way as many obfuscation engines, removing the jump table and other diagnostic information from the transformed image of the original binary executable file, which is a significant challenge for disassembly or reverse engineering the unknown binary. Cifuentes and Van Emmerik (1999) proposed a method based on program slicing and expression substitution to recover the lost jump table.

In response to the typical lack of accuracy in automated disassembly of unknown executable files, Krishnamoorthy, Debray, and Fligg (2009) described a machine-learning based approach that used decision trees to identify possible errors in static disassembly, and Zwanger et al. (2014) created a set of classifiers that could accurately locate code sections in many types of unknown files.

X. Chen, Andersen, Mao, Bailey, and Nazario (2008) discussed the challenges of disassembling unknown binary executable files due to both anti-virtualization and anti-debugging techniques that can be applied to binary executable files to thwart analysis. For this research, disassembly will include static disassembly techniques using generally available plug-ins for IDA-Pro (Hex-Rays SA, 2015b) and dynamic disassembly techniques based on tools written for the QEMU whole machine emulator (Bellard, 2005). Beyond the specific techniques discussed in the following subsections on dynamic and static analysis, research on deobfuscation techniques has been done in (B. Lee, Kim, & Kim, 2010; Lin, Zhang, & Xu, 2010; Luo, Ming, Wu, Liu, & Zhu, 2014;

Popa, 2012; H. Qiu & Osorio, 2013; Vigna, 2007; Xie, Li, Wang, Su, & Lu, 2012) with varying levels of success on specific variants of specific obfuscations.

Dynamic Analysis

Dynamic analysis of a binary executable file involves executing the file on some system that is tracing the execution path and logging instructions and system state as the execution progresses. Dyninst was an early example of a tool for dynamic analysis of known and unknown binary executable files. It was written to be a tool to enable structured modification of a binary file outside of the standard development environment of source code editor and compiler. The structured modification capability was implemented as a set of C++ class libraries that could be linked together to allow for program instrumentation (Buck & Hollingsworth, 2000). Dyninst enabled the instrumentation of an unknown binary for the extraction and logging of instructions that were flagged as instrumented. Harris and Miller (2005) offered new insights into the analysis of stripped binary code using an analysis framework based on Dyninst. It was reported that this framework was able to produce a 95.6% average recovery rate for recreating the symbol table stripped from the original binary file. Rosenblum et al. (2008) was able to extend the capability of Dyninst to build system that could identify function entry points (FEPs) in unknown, obfuscated programs. Due to the nature of most obfuscation routines, the symbol information needed to accurately identify FEPs in an executable are missing which is a challenge for static analysis.

QEMU is a software-only machine emulator that could be used to trace through the execution of computer programs (Bellard, 2005). QEMU has been used for many types of analysis, including performance tuning, debugging, and security-based analysis

of binary executable files (Bellard, 2005). Egele, Kruegel, Kirda, Yin, and Song (2007) extended the QEMU environment by creating a dynamic taint propagation engine that could reliably classify unknown browser helper objects that behaved like spyware. Taint analysis involves running a program while observing which computations are affected by predefined taint sources such as user input. Techniques in taint propagation are used to identify how tainted data flows through a program and what impacts this tainted data may have on normal program execution (Schwartz et al., 2010). Moser, Kruegel, and Kirda (2007b) then extended the QEMU environment further by creating a system that could explore multiple execution paths by taking note of instruction branches and the values that determined the branch decision and later returning to the decision point and replaying the execution with new values. By returning to the decision point and following the alternate execution path, a typical limitation of dynamic analysis could be minimized. A computer program source code retrieval system called Top was introduced in Zeng et al. (2013), which used QEMU and PIN to reconstruct sources from binary executable files using dynamic binary instrumentation (DBI). Henderson et al. (2014) extended QEMU further with DECAF, a virtual machine based, multi-target, and whole-system emulating dynamic binary analysis system.

Another tool built upon QEMU is TEMU (TEMU, 2009), but for TEMU, the emulator is able to provide whole system dynamic taint analysis. TEMU was the foundation that BitBlaze, a binary analysis framework, was built on (Song et al., 2008). M. G. Kang, Poosankam, and Yin (2007) created Renovo, which extended the TEMU/BitBlaze tool set to iteratively unpack binaries that were obfuscated with

multiple layers of obfuscation (see Table 2, complexity type II). The tool showed encouraging results, but was unable to fully unpack binary files that utilized the obfuscation technique of "embedded virtual machine" (see Table 1). Vine was the component of BitBlaze that provided static analysis capability for use with unknown binary executable files. BAP (short for Binary Analysis Platform) was created as a re-write of Vine to facilitate the analysis of assembly language instructions (Brumley, Jager, Avgerinos, & Schwartz, 2011). The Phoenix decompiler is one of the projects that use the intermediate representation that is output by BAP (Schwartz, Lee, Woo, & Brumley, 2013).

As previously introduced, PIN is another DBI tool that is used for the instrumentation of binary executable files on the x86/x64 architectures (Luk et al., 2005). PIN is a flexible toolkit that has often been used in security related research including (R. Balakrishnan & Anbarasu, 2013; Kwon & Su, 2010; Veeralakshmi & Sindhuja, 2013).

Bhansali et al. (2006) proposed a runtime framework that would collect a complete trace, down to the instruction level, of a program's execution. This framework, named Nirvana, had good performance and accuracy characteristics, and the trace output could be used for analysis purposes. Nanda et al. (2006) presented BIRD, the Binary Interpretation using Runtime Disassembly system, that consisted of two passes: the first uses recursive traversal disassembly, and the second pass treats all unreachable bytes as instructions and performs another iteration of the same recursive traversal disassembly on these bytes.

Martignoni, Christodorescu, and Jha (2007) built OmniUnpack which was built on, and addressed the limitations of the static analysis tool PolyUnpack (Royal, Halpin, Dagon, Edmonds, & Lee, 2006). An additional benefit of the dynamic unpacking tool OmniUnpack was that it could properly unpack executable files with anti-debugging and other offensive capabilities (Martignoni et al., 2007).

Sharif, Yegneswaran, et al. (2008) describe Eureka, a framework that combined coarse-grained execution tracing and statistical bigram analysis to accurately unpack binary executable files of unknown origin, obfuscation, and authorship. Unlike other contemporary binary analysis tools that monitored the progress of unpacking the target binary on an instruction-by-instruction (fine-grained) basis, Eureka only hooked the system call interface, specifically the NtTerminateProcess call (coarse-grained), to trigger a write to disk of the contents of the process memory. The assumption was that the unknown binary would be at its most unpacked state just prior to exiting.

Dinaburg et al. (2008) presented Ether, a malware analysis framework based on hardware virtualization extensions. By using hardware extensions, Ether was able to address the shortcomings of QEMU-based solutions. Alazab (2015) used Ether to study and characterize types of malware and variants in an effort to create a system that could accurately profile malware, allowing for the implementation of countermeasures in a timely and appropriate manner.

Rolles (2009) offered guidelines that could be followed to unpack binary executable files that were obfuscated with virtual machine type obfuscation.

Sharif, Lanzi, Giffin, and Lee (2009) demonstrated a proof-of-concept system called Rotalumé that had good success in extracting bytecode traces, syntax, and semantic information from the virtual machine obfuscated binary executable file specimens. Coogan, Lu, and Debray (2011) extended this previous work and proposed a method that would return 50-75% of the original binary executable file from its virtual machine based obfuscation.

Y. Wu, Chiueh, and Zhao (2009) created Uncover, a dynamic binary instrumentation tool that could accurately evaluate the workings of packed binaries by entropy analysis during execution to accurately track unpacking process during runtime. Uncover was shown to work well, but made assumptions that would prove troublesome for malware disassembly, including the inability to unpack sections of the executable at a time and that there would be no anti-disassembly routines embedded in the binary.

Coogan, Debray, Kaochar, and Townsend (2009) proposed a system that combined dynamic and static analysis by having the unpacking code identified using static analysis methods and then as a second step used these results to construct a customized unpacker for that binary. They report good results using this hybrid approach to disassembling unknown binary executable files.

Kawakoya, Iwamura, and Itoh (2010) developed a debugger that unobtrusively monitored memory accesses of a process for the purpose of returning page execute access requests in order to address the concern that some obfuscated binaries unpack only portions of their code at a time, or recursively pack their content.

Charif-Rubial et al. (2013) proposed MIL, based on MAQAO (Barthou, Rubial, Jalby, Koliai, & Valensi, 2010), which is a binary instrumentation tool that can accurately disassemble optimized code used in parallel processing applications.

Static Analysis

Alternatively, a binary executable file can be studied using static analysis techniques, which analyze the file without it being executed by the computer. Cifuentes and Gough (1995) described the structure of a decompiler for computer programs: a software program that could revert the binary executable back to its human readable source code. Cifuentes also presented an integrated environment for the reverse engineering of programs in (Cifuentes, 1995, 1996).

Schwarz et al. (2002) discussed the primary methods for disassembly: linear sweep, where the disassembler decodes everything in order that bytes appear in the binary file sections, and recursive traversal, where the disassembler starts at the entry-point of the binary file and traverses down through the executable following the flow and logic of the file. Schwarz et al. (2002) offered weaknesses in both methods and proposed to improve the accuracy of the disassembly process using either an extended linear sweep algorithm or a hybrid approach that was able to check the results of disassembly as the disassembly was happening. Vinciguerra et al. (2003) expanded on this discussion of the methods of disassembly, including Linear Sweep (LS), Extended Linear Sweep (ELS), Recursive Traversal (RT), Data-Flow guided RT (DRT), and Hybrid ELS/RT and the tools and procedures used to disassemble binary executable files.

Kruegel, Robertson, Valeur, and Vigna (2004) implemented a disassembler that used a control flow graph from an obfuscated binary executable as well as statistical techniques that could accurately disassemble programs obfuscated with the techniques described in (Linn & Debray, 2003).

Udupa, Debray, and Madou (2005) proposed a method that combined aspects of static and dynamic analysis and was based on the specific obfuscation mechanism being addressed. Royal et al. (2006) demonstrated PolyUnpack, a static unpacking tool that could be used to force a malware sample to self-identify as being packed by comparing its runtime execution against its static code model.

Caballero, Johnson, McCamant, and Song (2009) created a tool using hybrid disassembly, symbolic execution, jump table identification, and inference to extract functions from binary executable files for use in other source-based applications.

As defined in both the Obfuscation and the Dynamic analysis sections, virtual machine obfuscation continues to present challenges for accurate disassembly, and (Kinder, 2012) discussed potential strategies that might be employed to accurately analyze binaries using static analysis methods that have been obfuscated with virtualization type obfuscators.

Representation & Comparison

In much the same way as natural language authorship attribution first extracts measurable features from a document specimen for later comparison of authorship features, binary files must also first be analyzed before comparison. The first binary file comparison techniques involved simple byte/location based matching (Coppieters, 1995). Shortly following this byte-matching process, Zheng Wang et al. (1999)

recognized that programs could be broken down into execution blocks and that execution blocks from two programs could be very similar, if not identical. The researchers observed that a program could have "primary changes" – changes directly attributed to the programmer: addition of new code, deletion or modification of existing code. In addition to primary changes, "secondary changes" could also be observed as a result of the changes in control flow instruction targets, pointers, register allocation, etc. These secondary changes posed challenges to simple matching techniques. The researchers introduced BMAT, a tool that could match binaries with a measured accuracy of over 98%.

A further extension of the aforementioned research is presented by Flake (2004), where Flake described a system that created control flow graphs that represent binary programs. By transforming the binary program into a control flow graph, all compiler-introduced variability is removed, leaving only the specific flow of execution through the binary. Flake showed that this transformation to control flow graphs allowed for higher accuracy binary comparison as well as being able to identify differing regions in two file specimens.

Dullien and Rolles (2005) extended Flake's work by adding the capability to accurately represent differences between versions of files based on instruction ordering. Another challenge encountered with control flow graph based binary analysis is that certain changes to code, like loop iteration counts and buffer sizes do not change the graph representation, causing analysis to improperly match code that is different. In response to this challenge, H. Park, Choi, Seo, and Han (2008) proposed a solution using control flow graph analysis with the addition of constant values that have been

extracted from the code during transformation. These extracted values are then added to the analysis layer to further refine the matching process.

Gao et al. (2008) presented BinHunt, a tool that reported high levels of accuracy in identifying semantic differences between files based on the control flow of the program using a technique that involves a new graph isomorphism technique, symbolic execution, and theorem proving. Once transformed into graphs, the two specimen files were executed symbolically and the output of each graph was compared using a theorem-proving solver.

Another of the principle challenges in the use of control flow graphing as a mechanism to compare binary files, is the computational complexity of matching graphs (Unger, 1964). B. Kang, Kim, Kim, Kwon, and Im (2011) proposed the addition of a Bloom filter to the control flow graph analysis to reduce the processing overhead found in previous implementations. Similarly, Bourquin, King, and Robbins (2013) proposed a method of binary file comparison using control flow graphs with Hungarian equations as a method of reducing the computational complexity of the matching. This new method produced a tool called BinSlayer that, while still in development, showed great promise in terms of both matching accuracy and speed of comparison.

Alam, Horspool, Traore, and Sogukpinar (2014) put forward additional improvements in performance of control flow graph comparisons by adding annotations to the control flow graph and the addition of a sliding window of comparison to the generic control flow graph analysis technique. These improvements allowed for both performance increases as well as higher levels of accuracy in comparison, even when similar binaries have undergone external obfuscation or different compiler

optimizations.

Ryder (1979) presented an alternate, but similar graph based strategy for program analysis: call graphs. Call graphs are similar to program flow graphs, except that instead of building the graph based on conditional branching instructions, the graphs are built around system and function calls. As previously introduced, Harris and Miller (2005) created a tool called Dyninst that could be used to reconstruct stripped executable files using a combination of both control flow graphs and call graphs. (Kinable, 2010; Kinable & Kostakis, 2011) used call graphs to identify and classify malware. Their research demonstrated that function and system calls, and their frequency and patterns of use, could be used to create signatures that could be used to identify similarities in malware.

With an appreciation that call graph analysis suffers from the same computational complexity challenge as control flow graphs, Kostakis, Kinable, Mahmoudi, and Mustonen (2011) applied simulated annealing to the call graph analysis in an effort to reduce computational complexity. Simulated annealing takes a known configuration (in this case a set of call graphs), a configuration changing mechanism that operates at random, a comparison function to measure the improvement (if any) in the change in configuration, and a schedule to determine how long to run the simulation in order to automatically find the optimal matching solution (Kirkpatrick et al., 1983). Nascimento, Prado, Boccardo, Carmo, and Machado (2012) chose to implement a comparison system for binary files that extracted the program features from the call and control flow graphs and, instead of comparing sub-graphs for similarity, fed the extracted features into a feed forward neural network for classification of similarity.

In an effort to increase the match capability of the system, Joonhyouk Jang, Choi, and Hong (2012) proposed to add additional structural features of the files under test to the call graph based comparison. D. Qiu, Li, and Sun (2013) attempted to improve the matching capability of the algorithm by adding both the structure of the binary and a computed class diagram to the graph of the binary under examination.

Y. R. Lee, Kang, and Im (2013) worked toward increasing the speed of analysis by creating call graphs that were used not for comparison purposes, but for creating n-grams of the sub-trees that could then be used for comparison. These call-graph spawned n-grams have been shown to be as effective in matching binaries as a pure graph comparison approach while not being as computationally intensive as the pure call graph solving method. Similarly, Zhimin Yin, Yu, and Niu (2013) created call graphs of binaries, and then transformed the call graphs into a set of vectors that could be visualized using colors and shapes. Similar features within binaries are made apparent in the patterns and colors of the visualizations. Instead of control flow graphs as the input mechanism for signatures, Cesare and Xiang (2011) created signatures from call graphs.

Zhang and Reeves (2007) created a system called Metaaware that was able to identify metamorphic variants of a particular malware type based on the pattern of library and system functions that were called. Ahmed, Hameed, Shafiq, and Farooq (2009) demonstrated that spatial-temporal measurements of API calls in binary programs could also accurately and uniquely provide identification of a binary file. (Alazab, Venkataraman, & Watters, 2010; Iwamoto & Wasaki, 2012) recognized that the behavior of a piece of malware could be understood by extracting the API calls from

the binary. Han, Kang, and Im (2011) further demonstrated that malware could be classified by identifying the frequencies of instructions within the program.

One of the latest advancements in the field of call graph analysis for malware similarity comes from Kostakis (2014), where they created a new system named "Classy", a distributed system that allows for the clustering of large streams of call graphs. This distributed system has shown very good accuracy while maintaining acceptable levels of performance.

Other researchers have looked towards binary file comparison with the idea that if low-level representations of code were similar, then higher levels of code, including source code, should be similar as well (Chaki et al., 2011; Juricic, 2011). (Davies et al., 2011; Godfrey, 2013) proposed digital "Bertillonage", whereby coarser measurements of program similarity are compared with the assumption that if these coarser measurements are similar then it is reasonable to conclude that the sources were also similar.

Another means of classifying and comparing binary files involves the study of the binary file's behavior as it executes on a machine. Egele, Scholte, Kirda, and Kruegel (2012) provided an overview of the most common methods. Among the common methods is the family of methods that involve running the unknown binary in some type of sandbox, usually a virtualized machine, and recording its output (Moser, Kruegel, & Kirda, 2007a; Rieck, Trinius, Willems, & Holz, 2011; Trinius, Holz, Gobel, & Freiling, 2009; Trinius, Willems, Holz, & Rieck, 2009). Palahan, Babić, Chaudhuri, and Kifer (2013) were able to build a system call dependency graph that identified significant behaviors from binary specimens executed within a sandbox. However,

Huang, Lee, Kao, Tsai, and Chang (2011) noted that analysis done from within a virtualized system may result in inaccurate conclusions due to subtle differences in execution environments or program design. An analysis method that attempts to balance the fully managed environment of a virtual machine against the accuracy of "bare metal" execution is done by instrumenting the system in such a way that the system provides extensive monitoring and logging of its own state, such that the executable is being observed, not through its own execution, but rather by interaction with the running system below it (Apel, Bockermann, & Meier, 2009; Bai, Hu, Jing, Li, & Wang, 2014; Ding, Dai, Yan, & Zhang, 2014; Egele, Woo, Chapman, & Brumley, 2014; Qiao, Yang, He, Tang, & Liu, 2014).

One of the other longstanding methods of analyzing binary files for comparison purposes is the creation and matching of signatures derived from the binary file itself. Walenstein and Lakhotia (2007) and Walenstein, Venable, Hayes, Thompson, and Lakhotia (2007) used an approach based on vector comparisons to identify similar binaries. The vectors are based on both n-grams and n-perms (defined previously).

Zhiyi Yin et al. (2008) created a fingerprint of an executable using "color moments" in an effort to easily and accurately identify similar versions of files, while Barr, Cardman, and Martin Jr (2008) used an ensemble of clustering algorithms to create a technique that correlates both functions and global data references to provide quick and accurate comparison of two files.

(Acosta & Medina, 2012; Acosta, Medina, & Mendoza, 2012) developed an algorithm for comparing binary files by using the quantity and quality of the set of longest common substrings in each file as a basis for comparison. Blokhin, Saxe, and

Mentis (2013) extended this research of longest common string analysis methods with the addition of using run-time logs that featured call-stack data to create traces that can be compared for program similarity. Similarly, Adkins, Jones, Carlisle, and Upchurch (2013) used a sliding window approach to identify matching sections of code for the purpose of creating a similarity ratio and Liangboonprakong and Sornil (2013) used n-gram sequential patterns to identify features of malware. Luo et al. (2014) created a system named CoP that implemented two levels of analysis to create a more accurate, obfuscation resilient ensemble method for determining how similar two binaries were. First, the system symbolically modeled the program's basic block with a theorem solver, and then implemented a longest common subsequence algorithm that could accept non-matching portions of the sequence, effectively negating the effects of common code obfuscation methods.

Other methods, including the use of pushdown systems (stacks) (Macedo & Touili, 2013) and hidden Markov models to identify hidden states that could identify similar features of metamorphic malware have been investigated in an effort to accurately compare binary executable files (Austin, Filiol, Josse, & Stamp, 2013).

As discussed in the Compiler *Chain Provenance* section, (Farhadi, 2013; Farhadi, Fung, Charland, & Debbabi, 2014) have proposed a method to identify code clones in binary files. Code clones can be identified in a binary file, then used to provide reference to future clones, thus shortening the time required to complete a binary analysis.

McMillan et al. (2012) proposed the idea of comparing semantic anchors (for example API calls) between two files to identify their similarity. With API calls being

defined in purpose-oriented libraries, two programs using the same libraries and the same calls within the libraries share similarities. Additionally, the researchers defined weights based on commonality of API calls and patterns of co-occurrence of API calls in order to classify the similarity of Java programs. The combination of these three features produced similarity measures that were more accurate than other systems considered state of the art at the time. In an effort to identify similarity in binary fragments, Lakhotia, Preda, and Giacobazzi (2013) created a system whereby abstractions of the semantic actions of basic blocks could be compared measure similarity. These abstractions, call the 'juice' of a binary, could be used to compare basic blocks from within binary files because the abstraction process removed code variations like register renaming, memory address allocation, and constant replacement from the block representations. Their experimental results show that this new abstraction represented an improvement in previous uses of symbolic interpretation to discover similarity between code fragments. Chaki, Cohen, Gurfinkel, and Havrilla (2013) proposed a system to hash each basic block into a semantic signature for the purpose of identifying similarity in binary files that had different compiler optimizations enabled. Their use of hashes, unlike some other systems that compare graphs of basic blocks for similarity, resulted in good accuracy and good performance. Finally, Ruttenberg et al. (2014) created a system that demonstrated high accuracy results and was based on extracting semantic information from blocks of binary code for the purpose of identifying similar components in different specimens of malware.

Binary File Authorship Attribution

As early as 1993, Spafford and Weeber (1993) and again in Gray et al. (1997)

suggested that it should be possible to extract author identifying traits from computer binary programs for comparison purposes. Review of the literature suggests that most authorship attribution research has been directed toward the case where computer source code was analyzed rather than computer binary executable files. It appears that the first concerted effort to determine authorship in binary files was done by Rosenblum, Zhu, et al. (2011). In that research, a system was devised whereby code from an arbitrary binary file could be reduced to a set of predefined feature templates. Instead of dealing with the challenge of determining what features ought to be selected for author attribution *a priori*, Rosenblum, Zhu, et al. (2011) choose to focus on collecting all the features available in the program. These features were both numerous and very simple. After collection, the researchers allowed supervised machine learning algorithms to detemine the best features to classify authorship based on a set of programs with known authorship. The classification results were used to create a k-means clustering algorithm (Bishop, 2001) which was used to determine authorship in an unsupervised capacity.

Rosenblum, Zhu, et al. (2011) used feature templates as a means of describing patterns that would be instantiated into many concrete features of binary files. Their feature templates included idioms, graphlets, supergraphlets, call graphlets, n-grams, and external interations. While idioms captured low-level details of the instruction sequence underlying a program, graphlets captured details that represented program structure. Supergraphlets were made up of graphlets that were combined with a random neighbour and then collapsed. Call graphlets captured both the interprocedural control flow of a program as well as interactions with external libraries. Finally, n-grams are short sequences of bytes taken directly from the binary program itself and external

interactions with libraries were simply counted. All of the collected features from multiple examples of binaries written by a known author were then fed into a supervised learning classification system to determine how much each feature contributed to attribution and how accurately the identity of an author could be inferred from the generated model (Rosenblum, Zhu, et al., 2011).

The researchers also applied unsupervised clustering to determine how well the computed clusters reflected the actual authorship of a program and whether the stylistic characteristics learned from one set of authors could improve the clustering of programs written by different authors. The results of the classification system showed positive results with 77% accuracy in selecting the correct author from a pool of 20 and the improvement offered by the learned metric in the clustering algorithm was significant at a 95% confidence level for all measures (Rosenblum, Zhu, et al., 2011).

As part of their research, Alrabaee et al. (2014) found that the results published by Rosenblum, Zhu, et al. (2011) were overly optimistic compared to their efforts to replicate Rosenblum's findings. Alrabaee et al. (2014) concluded that many of the indicators of authorship identified by Rosenblum's classifier did not classify on programmer style, rather it classified based on similarities of function names or contants defined during compilation. As a result of these discrepancies, Alrabaee et al. (2014) proposed their own process for identifying the author of a binary file of unknown provenance.

The first steps of the process defined in Alrabaee et al. (2014) was to filter out the portions of the binary file that came from external libraries. The process of attribution continued on the user-contributed portion of the binary file with their Code

Analysis Layer (CAL), which compared the code in the binary to existing samples of programming contructs stored in a Software Template Library (STL). In addition to the comparison of code segments with those found in the STL, hashes were made of code blocks. Alrabaee et al. (2014) found that blocks of binary code from similarly coded programs will have a high ratio of matched hashes. Finally, code sections were reduced to Register Flow Graphs (RFG) for the purpose of creating generic signatures of blocks of code which would allow for inexact matching of signatures between two blocks. All of these data were directed towards a supervised machine learning classification algorithm with the goal of creating an author profile of each author of a set of specimen programs. Once these profiles were created, binary files of unknown provenance could be analyzed and compared to the profile data in order to match the author.

The classification scheme used in Alrabaee et al. (2014) is based on a hierarchical classification system as described in Barthélemy, Brucker, and Osswald (2004) and contains a manual step which allows for intervention into the classification process should the case arise where there the number of selected classes "for a given author are too numerous" (Alrabaee et al., 2014, p. 101). A second potential limitation in the classification strategy used in Alrabaee et al. (2014) is the requirement to define a similarity function for the valued class system. These similarity functions can introduce bias into the classification system, especially where there can be multiple interpretations and thus measurements of similarity between discrete attributes (Witten et al., 2011).

Alrabaee et al. (2014) report lower false positive rates of author identification in cases of attribution of a binary file of unknown provenance than the results reported in Rosenblum, Zhu, et al. (2011), but admitted there is additional research to be done

regarding binaries created using different programming languages, programs compiled using different compilers, and programs that have been obfuscated. In a subsequent survey paper on the feasibility of malware authorship attribution, a research team made up of three of the four original researchers conducted experiments that showed that some feature ranking was an incorrect byproduct of the IDA-Pro (Hex-Rays SA, 2015b) disassembly process rather than features of author style (Alrabaee et al., 2016).

In Caliskan-Islam, Yamaguchi, et al. (2015), the researchers added additional features to the the feature set that would be used in the classification of authorship in binary exectuabe files. They used the results of two automated decompilation tools (Hex-rays (SA, 2016) and Snowman (Derevenets, 2016)) to identify lexical and syntactic features of the files under examination. The syntactic features came from passing the decompiled pseudo-code into a tool called joern (Yamaguchi, 2016), a tool that is capable of producing fuzzy Abstract Syntax Trees (ASTs) from decompiled pseudo-code. ASTs have been used in previous research on author attribution of computer source code because ASTs have been shown to capture features within them that are highly effective in representing programming style (Caliskan-Islam, Harang, et al., 2015; Wisse & Veenman, 2015).

During the process of decompilation, Snowman (Derevenets, 2016) creates control flow graphs. Caliskan-Islam, Yamaguchi, et al. (2015) used features taken from the control flow graph data in addition to using the decompiled pseudo-code produced by Snowman during the classfication step.

Finally, in addition to the features contributed from the decompilation and control flow graph steps, Caliskan-Islam, Yamaguchi, et al. (2015) used two different

disassembler engines (radare2 (pancake, 2016) and nasm (NASM development team, 2016)) to extract the op-code instruction data from the executable files.

The sheer number of attributes that were created as a result of these analysis steps forced the use of information gain (Quinlan, 1986) calculations to reduce the dimensionality of the feature set. The reduced feature set was then used to train a random forest (Breiman, 2001) classifier using k-fold cross-validation (Witten et al., 2011). In this research, the number of folds (k) was determined by the number of binary samples contributed by each author. The number of trees created in the forest was set to 500, which was empirically determined to provide "the best tradeoff between accuracy and processing time" (Caliskan-Islam, Yamaguchi, et al., 2015, p. 6). These researchers have demonstrated higher classification accuracies on significantly larger author pools than previously attained, a limitation identified in the research of both Rosenblum, Zhu, et al. (2011) and Alrabaee et al. (2014).

Like the previous research, Caliskan-Islam, Yamaguchi, et al. (2015) leaves the issue of binary file obfuscation to future research. While this research shows promise in the area of authorship attribution in binary executable files, a full 58 percent of the features used in classification came from the decompilation phase. As previously discussed in the *Binary Obfuscation* section, binary obfuscators significantly reduce (if not eliminate, based on the complexity category (see Table 2)) the ability to accurately decompile binary files back to their human readable source (Collberg & Thomborson, 2002; Collberg et al., 1997).

Summary

This literature review began by outlining key research accomplishments in authorship attribution of natural language documents. Progress has been made, from the simple counting of characters and character combinations to the use of sophisticated machine learning algorithms to identify the most distinguishing attributes in a dataset of text to the classification of authors based on those attributes, with exceptional accuracy.

It's not surprising that researchers then looked towards identifying those stylometric attributes in computer source code for the purpose of attributing authorship in computer programs. Advances in this field were also accomplished in lock-step with advances in machine learning capabilities.

If one considers that executable binary files are simply these human readable source code files transformed by the compiler into the binary language equivalent for the target processor (Aho et al., 2006; Grune, Van Reeuwijk, Bal, Jacobs, & Langendoen, 2012), it's reasonable to attempt to apply these same classification procedures to the compiled code. The task of authorship attribution of machine readable executable files will necessarily need to separate the contributions of the author and of the compiler in order to achieve high classification accuracy.

Working against the accurate classification of binary executable files, the work product of human programmers (and the functional requirements of same) continue to evolve over time. Snippets of code, functions, or even full classes of human readable source code can be copied from any number of sources available on the Internet today. These copied components may not follow the author's native style of authorship. Both

the injection of plagiarized content and the natural evolution of programmers and programs must be accommodated in order to achieve the classification accuracy goals.

Further, these executable files can be obfuscated using a plethora of options, each of which contributes to an increase in difficulty of reverse engineering the executable file for the purpose of analysis, description, and attribution. As Collberg et al. (1997) suggested, extracting the original executable file from the obfuscated derivative may not be easily done.

Obfuscated or not, one of the primary methods for reverse engineering these executable files back into some form that is human-readable is known as disassembly. There are two primary methods of disassembly: dynamic and static disassembly. Dynamic disassembly involves tracing the program instructions as they are executed by the processor. The disadvantage of dynamic disassembly is that decision branches that are not executed are not included in the disassembly, which reduces the percentage of the file that is disassembled, thus reducing the number of instructions available for analysis. Static disassembly involves tracing the program execution using an algorithm to traverse the instructions. The disadvantage of this method is that Intel X86/X64 assembly code is self-healing (Linn & Debray, 2003), which allows for the possibility of inaccurate or incomplete disassembly based on an incorrect offset into an instruction or block of instructions.

Regardless of the method of disassembly, the human-readable program that is recovered during disassembly must be presented and stored in a format that allows for accurate and timely comparison. The challenges discussed in this section regarding these two competing requirements demonstrate that accurate representation and

comparison of binary file artifacts is an area of research that still holds significant opportunity for advancement.

Research has extended the study of attribution to binary files, with good success, but an open and remaining problem in binary file attribution is applying this attribution capability to files that have been obfuscated.

Chapter 3

Methodology

Overview

In this chapter, a detailed methodology of the experiments that were designed to support the research objectives are described. The research objectives, as captured by the Research questions and Dissertation goal sections, were to 1) classify authorship of obfuscated programs, 2) identify attributes that lend themselves towards classifying the author of an obfuscator and, by extension, fingerprinting the obfuscator, and 3) identify the impact of tool-chain provenance on accuracy of attribution.

Both the Google Code Jam (Google, 2015) specimen files and the selected obfuscator tools were downloaded from the Internet. The Google Code Jam (GCJ) files were compiled using compiler-specific default settings. A baseline authorship attribution accuracy was measured using the procedure to be described in the following subsections. After this baseline was determined using the executable files as they existed prior to obfuscation, the executable files were transformed using the downloaded obfuscator tools.

The raw data used in the analysis of authorship attribution of the obfuscated executable files was collected using two different methods. The first method was based on static analysis, an analysis technique in which the file is not executed during the disassembly or data collection process. The second method was based on dynamic analysis techniques, which does include executing the file during the process of data

collection. The raw instruction data underwent feature extraction, primary component analysis, and mapping to feature vectors. These feature vectors were used in a supervised machine learning environment for the determination of authorship. In the following sections, the preparation, compilation, feature extraction and analysis, and machine learning algorithms will be discussed in greater depth.

Following the descriptions of the experiment preparation steps and experiments themselves, this chapter concludes with the following sections: Instrument development and validation, Format for presenting results, Resource requirements, and Summary. These final sections serve the purpose of adding richer technical context to the discussions of specific experiments and set the foundation for the findings to be discussed in the next chapter.

The Obfuscation Tools

Selection and Collection of Obfuscation Tools

As discussed in the Literature Review section, there are literally hundreds of obfuscation tools in use today, each implementing different features designed towards making the obfuscated binary resistant to analysis or reverse engineering. Obfuscator samples were selected using purposeful maximal sampling (Creswell, 2012) of specific obfuscators from within the full set of available binary file obfuscators. Purposeful maximal sampling refers to the selection of specific cases that show as wide a selection of perspectives as possible (Creswell, 2012).

By using purposeful maximal sampling, uniform coverage from all the complexity types listed in Table 2 was assured. With assistance from the researchers who authored Ugarte-Pedrero et al. (2015), a list of obfuscation tools and their complexity types was

compiled and can be found in Table 3. The experiments listed in the remainder of this Methodology chapter used these specific obfuscators as representative samples for each complexity type.

As shown in Table 3, a mix of obfuscators from each of the six complexity categories were downloaded from sites on the Internet. The exact download locations of the obfuscators can be found in Appendix E.

Complexity Type	Obfuscator
Type 1	UPX 3.91 ASPack 2.40 PESpin 1.33
Type 2	YodasCrypter 1.3 JDPack 1.01 DotFix NiceProtect 6.0
Type 3	UPolyX 0.5 PECompact 3.02.2 ASProtect 2.76
Type 4	ACProtect 1.10 ExeStealth 3.17
Type 5	Beria 0.7 EXPressor 1.8.0.1
Type 6	VMProtect 3.03 Themida 2.4.6.0 Armadillo 8.0.4

Table 3 – Obfuscation samples by complexity type

The Data Set

The specimen files came from the Google Code Jam (Google, 2015) contest, a contest that has been running since 2008, which features multiple submissions by individual developers solving specific computing challenges. Submissions from contestants who did not advance to the finalist stage were excluded due to an insufficient number of submissions with which to build an accurate characterization of the author. The number of problems and the number of competitors who reached the finalist stage and have taken part in the competition over multiple years did provide an

ample corpus of source code that could be compiled into binary executable files and then obfuscated for both describing stylometric similarity and authorship attribution testing.

Preparation of Data Set

A Python script made available by the researchers who authored Caliskan-Islam, Harang, et al. (2015) was used to scrape the Google Code Jam pages for the contest standings and links to the file submissions. Once these standings and submissions were collected, a second python script was used to download each submission in .zip file format. The submissions were downloaded into a directory structure that captured the year of submission and the username of the submitter:

- 2008
  - ACRush
    - 32001.24439.0.zip
    - 32002.24444.1.zip
    - …
  - Ahyangyi
    - …
- 2009
  - ACRush
    - …
  - AdrianKuegel
    - 186264.171116.0.zip
    - 186264.171116.1.zip

…

- 2010

…

Compilation of Data Set Specimens

This research required three different configurations of compilation of the specimen files, using three different compiler tools. The three configurations were "default", "optimized for execution speed", and "optimized for (minimal) file size". The compiler configuration flags can be found in Table 4.

	Default	Optimize for Speed	Optimize for Size
MS VC++ (Microsoft, 2016b)	<n/a>	/O2	/O1
Intel ICC (Intel, 2016)	/0d	/01	/02
GNU GCC (GCC, 2016)	-O0	-O2	-Os

Table 4 - Compiler and optimization settings

For each compiler product, a Microsoft Windows 7 x86 virtual machine was created, configured with two processor cores and two gigabytes (GB) of memory. Refer to Table 5 for product and patch version information for each compiler product. Required run-time dynamic link libraries (as determined by the compiler product) will be copied to the DECAF (Henderson et al., 2014) system to ensure successful program execution.

	Version	Patch Level
MS VC++ (Microsoft, 2016b)	2013	VS2013 Update 5
Intel ICC (Intel, 2016)	2016	Update 2
GNU GCC (GCC, 2016)	4.9	.3

Table 5 - Compiler version and patch level

While the GCJ contest does not place restrictions on the programming language of the submissions, this research used only those submissions that were written in C/C++. Any submission that did not compile cleanly using the compiler options

specified in this section were excluded. Submissions that compiled successfully but issued warnings during compilation were accepted as specimens for this research.

For each coding challenge, participants were allowed to submit a "short version" and a "long version" of their solution (Furrow & Naverniouk, 2016). Each authorship attribution experiment used only one of the two submissions per coding problem per author in the training set.

Static Disassembly and De-obfuscation

Many of the obfuscation techniques listed in Table 1 break the algorithm used for disassembling code, whether the algorithm is based on linear sweep or recursive descent (Eagle, 2008). The use of these obfuscation techniques mandates the use of an unpacking utility prior to disassembly. Even though some obfuscation tools offer companion utility programs that can be used to de-obfuscate an executable file that was obfuscated by that specific obfuscator, to maintain consistency across all the obfuscators these companion tools were not used, even though these tools may have resulted in slightly higher prediction accuracies. Instead, all the obfuscated specimens were first deobfuscated using the GUnpacker tool (Quick Unpack, 2017), as described in Figure 1.

Dynamic Disassembly and De-obfuscation

In addition to static disassembly, each specimen file was disassembled using dynamic analysis techniques using a full system emulator called DECAF (Henderson et al., 2014), which is based on QEMU (Bellard, 2005). As discussed in the Literature Review section, dynamic analysis has the advantage of capturing execution at the instruction level as the binary application is being executed.

Feature Extraction

Previous research has defined a set of feature types that exist within binary executable files and can be used to accurately attribute the author of the executable file (Alrabaee et al., 2014; Caliskan-Islam, Yamaguchi, et al., 2015; Rosenblum, Zhu, et al., 2011). The feature types identified in these past research efforts are listed in Table 6, but have been organized by the similarity of the feature type rather than by the original researcher to identify the feature type. In this research, n-grams (Rosenblum, Zhu, et al., 2011), disassembly unigrams, and bigrams (Caliskan-Islam, Yamaguchi, et al., 2015) have been simplified and reduced to the single feature type of opcode.

Feature	(Rosenblum, Zhu, et al., 2011)	(Alrabaee et al., 2014)	(Caliskan-Islam, Yamaguchi, et al., 2015)	This research
N-gram	X			Implemented as opcode
Disassembly unigram			X	
Disassembly bigram			X	
Stuttering layer		X		Implemented with FLIRT (Hex-Rays SA, 2015a) and data post-processing.
Graphlets	X			Implemented as control flow graph hashes.
SuperGraphlets	X			
Call graphlets	X			
Exact match		X		
Inexact match		X		Implemented as Register Flow Graphs.
RFG		X		
Idioms	X			
Decompiled features			X	Not used.
Library calls	X			Not used.
STL repository		X		Not used.

Table 6 - Features identified in binary executables for author attribution

In Alrabaee et al. (2014), the stuttering layer was primarily responsible for separating user and system portions of the application. In this research, FLIRT (Hex-Rays SA, 2015a) was used to separate user and system code in the static method, and a post-processing step which removes all instructions located in the system portion of the memory map was used in the dynamic method.

Graphlets, supergraphlets, call graphlets (Rosenblum, Zhu, et al., 2011), and exact matches (Alrabaee et al., 2014) have been reinterpreted as hashes of control flow graphs. Classification is computationally simplified by taking the hash of the control flow graph (Alrabaee et al., 2014). Idioms (Rosenblum, Zhu, et al., 2011) and inexact matches (Alrabaee et al., 2014) have been interpreted as equivalent to Register Flow Graphs (Alrabaee et al., 2014) and have been captured as a feature type in this research. Register Flow Graphs capture the flow and semantics of a control flow graph (Alrabaee et al., 2014).

Obfuscated programs cannot be readily (or accurately) decompiled, thus none of feature types based on program decompilation are included in this research (Caliskan-Islam, Yamaguchi, et al., 2015). Similarly, library calls are stored in the Imports section of the Portable Executable (PE) file (Pietrek, 1994), but obfuscators typically remove this section (Roundy & Miller, 2013). Finally, the Software Template Library (STL) (Alrabaee et al., 2014) stored collections of opcodes that, when used together, typically represent some common programming syntax. Since obfuscators insert junk operations, unroll loops, and insert additional subroutines (Collberg et al., 1997; Roundy & Miller, 2013), no feature types based on the STL will contribute to the classification process.

However, because the obfuscated samples will still have an observable behavior that is identical to the original sample (Collberg et al., 1997), elements of original structure of the sample must still exist within the obfuscated sample. Even though these original elements were obfuscated, the nature of the supervised learning classification algorithms used in this research allowed for the identification of similarities, and those similarities were weighed, combined with others, and then were used to compute a likelihood of similar authorship in the samples.

Having chosen the feature types to use for author classification, the assembly language formatted output of the static disassembly and dynamic instruction collection was put through an automated process to separate the user instructions from the system instructions, collate the opcodes, separate the program into a set of control flow graphs using the procedure described in Allen (1970) and compute the hash of each graph, and generate a set of Register Flow Graphs (RFG) using the procedure described in Alrabaee et al. (2014).

Primary Component Analysis

For each experiment, a set of specimens was chosen at random for each of the selected authors, with each author also having been chosen at random. The management tool, discussed in detail in the Instrument design and validation section, used a random number generator to choose the files from the set of unique submissions from each author. Once the full set of specimens was determined, a set of distinct feature vectors was created from the union of all the feature vectors extracted (as described in the previous subsection) from each specimen in the experiment set. These feature vectors were then converted into a probability map based on the frequency of occurrence in each

specimen (Tanguy, Urieli, Calderone, Hathout, & Sajous, 2011). For every obfuscated specimen analyzed using the dynamic method, the set of feature vectors exceeded 100,000 elements. Similarly, for many of the obfuscated specimens analyzed using the static method, the set of feature vectors exceeded 100,000 elements. Analysis to determine the principal components of this full set of feature vectors was conducted to reduce the number of feature vectors used in the classification step. The algorithm used to compute how much any single feature vector contributed to the accurate classification result is called Information Gain and is described by the following formula:

$$IG(A, M_i) = H(A) - H(A| M_i) \quad (1)$$

where A is the class that represents the author, H is Shannon Entropy, and M_i is the i^{th} feature vector in the experiment's data set (Quinlan, 1986; University of Waikato, 2016). The full set of feature vectors was then ranked according to their Information Gain, or the relative amount that the feature vector contributed to the classification of the author. The number of feature vectors used in the final classification was studied through experimentation, and it was found that limiting the number of feature vectors to a maximum of 500 had no noticeable effect on prediction accuracy, and kept the classifier training time to an acceptably small duration. Caliskan-Islam, Yamaguchi, et al. (2015) also concluded that 500 feature vectors was an optimal number of feature vectors, balancing both accuracy and training time.

Classification

Prior research into attribution of authorship to binary files has used the supervised classification algorithms of support vector machine (SVM), single hierarchical classification, and random forests. Given the potential performance

limitations when using an SVM on very large dimensioned datasets, and based on the previously discussed limitations of hierarchical classifiers, and the natural fit of random forests for datasets that have weak inputs, this research used a random forest classifier. The specific implementation of the random forest classifier was the one found in Weka (University of Waikato, 2016).

Baseline Attribution

Experiments were undertaken to address the Research Questions posed earlier in this Research Report. For each of these experiments, described in the following paragraphs, a common procedure of comparing the results of the stylometric similarity process (feature extraction and classification for stylometric similarity) was conducted.

In each experiment, the disassembly data of the original executable (not obfuscated), the disassembly data recreated via static analysis, and the disassembly data recovered using dynamic analysis techniques were used as input to the feature extraction and classification process (described earlier). The results of the classification of the original sample executable (not obfuscated) was used as a sort of baseline measure of classification accuracy.

Attribution of Obfuscated Files

The first experiment involved assessing the stylometric similarity between native, never-having-been-obfuscated versions of the specimens of each of the contributing authors using a stratified tenfold cross-validation process (Witten et al., 2011). Tenfold cross-validation involves splitting the dataset into ten equal shares and each of the ten shares is withheld from the set for testing purposes after training the classifier with the other nine shares (Witten et al., 2011). The final result was computed

as the average from the ten iterations of the test (Witten et al., 2011). The results of this experiment were used as a baseline of attribution accuracy for use in comparison with the attribution accuracy of obfuscated examples.

Whereas the first experiment involved assessing for stylometric similarity in native, never-having-been-obfuscated specimens, the second experiment assessed for stylometric similarity in the obfuscated specimens themselves. For each of the complexity types, and for each of the contributing authors, a tenfold cross-validation process (Witten et al., 2011) of collecting stylometric similarity markers was undertaken using the disassembled opcode instructions in mnemonic form of the obfuscated versions of the specimen files. Obfuscation tools are designed to hide both the intent and specific implementation details of the executable files that are to be obfuscated, but should also reduce or remove stylometric markings as well. It was expected that an effective obfuscation tool would remove all traces of any stylometric feature that could have been used to attribute authorship.

These first two experiments provided data that was used to address Research Questions One and Two. Research Question One focused on classes of stylometric markings that persisted through the obfuscation/de-obfuscation process. Similarly, Research Question Two concerned the extent to which the method of analysis (static disassembly vs. dynamic reconstruction) had an impact on the accuracy of authorship attribution. Since these first two experiments extracted data from the executable files using two separate procedures, one using static disassembly methods, and one using dynamic instruction collection methods, it was expected that there would be differences in attribution accuracy between these two methods. It was hypothesized that the

accuracy of predictions of author attribution will be both higher and more stable across different obfuscation tools using the dynamic method of data collection than when using the static method of disassembly. This hypothesis is supported by the previous discussion on the practical implementation problems inherent with Intel X86 disassembly and the ground-truth execution platform that the CPU, even an emulated CPU, must provide to the higher-level operating system and application programs.

Stylometric Markers within the Obfuscation Routines

Research Question Three posed questions regarding challenges that may have existed with respect to fingerprinting the obfuscator code based on the transformations done to the original file. Two experiments were designed to provide insight into this question.

Experiment three used a single obfuscator, but the experiment was repeated for each obfuscator in the list of selected obfuscation tools (see Table 3). In much the same way as Collberg described an obfuscated binary as being made up of two distinct elements: the original binary file and the obfuscation routines (Collberg et al., 1997), this experiment used samples randomly selected from different authors solving different problems but all having been obfuscated by the same obfuscator. By using specimens from different authors solving different coding problems, the information gain on features not contributed by the obfuscator were minimized. This experiment followed the same stratified tenfold cross-validation process (Witten et al., 2011) described previously. It was expected that the transformations and additional instructions, and/or subroutines applied to the original file by the obfuscator would be separable from the original program. By being able to separate the features of the original program from

the transformations applied by the obfuscator, the machine learning classifier would be able to accurately predict the obfuscation tool used to create the sample.

The second experiment designed to address Research Question Three involved obfuscating a single author's specimen files with different versions of a single obfuscator. This set of obfuscators included both simple generational iterations and forked versions that potentially had different features implemented. A set of submissions was created, containing the programming challenge submissions from a single participant from the Google Code Jam Contest (Google, 2015). By using specimens from the same author, the information gain on features not contributed by the obfuscator was minimized. Once selected, these specimens were obfuscated with different versions of the same obfuscator and then used to train a machine-learning algorithm to detect variations in obfuscation techniques. Given the hypothesized result from the last experiment, where the classifier was expected to differentiate between obfuscation code and original code, it was hypothesized in this experiment that the machine learning classifier would be unable to differentiate between versions of a single obfuscator due to stylometric consistency across their generational versions.

This experiment focused solely on the complexity type 1 obfuscator, UPX (Markus F.X.J. Oberhumer, 2017a). Multiple, archived versions of UPX were available for download, unlike most other obfuscators being studied. The archived versions were available at Sourceforge (2015).

Tool-chain Impacts to Attribution

The fourth Research Question involved the changes to binary files when using different compilers and/or different optimization and compilation options that could

affect the accuracy of authorship attribution. Based on the similarities between compilers and obfuscators, it seemed scientifically reasonable to apply the same process of analysis to files generated by different compilers and their different settings as those files that have been obfuscated using some obfuscation tool. A fixed number of compiler options were used, those that are commercially available for MS Windows:

1. Microsoft Visual Studio (Microsoft, 2015)
2. Intel Compiler (Intel, 2015)
3. GCC (GCC, 2015).

Three experiments were designed to address the question posed in Research Question Four. The first experiment involved setting a baseline for the additional two compilers listed above. Experiment one from Research Question One was reproduced using these additional compilers. The expected result was that each different compiler, while possibly encoding features using a different format or schema, would be consistent in those encodings and provide similar attribution accuracy results as the baseline identified in experiment one.

The second experiment designed to address Research Question Four involved using two optimization classes for each compiler. Two classes were examined: optimizations applied to ensure fastest execution speed, and optimizations applied to ensure the smallest size of executable (see Table 4). For each of the compilers, for each of the two optimization classes, experiment one from Research Question One were reproduced in order to identify the effect that these two types of optimization had on the measured accuracy of attribution classification. It was hypothesized that as the level of optimization changed, the level and number of stylometric markers capable of

substantively contributing to the accurate classification of authorship would be changed, likely resulting in weaker prediction results.

The last two experiment designed to address Research Question Four involved testing for authorship attribution on files that were compiled using different compilers and different optimizations. In these previous experiments, the compilation process had been common and consistent for all the samples in all the author groups. In this experiment, each sample being classified for authorship was randomly compiled using one of the available compiler and optimization configurations from the previous two experiments. Given the three different compilers listed above, with each compiler having the two optimization settings (default, speed, and size, see Table 4), samples from each compiler/optimization class were compared to randomly chosen samples from each of the eight other compiler/optimization classes using the stratified tenfold cross-validation process (Witten et al., 2011) described earlier. As discussed in the previous experiment, differences in compiler optimization, for either executable size or execution speed, should negatively affect the number and quality of stylometric markers available for classification; however, this reduction should have been somewhat mitigated by the commonality of the compiler itself. In this experiment, the learning algorithm will not have any consistent compiler or optimization features that will naturally be de-emphasized due to their existing in every sample. It is expected that there will be no significant predictive correlation between executable file and author given the broad variation in compiler and optimization settings.

Instrument Development and Validation

A schematic diagram of the experimental setup is presented in Figure 1. The primary components of this setup include the management tool, the automation bridge, and the DECAF environment, which includes the trace_cap plugin.

The management tool was developed specifically for this research, and allowed for the creation, modification, and exploration of the experimental data. Details of the Google Code Jam problems, submission and submitter details, experiment details, and preliminary data could be viewed using this management tool. The management tool was also capable of defining experiments, selecting samples at random (using the built-in random number library) based on experimental criteria, and generating the ARFF formatted files used by Weka (Henderson et al., 2014).

A key component of the dynamic analysis system, the automation bridge was connected to the primary data store to pull contest submission control data from the data store. This control data was sent to the virtualized Windows 7 machine running within the DECAF environment (Henderson et al., 2014) to control the execution of the samples and the data collection. By controlling the execution of the samples in this way, reliable and repeatable execution of hundreds of file samples, each potentially having their own specific input and output requirements, could be accomplished.

DECAF (Henderson et al., 2014) is a dynamic analysis framework built on QEMU (Bellard, 2005) and was used for all dynamic analysis tasks. DECAF was downloaded from the Git repository (Sycurelab, 2016) of the project on 2015-12-20. DECAF was run from within a virtual machine image running an up-to-date (as of 2015-12-20) Ubuntu Desktop 14.04.3 LTS (Ubuntu, 2016) system with the current version

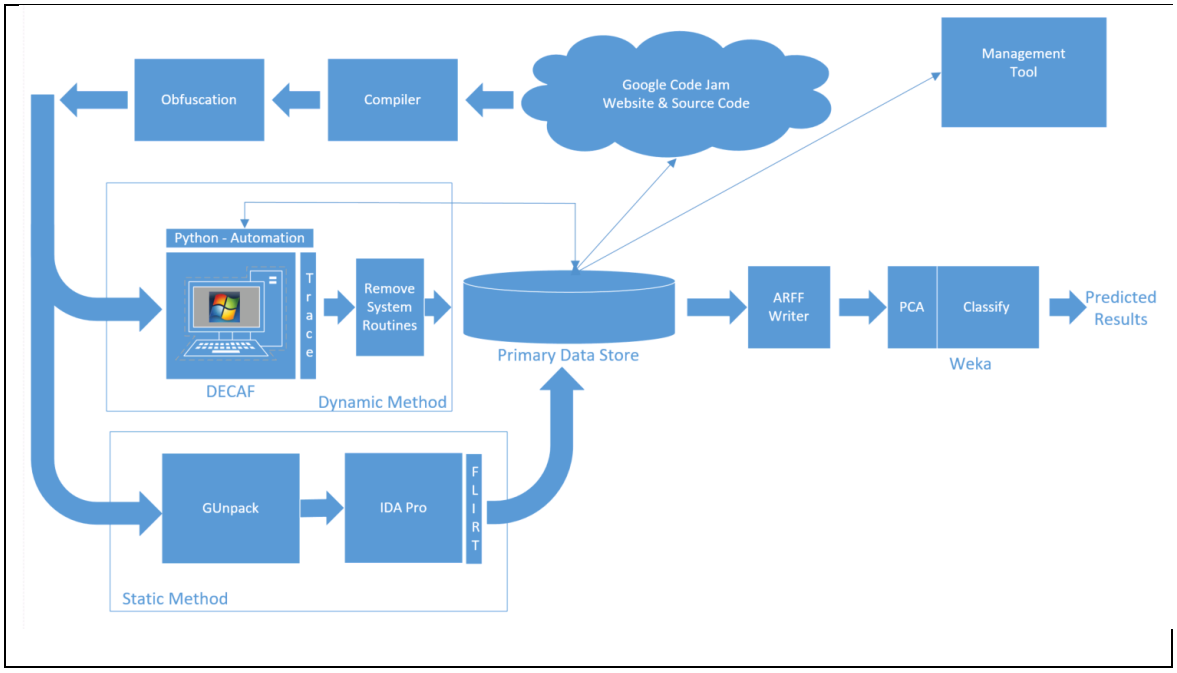

Figure 1 - Schematic diagram of experimental setup

of QEMU (version 2.0.0) and its development tools installed as a layered product.

One of the plugins that is bundled with DECAF (Henderson et al., 2014) is called "tracecap". This tracecap plugin records every instruction that is executed by the program under test by intercepting instructions as they are processed by the DECAF platform (Henderson et al., 2014). Tracecap writes all the instruction detail to a binary file on the host computer. A helper utility is also provided, trace_reader, that is used to extract information from the binary trace file and present it in a human readable format. Trace_reader was used to recover the binary instructions from the specimen executable. Once the instructions were translated into the standard mnemonic format, the tools described previously were used to format the data for use in Weka (University of Waikato, 2016).

A standard Microsoft Windows 7 X86 image (with Internet Explorer 11) was downloaded from the Internet Explorer Development home page (Microsoft, 2016a) and was used, without modification (beyond formatting the emulated machine image for the execution of the test specimens). The build version of machine image used for this research is 20141027. The image downloaded from Microsoft was in the industry standard OVF format (Distributed Management Task Force, 2016), which required a translation (prior to first use) from this format to the QEMU .qcow2 version 0.10 format (WikiBooks, 2016). A utility for translating these image files is bundled with QEMU, and was used to translate the stock Microsoft Windows 7 image to an equivalent image in the format supported by DECAF (Henderson et al., 2014).

The management tool (including the .ARFF (University of Waikato, 2016) writing capability) was written in C# using Microsoft Visual Studio 2013 (Microsoft, 2015).

Python (Python Software Foundation, 2015) was used as the default scripting language for tasks requiring simple transfer and transformation capabilities. The specimen files were collated and transferred from the Google code repository using scripts written in Python (Python Software Foundation, 2015). Scripts allowing for the automation of the DECAF (Henderson et al., 2014) analysis tasks were also written in Python (Python Software Foundation, 2015).

Format for Presenting Results

The summary results from each experiment are presented in graph form for maximum clarity and will reside with the summary description of the experiment. The results record the prediction accuracy of authorship attribution for the given obfuscator

in the given scenario. Comparable results will be displayed within common tables and graphs to facilitate comparison between analysis techniques and/or obfuscators.

Tables containing all the detailed results of each experiment can be found in the appendices.

Resource Requirements

Beyond the tools developed specifically for this research and described in the Instrument Development and Validation section, the other external tools and software packages required to complete this research included IDA-Pro (Hex-Rays SA, 2015b), QEMU (Bellard, 2005), DECAF (Henderson et al., 2014), and Weka (University of Waikato). Besides IDA-Pro, these tools are available freely by download. Visual Studio (Microsoft, 2015) and the Intel Compiler Set (Intel, 2015) were downloaded and used during a limited time evaluation period. GCC (GCC, 2015) is free and open-source and was simply downloaded. TortoiseSVN (TortoiseSVN, 2016) was used as a client side tool to track and manage all versions of the software written for this research.

Experiments were performed on a 64bit, 4-core Intel Core i5 processor with 3.0 GHz clock speed, 8GB of RAM, and Microsoft Windows 7 operating system. Extracted feature data and experiment configuration data was stored in an MSSQL 2012 (Microsoft, 2016c) database. Backup copies of the tools, specimen files, and data were stored to Google Drive as off-site backups.

Summary

This research had three primary objectives: 1) classify authorship of obfuscated programs, 2) identify attributes that lend themselves towards fingerprinting an

obfuscator, and 3) identify the impact of tool-chain provenance on accuracy of attribution. These objectives were mapped directly from Research questions one through four.

By using computer programs from the Google Code Jam (Google, 2015) contest, and accepting the assumption that each submission was the individual work product of the participant of the contest, the executable file specimens could be attributed to a single programmer. Features embedded in these executable files could be identified and extracted to classify for authorship attribution prior to obfuscation. These classifications were used as a baseline of attribution accuracy against which the classification of obfuscated samples was compared.

Previous research had consistently demonstrated that there existed a sufficient number and quality of features in computer source code to attribute authorship. Even though compilation is generally non-deterministic (Lamb, Lunar, Levsen, Cascadian, & Luo, 2016), there would exist some subset of those source code-based authorship features in the resulting compiled executable. In this research, the raw data set from which these features were identified and extracted are the processor instructions as decoded by a disassembler, in the static analysis case, and by dynamic instruction collection from within an emulated system, in the dynamic analysis case. These two different methods of raw data extraction, static and dynamic, were followed in order to shed the light of understanding on how the method of extraction affects the final measurement of accuracy of attribution. Data extracted via the static method might not have been accurate due to remaining artifacts of obfuscation or errors in the disassembly algorithm (either due to algorithmic flaws or purposeful injection of junk material into

the byte stream). Similarly, data collected via dynamic analysis methods might have been incomplete due to the execution of only a single decision path during program execution.

The collected data was then analyzed using the latest machine learning algorithms, first to determine the features that were the most author discriminating, and second to build a classifier that could be used to predict the author of unknown software samples.

Research Question five remains unaddressed in this research, as there were significant challenges in providing a rigorous response to that question. While the Terms and Conditions (Appendix D) of the Google Code Jam (Google, 2015) contest prohibit jointly developed submissions, there was no way to ensure this prohibition was enforced. Similarly, there were not enough samples from each contributor, over enough contest years, to ensure accurate results with good levels of generalizability. As such, Research Question five will be addressed in future research.

Chapter 4

Results

Overview

This chapter presents the results of the experiments described in the Methodology chapter. There are three major sections to this chapter: Data analysis, Findings, and Summary. In the Data analysis section, the results of the experiments will be presented. The results will be presented without extended discussion in the Data Analysis section, but will be thoroughly discussed in the Findings section. Finally, the Summary will offer a reiteration of the important aspects of these findings.

Data Analysis

The Data Set

A set of 1863 programs were selected from the Google Code Jam (Google, 2015) contest from between 2008 and 2015. Each of these programs was chosen randomly, using the management tool described in the previous chapter, from the set of all programs written by an author, again chosen at random to meet the requirements for sample size. All of samples were collected from the Google Code Jam website (Google, 2015) using the procedure described in the Methodology section of this report. Each of the selected programs, written in either C or C++ were compiled per the requirement of the specific experiment.

Experiment #1: The Baseline Study

This first experiment set a baseline for accuracy of author attribution given the methodology developed in this research. In addition to establishing a baseline accuracy,

comparisons between the methods of data collection were done. The results shown in Figure 2 show the prediction accuracy of author identification in executable files that have not been obfuscated.

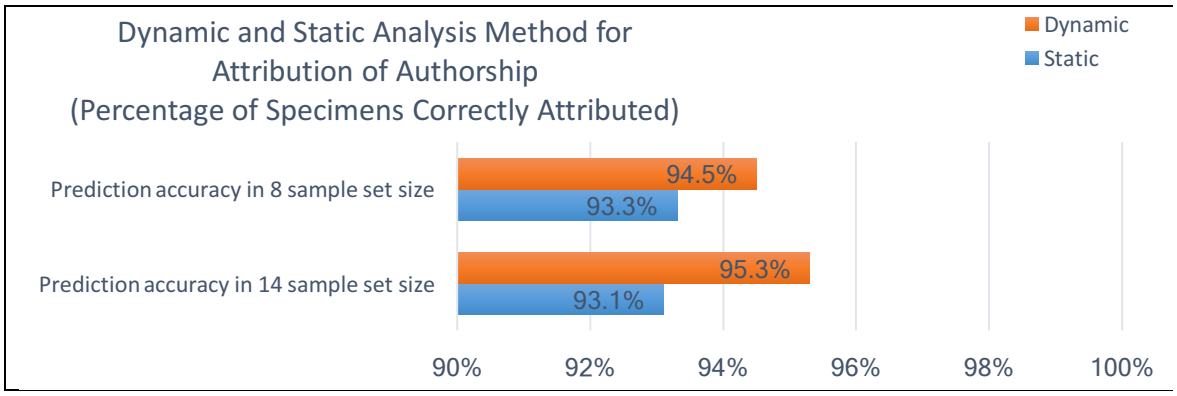

Figure 2 - Results of baseline accuracy of author attribution using dynamic and static methods

Experiment #2: Attribution of Packed Binary Executables

After defining the baseline prediction accuracy of author attribution using executables not packed at all, each of the samples was obfuscated using the obfuscators chosen from the set of available obfuscators from each complexity type. The results of each category are presented under their own sub-heading. The data collection statistics for each of the obfuscators under test can be found in Appendix A.

Obfuscation Complexity Type 1

Complexity type one obfuscators include ASPack (ASPack Software, 2017a), PESpin (cyberbob, 2017), and UPX (Markus F.X.J. Oberhumer, 2017a). The results of the authorship attribution experiments for these three examples of obfuscator complexity type one are shown in Figure 3. Note that files obfuscated using ASPack (ASPack Software, 2017a) could not be disassembled using the standardized procedure and tools described in the static analysis procedure, so prediction accuracies are not provided.

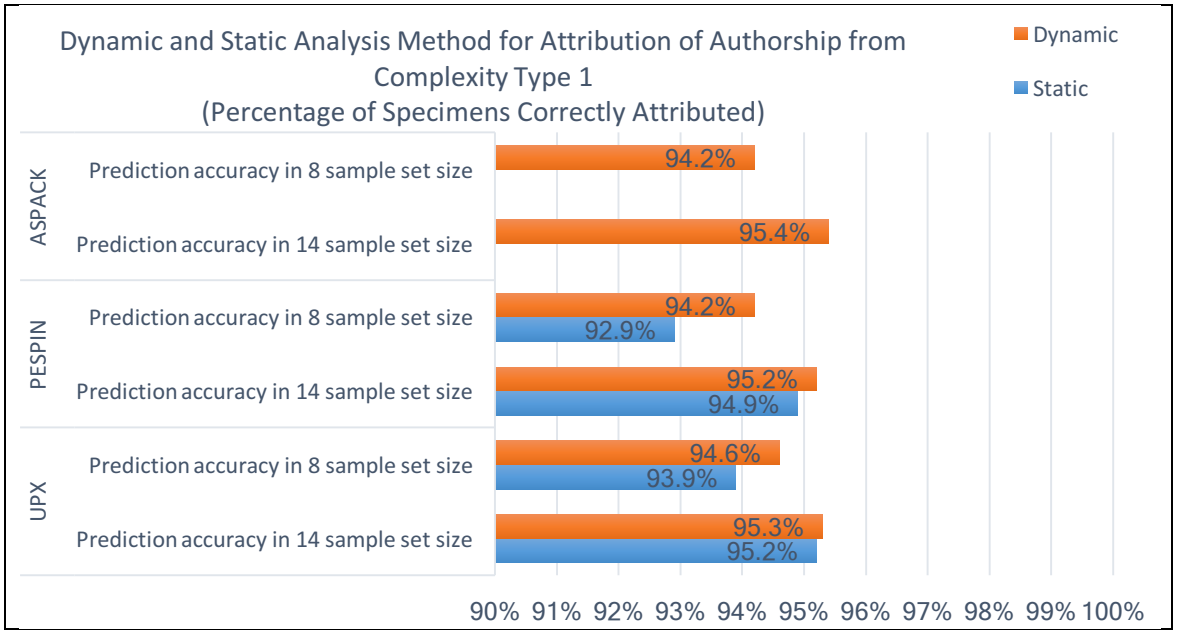

Figure 3 - Results of author attribution in specimens obfuscated using complexity type 1 obfuscators

Obfuscation Complexity Type 2

Yoda's crypter (Yoda, 2017), JDPack (JDPack, 2016), and DotFix's NiceProtect (DoxFix Software, 2017) are three examples of obfuscator tools that implement obfuscation in accordance with the definition of complexity type two. Each of these examples was used to obfuscate the test specimens for analysis using the static and dynamic methods described previously. The results of the authorship attribution experiments for these three examples of obfuscator complexity type two are shown in Figure 4. For the NiceProtect obfuscator (DoxFix Software, 2017), the obfuscated executables appeared to execute correctly, but the instruction traces were not captured by the DECAF dynamic collection platform (Henderson et al., 2014), so no results are reported. As documented in Table 8, many specimens obfuscated using Yoda's crypter (Yoda, 2017) aborted with a runtime error at time of execution, yielding no usable trace data. Other specimens obfuscated with Yoda's crypter (Yoda, 2017) did execute, but the size of the captured trace file, and the lack of expected output, suggest that a

problem between the DECAF platform (Henderson et al., 2014), this version of Yoda's crypter (Yoda, 2017), and version of Microsoft Windows (Microsoft, 2016a), exist.

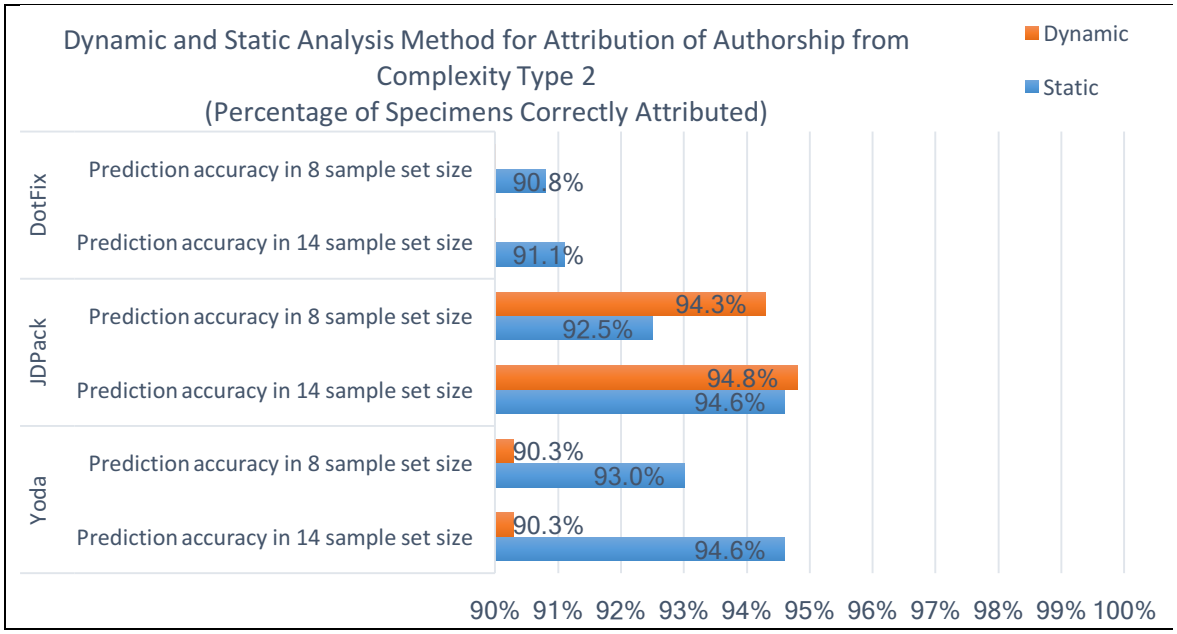

Figure 4 - Results of author attribution in specimens obfuscated using complexity type 2 obfuscators

Obfuscation Complexity Type 3

According to the statistics published by Ugarte-Pedrero et al. (2015), malware obfuscated by complexity type three obfuscators are the most commonly encountered in the wild. Of the numerous potential choices for instances of complexity type three obfuscation tools, ASProtect (ASPack Software, 2017b), PECompact (Collake, 2017), and UPolyX (unknown) were chosen. Author attribution prediction accuracy measurements for specimens obfuscated by these instances can be found in Figure 5. Like the complexity type two example ASPack (ASPack Software, 2017a), specimens obfuscated using ASProtect (ASPack Software, 2017b) could not be disassembled using the procedure and tools used in the static analysis case.

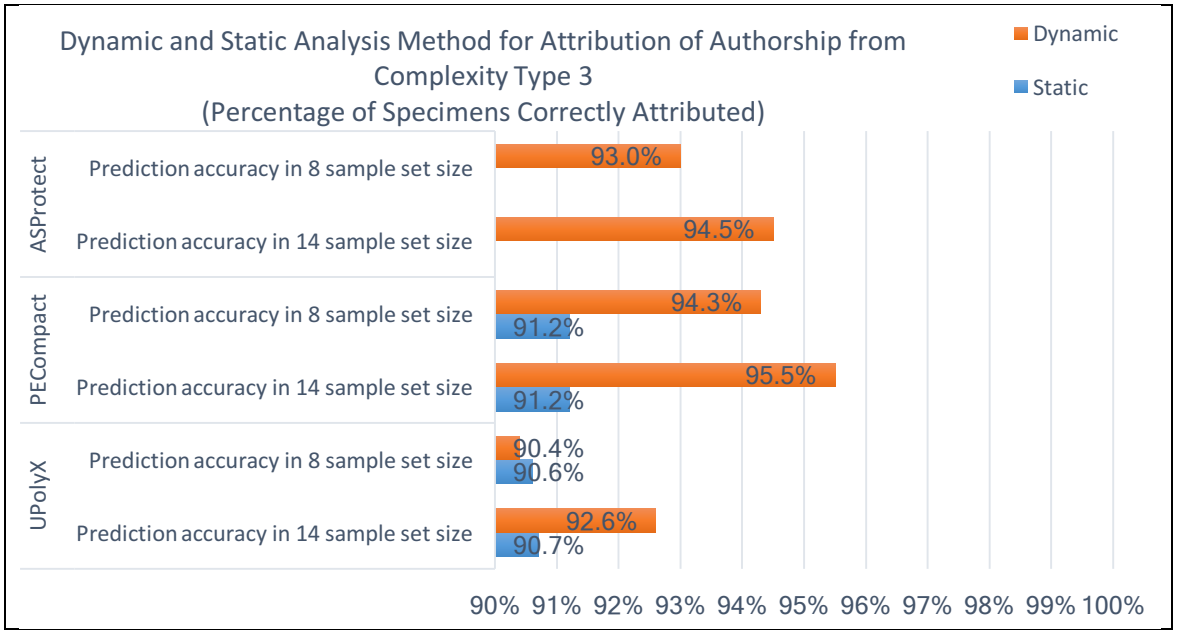

Figure 5 - Results of author attribution in specimens obfuscated using complexity type 3 obfuscators

Obfuscation Complexity Type 4

Obfuscators that implement complexity type four and five are rare, according to the statistics presented in (Ugarte-Pedrero et al., 2015). As such, only two instances for each of type four and type five obfuscators were located for use in this research. For type four, the two instances were ACProtect (Ultra-Protect, 2016) and Exestealth (Hanspeter Imp, 2017). The authorship attribution prediction accuracy measures are detailed in Figure 6. Even though the initial executable files appeared to properly transform into their obfuscated counterparts using the ACProtect (Ultra-Protect, 2016) tool and the obfuscated versions were successfully disassembled as part of the static analysis procedure, all the obfuscated specimens failed to execute within the DECAF (Henderson et al., 2014) environment, each returning a Microsoft Windows runtime error. As such, no prediction accuracies from the dynamic analysis are reported.

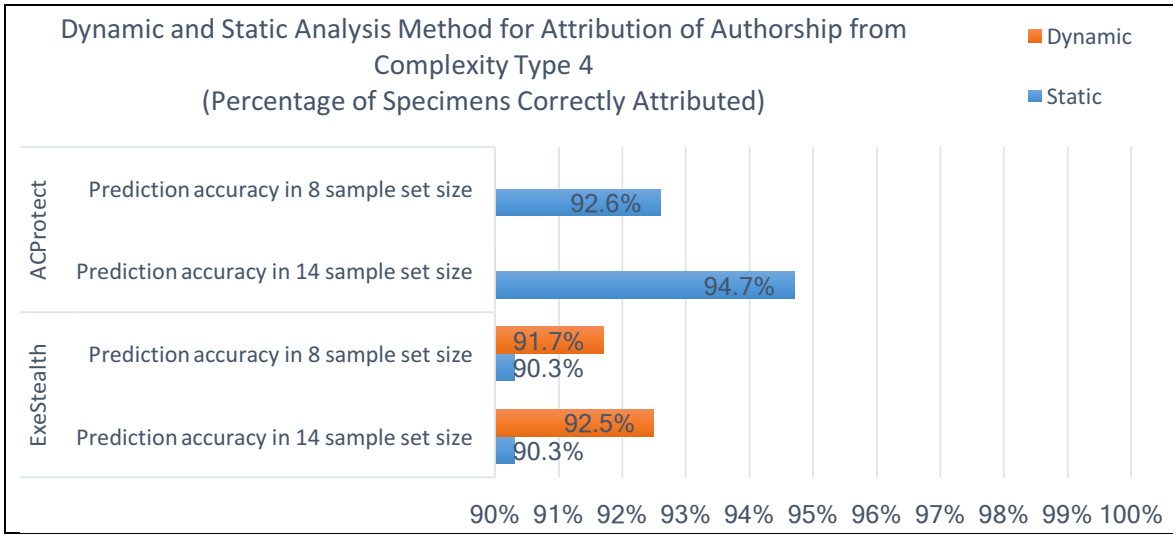

Figure 6 - Results of author attribution in specimens obfuscated using complexity type 4 obfuscators

Obfuscation Complexity Type 5

As evidenced by only 0.9 % of samples being obfuscated using type five complexity (Ugarte-Pedrero et al., 2015), only two such tools could be located, Beria (haggar, 2016) and EXPressor (CGSoftLabs). The author attribution prediction accuracy findings for each of these obfuscators can be found in Figure 7.

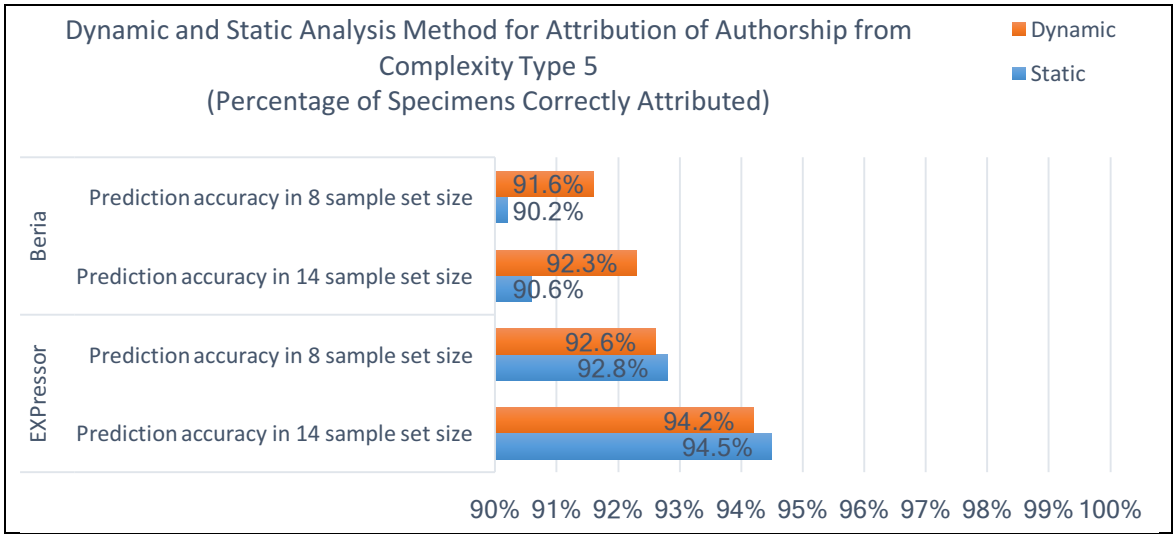

Figure 7 - Results of author attribution in specimens obfuscated using complexity type 5 obfuscators

Obfuscation Complexity Type 6

As previously described, executable files obfuscated with complexity type six, which include virtual machine obfuscation, are resistant to disassembly (Coogan et al., 2011; Fang et al., 2011; M. G. Kang, Yin, Hanna, McCamant, & Song, 2009; Rolles, 2009). As such, the three instances of obfuscators used in this research, Armadillo (Silicon Realms), Themida (Oreans Technologies, 2017), and VMProtect (VMProtect Software, 2017), all lack author attribution prediction measurements gathered from the static analysis method. The results from the dynamic analysis are shown in Figure 8.

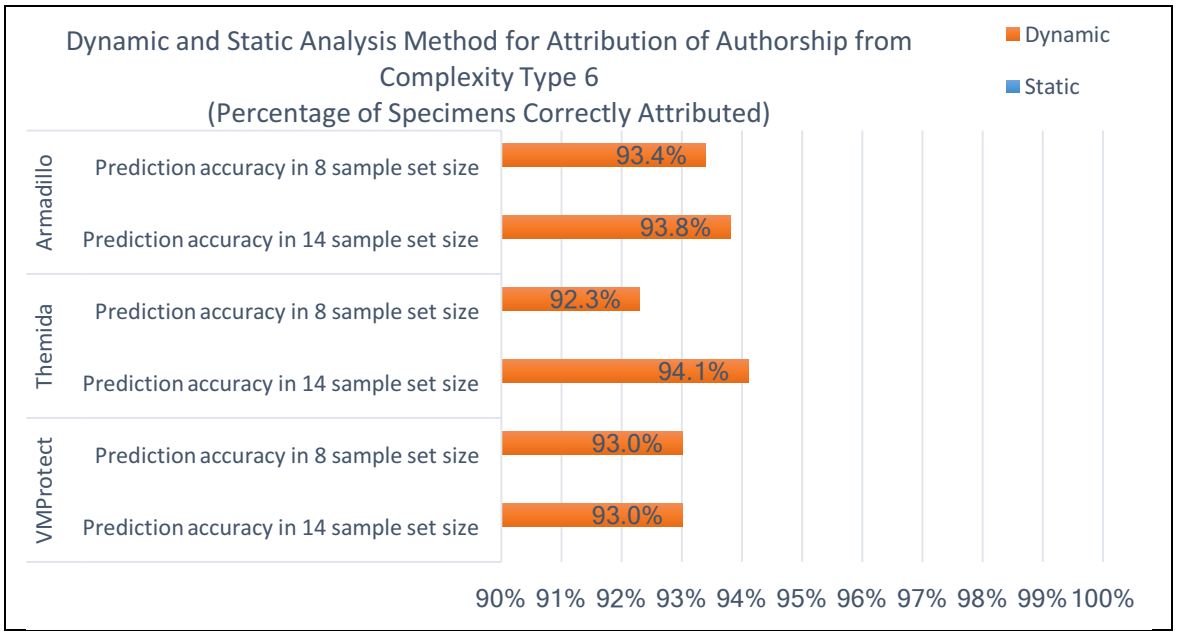

Figure 8 - Results of author attribution in specimens obfuscated using complexity type 6 obfuscators

Summary

The average of the author attribution prediction accuracies for each of the complexity groups was calculated for both the static and dynamic methods of analysis. The averages for each complexity type are given in Figure 9 for the 14 sample groups. Refer to Appendix B for results from the 8 sample groups.

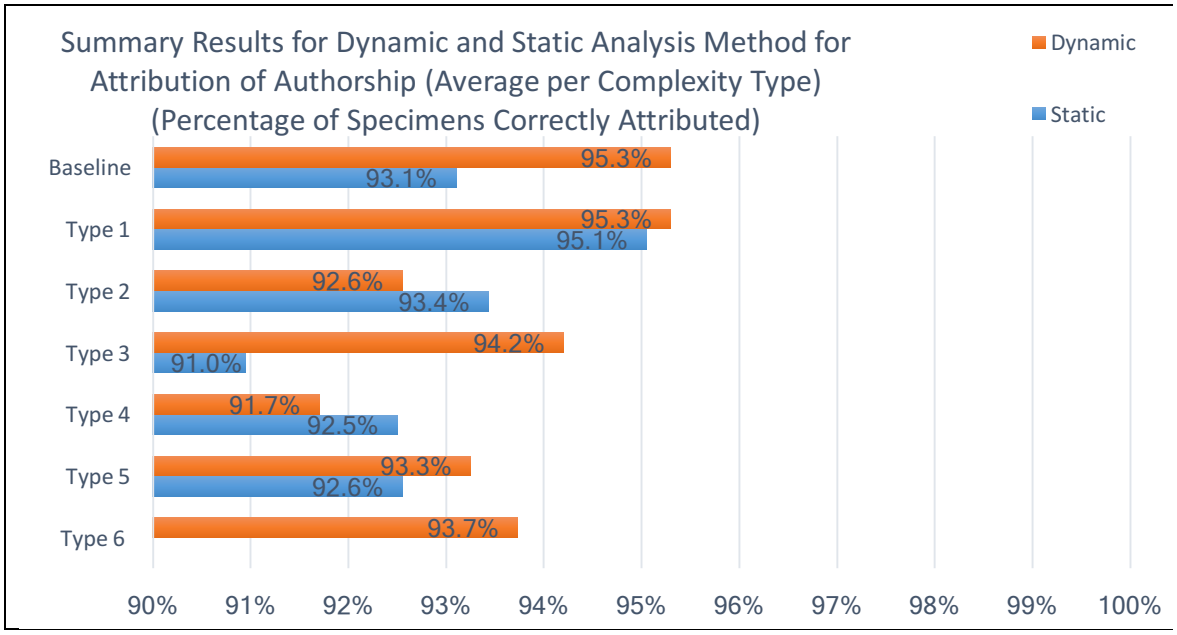

Figure 9 - Summary results of obfuscator attribution by complexity type (Set size of 14)

Experiment #3: Fingerprinting the Obfuscator

Experiment three was designed to identify features common in all samples that were obfuscated with the same obfuscator allowing for the identification of the obfuscation tool itself. Results shown in Figure 10 only describe obfuscator prediction accuracies from the 14 sample groups. For prediction accuracies measured using the eight sample groups, refer to Appendix C.

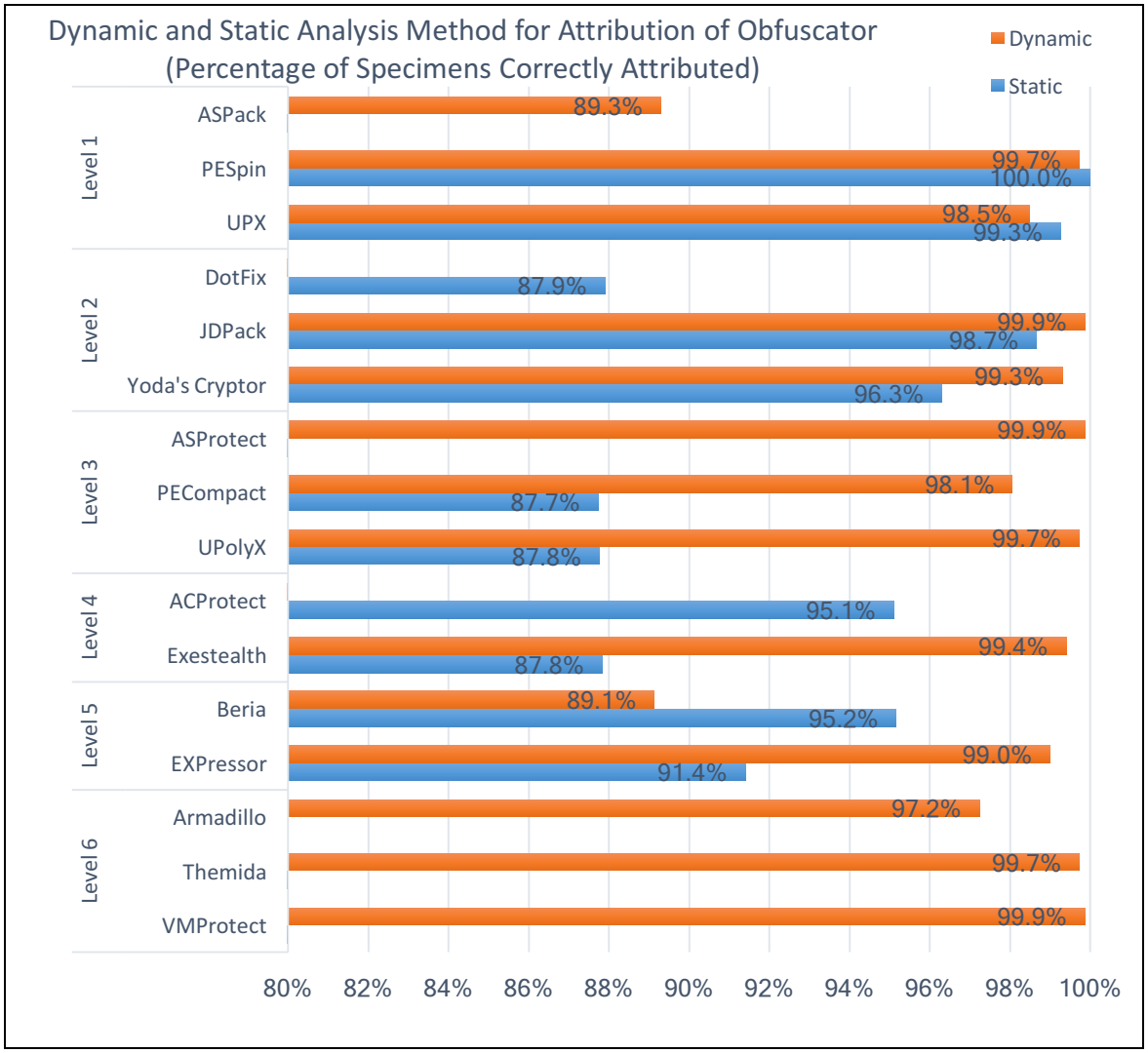

Figure 10 - Results of obfuscator attribution in randomly selected and obfuscated samples

Experiment #4: Stylometric consistency over generational versions of an obfuscator

This experiment was designed to test the stability over time of stylometric features in obfuscators themselves. A machine learning algorithm was trained to recognize differences introduced by the obfuscator itself. The results of this experiment are shown in Figure 11 and the data collection statistics for this experiment can be found in Appendix D.

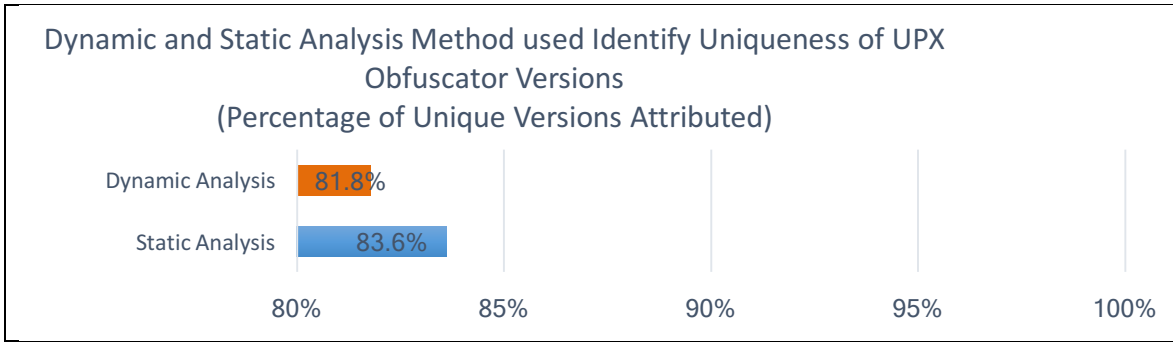

Figure 11 - Results of obfuscator version uniqueness identification using the UPX obfuscator

Experiment #5: Baseline attribution using different toolchains

For this experiment, the same specimens were compiled using both the Intel C Compiler and the GNU GCC compiler. The results of authorship attribution prediction accuracy using these different toolchains are illustrated in Figure 12. For comparison purposes, the earlier results from the Microsoft compiler baseline have been added.

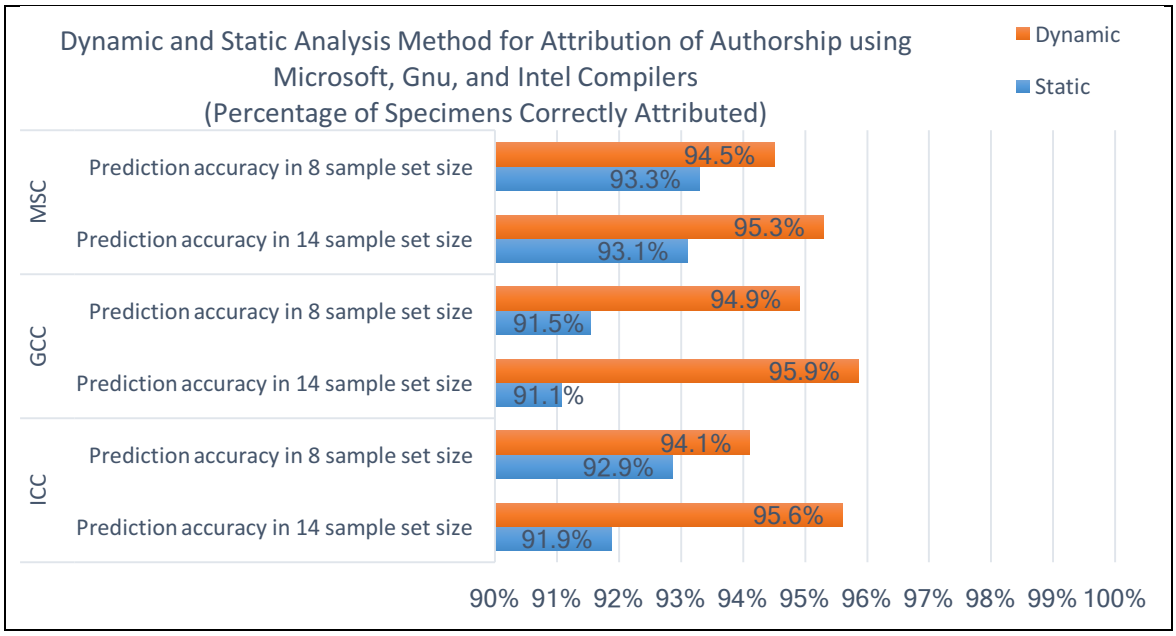

Figure 12 - Results of author attribution in specimens compiled using different compilers

Experiment #6: Attribution when using different optimization settings

In much the same way as obfuscators transform executable programs into observably equivalent, but different versions of themselves, the executable file output from a compiler configured for different optimization settings will be different, yet

observably equivalent. To quantify the effects of compiler optimizations options, two optimization configurations were created, one for fast execution and one for small file size.

Optimization for Speed

The results describing author prediction accuracies when all the samples in the author sets are compiled for speed is shown in Figure 13.

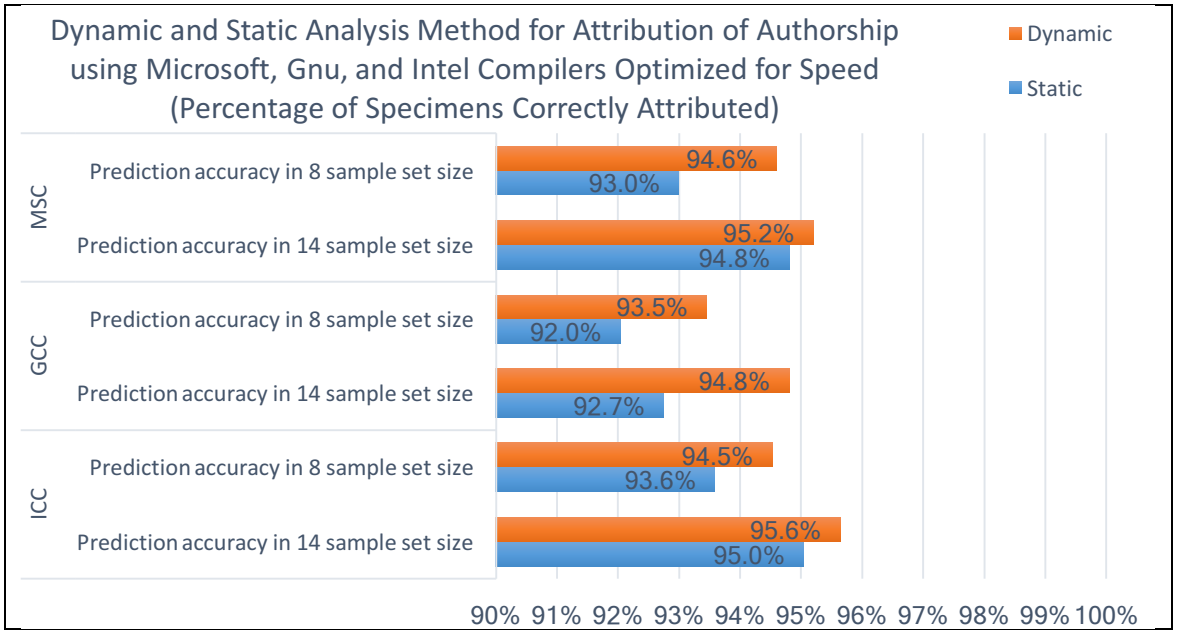

Figure 13 - Results of author attribution in specimens compiled using optimization for execution speed

Optimization for File Size

The results describing author prediction accuracies when all the samples in the author sets are compiled for file size is shown in Figure 14.

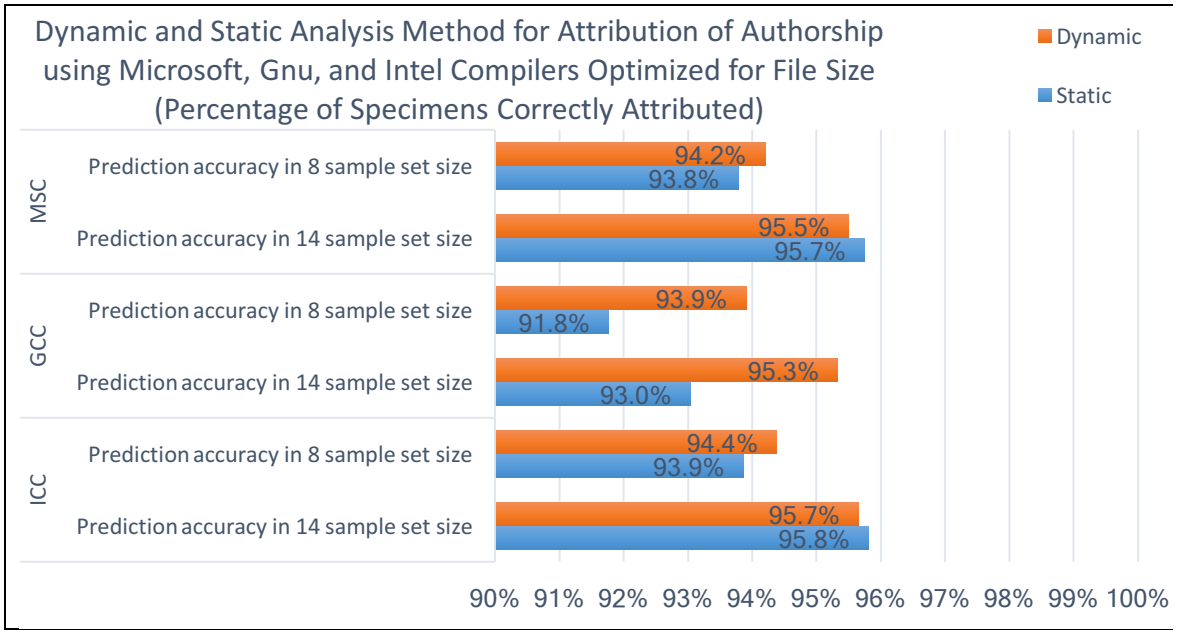

Figure 14 - Results of author attribution in specimens compiled using optimization for file size

Experiment #7: Attribution with random composition of all tool chains

Having measured the attribution accuracies of authorship in programs that were consistent across compiler selection and optimization settings, the final experiment sought to identify author attributing similarities in samples coming from like authors, but having been built using different compilers and/or different optimizations. Results of this experiment can be seen in Figure 15.

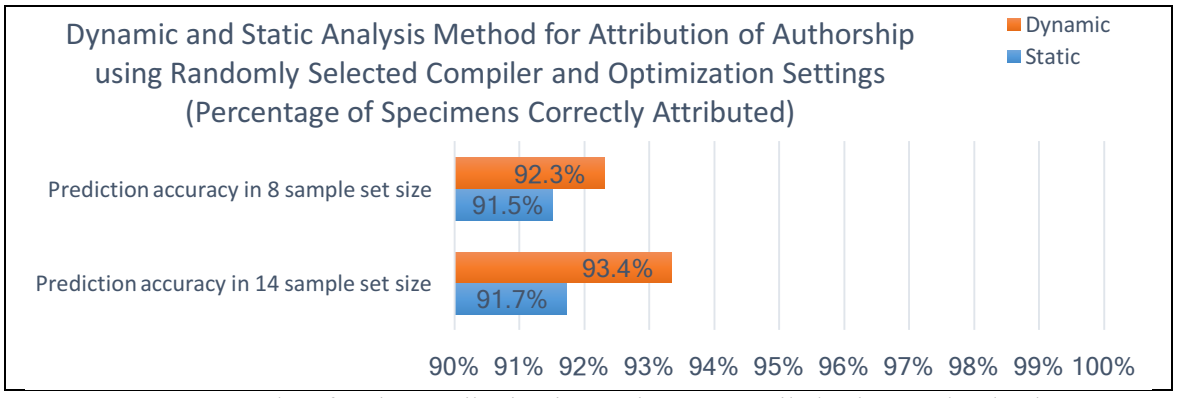

Figure 15 - Results of author attribution in specimens compiled using randomly chosen compiler / optimization settings

Findings

This section discusses the results presented in the previous section and provides an analysis and insight of the data. Observations collected during the execution of these experiments are also presented in this section.

The Data Set

The decision to use the Google Code Jam contest as a source of files for this research was made due to the previous research in executable file author attribution having used these files (Alrabaee et al., 2014; Caliskan-Islam, Yamaguchi, et al., 2015; Rosenblum, Zhu, et al., 2011). Similarly, sample and dataset sizes where kept constant and consistent with these previous research efforts to maximize the comparability of results. However, these past research efforts in author attribution in executable files included only static analysis methods, whereas this research included both static and dynamic methods.

The first challenge in using a dynamic method for collection of program trace data was that the specimens were required to compile and execute without error. A nontrivial number of published submissions to the Google Code Jam contest could not be compiled and/or executed within this research environment. To be included in one the experiments of this research, the not-obfuscated version of the specimen had to execute on the DECAF platform (Henderson et al., 2014) without error.

For each Code Jam problem, Google only describes the problem and supplies a single set of inputs with their expected output. As a result, there were large variations of input parameter and output handling. Some contributions accepted interactive user input while executing the specimen program, others wrote the input parameters to a file (with

hardcoded filenames) for reading during execution, while others accepted input and output files as command line parameters passed at runtime. The automation bridge, described in the previous chapter, was implemented to ensure that each of these cases was dealt with in a uniform and consistent fashion.

Data Collection

To accurately predict the authorship of executable files, author identifying feature data contained in the executable file must first be extracted from the executable file. As described in the Methodology section, this raw feature data extraction is done using two distinct methods, the first using static analysis techniques (the file being analyzed is not executed), and dynamic analysis techniques (the file being analyzed is executed).

Static Analysis Method

To extract feature data from the executable using the static analysis technique, the executable is read, analyzed, and then disassembled by IDA-Pro (Hex-Rays SA, 2015b). If the executable under analysis has been obfuscated, IDA-Pro will be unable to accurately analyze the file, which results in an aborted disassembly. Thus, an additional step of unpacking the executable was required. Previous research has demonstrated (and discussed previously in the document) that IDA-Pro can introduce inaccuracies during the disassembly of binary executable files. Similarly, the unpacking routine can also introduce translation inaccuracies in the unpacked version. These inaccuracies may have reduced the level of accuracy in attributing the authorship of some binary executable files.

Dynamic Analysis Method

The dynamic analysis method based the raw feature data collection on capturing CPU instructions as they are executed from within an instrumented platform, in this research, DECAF (Henderson et al., 2014). As with the static analysis method, author attribution prediction accuracies may have been affected by the following limitations. First, not all execution paths will be traversed when using dynamic analysis techniques (the discarded alternate path of any conditional statement is never executed. A second limitation lies in the potential for certain emulated instructions to be incorrectly implemented or instrumented which would have introduced inaccuracy in the disassembled instructions. (Chyłek, 2009; M. G. Kang et al., 2009).

The Obfuscators

The researchers who defined the packer complexity types assisted this research by providing additional examples of obfuscator products for some of the less often observed complexity types (Ugarte-Pedrero et al., 2015). All but two of these obfuscators could be located on the Internet. Alternative obfuscators could have been used, but at the time of the experiments, no tool for ad hoc classification of an unknown obfuscator was available.

While there is significant coverage of obfuscation tools on the open Internet, mostly related to facilitating reverse engineering, or de-obfuscating, a malware sample, some of the obfuscation tools themselves could not be located. Similarly, in cases where more than three examples of a complexity type were identified, the most available examples, as defined by Google search result rankings, were chosen for use in this

research. No effort was made to join any malware development forums, regardless of whether the forum was open or closed to new members.

Based on the timestamps of Internet forum postings, it appeared as though several obfuscation tools have simply not been kept current and in working order for the newer operating systems and compilers. As a result, some specimen files that were successfully compiled and executed could not be packed and others, after having been reported as successfully packed would not execute. Still other specimens, after having been obfuscated without error, abnormally end their execution with either a generic Microsoft Windows run time error or an error of type R6002.

An R6002 error is known to be caused by an illegal setting of an access attribute on a page of memory during a program's execution (blackd0t, 2009a, 2009b). These errors have been linked to coding errors in programs that modify Portable Executable file sections as these obfuscation tools do (Castán, 2014; Rodriguez, 2015). These issues, experienced while undertaking this research on authorship attribution are consistent with previous research done on the software quality of packed programs (Castán, 2014; Rodriguez, 2015).

One of the distinctive features of obfuscators in general is that they add additional "do little" or "do nothing" code to programs to increase the difficulty of analysis (Collberg et al., 1997; Roundy & Miller, 2013). In addition, many obfuscators are loop intensive, which adds a considerable number of additional instructions to the program being executed (Collberg et al., 1997; Roundy & Miller, 2013). For some obfuscated specimens, across a wide range of obfuscation tools spanning all the complexity types, the additional execution overhead introduced with these extra

instructions caused issues in the collection and analysis phases of the research. The computer hosting the DECAF environment had approximately 140 gigabytes (GB) of free disk space to store program traces. During the execution of some of the obfuscated specimens, all the available disk space was consumed causing the tracing to abort due to insufficient disk space. As a result, the trace collection of these files had a maximum file size of 125GB. The traces that were capped to this maximum size are marked as "partial" in the tables containing the data collection and analysis details (Appendix A).

Finally, some of the obfuscation tools used in this research were evaluation versions of the commercially available tool. All the evaluation versions of obfuscation tools had a runtime pop-up window identifying the specimen as having been obfuscated with an evaluation version. A review of the documentation provided with the evaluation version of the obfuscation tools suggested that only advanced features like time-locked execution, node-locked execution, licensing, and advanced features like anti-debug and anti-virtual machine execution were unavailable in the evaluation versions. Beyond this documentation review, no exhaustive investigation of differences in the protection strategies offered between evaluation and full versions of the obfuscation tools was done.

Experiment #1: The Baseline Study

The accuracy measures of predicted author in this research are lower than accuracy measures reported in the most recently published research in this area of study (Caliskan-Islam, Yamaguchi, et al., 2015). This is to be expected, as the number of feature types is significantly less in this research than that previous research. As described in the previous chapter, the feature types selected for this research were

chosen from the full set of feature types identified in previous research (Alrabaee et al., 2014; Caliskan-Islam, Yamaguchi, et al., 2015; Rosenblum, Zhu, et al., 2011). Each of these candidate feature types was assessed for its ability to survive the compilation and obfuscation steps.

Even with the reduced accuracy measures presented in the previous section, a baseline of the "best-case" output of author attribution prediction using these features was established. As discussed in the Methodology section, the use of random forest classifiers is encouraged when a great number of features, each with weaker predictive strength, is known to be the general composition of the data set. The trace data, collected using the DECAF (Henderson et al., 2014) analysis platform, conforms to this general composition. The program trace output in the static analysis process, however, does not. Within the static analysis process, the output of the disassembly step, which used IDA-Pro (Hex-Rays SA, 2015b), is more like computer source code in that loops are not unrolled and recurring functions are not duplicated. To maintain the consistency of results, a random forest classifier was used in the static analysis process as well, but the accuracy results would have been higher if another classifier was used (additional prediction measurements were made using a Naïve Bayes classifier, and those measurements have been included in the data collection detail tables found in Appendix A).

Experiment #2: Attribution of Packed Binary Executables

The findings for each of the complexity types are discussed together, as the results are generally consistent for all the tested obfuscation tools of the given type.

Following the per type discussions, a Summary of the findings overall is given, which includes a discussion of average prediction accuracies for each complexity type.

Obfuscation Complexity Type 1

Obfuscators having complexity type one have a single unpacking routine that is fully executed before transferring execution control to the now unpacked program that resides in memory (Ugarte-Pedrero et al., 2015). Consistent this this definition, the specimens used in this research, having been obfuscated with ASPack (ASPack Software, 2017a), PESpin (cyberbob, 2017), and UPX (Markus F.X.J. Oberhumer, 2017a), provide attribution prediction accuracies that are, on average, the same as the baseline measures observed in experiment one. These prediction accuracies are to be expected, given the nature of obfuscation for complexity type one packers. The single unpacking routine is executed at the start of the program run, deobfuscating and transferring the entire image to memory prior to the transfer back to the recreated original application. The instructions involved in unpacking will occur in all the samples, effectively being deemphasized by the classifier during training. Similarly, once the deobfuscation is done, all the remaining instructions are application code, and the differences in author style are identified and classified in the same manner as the original baseline samples. However, for specimens obfuscated using ASPack (ASPack Software, 2017a), the GUnpacker (Quick Unpack, 2017) tool was unable to create an image dump, so the static analysis process could not be completed.

Obfuscation Complexity Level 2

Whereas obfuscation tools of complexity type one have only a single packing routine, obfuscators of complexity type two have multiple packing routines organized in

a chain. The output of the last packing routine in the chain transfers execution control to the now de-obfuscated image in memory (Ugarte-Pedrero et al., 2015). One of the obfuscators used in this research, JDPack (JDPack, 2016), produced results in both static and dynamic analysis consistent with the expectation that the transfer point between the obfuscator code and original application could be identified. The measured results are 2.7 percent lower than the baseline for the dynamic result and just 0.3 percent higher than the baseline for the static case. These results suggest some links in the obfuscation chain modify the original code more fundamentally than just compression or encryption.

The NiceProtect (DoxFix Software, 2017) obfuscated specimens did undergo static analysis successfully, but return a lower attribution prediction accuracy. These reductions in authorship attribution prediction accuracy in samples that have undergone obfuscation provides a hint of support for the hypothesis provided in the Methodology chapter: a fully effective obfuscator should fully remove all stylometric markings such that a specific author cannot be predicted with accuracy; clearly, these examples of complexity type two obfuscation are not fully effective obfuscators. Unfortunately, the specimens packed with NiceProtect (DoxFix Software, 2017) could not be successfully analyzed using the procedure defined in this research and described previously. The specimens execute correctly through to their natural completion, complete with expected output, but the executable trace plugin from DECAF (Henderson et al., 2014) was not triggered as the specimen started. It is not known if this is an undocumented protection feature of NiceProtect (DoxFix Software, 2017) or a limitation of the DECAF platform (Henderson et al., 2014).

Consistent with the results measured from JDPack (JDPack, 2016), the prediction accuracy measures using the static analysis process from specimens obfuscated with Yoda's crypter (Yoda, 2017) are close, but lower, than the baseline. The results derived from the dynamic analysis of specimens obfuscated using Yoda's crypter (Yoda, 2017) are some of the lowest measured across all of the experiments. While this result could be interpreted as strongly supporting the research hypothesis, it is expected that the high percentage of specimens not analyzed due to errors (34.0 percent) or not fully analyzed (40.0 percent) due to trace capture storage limitations are to blame. The specific percentages of excluded samples are listed in Table 8 and an explanation of the types of failures was provided earlier in this chapter.

Obfuscation Complexity Type 3

Rather than having the multiple packing routines organized in a single chain, obfuscators of complexity type three organize the multiple packing routines in a more complex topology which includes loops (Ugarte-Pedrero et al., 2015). While this complex topology can result in having the original executable code in layers that are above the last layer of packing, there is still an observable transition between the set of packing routines and the application code (Ugarte-Pedrero et al., 2015).

Specimens obfuscated with ASProtect (ASPack Software, 2017b) could not be successfully dumped using the GUnpacker (Quick Unpack, 2017) tool, so attribution predictions derived from the static analysis process are not provided. The attribution prediction accuracies for ASProtect (ASPack Software, 2017b) and PECompact (Collake, 2017) when derived from the dynamic process are just 0.9 percent lower (average) than the established baseline.

The attribution accuracies for specimens obfuscated using UPolyX (unknown) have the same basic form as the others in this complexity class, with a markedly lower prediction accuracy when using the static method than when using the dynamic method. Overall, though, the measures were significantly lower than the results when using the other two obfuscators of this complexity type. Inasmuch as this reduction in attribution prediction accuracy supports the research hypothesis, these reductions can more likely be attributed to the significant number of samples that either did not pack or did not execute to their natural completion due to either a generic runtime error or the R6002 error discussed earlier. A complete accounting of all the data collection details can be found in Appendix A, but for UPolyX (unknown, 2016), only 26% of the samples were successfully executed. For the training set size of eight, this averages to just two samples per author, and for the training set size of 14, this averages to 3.6 samples per author. As discussed previously, and experimentally demonstrated in Caliskan-Islam, Yamaguchi, et al. (2015), prediction accuracies only approach the upper plateau of high accuracy predictions after samples set size of four.

Obfuscation Complexity type 4

Obfuscators developed within the definition of complexity type four can have either a single or multiple number of packing layers (Ugarte-Pedrero et al., 2015). The packing routines are interleaved into the main application and are executed immediately before the segment of memory that was obfuscated is required by the application during its execution. There is a moment during execution that the entire unpacked executable resides in memory (Ugarte-Pedrero et al., 2015). Examples of obfuscators of

complexity type four include ACProtect (Ultra-Protect, 2016) and Exestealth (Hanspeter Imp, 2017).

ACProtect (Ultra-Protect, 2016) created obfuscated specimens that appeared to be properly assembled portable executable files and GUnpacker (Quick Unpack, 2017) was able to parse these files, providing a deobfuscated dump of the file. The attribution prediction accuracy using the static method was very high, higher even than the baseline measure. These results suggest that some portion of the program transformation by the ACProtect (Ultra-Protect, 2016) obfuscator strengthens some (or all) of the features used by the machine learning algorithm to differentiate between samples having different authors. However, every one of the specimen executables would not execute within the DECAF (Henderson et al., 2014) environment, thus providing no trace data for use in authorship attribution testing.

Exestealth (Hanspeter Imp, 2017) returned a 2.8 percent reduction in author prediction accuracies when using the static analysis method. Similarly, the prediction accuracy results from the dynamic method showed a decrease of 3.6 percent from the baseline. Both findings are consistent with the research hypothesis, and suggest that this obfuscator is somewhat effective at removing stylometric features introduced by the author of the original program.

Obfuscation Complexity Type 5

Complexity type five obfuscation tools interleave the packing routines with the application code, only de-obfuscating the frame of the application as it is about to be executed. The only time the entire application is available in memory in a deobfuscated state is immediately prior to program completion (Ugarte-Pedrero et al., 2015).

Of the two complexity type five obfuscators tested, the Beria (haggar, 2016) obfuscator behaved in a manner consistent with the research hypothesis: the author prediction accuracies were reduced by 3.0 percent in the dynamic case. Similarly, the accuracy prediction results from the static analysis method were 2.5 percent lower than the baseline, which is noteworthy for both the obfuscators for this complexity type and the previous complexity type in that the results are consistent with previous research that had discussed the difficulties encountered when attempting to analyze files obfuscated within these higher complexity types (Bat-Erdene, Kim, Park, & Lee, 2017; Devi & Nandi, 2012; Kruegel et al., 2004; Lakhotia et al., 2013; B. Lee et al., 2010; O'Kane et al., 2011; Roundy & Miller, 2013; Udupa et al., 2005). Even at this level of complexity, both the dynamic analysis and static analysis method could still identify a subset of the stylometric markers that would have been found in the original, not obfuscated versions of the programs.

The other complexity type five example, EXPressor (CGSoftLabs), had static analysis prediction results that exceeded the dynamic analysis results. As previously discussed, this obfuscation tool appears to transform the original in such a way as to inadvertently amplify some of the stylometric markers the machine learning algorithm used to prediction authorship.

Obfuscation Complexity Type 6

Finally, there are obfuscators of complexity type six. These obfuscators pack the application code in such a way that only the smallest of frames, even down to a single instruction, are unpacked at any given time (Ugarte-Pedrero et al., 2015). The family of virtual machine obfuscators is included in the complexity type six category (Coogan et

al., 2011; Fang et al., 2011; M. G. Kang et al., 2009; Rolles, 2009; Ugarte-Pedrero et al., 2015).

As expected based on past research on program analysis of virtual machine obfuscation protected binaries, each of the three examples of complexity type six obfuscators tested, Armadillo (Silicon Realms), Themida (Oreans Technologies, 2017), and VMProtect (VMProtect Software, 2017), could not be analyzed using the static method (Coogan et al., 2011; Fang et al., 2011; M. G. Kang et al., 2009; Rolles, 2009). Collection of program traces using DECAF (Henderson et al., 2014) was successful for these three examples, however, on average, 34.0 percent of specimens aborted execution with errors and 17.0 percent of specimens could only have partial collection of trace data due to trace storage limitations. Even with these relative high percentages of missing and incomplete data, the average attribution accuracy reduction was only 1.6 percent, suggesting that even with this highest level complexity, the machine learning algorithm was able to separate portions of code contributed from the obfuscator and portions of the original application.

In the case of Themida (Oreans Technologies, 2017), a maximum runtime restriction is also imposed on programs obfuscated using the trial version of the software. After 20 minutes of execution, the specimen program is aborted. Specimens having had their execution time limited due to this restriction have been marked as having only partial execution analysis.

In the case of Armadillo (Silicon Realms), dynamic analysis prediction measures were close to those of the baseline measures, suggesting that some author attributing stylometric features have survived through compilation and obfuscation. The same can

be said for executables obfuscated using Themida (Oreans Technologies, 2017), having prediction results consistent with baseline measures. Finally, for VMProtect (VMProtect Software, 2017), the prediction results come in at about half way between the best and worst case measurements. These measures suggest that some, but not all, the stylometric features have been removed from the original code by the obfuscator. However, the lack of differentiation between the experimental results using smaller and larger sample size might suggest that the number of features selected for this research, and used consistently in all the experiments, is insufficient to provide a model that does improve prediction accuracy with additional samples from which to learn (Abu-Mostafa et al., 2012; Hastie et al., 2009).

Summary

In Figures 9 and 16 the average prediction accuracies per complexity type are displayed. While there are slight reductions in author prediction accuracies as the complexity type increases, the negative correlation between obfuscator complexity type and prediction accuracy for authorship attribution is not as strong as described in the experimental hypothesis. The results of this experiment support the original description offered by Collberg et al. (1997), where it is stated that an obfuscated program "consists of two programs merged into one: a real program which performs a useful task and a bogus program which computes useless information" (Collberg et al., 1997, p. 23), and the machine learning classifier has been given enough features from enough samples to correctly distinguish between the "real program" and the "bogus program".

Experiment #3: Fingerprinting the Obfuscator

This experiment was designed to fingerprint, or identify, the obfuscation tool that was used to obfuscate the original program by identifying features common among all samples obfuscated by a tool. The results, indicated by the measurements taken in this experiment, generally support the research hypothesis, that is, that obfuscation routines intermingled into the original application would have their own stylometric markers which could be used in attribution testing to identify the obfuscator itself.

As documented in Figure 10, when using the dynamic method of analysis, enough features were identified by the classifier from each of the obfuscators such that obfuscated samples could be classified with over 97.7 percent accuracy based on stylometric markers contributed by the obfuscator. Generally, the results from the static method of test were similarly supportive of the research hypothesis, but as expected, the prediction accuracies are lower at 92.7 percent.

Experiment #4: Stylometric consistency over generational versions of an obfuscator

This experiment was designed to test the stability of stylometric features in an obfuscator over time. It was hypothesized that different obfuscator tools are developed over time by the same team of developers, and that different versions of the obfuscator should have very few individual, unique stylometric features. To that end, a set of specimens, all written by the same author, were obfuscated using each of the available versions of the obfuscator and tested for stylometric similarity. All specimens in this experiment were written by the same author to minimize the effect of any stylometric features from the author. The machine learning algorithm was then trained to recognize differences introduced by the obfuscator itself. As shown in Figure 11, these results

show a prediction accuracy of just over 80 percent, which suggests that there are fewer unique stylometric identifiers in each version of UPX (compared to the greater than 97 percent accuracy achieved in predicting an obfuscator), thus lending some support to the hypothesis that these generational versions have consistent stylometric features which supports the hypothesis that the same author(s) wrote all the versions of UPX that were assessed in this research.

As previously discussed, a determination of whether a single author or a stable group of authors wrote all the generational versions cannot be made, but attribution problems such as this are presently being researched (Dauber, Caliskan-Islam, Harang, & Greenstadt, 2017; Meng, 2016; Meng, Miller, & Jun, 2016).

A threat to the validity of this experiment lies in the release schedule of some of the older UPX versions used in this experiment. Based on revision history posted at (Markus F.X.J. Oberhumer, 2017b), many of the release versions were primarily due to bug-fixes. It is expected that some of the issues encountered while trying to capture trace data using the dynamic method were caused by bugs present in the UPX version and presenting themselves while executing within the DECAF (Henderson et al., 2014) environment. In this experiment, as with all the experiments included in this research, specimens unable to provide data (in both the static and dynamic analysis cases), were excluded from the dataset. These exclusions may have affected both the in-sample and out-of-sample error rates of the predictions of stylometric stability over time (Abu-Mostafa et al., 2012).

Experiment #5: Baseline attribution using different toolchains

Previous research has searched for and studied artefacts found in applications left by various toolchains as a method for determining which tools were used to create the application (Chaki et al., 2011; H. Chen, 2013; Rahimian et al., 2015; Rosenblum et al., 2010; Rosenblum, Miller, et al., 2011), but no previous research could be located to causally connect the toolchain to the prediction strength of author attribution. This research posed the possibility of the toolchain affecting the accuracy of author attribution as a Research Question, and this experiment began a set of experiments designed to identify any causal relationships between author attribution and toolchain selection. The results from this experiment, shown in Figure 12, show that author attribution prediction accuracy is similar for each of the three compilers chosen for this research. These results are consistent with the research hypothesis as well as what one would generally expect, as the program itself should always behave in a consistent fashion, regardless of the tools used to transform the source code into the corresponding machine code.

Experiment #6: Attribution using different optimization settings

Earlier in this document, an analogy was put forward that compiler optimizations would act in a manner like an obfuscator, taking in a source program and creating a substantively different variant from the original. However, after having compiled every sample program using these different optimization settings, the results of author prediction testing are not consistent with the expected result. The author prediction rates in both optimization cases examined here are nearly identical to the baseline measures for each compiler.

Experiment #7: Attribution with random composition of all tool chains

Experiment seven was designed to mimic a real-world attribution scenario where an unknown sample, compiled with an unknown compiler using unknown optimization settings must be attributed for authorship. To that end, the eight and 14 sample experiments done in experiments five and six were redone but instead of the sample set being made up of specimens compiled with the same version of compiler and optimization settings, each sample in the set was randomly chosen from the set containing all nine (i.e. three compiler types, with three configurations for each) of the different versions of that sample. It was hypothesized that the machine learning classifier would be less able to accurately predict the author of a file with an unknown author and unknown toolchain provenance. The measured results are lower than those measured in the case of uniform compiler and optimization settings, but not by much. For the dynamic method of attribution, a reduction of 2.2 percent was measured for both the eight and 14 sample groups. The results from the static method show only a reduction of only 0.3 percent for the 14 sample group and a 1.0 percent reduction for the eight sample group. These findings suggest that few of the stylometric indicators stored in code are affected by compiler choice or optimization level.

Summary

This research had the goal of answering several research questions involved in quantifying the effects of binary file obfuscation on the accuracy of author attribution predictions. The primary tasks of this research were to compile and obfuscate the specimens, devise a process for and build tools to identify and capture program feature data, and analyze these data to make predictions on the accuracy of authorship

attribution. Not only did this research capture and analyze data from obfuscated samples, it captured and analyzed data from specimens compiled using various compilers, each with differing options for output optimization.

The data consisted of 1863 samples pulled from the Google Code Jam (Google, 2015) code repository. The samples included in this research were only those that were written in C/C++ and could be successfully compiled and executed within the test platform.

There were three notable outcomes of the experiments designed to provide a baseline accuracy of authorship attribution. The first outcome validated that the selected features could, with reasonable accuracy, predict authorship in unknown samples. The second outcome validated that this novel method of capturing instruction trace data could be used to predict authorship in unknown samples. The third outcome, arguably the most important, demonstrated that the instruction traces captured using an instrumented virtual machine provided higher accuracy predictions than the predictions based on trace data extracted using the IDA-Pro disassembler (Hex-Rays SA, 2015b) and using the same primary component analysis and classification process. These baseline experiments set the "high-water" mark for the other experiments that used the same methodology on specimens that were obfuscated using a variety of different obfuscation tools.

The results of the attribution testing for samples obfuscated with the type one complexity measure generally had prediction accuracies very close to the baseline. In the case of ASPack (ASPack Software, 2017a), static analysis could not be completed, so no prediction results are offered.

Results of attribution testing for samples obfuscated with the type two obfuscators were not consistent with the hypothesized results. While some results continued to show strong prediction accuracies when using the dynamic method, the static method of author attribution showed no decrease in accuracy.

Even though the specimens obfuscated with NiceProtect (DoxFix Software, 2017) executed correctly, the DECAF (Henderson et al., 2014) environment could not trigger on program start, resulting in a lack of dynamic analysis results for this obfuscator . Similarly, a significant percentage of samples obfuscated with Yoda's crypter (Yoda, 2017) failed to execute correctly.

Samples obfuscated using complexity type three tools also failed to remove enough stylometric markers from the original code to adequately hide the identity of the author. For each of these complexity type three obfuscators, the results from dynamic testing were close to the baseline measure. However, the obfuscation tools selected as type three examples did reduce the author prediction accuracies in the static case to some of the lowest accuracies measured.

The results from one of the two examples of complexity type four obfuscator, Exestealth (Hanspeter Imp, 2017) were much the same as those described for complexity type three: the obfuscator had successfully reduced the number of stylometric features that could be identified from within the static method. The strength of stylometric markings were also reduced in the dynamic case, but not nearly as much as in the static case.

For the second obfuscator, ACProtect (Ultra-Protect, 2016), the results from the static method are not consistent with those measured with Exestealth (Ultra-Protect,

2016), and no conclusions can be drawn from the dynamic method due to the consistently invalid executable files produced by this version of the obfuscator.

The results from samples obfuscated using complexity type five tools are similarly polarized. The first example, Beria (haggar, 2016) offers stronger anonymization in the static analysis case and weaker anonymization in the dynamic case, much like what was measured using Exestealth (Hanspeter Imp, 2017). The results of both static and dynamic analysis show only slight attenuation of the stylometric markers being identified during the testing of the specimens packed with Expressor (CGSoftLabs, 2017).

Type six obfuscators performed as expected, providing no useable results from static analysis and a slightly lower prediction accuracy from the dynamic method. Additionally, there are some indications that the authorship predictions for this complexity type could benefit from additional features being provided to the classifier.

Based on the result presented above, the obfuscators selected in this research seemed to prioritize adding additional layers of redirection and "junk" code over rewriting the original instructions into functionally equivalent, but syntactically different, versions of themselves. As a result, the machine learning algorithm could separate the contributions from the obfuscator and the contributions from the original application. Having the contributions supplied from the obfuscator being ubiquitous across the sample population, the obfuscator effects could be deemphasized by the classifier allowing the original stylometric attributes to be selected for which resulted in high accuracy author predictions.

There were several systemic issues that potentially challenge the validity of these results. First, a considerable number of specimens, from all complexity types, would not execute in the DECAF (Henderson et al., 2014) environment. While previous research has shown that the software engineering quality of packed (or obfuscated) specimens is considerably lower than compiled samples, (Castán, 2014; Rodriguez, 2015), it is not known whether packer quality issues or unknown issues with DECAF (Henderson et al., 2014) are to blame. A second issue was the amount of disk space needed to hold a complete trace of some of the executables under test. Each of the Google Code Jam samples can be considered a simple program, the largest of which has no more than a thousand lines of source code, but once obfuscated and being traced by the DECAF (Henderson et al., 2014) tool, requires more than 140 gigabytes (GB) of disk space. As a result of limitations in the collection environment used in this research, some specimens could only have partial execution traces analyzed. Finally, there were cases where the tracing mechanism within DECAF (Henderson et al., 2014) was unable to trigger the trace plugin when a program was initiated, resulting in no trace data being collected for a specimen.

In the first of the two remaining experiments using the obfuscators, it was demonstrated that evolutionary versions of a packing routine are still separable, however, each version does provide fewer unique stylometric features for a machine learning algorithm to use in classification. In the second, it was shown that all of obfuscators in this research have identifiable stylometric features that can be measured, albeit with reduced accuracy in a few cases.

A second group of experiments were conducted to ascertain whether a program's toolchain affected the accuracy of author attribution prediction. A baseline experiment was done, resulting in the conclusion author prediction accuracies were stable and mostly the same across different brands of compiler. A second experiment showed that optimizations for execution speed and file size do not substantially change the author prediction accuracies.

Finally, a random mixture of specimens having come from differing toolchains and levels of optimization were tested for authorship attribution prediction accuracy. It was shown that predicting an author of an unknown program is only slightly harder in the case where the training data is not made up of programs compiled using the same compiler brand and the same optimization settings.

Chapter 5

Conclusions, Implications, Recommendations, and Summary

Conclusions

Attribution of authorship in computer code, both human- and machine-readable formats continues to rise in importance in parallel with the rise of both intellectual property disputes and computer crime. Past research has considered whether, and to what extent, stylometric features found in human-readable versions of code carry into their machine-readable versions, but no research could be found that attempted to qualify or quantify the effects of obfuscation of authorship attribution.

The goal of this research was to provide insight into the effect, if any, of obfuscation on the accuracy of authorship attribution of machine-readable computer programs. Once the challenge of creating obfuscated specimens for use in testing were overcome, a methodology and tools were created that would analyze and de-emphasize the effects of obfuscation such that author attribution could proceed. A methodology using DECAF (Henderson et al., 2014) for instruction collection was devised to avoid the common difficulties associated with using static disassembly methods on obfuscated files, described earlier in the Literature Review section. As a comparative baseline, these same obfuscated specimens were analyzed using typical methodology based on static analysis, one that relied on generic unpacking using GUnpacker (Quick Unpack, 2017) prior to static disassembly using IDA-Pro (Hex-Rays SA, 2015b).

The results of this research demonstrated that enough stylometric indicators survive from source code through compilation and obfuscation to make accurate predictions of authorship feasible. Instead of making considerable changes to the

original instructions, it appears that obfuscators mostly wrap the existing instructions in multiple layers of additional do-nothing instructions. These extra layers do confound human reverse engineers, and to a lesser extent, the leading unpacking and disassembly tools. However, these extra layers seem to be readily identified by the classifier and due to their prevalence across all obfuscated samples, are deemphasized to nothing in principal component analysis. Some rewriting of code is done, for instance, instruction obfuscations, which include changes in function calls from direct to indirect (Roundy & Miller, 2013), and are observed with less frequency or in only a percentage of the samples, thus negatively impacting attribution prediction accuracy.

Further, a method of collecting execution traces was devised which used an instrumented, software-emulated machine platform for data collection. This new method enabled accurate analysis of executables even when those executables were obfuscated with tools that implement complex obfuscation algorithms which have foiled some of the best static analysis methods described in past research.

Implications

Prior research has done a lot to demonstrate that an author of an executable file can be identified. However, those researchers avoided the additional complexity of executable file obfuscation. While there are many applications of author attribution in cases where the executable files are not obfuscated, a very large opportunity exists in the attribution of authorship in malware. Recent research has concluded that more than 80 percent of malware is obfuscated (Bat-Erdene, Park, Li, Lee, & Choi, 2016).

Following the trend established in previous research, submissions from the Google Code Jam (Google, 2015) contest were used (Alrabaee et al., 2014; Caliskan-

Islam, Yamaguchi, et al., 2015; Rosenblum, Zhu, et al., 2011). While there have been criticisms on the use of Google Code Jam (Google, 2015) submissions for author attribution research (Dauber et al., 2017), an alternate dataset does not readily exist. Research is currently being undertaken to address the applicability of Github (GitHub Inc., 2017) projects as a reasonable alternative to the Google Code Jam, but problems of ground truth attribution exist within Github (GitHub Inc., 2017) as well (Dauber et al., 2017).

Even though all the submissions to this contest were validated by the Google Code Jam (Google, 2015) online submission portal at time of submission, some challenges with respect to building and executing a percentage of these submissions were encountered. Further challenges were encountered when attempting to obfuscate these submissions with a variety of obfuscation tools, some of which were first introduced over a decade ago. These challenges, when compared to the challenges described in the Literature Review section by other researchers doing similar research, may seem relatively insignificant based on the results that were produced using this approach based on instrumented execution within a virtualized environment.

With damages and payouts exceeding six hundred millions of dollars due to ransomware attacks in 2016 (Matthews, 2017), the need to accurately identify the author of a program, especially those that have been obfuscated, has never been higher.

Recommendations

Some of the future research opportunities that can be derived from these results include optimizing the collection and analysis of instruction traces, identification of additional features in obfuscated executables that can improve prediction accuracy,

classification methods that support the notion of deep learning, and modifications of tools and processes based on findings resulting from the study of common practice of plagiarism and code sharing.

As previously discussed, the steps taken to capture execution traces consumed significant amounts of disk space and time. The programs written to solve the problems presented in the Google Code Jam did not exceed a few thousand lines of code. A recent study found that malware has been growing in size, measured in lines of source code, from several thousand lines of code in the early 2000s, to tens of thousands of lines of code in recent samples, at an average rate of 16% per year growth (Calleja, Tapiador, & Caballero, 2016). Such large specimens would require a prohibitively large analysis platform and would take more time to analyze than could be reasonably allowed, given the number of new malware samples requiring analysis on a daily basis (Kaspersky Labs recent published their statistic of 323,000 new samples found per day (DarkReading, 2016)).

The features selected for use in attribution testing were chosen for their expected ability to survive obfuscation. Past research has always used more features than just those used in this research (Alrabaee et al., 2014; Caliskan-Islam, Yamaguchi, et al., 2015; Rosenblum, Zhu, et al., 2011). Each of the features used in previous research were assessed to determine whether the feature would survive through obfuscation. As discussed in the Methodology section, only a small subset of features was consistent through obfuscation, resulting in these other features to be excluded. However, improvements in prediction accuracy will likely come from the addition of other

features that capture additional stylometric features encoded within the obfuscated binary files.

Throughout this Dissertation Report, the idea of stylometric features as a tangible entity tempts the reader to request a listing of the features that were most influential in giving an accurate prediction of authorship. However, at present, it is not possible, in most cases, to extract the decision-making criteria from a supervised machine learning classifier (Knight, 2017). Research opportunities exist in creating a mechanism to extract and encode those stylometric features from the classifier which accurately and parsimoniously describe an author.

Drawing conclusions from the intersection of findings from the attribution experiments, the obfuscator fingerprinting experiment, and the toolchain impact experiments, an opportunity for future research may include the use of deep learning to improve the prediction accuracy of authorship, where separate processing layers within a larger classifier are trained to discriminate between different aspects of the data, in this case, the compiler and its level of optimization, the original executable, and the obfuscator (LeCun, Bengio, & Hinton, 2015; Schmidhuber, 2015). Application of this technology may improve both the author attribution accuracies described in this research and may also address the scenario where a single author uses different obfuscation tools for different software projects.

Even though Research Question Five remained unaddressed in this research, other researchers have started to explore the area of author attribution in the case of intermingled contributions within a single file, both in source code (Dauber et al., 2017)

and binary format (Meng, 2016; Meng et al., 2016). In both of these efforts, obfuscation has not been addressed, and will surely need to be at some future time.

Summary

In human readable files, authorship attribution relies on an analysis of stylometry, vocabulary diversity, word choice, and frequency of function words (Bozkurt et al., 2007; Stamatatos, 2009). Past research has also shown that similar analysis can be done on computer source code resulting accurate predictions of authorship (Burrows et al., 2014; Caliskan-Islam, Harang, et al., 2015; Đurić & Gašević, 2013; Frantzeskou et al., 2004; Frantzeskou et al., 2008; Frantzeskou et al., 2007; Ji, Park, Woo, & Cho, 2007; Kilgour et al., 1998; Meng et al., 2013; Ohmann & Rahal, 2014; Tennyson, 2013; Matthew F Tennyson & Francisco J Mitropoulos, 2014; Matthew F Tennyson & Frank J Mitropoulos, 2014). Even though the process of compilation removes many of these elements of author style (Rosenblum et al., 2010; Rosenblum, Miller, et al., 2011), research has also shown stylometric indicators can be located in executable files (Alrabaee et al., 2014; Caliskan-Islam, Yamaguchi, et al., 2015; Rosenblum, Zhu, et al., 2011). This research set out to extend the body of knowledge in this domain of author attribution in executable files by examining the impact that binary executable obfuscation had on the accuracy of author attribution.

The goal of binary executable obfuscation is to dramatically increase the difficulty of analysis, thus requiring additional effort to be spent in order to understand the capability and mechanisms of the obfuscated software (Collberg & Thomborson, 2002; Collberg et al., 1997; Roundy & Miller, 2013). Consequently, many analysis tools fail to provide an accurate representation of the original program after it has been

obfuscated (Acosta & Medina, 2012; Acosta et al., 2012; Alam et al., 2014; Alazab, Venkatraman, Watters, & Alazab, 2011; Egele et al., 2012; Gandotra, Bansal, & Sofat, 2014; Huang et al., 2011; Iwamoto & Wasaki, 2012; Ki, Kim, & Kim, 2015; Mohaisen, Alrawi, & Larson, 2013; Moser et al., 2007a; Pfeffer et al., 2012; Rieck et al., 2011; Rolles, 2009; Sharif, Lanzi, et al., 2008; Walenstein & Lakhotia, 2007; Xie et al., 2012; Zeng et al., 2013; Zhang & Reeves, 2007). For this reason, an approach of using dynamic analysis techniques to test for authorship was used. In much the same was as (Dinaburg et al., 2008; Henderson et al., 2014; Zeng et al., 2013) used virtual machine execution to analyze malware for its capabilities, DECAF (Henderson et al., 2014) was used in this research to collect executable instruction traces for the purpose of author attribution. This novel approach to author attribution has not yet received much attention in prior research (Alrabaee et al., 2016). The experiments using execution of programs in a virtualized environment to supply instruction traces were replicated using instruction traces collected using static methods to provide a baseline for comparison with both the methodology and the effects of program obfuscation on authorship attribution.

Some challenges existed in creating an environment useful for author attribution based on dynamic analysis techniques. Each of the individual challenges posed to participants of the Google Code Jam contest are described in written English and a sample of input parameters and expected output are provided. While guidance is given in the Quick-Start Guide (Google Inc., 2017), no further restrictions are applied, so participants are free to decide how the input will be supplied to the program and how the output would be presented. These differences in input and output specification became

another significant challenge, as each executable needed to be analyzed in a reliable and repeatable fashion. The number of samples, the whole corpus of submissions totaled to over 4400 files, combined with the number of obfuscation tools and different compiler settings, dictated the need for automation. With these requirements for instrumentation and automation defined, processes and tools were created for use in these experiments.

The feasibility of this methodology and tools for predicting authorship in files of unknown provenance was demonstrated by comparing its predictions to those measured using methods based on static disassembly of the specimens. This new method was then used to predict the author of executable files that were obfuscated using a variety of obfuscation tools organized into complexity types as defined in (Ugarte-Pedrero et al., 2015). Each of these attribution tests was redone using the methodology and tools based on static disassembly of specimens for comparison. Additional experiments were done to show the stability of programmer style over the evolution of a single obfuscation tool and how amenable to fingerprinting a specific obfuscator's author might be solely based on data captured from the execution of randomly chosen specimens. Finally, it was hypothesized that differences in executable files introduced by either differing compilers or compiler options would be factored out during the machine learning classification phase, thus rendering toolchain impacts to author attribution insignificant.

It was hypothesized in the Methodology chapter than an effective obfuscation tool would remove all traces of any stylometric feature that could have been used to attribute authorship. Given this hypothesis and the scale of increasing obfuscator complexity types (Ugarte-Pedrero et al., 2015), it was expected that as the complexity type increased, the number and quality of stylometric markers would decrease. The

measured results were not consistent with the expected outcomes; executables obfuscated with tools from the lower levels of packer complexity types had author prediction results almost equal to those measured in the baseline experiments and executables obfuscated with tools from the higher levels of packer complexity types had author prediction rates only one to three percent lower than the baseline measurements. It was, however, shown that author prediction accuracies resulting from the static analysis method were both unpredictable and lower than those resulting from the dynamic method. Generic unpacking tools can often deobfuscate executable files that have been transformed using tools in the lower complexity types, but are largely ineffective when used on those obfuscated with the higher level of complexity type obfuscators.

Support for the notion of programmer stability (J. H. Hayes & Offutt, 2010; Jane Huffman Hayes, 2009) was not sufficiently demonstrated in this research as the machine learning algorithms trained to provide predictions of the obfuscator version from a set of generational versions of the obfuscator succeeded with an accuracy of 80 percent. Further, this research did not test the full implementation of the obfuscator itself for stylometric stability, it only tested the specific obfuscation routines that were added to the samples.

The experiment designed to fingerprint the obfuscator found that, generally, the dynamic method of analyzing obfuscated executables allowed for the accurate identification of stylometric features within the obfuscator code allowing the obfuscator to be fingerprinted with high accuracy.

A threat to the validity of these results existed in that a significant number of obfuscated specimens, up to 70 percent in the worst case, and on average 23.5 percent, failed to execute completely, potentially allowing some features to be excluded in the classification steps. Some incomplete, partial results could be attributed to constraints imposed by the testing environment itself, the test environment only had about 140 gigabytes (GB) of available disk space in which to temporarily store execution traces. In some instances, over most of the complexity types, the amount of processing overhead added by the obfuscator pushed execution trace sizes from a few hundred megabytes (MB) to over a hundred gigabytes (GB). As a result, these execution traces were capped at 125 GB. For the cases where the obfuscated specimens would not execute, or abort due to runtime errors, it is not known whether software quality issues in the obfuscation tools themselves or whether inaccurate or incomplete implementation of the software emulated environment are responsible.

While the results of this research demonstrate that dynamic methods of analysis of executables can be used to predict authorship with good accuracy, it is still a resource intensive task. As just described, the disk space requirements for each execution of each file were significant. Not surprisingly, the amount of time required to execute each sample is also significant; many specimens took over four hours to execute within the virtual environment, even though the environment consisted of modern computer equipment with multiple processing cores and adequate amounts of memory. Future research opportunities may include improving the success rate of analyzed samples and the accuracy of author prediction.

Obfuscation tools remove many of the artifacts that have been used in previous research to predict authorship (Alrabaee et al., 2014 ; Caliskan-Islam, Yamaguchi, et al., 2015; Rosenblum, Zhu, et al., 2011), thus requiring the identification of additional structures or behaviors that can lead to higher accuracy prediction. The results of this research provide a solid first step and encouragement that authorship in obfuscated executable files can be attributed with good accuracy, it is expected future research will provide improvements to the foundations described herein.

Appendix A:

Experiment 2 Data Collection Details

			ASPack	PESpin	UPX
Sample Size (files)			1863	1863	1863
Not Packed (%)			0	0	1.27
Static Analysis	Not Dumped (%)		100	3.92	0.42
	8 sample set	Accuracy (%): Naïve Bayes	n/a	93.9	93.5
		Accuracy (%): Naïve Bayes, maximum 500 features	n/a	94.1	93.9
		Accuracy (%): Random Forest (500 trees), maximum 500 features	n/a	92.9	93.9
	14 sample set	Accuracy (%): Naïve Bayes	n/a	94.5	94.1
		Accuracy (%): Naïve Bayes, maximum 500 features	n/a	94.9	94.8
		Accuracy (%): Random Forest (500 trees), maximum 500 features	n/a	94.9	95.2
Dynamic Analysis	Runtime Errors (%)		0	0.42	0.42
	R6002 Errors (%)		0	0.85	0
	No Capture Errors (%)		0	0	0
	8 sample set	Partial samples (%)	0	3.75	0
		Accuracy (%): Naïve Bayes	92.8	94.0	92.8
		Accuracy (%): Naïve Bayes, maximum 500 features	93.3	93.8	93.2
		Accuracy (%): Random Forest (500 trees), maximum 500 features	94.2	94.2	94.6
	14 sample set	Partial samples (%)	0	3.51	0
		Accuracy (%): Naïve Bayes	93.0	94.6	92.7
		Accuracy (%): Naïve Bayes, maximum 500 features	94.8	94.7	94.5
		Accuracy (%): Random Forest (500 trees), maximum 500 features	95.4	95.2	95.3

Table 7 – Data collection details for complexity type 1 results

			NiceProtect DotFix	JDPack	Yoda's crypter
Sample Size (files)			1863	1863	1863
Not Packed (%)			0.14	0.14	11.55
Static Analysis	Not Dumped (%)		10.9	10.0	0.14
	8 sample set	Accuracy (%): Naïve Bayes	91.1	94.2	94.3
		Accuracy (%): Naïve Bayes, maximum 500 features	91.0	94.3	94.4
		Accuracy (%): Random Forest (500 trees), maximum 500 features	90.8	92.5	93.0
	14 sample set	Accuracy (%): Naïve Bayes	91.4	94.5	94.6
		Accuracy (%): Naïve Bayes, maximum 500 features	91.0	94.9	94.9
		Accuracy (%): Random Forest (500 trees), maximum 500 features	91.1	94.6	94.6
Dynamic Analysis	Runtime Errors (%)		0	3.8	21.07
	R6002 Errors (%)		0	7.04	1.43
	No Capture Errors (%)		100	0	0
	8 sample set	Partial samples (%)	0	3.13	38.75
		Accuracy (%): Naïve Bayes	n/a	92.5	90.3
		Accuracy (%): Naïve Bayes, maximum 500 features	n/a	92.9	90.3
		Accuracy (%): Random Forest (500 trees), maximum 500 features	n/a	94.3	90.3
	14 sample set	Partial samples (%)	0	2.14	40.0
		Accuracy (%): Naïve Bayes	n/a	92.7	90.3
		Accuracy (%): Naïve Bayes, maximum 500 features	n/a	94.2	90.3
		Accuracy (%): Random Forest (500 trees), maximum 500 features	n/a	94.8	90.3

Table 8 - Data collection details for complexity type 2 results

			ASProtect	PECompact	UPolyX
Sample Size (files)			1863	1863	1863
Not Packed (%)			12.13	0.14	3.21
Static Analysis	Not Dumped (%)		100	7.76	12.5
	8 sample set	Accuracy (%): Naïve Bayes	n/a	91.8	92.2
		Accuracy (%): Naïve Bayes, maximum 500 features	n/a	91.8	90.6
		Accuracy (%): Random Forest (500 trees), maximum 500 features	n/a	91.2	90.6
	14 sample set	Accuracy (%): Naïve Bayes	n/a	92.0	92.3
		Accuracy (%): Naïve Bayes, maximum 500 features	n/a	92.0	90.6
		Accuracy (%): Random Forest (500 trees), maximum 500 features	n/a	91.2	90.7
Dynamic Analysis	Runtime Errors (%)		10.71	0.14	70.71
	R6002 Errors (%)		0.28	0.14	0
	No Capture Errors (%)		0	0	0
	8 sample set	Partial samples (%)	15	0	13.13
		Accuracy (%): Naïve Bayes	92.8	92.8	90.8
		Accuracy (%): Naïve Bayes, maximum 500 features	92.6	93.0	90.3
		Accuracy (%): Random Forest (500 trees), maximum 500 features	93.0	94.3	90.4
	14 sample set	Partial samples (%)	14.64	0	13.93
		Accuracy (%): Naïve Bayes	92.9	93.4	92.4
		Accuracy (%): Naïve Bayes, maximum 500 features	93.7	94.6	91.9
		Accuracy (%): Random Forest (500 trees), maximum 500 features	94.5	95.5	92.6

Table 9 - Data collection details for complexity type 3 results

			ACProtect	Exestealth
Sample Size (files)			1863	1863
Not Packed (%)			8.17	0
Static Analysis	Not Dumped (%)		0	0
	8 sample set	Accuracy (%): Naïve Bayes	93.8	90.3
		Accuracy (%): Naïve Bayes, maximum 500 features	94.0	90.3
		Accuracy (%): Random Forest (500 trees), maximum 500 features	92.6	90.3
	14 sample set	Accuracy (%): Naïve Bayes	94.7	90.3
		Accuracy (%): Naïve Bayes, maximum 500 features	95.1	90.3
		Accuracy (%): Random Forest (500 trees), maximum 500 features	94.7	90.3
Dynamic Analysis	Runtime Errors (%)		91.83	40.36
	R6002 Errors (%)		0	0
	No Capture Errors (%)		0	0
	8 sample set	Partial samples (%)	0	0
		Accuracy (%): Naïve Bayes	n/a	93.3
		Accuracy (%): Naïve Bayes, maximum 500 features	n/a	93.1
		Accuracy (%): Random Forest (500 trees), maximum 500 features	n/a	91.7
	14 sample set	Partial samples (%)	0	0
		Accuracy (%): Naïve Bayes	n/a	92.6
		Accuracy (%): Naïve Bayes, maximum 500 features	n/a	93.1
		Accuracy (%): Random Forest (500 trees), maximum 500 features	n/a	92.5

Table 10 - Data collection details for complexity type 4 results

			Beria	Expressor
Sample Size (files)			1863	1863
Not Packed (%)			1.07	1.07
Static Analysis	Not Dumped (%)		11.79	10.36
	8 sample set	Accuracy (%): Naïve Bayes	90.9	93.8
		Accuracy (%): Naïve Bayes, maximum 500 features	91.0	94.1
		Accuracy (%): Random Forest (500 trees), maximum 500 features	90.2	92.8
	14 sample set	Accuracy (%): Naïve Bayes	91.0	94.3
		Accuracy (%): Naïve Bayes, maximum 500 features	91.4	94.7
		Accuracy (%): Random Forest (500 trees), maximum 500 features	90.6	94.5
Dynamic Analysis	Runtime Errors (%)		17.86	21.07
	R6002 Errors (%)		0.71	0
	No Capture Errors (%)		0	0
	8 sample set	Partial samples (%)	79.38	8.13
		Accuracy (%): Naïve Bayes	91.5	94.0
		Accuracy (%): Naïve Bayes, maximum 500 features	91.4	92.6
		Accuracy (%): Random Forest (500 trees), maximum 500 features	91.6	92.7
	14 sample set	Partial samples (%)	79.29	9.29
		Accuracy (%): Naïve Bayes	91.3	94.4
		Accuracy (%): Naïve Bayes, maximum 500 features	91.8	94.2
		Accuracy (%): Random Forest (500 trees), maximum 500 features	92.3	94.2

Table 11 - Data collection details for complexity type 5 results

			Armadillo	Themida	VMProtect
Sample Size (files)			1863	1863	1863
Not Packed (%)			0	0	0
Static Analysis	Not Dumped (%)		100	100	100
	8 sample set	Accuracy (%): Naïve Bayes	n/a	n/a	n/a
		Accuracy (%): Naïve Bayes, maximum 500 features	n/a	n/a	n/a
		Accuracy (%): Random Forest (500 trees), maximum 500 features	n/a	n/a	n/a
	14 sample set	Accuracy (%): Naïve Bayes	n/a	n/a	n/a
		Accuracy (%): Naïve Bayes, maximum 500 features	n/a	n/a	n/a
		Accuracy (%): Random Forest (500 trees), maximum 500 features	n/a	n/a	n/a
Dynamic Analysis	Runtime Errors (%)		14.64	43.93	43.57
	R6002 Errors (%)		0	0	0
	No Capture Errors (%)		0	0	0
	8 sample set	Partial samples (%)	46.25	0	5.63
		Accuracy (%): Naïve Bayes	93.9	90.6	91.2
		Accuracy (%): Naïve Bayes, maximum 500 features	93.2	92.2	92.3
		Accuracy (%): Random Forest (500 trees), maximum 500 features	93.5	92.3	93.3
	14 sample set	Partial samples (%)	44.29	0	6.79
		Accuracy (%): Naïve Bayes	93.8	90.6	92.0
		Accuracy (%): Naïve Bayes, maximum 500 features	93.3	93.3	93.0
		Accuracy (%): Random Forest (500 trees), maximum 500 features	93.9	94.0	93.0

Table 12 - Data collection details for complexity type 6 results

Appendix B:

Additional Results from Experiment 2

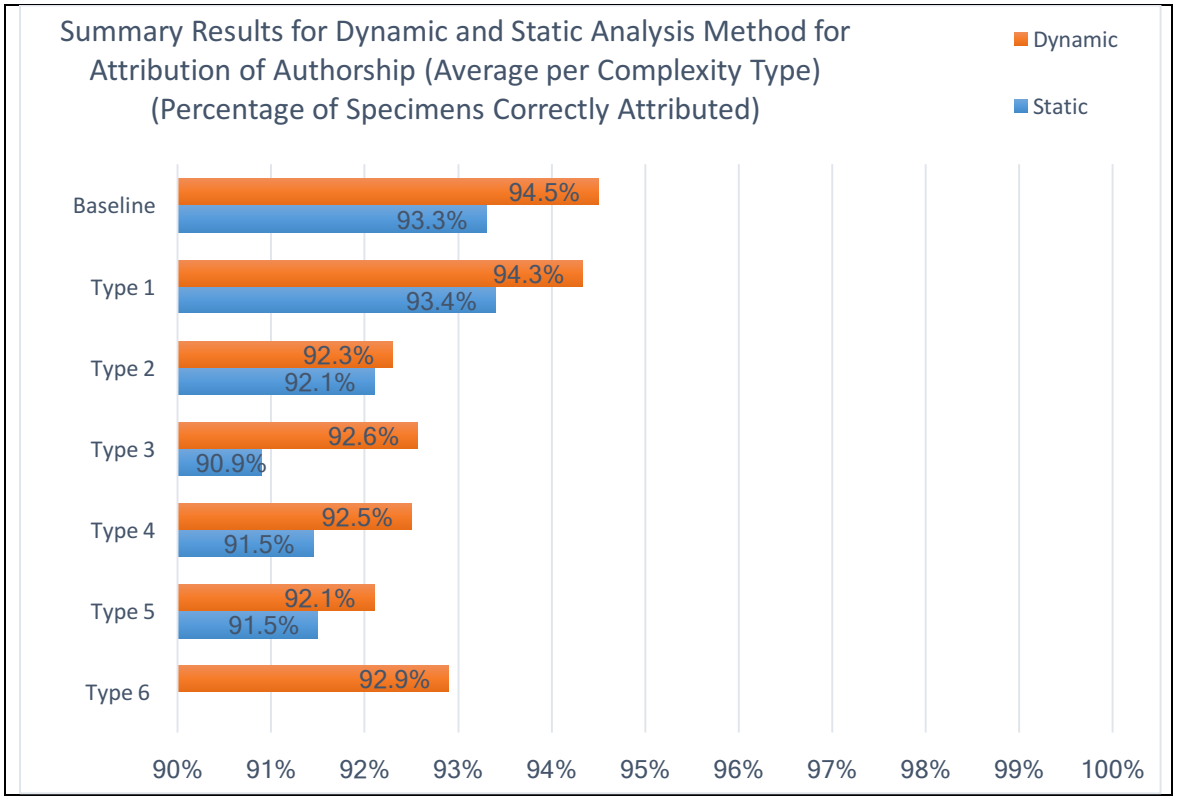

Figure 16 – Summary results of obfuscator attribution by complexity type (Set size of 8)

Appendix C:

Additional Results from Experiment 3

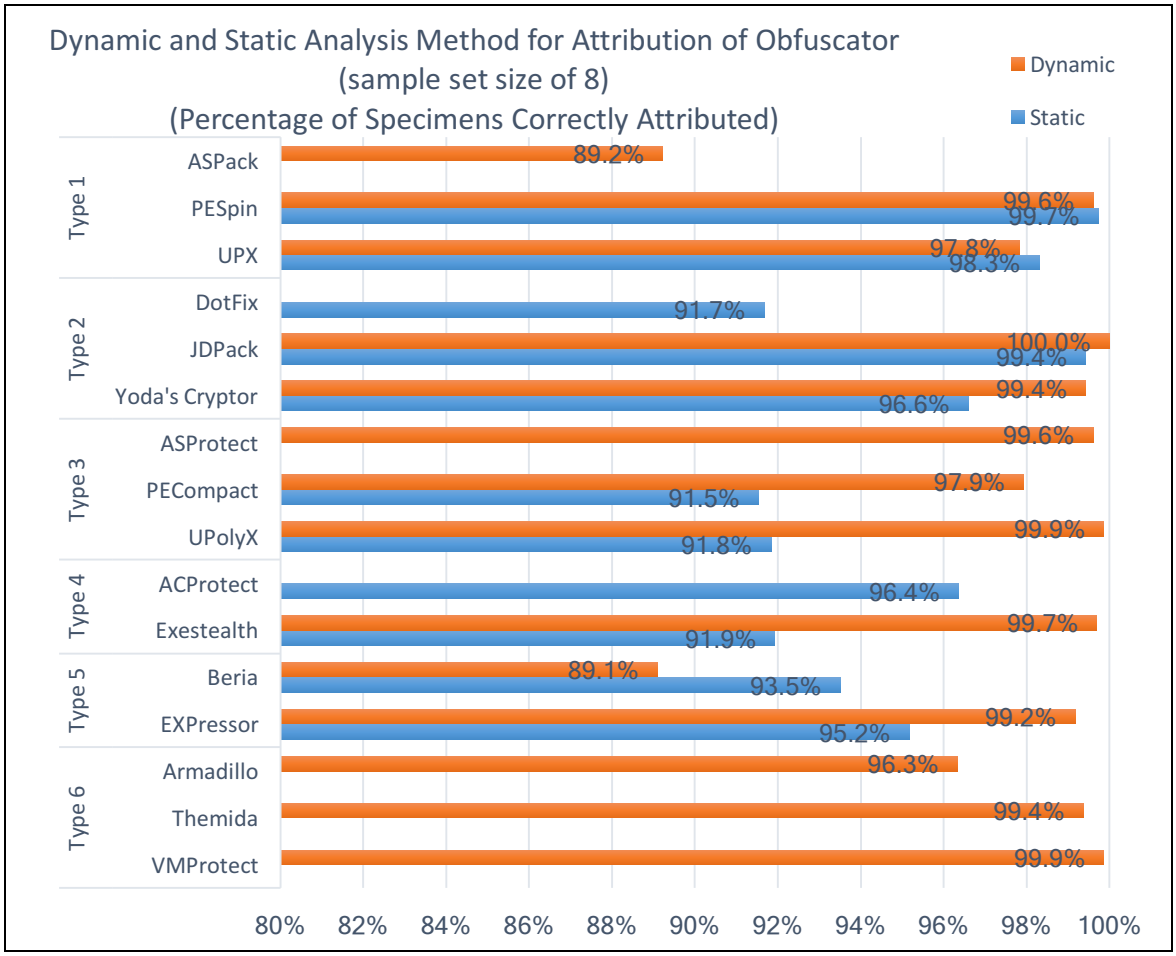

Figure 17 - Results of obfuscator attribution in randomly selected and obfuscated samples (Set size of 8)

Appendix D:

Experiment 4 Data Collection Details

			Static Analysis	Dynamic Analysis		
Version	Total Files	Not Packed (%)	Not Dumped (%)	Partial Collection (%)	Runtime Error (%)	R6002 Error (%)
1.20	70	0	21.4	4.29	68.57	0
1.24	70	0	7.14	2.86	40.0	8.57
1.25	70	0	14.29	8.57	35.71	0
2.02	70	67.14	2.86	0	2.86	0
2.03	70	0	24.29	0	4.29	0
3.06	70	0	7.14	2.86	0	0
3.07	70	11.43	0	4.29	0	0
3.08	70	0	7.14	0	2.86	0
3.09	70	11.43	0	2.86	0	0
3.91	70	2.86	0	2.86	0	0

Table 13 - Data collection details from Experiment 3: UPX versions

	Accuracy Naïve Bayes (%)	Accuracy Naïve Bayes max 500 features (%)	Accuracy Random Forest max 500 features (%)
Dynamic Method	70.7	83.61	81.78
Static Method	72.2	83.0	83.6

Table 14 - Results of UPX Version stylometric similarity, different machine learning algorithms

Appendix E:

Terms and Conditions from Google Code Jam

Taken from https://code.google.com/codejam/terms.html, October 22, 2015.

GOOGLE CODE JAM TERMS AND CONDITIONS

Welcome to Google Code Jam!

These Terms and Conditions ("Terms") apply to Google Code Jam and all related Code Jam contests sponsored by Google Inc. (each referred to as a "Contest"). Please read these Terms carefully as they form a binding legal agreement between you and Google Inc., located at 1600 Amphitheatre Parkway, Mountain View, CA 94043, United States ("Google") with respect to the Contest.

Our Code Jam Contests vary and we may post additional terms and conditions on the Contest website, which become part of these Terms and your agreement with us with respect to the Contest.

YOU MAY NOT SUBMIT AN ENTRY TO A CONTEST AND ARE NOT ELIGIBLE TO RECEIVE PRIZES UNDER A CONTEST UNLESS YOU AGREE TO THESE TERMS. YOUR SUBMISSION OF AN ENTRY IN A CONTEST CONSTITUTES YOUR AGREEMENT TO THESE TERMS.

The words "include" and "including" as used in these Terms mean "including but not limited to."

1. 1. **Eligibility**.
    1. 1.1 VOID WHERE PROHIBITED. The Contest is void in Crimea, Cuba, North Korea, Sudan, Syria, Quebec, and where prohibited by law.
    2. 1.2 NO PURCHASE NECESSARY TO ENTER OR WIN. You do not need to purchase any Google product or service to enter or win the Contest.
    3. 1.3 Ineligible Individuals.
        1. (A) You cannot participate in the Contest if:
            1. (1) You are a resident of Crimea, Cuba, North Korea, Sudan, Syria, or Quebec or anywhere that the Contest is prohibited by law;

2. (2) You are restricted by applicable export controls and sanctions programs;

3. (3) You are a current employee (including intern), contractor, officer, or director, of Google or its affiliates; or

4. (4) You are an immediate family member (including a parent, sibling, child, spouse, or life partner regardless of where you live) of one of the individuals listed in subsection (3) above or you are a member of their household (whether related or not).

2. (B) However, Section 1.3(A)(3) above does not apply if you are a Google Student Ambassador (whether paid or not) and you may participate in the Contest.

3. (C) If you gained information on a problem while working as an employee, intern, contractor, or official office-holder of Google,

    1. (1) you may be disqualified if you attempt to gain points on that problem in the Qualification Round; and

    2. (2) you may not participate in the Contest if that problem is used in any later rounds.

4. 1.4 Requirements to Enter and Receive a Prize.

    1. (A) In order to enter the Contest, you must have

        1. (1) access to the Internet, and

        2. (2) a valid email address.

    2. (B) In order to receive a prize, you must provide your name, phone number, a valid postal address, and any other information Google may need to award or send you a prize.

5. 1.5 Verifying Eligibility. Google reserves the right to verify your eligibility and to adjudicate on any dispute at any time. You agree to provide Google with any proof of eligibility requested by Google and your refusal or failure to provide such proof within 10 days of Google's request will result in your disqualification from the Contest and forfeiture of any prizes.

6. 1.6 Communications. All communications between Google and you, including the Contest website and email communications, will be in English.

2. 2. **How to Enter**.

1. 2.1 Registration. To enter the Contest, you must register at the Contest website and provide the required information about yourself.

    1. (A) Registration times are listed on the Contest website. YOU ARE RESPONSIBLE FOR DETERMINING THE CORRESPONDING TIME IN YOUR TIME ZONE.

    2. (B) You must register for each Contest before you can participate in that Contest. For example, if you registered for Code Jam 2014 and wanted to participate in Google Code Jam to I/O for Women, you would still need to register for the Google Code Jam to I/O for Women Contest.

    3. (C) You may only register for the Contest with one valid email address. If you compete in the same Contest with multiple email addresses, you will be disqualified.

2. 2.2 Screen Names. Google reserves the right to change or omit contestant screen names or nicknames for purposes of publication on Google websites or listserv, particularly if they are, in Google's sole opinion, obscene or violate the intellectual property rights of others.

3. 3. **Contest Structure**.

    1. 3.1 Overview. Each Code Jam Contest consists of one or more rounds as may be more fully described in rules posted on the Contest website. Each round consists of one or more problems. In each round, you will receive a score based on the answers you provide to the problem(s). If the Contest has more than one round and your score exceeds a specified threshold or is one of a specified number of highest scores in that round, you will advance to the next round.

    2. 3.2 Problems. In each round you will be asked to solve one or more algorithmic problems, which may include one or more small inputs, large inputs, or other inputs described in the problem statement. A problem may describe special rules for inputs other than small or large. You will be able to access the problems on the Contest website and download the relevant input files once the round begins.

    3. 3.3 Submissions. You must submit your solutions for problems through the Contest website.

        1. (A) Your submission must be in the format specified by the problem, the Contest website, and these Terms (with precedence given in that order). Deliberately obfuscated source code is not allowed.

2. (B) You should submit your solutions within enough time remaining in each time period to avoid latency issues between your computer and Google servers. Solutions submitted after the applicable time period expires will not be accepted.

4. 3.4 Modifying the Contest. Google may cancel or modify the structure and location of the Contest if technical difficulties prevent or make it unfair to run the Contest in accordance with these Terms.

4. 4. **Judging and Scoring**. The solutions you submit will be judged as follows.

1. 4.1 Points. Each problem is worth a certain value of points. For example, for a problem with a small input and a large input, the solution to the small input may be worth 10 points, while the solution to the large input may be worth 15 points. Your total score for a round will be the sum of the points you earned in that round for each correct solution to an input.

2. 4.2 Ranking Contestants.

1. (A) Highest Score First. Contestants will be ranked in order of the highest score first and lowest score last.

2. (B) Ties Ranked by Penalty Time. In the event of a tie between contestants, the one with the lowest penalty time will be ranked first while the one with the highest penalty time will be ranked last. Penalty times may be specified in the Contest Rules, the Contest website, or in the problem statement.

5. 5. **Advancement and Notice of Winners**.

1. 5.1 Notice of Advancement. You will be notified by email at least one day before the following round whether you have advanced to the next round.

2. 5.2 Announcement of Winners. The results of each round of the Contest will be announced on the Contest website at least one day prior to the next round. The results of the final round of the Contest will be posted on the Contest website after its completion. Posted results will include a list of the contests' names or nicknames in ranked order based on their scores.

6. 6. **Prizes**.

1. 6.1 Money Prizes. Money prizes will be awarded in U.S. dollars and may be delivered in the form of cash, check, gift card, or other cash

equivalent. You are responsible for any costs associated with currency exchange.

2. 6.2 Delivery. Google may either ship your prize to you or request that you come to a Google office to collect your prize. If Google informed you that it will ship your prize to you and you have not received your prize within 6 months after the end of the Contest, please email codejam@google.com. Google will not award any prizes beyond 12 months from the end of the Contest.

3. 6.3 Forfeiture. You will forfeit your prize and an alternative winner may be selected, if:

    1. (A) You fail to provide a phone number during registration or any other information requested by Google within 15 days of Google's request;

    2. (B) You fail to respond to Google's email notification of your prize within 15 days;

    3. (C) You fail to follow directions provided by Google for receiving the prize;

    4. (D) Your prize or prize notification is returned to Google; or

    5. (E) You fail to pay applicable taxes or timely submit applicable documentation to Google or the relevant tax authority.

4. 6.4 Substitute Prizes. Google may provide a substitute prize of equal or greater value at Google's discretion, or where required by law, or in the event all or part of a prize becomes unavailable.

5. 6.5 Taxes. You are solely responsible for complying with all applicable tax laws and filing requirements. To remain eligible for a prize, you must submit to Google or the relevant tax authority all documentation requested by Google or required by applicable law within 7 days of Google's request or earlier if required by law. You are solely responsible for paying all taxes imposed on prizes awarded to you. All prizes will be net of any taxes Google is required by law to withhold.

6. 6.6 No Warranties for Prizes. Except as required by law, Google makes no warranties, express or implied, for prizes.

7. 7. **Disqualification**.

    1. 7.1 You may be disqualified from the Contest and forfeit any prizes you may be eligible to receive if Google reasonably believes that you have

attempted to undermine the legitimate operation of the Contest, including:

1. (A) Providing false information about yourself during registration or concerning your eligibility;

2. (B) Breaching or refusing to comply with these Terms;

3. (C) Tampering or interfering with administration of the Contest (including monitoring at onsite rounds) or with the ability of other contestants to participate in the Contest;

4. (D) Sharing or using from others any information about a problem (including its content or solution) before the end of a round unless expressly permitted by these Terms;

5. (E) Submitting content that:

    1. (1) violates the rights of a third party,

    2. (2) is lewd, obscene, pornographic, racist, sexist, or otherwise inappropriate to the Contest, or

    3. (3) violates any applicable law; or

6. (F) Threatening or harassing other contestants or Google, including its employees and representatives. Harassing behavior includes, offensive, threatening, and/or hateful comments directed toward an individual or protected class (gender, sexual orientation, disability, gender identity, age, race, religion, ethnicity, veteran status), the use or display of sexual images in public spaces, deliberate intimidation, stalking, following, taking unwelcome photos/videos, sustained disruption of talks or other events, inappropriate physical contact, unwelcome sexual attention, and developing and/or promoting any applications designed to encourage any of these behaviors.

2. 7.2 You may report to Google at codejam@google.com any harassment, cheating, or violation of these Terms by another contestant. Google may investigate any such allegations and all decisions by Google in these matters are final and binding. If you are asked to stop any harassing behavior, you are expected to comply immediately.

8. 8. **Rights in Your Submissions; Privacy**.

1. 8.1 Ownership of Submissions. You retain all rights to your submissions of output files and source code that you had before submitting them to the Contest.

2. 8.2 License to Submissions. For any submission you make to the Contest, you grant Google a non-exclusive, worldwide, perpetual, irrevocable, free license (with the right to sublicense) to reproduce, prepare derivative works of, distribute, publicly perform, publicly display, and otherwise use such submission for the purpose of administering and promoting the Contest. Your submitted source code may be made available for anyone to view on the Internet and download and use at the end of the Contest.

3. 8.3 Privacy.

    1. (A) How Google May Use Your Information.

        1. (1) Google will use the information you provide during registration and in any subsequent communications to administer the Contest (including verifying your eligibility to participate in the Contest and delivering prizes). This data may be transferred into the United States and will be maintained in accordance with Google's Privacy Policy.

        2. (2) By accepting a prize, you agree that Google and its agents may use your name, likeness, and statements without compensation to promote the Contest, including displaying it on the Contest website.

    2. (B) Sharing Your Information. Your name and username you create during registration may be displayed publicly on the Contest website. If you win a prize, Google may share your name and address with third parties to fulfill awarding a prize to you.

    3. (C) Accessing Your Information. You may access, review, and update any of your personal data held by Google in connection with the Contest by emailing codejam-claims@google.com or writing to Google (Attention: Code Jam) at the address listed above.

    4. (D) Permission to Record You. If you participate in an onsite round of the Contest, you give permission to Google to make and have made audio, visual, and audiovisual recordings (in any format or media) of you and your activity on a computer while you are participating in the Contest (the "Recordings") and grant Google an unrestricted, sublicensable, transferable, perpetual, irrevocable, worldwide, free license to use the Recordings to promote the Contest. You waive all rights, including any right of prior approval,

and release Google and its agents from, and will neither sue nor bring any proceeding against Google or its agents for, any claim or cause of action, whether now known or unknown, for defamation, copyright infringement, and invasion of the rights to privacy, publicity, or personality or any similar matter, or based upon or relating to the use and exploitation of the Recordings.

9. 9. **Your Representations, Warranties, Indemnities**.

    1. 9.1 Representations and Warranties. You represent and warrant that:

        1. (A) the information you provide about yourself while registering or in subsequent communications with Google is truthful and accurate;

        2. (B) your submissions to the Contest are original and, unless expressly permitted by these Terms, not created with the assistance of any information about the problems not provided by Google;

        3. (C) you own all rights in your submissions or otherwise have the right to submit your submissions to Google and grant to Google the licenses granted in these Terms without violating any rights of any other person or entity or any obligation you may have with them; and

        4. (D) your submissions do not violate any applicable laws.

    2. 9.2 Indemnities. You will indemnify Google and its affiliates, directors, officers, employees against all liabilities, damages, losses, costs, fees (including legal fees), and expenses relating to any allegation or third-party legal proceeding to the extent arising from:

        1. (A) your acts or omissions in relation to the Contest (including your use or acceptance of any prize and your breach of these Terms); and

        2. (B) your submissions violating any rights of any other person or entity or any obligation you may have with them.

10. 10. **Disclaimers**.

    THE CONTEST WEBSITE AND ALL CONTENT (INCLUDING SOURCE CODE) IS PROVIDED ON AN "AS IS" AND "AS AVAILABLE" BASIS. GOOGLE DISCLAIMS ALL REPRESENTATIONS AND WARRANTIES (EXPRESS OR IMPLIED), INCLUDING ANY WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. GOOGLE IS NOT RESPONSIBLE FOR ANY INCOMPLETE, FAILED, OR

DELAYED TRANSMISSION OF YOUR APPLICATION INFORMATION OR SUBMISSIONS DUE TO THE INTERNET, INCLUDING INTERRUPTION OR DELAYS CAUSED BY EQUIPMENT OR SOFTWARE MALFUNCTION OR OTHER TECHNICAL PROBLEMS. YOU USE ALL SOURCE CODE AVAILABLE ON THE CONTEST WEBSITE AT YOUR OWN RISK.

11.11. **General**.

1. 11.1 Not an Offer or Contract Of Employment.
    1. (A) You acknowledge that your participation is voluntary.
    2. (B) You acknowledge that no confidential, fiduciary, agency or other relationship or implied-in-fact contract now exists between you and Google and that no such relationship is established by your submission of an entry to the Contest.
    3. (C) You understand and agree that nothing in these Terms, any Rules, a submission into the Contest, or an award of a prize may be construed as an offer or contract of employment with Google.
2. 11.2 Severability. If any term (or part of a term) of these Terms is invalid, illegal or unenforceable, the rest of the rules will remain in effect.
3. 11.3 Translations. In the event of any discrepancy between the English version of these Terms and a translated version, the English version will govern.
4. 11.4 Governing Law. ALL CLAIMS ARISING OUT OF OR RELATING TO THESE TERMS WILL BE GOVERNED BY CALIFORNIA LAW, EXCLUDING CALIFORNIA'S CONFLICT OF LAWS RULES, AND WILL BE LITIGATED EXCLUSIVELY IN THE FEDERAL OR STATE COURTS OF SANTA CLARA COUNTY, CALIFORNIA, USA; THE PARTIES CONSENT TO PERSONAL JURISDICTION IN THOSE COURTS.

Contest Rules for Code Jam 2015

Welcome to Google Code Jam 2015!

Please read carefully the Google Code Jam Terms and Conditions ("Code Jam Terms"), Google's general Privacy Policy, and these Rules for Code Jam 2015 (described below and referred to as the "Rules"), and all of which together are

referred to as the "Terms", as they form a binding legal agreement between you and Google Inc. with respect to the Google Code Jam 2015 Contest.

1. 1. **Age**. You may participate in the Contest only if you are 13 years of age or older at the time of registration, but you must be 18 years of age or older at the time of registration to be eligible for the onsite final round, if not, you are only eligible to win a t-shirt.

2. 2. **Registration**. Registration for the Contest opens on Tuesday, March 10, 2015 at 19:00 UTC (12:00 PM Pacific Time (PT) in the United States) and ends on Sunday, April 12, 2015 at 2:00 UTC (Saturday, April 11, 2015 at 7:00 PM PT).

3. 3. **Contest Structure**. The Contest consists of two competition tracks: Code Jam and Distributed Code Jam. Code Jam has 5 rounds: a qualification round, Round 1, Round 2, Round 3, and a final round. You must qualify for Round 3 of Code Jam to qualify for Distributed Code Jam. Distributed Code Jam has 2 rounds: Round 1 followed by an onsite final round. The qualification round, Round 1, Round 2, and Round 3 of each track will be conducted online. The final round of each track will take place at the Google office in Seattle, Washington, USA, or such other location as Google may designate.

    1. 3.1 Code Jam Structure.

        1. (A) Qualification Round. Code Jam will start with a qualification round on Friday, April 10, 2015 at 23:00 UTC (4:00 PM PT) and run for 27 hours, ending on Sunday, April 12, 2015 at 2:00 UTC (Saturday, April 11, 2015 at 7:00pm PT). In the Qualification Round, you must log in to the Contest website to attempt to solve a number of problems within the 27-hour period. If you earn a minimum number of points during the qualification round, which will be displayed on the Contest website, you will advance to Round 1 of Code Jam.

        2. (B) Round 1. Code Jam Round 1 is conducted online and is offered in three sub-rounds at the times specified athttps://code.google.com/codejam/schedule.html from Saturday, April 18, 2015 (Friday, April 17th PST) to Sunday, May 10, 2015. If you advanced to Code Jam Round 1, you can participate in any or all of the sub-rounds by logging into the Contest website to solve a number of problems until you qualify for Code Jam Round 2. However, once you qualify for Code Jam Round 2, you may not participate in any later sub-rounds of Code Jam Round 1. You will advance to Code Jam Round 2 if you are one of the top-scoring 1000 contestants from one of the sub-rounds in Code Jam Round 1. You will be notified by email after the end of each sub-round if

you are one of the 3000 contestants advancing to Code Jam Round 2.

3. (C) Round 2. Code Jam Round 2 is conducted online and will begin on Saturday, May 30, 2015 at 14:00 UTC and will end on Saturday, May 30 at 16:30 UTC. To participate in Code Jam Round 2, you must log in to the Contest website to solve problem sets. You will advance to Code Jam Round 3 if you are one of the top-scoring 500 contestants from Code Jam Round 2. You will be notified by email after the end of Code Jam Round 2 if you are one of the 500 contestants advancing to Code Jam Round 3.

4. (D) Round 3. Code Jam Round 3 is conducted online and will begin on Saturday, June 13, 2015 14:00 UTC and will end on Saturday, June 13, 2015 at 16:30 UTC. To participate in Code Jam Round 3, you must log in to the Contest website to solve problem sets. You will advance to the onsite final round of Code Jam if you are one of the top-scoring 25 contestants from Code Jam Round 3. You will be notified by email after the end of Code Jam Round 3 if you are one of the 25 contestants advancing to the final round of Code Jam.

5. (E) Final Round.

    1. (1) Time and Location. The final round of Code Jam will be held on Friday, August 14, 2015 at the Google offices in Seattle, Washington, USA. Google may change the date and location of the final round in its discretion.

    2. (2) Structure. During the final round of Code Jam, you will be asked to solve problem sets using only Google-provided computer equipment. You may choose to use your own keyboard and other materials permitted by Google. Google will send you a list of permitted materials by email at least 7 days before the beginning of the final round.

    3. (3) Last Year's Winner. If you are the winner of Google Code Jam 2014 and enter Google Code Jam 2015 and are otherwise eligible to participate, you will automatically advance to the final round of Code Jam.

2. 3.2 Distributed Code Jam Structure.

    1. (A) Round 1. If you advance to Round 3 of Code Jam, then you qualify to compete in the first round of Distributed Code Jam. Distributed Code Jam Round 1 is conducted online and will begin on Sunday, June 14, 2015 14:00 UTC and will end on Sunday, June 14, 2015 at 17:00 UTC. To participate in Distributed Code

Jam Round 1, you must log in to the Contest website to solve problem sets. You will advance to the onsite final round for Distributed Code Jam if you are one of the top-scoring 10 contestants from Distributed Code Jam Round 1. You will be notified by email after the end of Distributed Code Jam Round 1 if you are one of the 10 contestants advancing to the final round of Distributed Code Jam.

2. (B) Final Round.

    1. (1) Time and Location. The final round of Distributed Code Jam will be held on Saturday, August 15, 2015 at the Google offices in Seattle, Washington, USA. Google may change the date and location of the final round in its discretion.

    2. (2) Structure. During the final round of Distributed Code Jam, you will be asked to solve problem sets using only Google-provided computer equipment. You may choose to use your own keyboard and other materials permitted by Google. Google will send you a list of permitted materials by email at least 7 days before the beginning of the final round.

4. 4. **Problems**.

    1. 4.1 Submitting Solutions for Code Jam Problems.

        1. (A) Submission Requirements. For each Code Jam problem you must submit a correct output file and the source code used in its generation or a written explanation explaining how you solved the problem.

            1. (1) File Format and Size.. You may submit source code as one or more plaintext or zipped plain text files. The size of each source code file may not exceed 100KB, and the total size of your source code for an output after being unzipped may not exceed 1MB. If you use a standard library that is freely available on the Internet that is too large to include in your submission, you may exclude it as long as you put a comment in your source code explaining where the library is available.

            2. (2) Programming Language, Editor, and Compiler. For the qualification round of Code Jam, you may use any programming language to solve a Code Jam problem, using any development environment or text editor. For any round of Code Jam after the qualification round, the compiler or interpreter you use must be available such that anyone else can use it for free without a time limitation and without

violating any rights of any person or entity. However, the following are permissible:

1. (a) Visual Studio, Microsoft Excel, and MATLAB;

2. (b) compilers and interpreters that require Microsoft Windows or Mac OS X, as long as the compiler or interpreter itself is free, as described in this Subsection (2); and

3. (c) any further exceptions that Google may communicate to you by email on the Contest website.

2. (B) Time Limit. When you attempt to solve a particular input for a Code Jam, a timer will start as soon as you begin downloading the file. You must submit your solution in accordance with Subsection (A) above within:

   1. (1) 4 minutes for a small input;

   2. (2) 8 minutes for a large input; or

   3. (3) some other time as specified in the problem for other inputs, but in each case no later than the end of the round of the Contest.

3. (C) Small Inputs. In Code Jam, Google will judge your submissions for small inputs immediately and notify you if your submission is correct or incorrect. If the output file and source code file are not received by the end of the 4 minute period or if the output you submit is judged incorrect, you may choose to attempt to solve that problem again within the time remaining in the round, but will have to download a new small input.

4. (D) Large Inputs. In Code Jam, you can only attempt to solve each large input once. For each large input, you can make multiple output submissions within the 8 minute period, but only your last submission will be judged. Google will notify you whether your submission was correct or incorrect after the round ends by posting the results on the Contest website.

5. (E) Submission Erros and Discrepancies.

   1. (1) During a Round. Google may ignore certain incorrect submissions in Code Jam and notify you that the submission was malformed and that you may submit again in whatever time remains in the period. During a round of Code Jam, if you think you submitted the wrong source code for an output,

you may notify the judges using the "Ask a Question" link on the Contest website. For inputs other than large inputs, the judges may mark the submission as incorrect so that you can attempt the problem again. For a large input for onsite rounds of Code Jam, the judges may, in their discretion, permit resubmission of the source code.

2. (2) After a Round. You may not submit source code for any round after it ends. After the end of a round, if a discrepancy is discovered between the output file you submitted that was judged correct and the corresponding source code that you submitted, then a panel of two or more judges consisting of employees of Google will examine the source code for your submissions. The judges will determine whether a discrepancy exists, and if so will decide in their sole opinion whether the discrepancy is trivial or non-trivial. For a trivial discrepancy, you will be assessed a 4-minute penalty for that input. For a non-trivial discrepancy, you will be assessed a 4-minute penalty in a qualification round and you will forfeit all points for that input in all other rounds.

2. 4.2 Submitting Solutions for Distributed Code Jam Problems. For Distributed Code Jam Problems you must submit through the Contest website the source code for your solution and it will be compiled and run on Google Compute Engine. The Contest website will return a response code indicating whether your solution was accepted or not. The Contest website will provide a list of response codes and their meaning.

   1. (A) Submission Requirements. Your solution for a problem in Distributed Code Jam must be a single file of source code written in one of the approved languages listed on the Contest website which may list additional requirements for a particular language. Your solution:

      (1) cannot exceed 100KB in size;

      (2) must compile to 4MB or less in size;

      (3) must compile in less than 10 seconds under 64-bit Linux;

      (4) cannot fork or create threads;

      (5) cannot replace the currently executed binary with a different one (e.g. you cannot use exec);

      (6) can only use the libraries published on the Contest website;

(7) cannot embed assembler code;

(8) cannot use network functions (you must use the library published on the Contest website to communicate with other instances);

(9) cannot open files (including temporary files);

(10) cannot violate system security;

(11) cannot wait for user interaction;

(12) cannot use more memory than specified in the problem statement; and

(13) must make all instances exit with a 0 exit code.

2. (B) Time Limit. You must submit your solutions for problems in Distributed Code Jam within the round and your solutions must compile and run within the time specified in the problem, the Contest website, and these Terms (with precedence given in that order).

3. (C) Small Inputs. In Distributed Code Jam, you must provide a correct solution for the problem's small input before you can solve the large input. If your solution was not accepted, you may continue to submit new solutions for the small input until it is accepted or until the round expires.

4. (D) Large Inputs. Within a round of Distributed Code Jam, you can submit more than one solution for a large input, but only your last submission will be judged. Google will notify you whether your submission for the large input was correct or incorrect after the round ends by posting the results on the Contest website.

5. (E) Test Runs. You may submit test runs in Distributed Code Jam. The Contest website will provide more information on how to submit a test run and how often you may submit them.

3. 4.3 Penalty Time. Your penalty time for a round in Code Jam or Distributed Code Jam is equal to the time it took you to submit your last correct solution measured from the start of the round, plus 4 minutes for each incorrect solution you submitted for small inputs you eventually solved. Penalty times may also be specified in a problem statement.

5. 5. **Attendance at the Final Round and Related Events**.

1. 5.1 Required Attendance. To remain eligible to participate in the final rounds of the Contest and to receive a prize, you must commit to attending all related organized events as well as the awards ceremony at the end of the Contest. If you fail to attend the awards ceremony or any other event related to the final round that is not optional, Google may, in its discretion, move you to the bottom of the rankings for the final round and award your prize to the contestant with the next highest score until the top three prizes have been awarded. All events related to the final round are open only to contestants, except that you may bring a guest to the reception.

2. 5.2 Travel Expenses. As a finalist, Google will pay for certain expenses related to your travel to and participation in the final round as described below. You must submit to Google or its designated agent within 45 days after the final round any eligible expenses you want reimbursed with detailed receipts and any documentation requested by Google; otherwise, your expenses will not be reimbursed. You are responsible for all other expenses.

    1. (A) Visa. You are responsible for obtaining your own visa prior to arrival to the final round; however, Google will reimburse you for any visa application fee and up to $100 USD in travel expenses incurred each way in obtaining the visa. Google will not reimburse any fees or expenses related to obtaining a passport.

    2. (B) Flight. As a finalist, you will receive round-trip coach class air transportation from the major airport nearest your residence on a flight selected by Google. If you want to fly from an airport other than the major airport nearest your residence or on a flight other than the flight selected by Google, you will have to pay for the difference in cost. If Google changes the location of the final round, you are not entitled to any difference in the cost of airfare paid by Google. Google will reimburse you for the cost to travel to and from each airport up to $50 USD each way.

    3. (C) Accommodations. As a finalist, you will receive hotel accommodations for one at a hotel of Google's choice for the duration of the final round.

    4. (D) Meals. As a finalist, Google will reimburse you for the cost of your meals up to $30 USD per day on days you are travelling to and from the Contest and up to $60 USD per day on non-travel days during the final round.

3. 5.3 TRAVEL DISCLAIMER. YOU UNDERSTAND AND AGREE THAT TRAVEL CONTAINS SOME INHERENT ELEMENT OF RISK OF ACCIDENT, ILLNESS, INJURY, LOSS OR DEATH. IN NO EVENT

WILL GOOGLE, ITS AFFILIATES, OR AGENTS, BE LIABLE FOR ANY HARM, DAMAGE, CLAIM, LOSS OR OTHER ACTION ARISING OUT OF YOUR TRAVEL TO OR FROM THE FINAL ROUND.

6. 6. **Prizes**.

    1. 6.1 Code Jam Prizes.

        1. (A) T-Shirt. You are eligible to receive a t-shirt if you are one of the top-scoring 1000 contestants from Code Jam Round 2 or if you are the winner of Code Jam 2014 and eligible to participate in the final round pursuant to Section 3.1(E)(3) (Last Year's Winner).

        2. (B) Cash Prize. You are eligible to receive one of the following cash prizes if you advance to, attend, and compete in the final round of Code Jam.

Competitor(s)	**Prize**
1st Place	$15,000 USD
2nd Place	$2,000 USD
3rd Place	$1,000 USD
4th—26th Place	$100 USD

    2. 6.2 Distributed Code Jam Prizes. You are eligible to receive one of the following cash prizes if you advance to, attend, and compete in the final round of Distributed Code Jam.

Competitor(s)	**Prize**
1st Place	$3,000 USD

Competitor(s)	Prize
2nd Place	$1,000 USD
3rd Place	$500 USD

7. 7. **Disqualification**.

    1. 7.1 Qualification Round. You will not be disqualified in the qualification round of Code Jam for using or sharing information about problems.

    2. 7.2 Final Round. If you are disqualified from the final round of the Contest, Google may advance the next highest scoring contestant from the previous round. You may be disqualified from a final round if:

        1. (A) You fail to respond to any request for information from Google related to the final round within 5 days of Google's request;

        2. (B) You fail to confirm receipt of your visa letter from Google within 5 days of its receipt;

        3. (C) You fail to book your flight with Google's travel agent within 2 weeks of receiving the email invitation to participate in the final round;

        4. (D) You fail to respond to flight confirmation emails within 48 hours of Google or its agent sending such email;

        5. (E) You fail to provide all documentation and information related to your visa and passport requested by Google within 2 weeks of the final round; or

        6. (F) You fail to compete in the final round, or are unable to do so (for example, because you failed to obtain a passport in time).

Appendix F:

Download Locations of Obfuscation Tools

Obfuscator	Download location
UPX	https://upx.github.io/
ASPack	http://www.aspack.com/aspack.html
PESpin	http://www.pespin.com/
Yoda's Crypter 1.3	https://sourceforge.net/projects/yodap/files/Yoda%20Crypter/1.3/
JDPack 1.01	http://keygens.pro/crack/106500/
DotFix NiceProtect	https://www.niceprotect.com/
UPolyX 0.5	http://www.leetupload.com/database/Win32/Compression%20%20Cryptage/UPolyX%20v0.5.rar
PECompact	https://bitsum.com/portfolio/pecompact/
ASProtect	http://www.aspack.com/asprotect32.html
ACProtect	https://web.archive.org/web/20081216070813/http://www.ultraprotect.com/download.htm
ExeStealth	http://www.webtoolmaster.com/exestealth.htm
Beria 0.7	http://www.reversing.be/article.php?story=20051130102310455&query=iat
EXPressor	http://www.cgsoftlabs.ro/
VMProtect	http://vmpsoft.com/
Themida	http://www.oreans.com/downloads.php
Armadillo 8.0	https://www.bleepingcomputer.com/forums/t/80738/the-silicon-realms-toolworks-armadillo/

Table 15 - Download Locations for Obfuscation Tools used in this Research

References

Abbasi, A., & Chen, H. (2008). Writeprints: A stylometric approach to identity-level identification and similarity detection in cyberspace. *ACM Transactions on Information Systems (TOIS), 26*(2), 7.

Abu-Mostafa, Y. S., Magdon-Ismail, M., & Lin, H.-T. (2012). *Learning from data*: AMLBook Berlin, Germany.

Acosta, J. C., & Medina, B. G. (2012). Poster: Using Semantic Snippets of Malware Traces for Efficient Behavioral Analysis.

Acosta, J. C., Medina, B. G., & Mendoza, H. (2012). An Efficient Common Substrings Algorithm for On-the-Fly Behavior-Based Malware Detection and Analysis.

Adkins, F., Jones, L., Carlisle, M., & Upchurch, J. (2013). *Heuristic malware detection via basic block comparison.* Paper presented at the 8th International Conference on Malicious and Unwanted Software: The Americas (MALWARE).

Ahmed, F., Hameed, H., Shafiq, M. Z., & Farooq, M. (2009). *Using spatio-temporal information in API calls with machine learning algorithms for malware detection.* Paper presented at the Proceedings of the 2nd ACM workshop on Security and artificial intelligence.

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). Compilers: Principles, Techniques, and Tools.

Alam, S., Horspool, R. N., Traore, I., & Sogukpinar, I. (2014). A Framework for Metamorphic Malware Analysis and Real-Time Detection. *Computers & Security*.

Alazab, M. (2015). Profiling and classifying the behavior of malicious codes. *Journal of Systems and Software, 100*, 91-102.

Alazab, M., Venkataraman, S., & Watters, P. (2010). *Towards understanding malware behaviour by the extraction of API calls.* Paper presented at the Cybercrime and Trustworthy Computing Workshop (CTC).

Alazab, M., Venkatraman, S., Watters, P., & Alazab, M. (2011). *Zero-day malware detection based on supervised learning algorithms of api call signatures.* Paper presented at the Proceedings of the Ninth Australasian Data Mining Conference-Volume 121.

Allen, F. E. (1970). *Control flow analysis.* Paper presented at the ACM Sigplan Notices.

Alrabaee, S., Saleem, N., Preda, S., Wang, L., & Debbabi, M. (2014). OBA2: An Onion approach to Binary code Authorship Attribution. *Digital Investigation, 11*, S94-S103.

Alrabaee, S., Shirani, P., Debbabi, M., & Wang, L. (2016). *On the Feasibility of Malware Authorship Attribution.* Paper presented at the International Symposium on Foundations and Practice of Security.

Anckaert, B., Madou, M., De Sutter, B., De Bus, B., De Bosschere, K., & Preneel, B. (2007). *Program obfuscation: a quantitative approach.* Paper presented at the Proceedings of the 2007 ACM Workshop on Quality of Protection.

Anderson, B., Lane, T., & Hash, C. (2014). *Malware Phylogenetics Based on the Multiview Graphical Lasso.* Paper presented at the Advances in Intelligent Data Analysis (IDA) XIII: 13th International Symposium, Leuven, Belgium.

Apel, M., Bockermann, C., & Meier, M. (2009). *Measuring similarity of malware behavior.* Paper presented at the IEEE 34th Conference on Local Computer Networks (LCN).

Arabyarmohamady, S., Moradi, H., & Asadpour, M. (2012). *A coding style-based plagiarism detection.* Paper presented at the International Conference on Interactive Mobile and Computer Aided Learning (IMCL).

Argamon, S., Whitelaw, C., Chase, P., Hota, S. R., Garg, N., & Levitan, S. (2007). Stylistic text classification using functional lexical features. *Journal of the American Society for Information Science and Technology, 58*(6), 802-822.

ASPack Software. (2017a). ASPack. Retrieved from http://www.aspack.com/aspack.html

ASPack Software. (2017b). ASProtect. Retrieved from http://www.aspack.com/asprotect32.html

Austin, T. H., Filiol, E., Josse, S., & Stamp, M. (2013). *Exploring hidden Markov models for virus analysis: A semantic approach.* Paper presented at the 46th Hawaii International Conference on System Sciences (HICSS).

Bai, H., Hu, C.-Z., Jing, X.-C., Li, N., & Wang, X.-Y. (2014). Approach for malware identification using dynamic behaviour and outcome triggering. *Information Security, IET, 8*(2), 140-151.

Balakrishnan, G., & Reps, T. (2010). Wysinwyx: What you see is not what you execute. *ACM Transactions on Programming Languages and Systems (TOPLAS), 32*(6), 23.

Balakrishnan, R., & Anbarasu, J. (2013). *Automated diagnosis and prevention of unsafe dynamic software component loadings.* Paper presented at the IEEE Conference on Information & Communication Technologies (ICT).

Baldwin, J., Sinha, P., Salois, M., & Coady, Y. (2011). *Progressive user interfaces for regressive analysis: Making tracks with large, low-level systems.* Paper presented at the Proceedings of the Twelfth Australasian User Interface Conference-Volume 117.

Bao, T., Burket, J., Woo, M., Turner, R., & Brumley, D. (2014). BYTEWEIGHT: Learning to recognize functions in binary code. *Proceedings of USENIX Security 2014*.

Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., & Yang, K. (2001). *On the (im) possibility of obfuscating programs.* Paper presented at the Advances in cryptology—CRYPTO 2001.

Barat, M., Prelipcean, D.-B., & Gavriluţ, D. T. (2013). A study on common malware families evolution in 2012. *Journal of Computer Virology and Hacking Techniques, 9*(4), 171-178.

Baron, G. (2014). Influence of data discretization on efficiency of Bayesian classifier for authorship attribution. *Procedia Computer Science, 35*, 1112-1121.

Barr, S. J., Cardman, S. J., & Martin Jr, D. M. (2008). A boosting ensemble for the recognition of code sharing in malware. *Journal in Computer Virology, 4*(4), 335-345.

Barthélemy, J.-P., Brucker, F., & Osswald, C. (2004). Combinatorial optimization and hierarchical classifications. *Quarterly Journal of the Belgian, French and Italian Operations Research Societies, 2*(3), 179-219.

Barthou, D., Rubial, A. C., Jalby, W., Koliai, S., & Valensi, C. (2010). Performance tuning of x86 openmp codes with maqao *Tools for High Performance Computing 2009* (pp. 95-113): Springer.

Bat-Erdene, M., Kim, T., Park, H., & Lee, H. (2017). Packer Detection for Multi-Layer Executables Using Entropy Analysis. *Entropy, 19*(3), 125.

Bat-Erdene, M., Park, H., Li, H., Lee, H., & Choi, M.-S. (2016). Entropy analysis to classify unknown packing algorithms for malware detection. *International Journal of Information Security*, 1-22.

Beaucamps, P., & Filiol, É. (2007). On the possibility of practically obfuscating programs towards a unified perspective of code protection. *Journal in Computer Virology, 3*(1), 3-21.

Belkhouche, B., Nix, A., & Hassell, J. (2004). *Plagiarism detection in software designs.* Paper presented at the Proceedings of the 42nd Annual Southeast Regional Conference.

Bellard, F. (2005). *QEMU, a Fast and Portable Dynamic Translator.* Paper presented at the USENIX Annual Technical Conference, FREENIX Track.

Berghel, H., & Sallach, D. (1984). Measurements of program similarity in identical task environments. *ACM Sigplan Notices, 19*(8), 65-76.

Bhansali, S., Chen, W.-K., De Jong, S., Edwards, A., Murray, R., Drinić, M., . . . Chau, J. (2006). *Framework for instruction-level tracing and analysis of program executions.* Paper presented at the Proceedings of the 2nd international conference on Virtual execution environments.

Bhattathiripad, P. V. (2012). *A Proposal for Incorporating Programming Blunder as Important Evidence in Abstraction-Filtration-Comparison Test.* Paper presented at the Proceedings of the Conference on Digital Forensics, Security and Law.

Bishop, C. (2001). Bishop Pattern Recognition and Machine Learning: Springer, New York.

blackd0t. (2009a). # MS VC - challenge for PE packers. Retrieved from http://nezumi-lab.org/blog/?p=73

blackd0t. (2009b). PE packer causing runtime MSVC++ runtime R6002 error. Retrieved from http://www.openrce.org/forums/posts/1002

Blokhin, K., Saxe, J., & Mentis, D. (2013). *Malware Similarity Identification Using Call Graph Based System Call Subsequence Features.* Paper presented at the IEEE 33rd International Conference on Distributed Computing Systems Workshops (ICDCSW).

Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM, 13*(7), 422-426.

Bourquin, M., King, A., & Robbins, E. (2013). *BinSlayer: accurate comparison of binary executables.* Paper presented at the Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop.

Bozkurt, I. N., Baghoglu, O., & Uyar, E. (2007). *Authorship attribution.* Paper presented at the 22nd International Symposium on Computer and Information Sciences (ISCIS)

Breiman, L. (2001). Random forests. *Machine learning, 45*(1), 5-32.

Broadhurst, R., Grabosky, P., Alazab, M., Bouhours, B., & Chon, S. (2013). Organizations and Cybercrime. *Available at SSRN 2345525*.

Brumley, D., Jager, I., Avgerinos, T., & Schwartz, E. J. (2011). *BAP: a binary analysis platform.* Paper presented at the Computer aided verification.

Buck, B., & Hollingsworth, J. K. (2000). An API for runtime code patching. *International Journal of High Performance Computing Applications, 14*(4), 317-329.

Burg, D. B., McConkey, K., & Amra, J. (2016). Cybercrime. Retrieved from http://www.pwc.com/gx/en/services/advisory/forensics/economic-crime-survey/cybercrime.html

Burrows, S., Uitdenbogerd, A. L., & Turpin, A. (2009). *Temporally robust software features for authorship attribution.* Paper presented at the 33rd Annual IEEE International Computer Software and Applications Conference, 2009. (COMPSAC'09). .

Burrows, S., Uitdenbogerd, A. L., & Turpin, A. (2014). Comparing techniques for authorship attribution of source code. *Software: Practice and Experience, 44*(1), 1-32.

Caballero, J., Johnson, N. M., McCamant, S., & Song, D. (2009). *Binary code extraction and interface identification for security applications.* Paper presented at the Proceedings of the Network and Distributed System Security Symposium, NDSS San Diego, California, USA.

Caliskan, A., & Greenstadt, R. (2012). *Translate once, translate twice, translate thrice and attribute: Identifying authors and machine translation tools in translated text.* Paper presented at the IEEE Sixth International Conference on Semantic Computing (ICSC), 2012

Caliskan-Islam, A., Harang, R., Liu, A., Narayanan, A., Voss, C., Yamaguchi, F., & Greenstadt, R. (2015). De-anonymizing Programmers via Code Stylometry: Manuscript.

Caliskan-Islam, A., Yamaguchi, F., Dauber, E., Harang, R., Rieck, K., Greenstadt, R., & Narayanan, A. (2015). When Coding Style Survives Compilation: De-anonymizing Programmers from Executable Binaries. *arXiv preprint arXiv:1512.08546*.

Calleja, A., Tapiador, J., & Caballero, J. (2016). *A Look into 30 Years of Malware Development from a Software Metrics Perspective.* Paper presented at the International Symposium on Research in Attacks, Intrusions, and Defenses.

Cappaert, J., Preneel, B., Anckaert, B., Madou, M., & De Bosschere, K. (2008). Towards tamper resistant code encryption: Practice and experience. *lecture notes in Computer Science, 4991*, 86-100.

Castán, M. A. B. (2014). Análisis de rendimiento y niveles de protección de protectores de software.

Cesare, S., & Xiang, Y. (2011). *Malware variant detection using similarity search over sets of control flow graphs.* Paper presented at the IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom).

CGSoftLabs. (2017). eXPressor v. 1.8.0.1. Retrieved from http://www.cgsoftlabs.ro/

Chae, D.-K., Ha, J., Kim, S.-W., Kang, B., & Im, E. G. (2013). *Software plagiarism detection: a graph-based approach.* Paper presented at the Proceedings of the 22nd ACM international conference on Conference on information & knowledge management.

Chae, D.-K., Kim, S.-W., Ha, J., Lee, S.-C., & Woo, G. (2013). *Software plagiarism detection via the static API call frequency birthmark.* Paper presented at the Proceedings of the 28th Annual ACM Symposium on Applied Computing.

Chaki, S., Cohen, C., & Gurfinkel, A. (2011). *Supervised learning for provenance-similarity of binaries.* Paper presented at the Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining.

Chaki, S., Cohen, C., Gurfinkel, A., & Havrilla, J. (2013). 4 Semantic Comparison of Malware Functions. *Results of SEI Line-Funded Exploratory New Starts Projects*, 34.

Charif-Rubial, A. S., Barthou, D., Valensi, C., Shende, S., Malony, A., & Jalby, W. (2013). *Mil: A language to build program analysis tools through static binary instrumentation.* Paper presented at the 20th International Conference on High Performance Computing (HiPC).

Chatzicharalampous, E., Frantzeskou, G., & Stamatatos, E. (2012). *Author Identification in Imbalanced Sets of Source Code Samples.* Paper presented at the IEEE 24th International Conference on Tools with Artificial Intelligence (ICTAI).

Chen, H. (2013). *The Influences of Compiler Optimization on Binary Files Similarity Detection.* Paper presented at the 2013 the International Conference on Education Technology and Information System (ICETIS 2013).

Chen, X., Andersen, J., Mao, Z. M., Bailey, M., & Nazario, J. (2008). *Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware.* Paper presented at the IEEE International Conference on Dependable Systems and Networks With FTCS and DCC.

Chilowicz, M., Duris, E., & Roussel, G. (2009). *Syntax tree fingerprinting for source code similarity detection.* Paper presented at the IEEE 17th International Conference on Program Comprehension, ICPC'09. .

Chouchane, R., Stakhanova, N., Walenstein, A., & Lakhotia, A. (2013). Detecting machine-morphed malware variants via engine attribution. *Journal of Computer Virology and Hacking Techniques*, 1-21.

Chyłek, S. (2009). *Collecting program execution statistics with Qemu processor emulator.* Paper presented at the International Multiconference on Computer Science and Information Technology, 2009. (IMCSIT'09). .

Cifuentes, C. (1995). *An environment for the reverse engineering of executable programs.* Paper presented at the 1995 Asia Pacific Software Engineering Conference, 1995. .

Cifuentes, C. (1996). *Partial automation of an integrated reverse engineering environment of binary code.* Paper presented at the Proceedings of the Third Working Conference on Reverse Engineering.

Cifuentes, C., & Gough, K. J. (1995). Decompilation of binary programs. *Software: Practice and Experience, 25*(7), 811-829.

Cifuentes, C., & Van Emmerik, M. (1999). *Recovery of jump table case statements from binary code.* Paper presented at the Seventh International Workshop on Program Comprehension.

Cohen, F. B. (1993). Operating system protection through program evolution. *Computers & Security, 12*(6), 565-584.

Collake, J. (2017). PECompact – Windows (PE) Executable Compressor. Retrieved from https://bitsum.com/portfolio/pecompact/

Collberg, C., & Thomborson, C. (2002). Watermarking, tamper-proofing, and obfuscation-tools for software protection. *IEEE Transactions on Software Engineering., 28*(8), 735-746.

Collberg, C., Thomborson, C., & Low, D. (1997). A taxonomy of obfuscating transformations.

Collberg, C., Thomborson, C., & Low, D. (1998a). *Breaking abstractions and unstructuring data structures.* Paper presented at the International Conference on Computer Languages.

Collberg, C., Thomborson, C., & Low, D. (1998b). *Manufacturing cheap, resilient, and stealthy opaque constructs.* Paper presented at the Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.

Coogan, K., Debray, S., Kaochar, T., & Townsend, G. (2009). *Automatic static unpacking of malware binaries.* Paper presented at the 16th Working Conference on Reverse Engineering (WCRE'09)

Coogan, K., Lu, G., & Debray, S. (2011). *Deobfuscation of virtualization-obfuscated software: a semantics-based approach.* Paper presented at the Proceedings of the 18th ACM conference on Computer and communications security.

Coppieters, K. (1995). A cross-platform binary diff. *Dr Dobb's Journal-Software Tools for the Professional Programmer, 20*(5), 32-39.

Creswell, J. W. (2012). *Qualitative inquiry and research design: Choosing among five approaches*: Sage.

cyberbob. (2017). PESpin. Retrieved from http://www.pespin.com/

DarkReading. (2016). Kaspersky Lab: 323,000 New Malware Samples Found Each Day. Retrieved from http://www.darkreading.com/vulnerabilities---threats/kaspersky-lab-323000-new-malware-samples-found-each-day/d/d-id/1327655

Darmetko, C., Jilcott, S., & Everett, J. (2013). *Inferring Accurate Histories of Malware Evolution from Structural Evidence.* Paper presented at the The Twenty-Sixth International FLAIRS Conference.

Dauber, E., Caliskan-Islam, A., Harang, R., & Greenstadt, R. (2017). Git Blame Who?: Stylistic Authorship Attribution of Small, Incomplete Source Code Fragments. *arXiv preprint arXiv:1701.05681*.

David, Y., & Yahav, E. (2014). *Tracelet-based code search in executables.* Paper presented at the ACM Sigplan Notices.

Davies, J., German, D. M., Godfrey, M. W., & Hindle, A. (2011). *Software bertillonage: finding the provenance of an entity.* Paper presented at the Proceedings of the 8th working conference on mining software repositories.

Derevenets, Y. (2016). Snowman. Retrieved from https://derevenets.com/

Devi, D., & Nandi, S. (2012). *Detection of packed malware.* Paper presented at the Proceedings of the First International Conference on Security of Internet of Things.

Dinaburg, A., Royal, P., Sharif, M., & Lee, W. (2008). *Ether: malware analysis via hardware virtualization extensions.* Paper presented at the Proceedings of the 15th ACM conference on Computer and communications security.

Ding, Y., Dai, W., Yan, S., & Zhang, Y. (2014). Control flow-based opcode behavior analysis for Malware detection. *Computers & Security, 44*, 65-74.

Distributed Management Task Force, I. (2016). Open Virtualization Format. Retrieved from http://www.dmtf.org/standards/ovf

Donaldson, J. L., Lancaster, A.-M., & Sposato, P. H. (1981). *A plagiarism detection system.* Paper presented at the ACM Sigcse Bulletin.

DoxFix Software. (2017). DotFix NiceProtect. Retrieved from https://www.niceprotect.com/

Dullien, T., Carrera, E., Eppler, S.-M., & Porst, S. (2010). Automated attacker correlation for malicious code. http://www.researchgate.net/publication/235169390_Automated_Attacker_Correlation_for_Malicious_Code Retrieved from http://www.researchgate.net/publication/235169390_Automated_Attacker_Correlation_for_Malicious_Code

Dullien, T., & Rolles, R. (2005). Graph-based comparison of executable objects (English version). *Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC), 5*, 1-3.

Dumitras, T., & Neamtiu, I. (2011). *Experimental Challenges in Cyber Security: A Story of Provenance and Lineage for Malware.* Paper presented at the 4th Workshop on Cyber Security Experimentation and Test (CSET).

Đurić, Z., & Gašević, D. (2013). A Source Code Similarity System for Plagiarism Detection. *The Computer Journal, 56*(1), 70-86.

Eagle, C. (2008). *The IDA pro book: the unofficial guide to the world's most popular disassembler*: No Starch Press.

Eddy, S. R. (1996). Hidden markov models. *Current opinion in structural biology, 6*(3), 361-365.

Egele, M., Kruegel, C., Kirda, E., Yin, H., & Song, D. X. (2007). *Dynamic Spyware Analysis.* Paper presented at the USENIX annual technical conference.

Egele, M., Scholte, T., Kirda, E., & Kruegel, C. (2012). A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys (CSUR), 44*(2), 6.

Egele, M., Woo, M., Chapman, P., & Brumley, D. (2014). *Blanket execution: dynamic similarity testing for program binaries and components.* Paper presented at the 23rd USENIX Security Symposium (USENIX Security 14),(San Diego, CA), USENIX Association.

Emmerik, M., & Waddington, T. (2004). *Using a decompiler for real-world source recovery.* Paper presented at the 11th Working Conference on Reverse Engineering.

Engels, S., Lakshmanan, V., & Craig, M. (2007). *Plagiarism detection using feature-based neural networks.* Paper presented at the ACM Sigcse Bulletin.

Evans, M., & Scott, P. (2017). Fraud and cyber crime are now the country's most common offences. Retrieved from http://www.telegraph.co.uk/news/2017/01/19/fraud-cyber-crime-now-countrys-common-offences/

Fang, H., Wu, Y., Wang, S., & Huang, Y. (2011). Multi-stage binary code obfuscation using improved virtual machine *Information Security* (pp. 168-181): Springer.

Farhadi, M. R. (2013). *Assembly Code Clone Detection for Malware Binaries.* Concordia University.

Farhadi, M. R., Fung, B., Charland, P., & Debbabi, M. (2014). *BinClone: Detecting Code Clones in Malware.* Paper presented at the Eighth International Conference on Software Security and Reliability.

Federal Bureau of Investigation. (2014). Four Members of International Computer Hacking Ring Indicted for Stealing Gaming Technology, Apache Helicopter Training Software Retrieved from http://www.fbi.gov/baltimore/press-releases/2014/four-members-of-international-computer-hacking-ring-indicted-for-stealing-gaming-technology-apache-helicopter-training-software

Flake, H. (2004). Structural comparison of executable objects. http://static.googleusercontent.com/media/www.zynamics.com/en//downloads/dimva_paper2.pdf Retrieved from http://static.googleusercontent.com/media/www.zynamics.com/en//downloads/dimva_paper2.pdf

Frantzeskou, G., Gritzalis, S., & MacDonell, S. G. (2004). *Source code authorship analysis for supporting the cybercrime investigation process.* Paper presented at

the 1st International Conference on E-Business and Telecommunication Networks, Setúbal, Portugal.

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., Chaski, C. E., & Howald, B. S. (2007). Identifying authorship by byte-level n-grams: The source code author profile (scap) method. *International Journal of Digital Evidence, 6*(1), 1-18.

Freire, M. (2008). *Visualizing program similarity in the AC plagiarism detection system.* Paper presented at the Proceedings of the working conference on Advanced visual interfaces.

Frizell, S. (2014). 600 Retailers Ensnared in Major New Malware Attack, Cybersecurity Firm Says. Retrieved from http://time.com/3070555/malware-backoff-dhs-hacking-retail/

Furchtgott-Roth, D. (2017). Why intellectual property theft is one of the biggest crimes threatening the US economy. Retrieved from http://www.businessinsider.com/intellectual-property-theft-biggest-crimes-us-economy-2017-1

Furrow, B., & Navernioukk, I. (2016). Problem Preparation Guide. Retrieved from https://code.google.com/codejam/problem-preparation.html

Gandotra, E., Bansal, D., & Sofat, S. (2014). Malware Analysis and Classification: A Survey. *Journal of Information Security, 2014*.

Gao, D., Reiter, M. K., & Song, D. (2008). Binhunt: Automatically finding semantic differences in binary programs *Information and Communications Security* (pp. 238-255): Springer.

GCC. (2015). GCC, the GNU Compiler Collection. Retrieved from https://gcc.gnu.org/

GCC. (2016). 3.10 Options That Control Optimization. Retrieved from https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

Gitchell, D., & Tran, N. (1999). *Sim: a utility for detecting similarity in computer programs.* Paper presented at the ACM Sigcse Bulletin.

GitHub Inc. (2017). The world's leading software development platform · GitHub. Retrieved from https://github.com/

Godfrey, M. W. (2013). Understanding software artifact provenance. *Science of Computer Programming*.

Google. (2015). Google Code Jam 2015. Retrieved from https://code.google.com/codejam

Google Inc. (2017). Quick-Start Guide. Retrieved from https://code.google.com/codejam/resources/quickstart-guide - io-tutorial

Grabosky, P. (2014). The Evolution of Cybercrime, 2004-2014. *RegNet Research Paper*(2014/58).

Gray, A., Sallis, P., & MacDonell, S. (1997). Software Forensics: Extending Authorship Analysis Techniques to Computer Programs. *Journal of Law and Information Science, 13*(1), 34-69.

Grier, S. (1981). *A tool that detects plagiarism in Pascal programs.* Paper presented at the ACM Sigcse Bulletin.

Grune, D., Van Reeuwijk, K., Bal, H. E., Jacobs, C. J., & Langendoen, K. (2012). *Modern compiler design*: Springer Science & Business Media.

Gupta, A., Kuppili, P., Akella, A., & Barford, P. (2009). *An empirical study of malware evolution.* Paper presented at the First International Communication Systems and Networks and Workshops (COMSNETS).

haggar. (2016). berial 0.07 - manually unpacking. Retrieved from http://www.reversing.be/article.php?story=20051130102310455&query=iat

Halstead, M. H. (1972). Natural laws controlling algorithm structure? *ACM Sigplan Notices, 7*(2), 19-26.

Halstead, M. H. (1977). *Elements of Software Science (Operating and programming systems series)*. New York, NY: Elsevier Science Inc.

Han, K. S., Kang, B., & Im, E. G. (2011). *Malware classification using instruction frequencies.* Paper presented at the Proceedings of the 2011 ACM Symposium on Research in Applied Computation.

Haneda, M., Knijnenburg, P. M., & Wijshoff, H. A. (2005). *Optimizing general purpose compiler optimization.* Paper presented at the Proceedings of the 2nd conference on Computing frontiers.

Hannabuss, S. (2001). Contested texts: issues of plagiarism. *Library management, 22*(6/7), 311-318.

Hanspeter Imp. (2017). Exestealth. Retrieved from http://www.webtoolmaster.com/exestealth.htm

Harris, L. C., & Miller, B. P. (2005). Practical analysis of stripped binary code. *ACM SIGARCH Computer Architecture News, 33*(5), 63-68.

Hastie, T. J., Tibshirani, R. J., & Friedman, J. H. (2009). *The elements of statistical learning: data mining, inference, and prediction*: Springer.

Hayes, J. H. (2008). Authorship Attribution: A Principal Component and Linear Discriminant Analysis of the Consistent Programmer Hypothesis. *IJ Comput. Appl., 15*(2), 79-99.

Hayes, J. H., & Offutt, J. (2010). Recognizing authors: an examination of the consistent programmer hypothesis. *Software Testing, Verification and Reliability, 20*(4), 329-356.

Hayes, M., Walenstein, A., & Lakhotia, A. (2009). Evaluation of malware phylogeny modelling systems using automated variant generation. *Journal in Computer Virology, 5*(4), 335-343.

Henderson, A., Prakash, A., Yan, L. K., Hu, X., Wang, X., Zhou, R., & Yin, H. (2014). *Make it work, make it right, make it fast: Building a platform-neutral whole-system dynamic binary analysis platform.* Paper presented at the Proceedings of the 2014 International Symposium on Software Testing and Analysis.

Hex-Rays SA. (2015a, 2015-08-14). IDA F.L.I.R.T. Technology: In-Depth. Retrieved from https://www.hex-rays.com/products/ida/tech/flirt/in_depth.shtml

Hex-Rays SA. (2015b). IDA: About. Retrieved from https://www.hex-rays.com/products/ida/

Horspool, R. N., & Marovac, N. (1980). An approach to the problem of detranslation of computer programs. *The Computer Journal, 23*(3), 223-229.

Hosic, J., Tauritz, D. R., & Mulder, S. A. (2014). *Evolving Decision Trees for the Categorization of Software.* Paper presented at the IEEE 38th International on Computer Software and Applications Conference Workshops (COMPSACW),.

Hoste, K., & Eeckhout, L. (2008). *Cole: compiler optimization level exploration.* Paper presented at the Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization.

Huang, H.-D., Lee, C.-S., Kao, H.-Y., Tsai, Y.-L., & Chang, J.-G. (2011). *Malware behavioral analysis system: TWMAN.* Paper presented at the IEEE Symposium on Intelligent Agent (IA).

Iliopoulos, D., Adami, C., & Szor, P. (2011). Darwin inside the machines: Malware evolution and the consequences for computer security. *arXiv preprint arXiv:1111.2503*.

Intel. (2015). Intel C++ Compilers. Retrieved from https://software.intel.com/en-us/c-compilers

Intel. (2016). Step by Step Performance Optimization with Intel® C++ Compiler. Retrieved from https://software.intel.com/en-us/articles/step-by-step-optimizing-with-intel-c-compiler

Iqbal, F., Binsalleeh, H., Fung, B., & Debbabi, M. (2010). Mining writeprints from anonymous e-mails for forensic investigation. *Digital Investigation, 7*(1), 56-64.

Iqbal, F., Khan, L. A., Fung, B., & Debbabi, M. (2010). *E-mail authorship verification for forensic investigation.* Paper presented at the Proceedings of the 2010 ACM Symposium on Applied Computing.

Iwamoto, K., & Wasaki, K. (2012). *Malware classification based on extracted api sequences using static analysis.* Paper presented at the Proceedings of the Asian Internet Engineeering Conference.

Jacobson, E. R., Rosenblum, N., & Miller, B. P. (2011). *Labeling library functions in stripped binaries.* Paper presented at the Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools.

Jadalla, A., & Elnagar, A. (2008). PDE4Java: Plagiarism Detection Engine for Java source code: a clustering approach. *International Journal of Business Intelligence and Data Mining, 3*(2), 121-135.

Jane Huffman Hayes, J. O. (2009). Recognizing Authors: A Case Study of the Consistent Programmer Hypothesis. *Journal of Software Testing Verification and Reliability, 20*(4), 329-356.

Jang, J., Choi, S., & Hong, J. (2012). *A method for resilient graph-based comparison of executable objects.* Paper presented at the Proceedings of the 2012 ACM Research in Applied Computation Symposium.

Jang, J., Woo, M., & Brumley, D. (2013). *Towards Automatic Software Lineage Inference.* Paper presented at the USENIX Security.

Jankowitz, H. T. (1988). Detecting Plagiarism in Student Pascale Programs. *The Computer Journal, 31*(1), 1-8.

JDPack. (2016). JDPack. Retrieved from http://keygens.pro/crack/106500/

Ji, J., Park, S., Woo, G., & Cho, H. (2007). *Understanding the evolution process of program source for investigating software authorship and plagiarism.* Paper presented at the Digital Information Management, 2007. ICDIM'07. 2nd International Conference on.

Jones, S. (2014). Energy companies hit by cyber attack from Russia-linked group. Retrieved from http://www.ft.com/cms/s/0/606b97b4-0057-11e4-8aaf-00144feab7de.html - axzz3Ces5rS4M

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

Juricic, V. (2011). *Detecting source code similarity using low-level languages.* Paper presented at the 33rd International Conference on Information Technology Interfaces (ITI).

Kang, B., Kim, H. S., Kim, T., Kwon, H., & Im, E. G. (2011). *Fast malware family detection method using control flow graphs.* Paper presented at the Proceedings of the 2011 ACM Symposium on Research in Applied Computation.

Kang, M. G., Poosankam, P., & Yin, H. (2007). *Renovo: A hidden code extractor for packed executables.* Paper presented at the Proceedings of the 2007 ACM workshop on Recurring malcode.

Kang, M. G., Yin, H., Hanna, S., McCamant, S., & Song, D. (2009). *Emulating emulation-resistant malware.* Paper presented at the Proceedings of the 1st ACM workshop on Virtual machine security.

Kantchelian, A., Afroz, S., Huang, L., Islam, A. C., Miller, B., Tschantz, M. C., . . . Tygar, J. (2013). *Approaches to adversarial drift.* Paper presented at the Proceedings of the 2013 ACM workshop on Artificial Intelligence and Security.

Kanzaki, Y., Monden, A., Nakamura, M., & Matsumoto, K.-i. (2003). *Exploiting self-modification mechanism for program protection.* Paper presented at the 27th Annual International Computer Software and Applications Conference, 2003. (COMPSAC 2003). .

Kanzaki, Y., Monden, A., Nakamura, M., & Matsumoto, K. I. (2006). A software protection method based on instruction camouflage. *Electronics and Communications in Japan (Part III: Fundamental Electronic Science), 89*(1), 47-59.

Karim, M. E., Walenstein, A., Lakhotia, A., & Parida, L. (2005). Malware phylogeny generation using permutations of code. *Journal in Computer Virology, 1*(1-2), 13-23.

Kaspersky Labs. (2012). 2012 by the numbers: Kaspersky Lab now detects 200,000 new malicious programs every day. Retrieved from http://www.kaspersky.com/about/news/virus/2012/2012_by_the_numbers_Kaspersky_Lab_now_detects_200000_new_malicious_programs_every_day

Kawakoya, Y., Iwamura, M., & Itoh, M. (2010). *Memory behavior-based automatic malware unpacking in stealth debugging environment.* Paper presented at the Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on.

Khoo, W. M., & Lió, P. (2011). *Unity in diversity: Phylogenetic-inspired techniques for reverse engineering and detection of malware families.* Paper presented at the First SysSec Workshop (SysSec).

Ki, Y., Kim, E., & Kim, H. K. (2015). A Novel Approach to Detect Malware Based on API Call Sequence Analysis. *International Journal of Distributed Sensor Networks, 501*, 659101.

Kilgour, R., Gray, A., Sallis, P., & MacDonell, S. (1998). *A fuzzy logic approach to computer software source code authorship analysis.* Paper presented at the 1997 International Conference on Neural Information Processing and Intelligent Information Systems, Dunedin, New Zealand.

Kinable, J. (2010). *Malware Detection Through Call Graphs.* (Master in Security and Mobile Computing), Norwegian University of Science and Technology. Retrieved from https://daim.idi.ntnu.no/masteroppgaver/005/5741/masteroppgave.pdf

Kinable, J., & Kostakis, O. (2011). Malware classification based on call graph clustering. *Journal in Computer Virology, 7*(4), 233-245.

Kinder, J. (2012). *Towards static analysis of virtualization-obfuscated binaries.* Paper presented at the 19th Working Conference on Reverse Engineering. (WCRE).

Kinder, J., & Veith, H. (2008). *Jakstab: A static analysis platform for binaries.* Paper presented at the 20th International Conference on Computer Aided Verification (CAV), Princeton, NJ.

Kirk, J. (2013). Three indicted in alleged source code theft from trading house. Retrieved from http://www.computerworld.com/article/2485710/security0/three-indicted-in-alleged-source-code-theft-from-trading-house.html

Kirk, J. (2014). Six more US retailers attacked like Target, security firm says. Retrieved from http://www.pcworld.com/article/2089480/six-more-us-retailers-hit-by-targetlike-hacks-security-firm-says.html

Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by simulated annealing. *science, 220*(4598), 671-680.

Knight, W. (2017). The Dark Secret at the Heart of AI. Retrieved from https://www.technologyreview.com/s/604087/the-dark-secret-at-the-heart-of-ai/ - comments

Kobielus, J. (2014). The Ground Truth in Agile Machine Learning. Retrieved from http://www.ibmbigdatahub.com/blog/ground-truth-agile-machine-learning

Koppel, M., Schler, J., & Argamon, S. (2009). Computational methods in authorship attribution. *Journal of the American Society for Information Science and Technology, 60*(1), 9-26.

Koppel, M., Schler, J., & Argamon, S. (2013). Authorship Attribution: What's Easy and What's Hard? *Journal of Law and Policy, 21*(2).

Kostakis, O. (2014). Classy: fast clustering streams of call-graphs. *Data mining and knowledge discovery, 28*(5-6), 1554-1585.

Kostakis, O., Kinable, J., Mahmoudi, H., & Mustonen, K. (2011). *Improved call graph comparison using simulated annealing.* Paper presented at the Proceedings of the 2011 ACM Symposium on Applied Computing.

Krig, S. (2014). *Computer Vision Metrics: Survey, Taxonomy, and Analysis*: Apress.

Krishnamoorthy, N., Debray, S., & Fligg, K. (2009). *Static detection of disassembly errors.* Paper presented at the 16th Working Conference on Reverse Engineering (WCRE'09)

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

Kruegel, C., Kirda, E., Mutz, D., Robertson, W., & Vigna, G. (2006). *Polymorphic worm detection using structural information of executables.* Paper presented at the Recent Advances in Intrusion Detection.

Kruegel, C., Robertson, W., Valeur, F., & Vigna, G. (2004). *Static disassembly of obfuscated binaries.* Paper presented at the USENIX security Symposium.

Kučečka, T. (2011). *Obfuscating plagiarism detection: vulnerabilities and solutions.* Paper presented at the Proceedings of the 12th International Conference on Computer Systems and Technologies.

Kwon, T., & Su, Z. (2010). *Automatic detection of unsafe component loadings.* Paper presented at the Proceedings of the 19th international symposium on Software testing and analysis.

Lakhotia, A., Preda, M. D., & Giacobazzi, R. (2013). *Fast location of similar code fragments using semantic'juice'.* Paper presented at the Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop.

Lamb, C., Lunar, Levsen, H., Cascadian, V., & Luo, X. (2016). Provide a verifiable path from source code to binary. Retrieved from https://reproducible-builds.org/

Lamirel, J.-C., Cuxac, P., Chivukula, A. S., & Hajlaoui, K. (2015). Optimizing text classification through efficient feature selection based on quality metric. *Journal of Intelligent Information Systems, 45*(3), 379-396.

Lange, R. C., & Mancoridis, S. (2007). *Using code metric histograms and genetic algorithms to perform author identification for software forensics.* Paper presented at the Proceedings of the 9th annual conference on Genetic and evolutionary computation.

Layton, R., & Azab, A. (2014). *Authorship analysis of the Zeus botnet source code.* Paper presented at the Fifth Cybercrime and Trustworthy Computing Conference (CTC).

Layton, R., Watters, P., & Dazeley, R. (2010). *Automatically determining phishing campaigns using the USCAP methodology.* Paper presented at the eCrime Researchers Summit (eCrime), 2010.

Layton, R., Watters, P., & Dazeley, R. (2012). *Unsupervised authorship analysis of phishing webpages.* Paper presented at the International Symposium on Communications and Information Technologies (ISCIT).

Layton, R., Watters, P., & Dazeley, R. (2013). Automated unsupervised authorship analysis using evidence accumulation clustering. *Natural Language Engineering, 19*(01), 95-120.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature, 521*(7553), 436-444.

Lee, B., Kim, Y., & Kim, J. (2010). *binOb+: a framework for potent and stealthy binary obfuscation.* Paper presented at the Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security.

Lee, Y. R., Kang, B., & Im, E. G. (2013). *Function matching-based binary-level software similarity calculation.* Paper presented at the Proceedings of the 2013 Research in Adaptive and Convergent Systems.

Lehman, M. M. (1980). Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE, 68*(9), 1060-1076.

Lehman, M. M., & Ramil, J. F. (2001). Rules and tools for software evolution planning and management. *Annals of software engineering, 11*(1), 15-44.

Li, J., Zheng, R., & Chen, H. (2006). From fingerprint to writeprint. *Communications of the ACM, 49*(4), 76-82.

Liangboonprakong, C., & Sornil, O. (2013). *Classification of malware families based on N-grams sequential pattern features.* Paper presented at the 8th IEEE Conference on Industrial Electronics and Applications (ICIEA).

Lilly, P. (2012). Computer Programmer Faces 10 Years in Prison for Alleged Source Code Theft. Retrieved from http://www.maximumpc.com/article/news/computer_programmer_faces_10_years_prison_alleged_source_code_theft

Lin, Z., Zhang, X., & Xu, D. (2010). *Automatic reverse engineering of data structures from binary execution.* Paper presented at the Network and Distributed System Security Symposium, San Diego, CA.

Lindorfer, M., Di Federico, A., Maggi, F., Comparetti, P. M., & Zanero, S. (2012). *Lines of malicious code: insights into the malicious software industry.* Paper presented at the Proceedings of the 28th Annual Computer Security Applications Conference.

Linn, C., & Debray, S. (2003). *Obfuscation of executable code to improve resistance to static disassembly.* Paper presented at the Proceedings of the 10th ACM conference on Computer and communications security.

Liu, C., Chen, C., Han, J., & Yu, P. S. (2006). *GPLAG: detection of software plagiarism by program dependence graph analysis.* Paper presented at the Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining.

Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., . . . Hazelwood, K. (2005). Pin: building customized program analysis tools with dynamic instrumentation. *ACM Sigplan Notices, 40*(6), 190-200.

Luo, L., Ming, J., Wu, D., Liu, P., & Zhu, S. (2014). *Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection.* Paper presented at the Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering.

Luyckx, K., & Daelemans, W. (2005). Shallow Text Analysis and Machine Learning for Authorship Attribtion. *LOT Occasional Series, 4*, 149-160.

Luyckx, K., & Daelemans, W. (2008). *Authorship attribution and verification with many authors and limited data.* Paper presented at the Proceedings of the 22nd International Conference on Computational Linguistics-Volume 1.

MacDonell, S. G., & Gray, A. R. (2001). Software forensics applied to the task of discriminating between program authors. *Journal of Systems Research and Information Systems, 10*, 113 - 127.

Macedo, H. D., & Touili, T. (2013). Mining malware specifications through static reachability analysis *Computer Security–ESORICS 2013* (pp. 517-535): Springer.

Maitra, P., Ghosh, S., & Das, D. (2015). Authorship Verification: An Approach based on Random Forest. *Working Notes Papers of the CLEF*.

Markus F.X.J. Oberhumer, L. M., John F. Reiser. (2017a). UPX the Ultimate Packer for eXecutables. Retrieved from https://upx.github.io/

Markus F.X.J. Oberhumer, L. M., John F. Reiser. (2017b). User visible changes for UPX. Retrieved from https://upx.github.io/upx-news.txt

Martignoni, L., Christodorescu, M., & Jha, S. (2007). *Omniunpack: Fast, generic, and safe unpacking of malware.* Paper presented at the Twenty-Third Annual Computer Security Applications Conference, (ACSAC) 2007. .

Matthews, L. (2017). 2016 Saw An Insane Rise In The Number Of Ransomware Attacks. Retrieved from https://www.forbes.com/sites/leemathews/2017/02/07/2016-saw-an-insane-rise-in-the-number-of-ransomware-attacks/ - 4eec647f58dc

Mavrogiannopoulos, N., Kisserli, N., & Preneel, B. (2011). A taxonomy of self-modifying code for obfuscation. *Computers & Security, 30*(8), 679-691.

McMillan, C., Grechanik, M., & Poshyvanyk, D. (2012). *Detecting similar software applications.* Paper presented at the 34th International Conference on Software Engineering (ICSE).

Meng, X. (2016). *Fine-grained binary code authorship identification.* Paper presented at the Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering.

Meng, X., Miller, B. P., & Jun, K.-S. (2016). Identifying Multiple Authors in a Binary Program.

Meng, X., Miller, B. P., Williams, W. R., & Bernat, A. R. (2013). *Mining Software Repositories for Accurate Authorship.* Paper presented at the 29th IEEE International Conference on Software Maintenance (ICSM).

Microsoft. (2015). Visual Studio. Retrieved from http://www.visualstudio.com/

Microsoft. (2016a). Download virtual machines. Retrieved from https://dev.windows.com/en-us/microsoft-edge/tools/vms/windows/

Microsoft. (2016b). /O Options (Optimize Code). Retrieved from https://msdn.microsoft.com/en-us/library/k1ack8f1(v=vs.120).aspx

Microsoft. (2016c). SQL Server Editions. Retrieved from https://www.microsoft.com/en-us/server-cloud/products/sql-server-editions/overview.aspx

Mohaisen, A., Alrawi, O., & Larson, M. (2013). *AMAL: Highfidelity, behavior-based automated malware analysis and classification*. Retrieved from

Moseley, T., Grunwald, D., & Peri, R. (2009). *OptiScope: performance accountability for optimizing compilers.* Paper presented at the Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization.

Moser, A., Kruegel, C., & Kirda, E. (2007a). *Exploring multiple execution paths for malware analysis.* Paper presented at the IEEE Symposium on Security and Privacy.

Moser, A., Kruegel, C., & Kirda, E. (2007b). *Exploring multiple execution paths for malware analysis.* Paper presented at the IEEE Symposium on Security and Privacy (SP'07).

Murphy, K. P. (2012). *Machine learning: a probabilistic perspective*: MIT press.

Nanda, S., Li, W., Lam, L.-C., & Chiueh, T.-c. (2006). *BIRD: Binary interpretation using runtime disassembly.* Paper presented at the Proceedings of the International Symposium on Code Generation and Optimization.

Nascimento, T. M., Prado, C. B., Boccardo, D. R., Carmo, L. F., & Machado, R. C. (2012). Program Equivalence Using Neural Networks *Bio-Inspired Models of Network, Information, and Computing Systems* (pp. 637-650): Springer.

NASM development team. (2016). NASM. Retrieved from http://www.nasm.us/

O'Kane, P., Sezer, S., & McLaughlin, K. (2011). Obfuscation: The hidden malware. *Security & Privacy, IEEE, 9*(5), 41-47.

Ohmann, T., & Rahal, I. (2014). Efficient clustering-based source code plagiarism detection using PIY. *Knowledge and Information Systems*, 1-28.

Oreans Technologies. (2017). Oreans Technologies. Retrieved from http://www.oreans.com/downloads.php

Ottenstein, K. J. (1976). An algorithmic approach to the detection and prevention of plagiarism. *ACM Sigcse Bulletin, 8*(4), 30-41.

Palahan, S., Babić, D., Chaudhuri, S., & Kifer, D. (2013). *Extraction of statistically significant malware behaviors.* Paper presented at the Proceedings of the 29th Annual Computer Security Applications Conference.

Paleari, R., Martignoni, L., Fresi Roglia, G., & Bruschi, D. (2010). *N-version disassembly: differential testing of x86 disassemblers.* Paper presented at the Proceedings of the 19th international symposium on Software testing and analysis.

pancake. (2016). radare. Retrieved from http://radare.org/r/index.html

Park, H., Choi, S., Seo, S., & Han, T. (2008). *SCV: Structure and constant value based binary diffing.* Paper presented at the International Conference on Information Security and Assurance (ISA).

Park, Y., Reeves, D. S., & Stamp, M. (2013). Deriving common malware behavior through graph clustering. *Computers & Security, 39*, 419-430.

Parker, A., & Hamblen, J. (1989). Computer algorithms for plagiarism detection. *IEEE Transactions on Education., 32*(2), 94-99.

Parr, T. (2009). *Language implementation patterns: create your own domain-specific and general programming languages*: Pragmatic Bookshelf.

Payer, M., Crane, S., Larsen, P., Brunthaler, S., Wartell, R., & Franz, M. (2014). Similarity-based matching meets Malware Diversity. *arXiv preprint arXiv:1409.7760*.

Pfeffer, A., Call, C., Chamberlain, J., Kellogg, L., Ouellette, J., Patten, T., . . . Bay, J. (2012). *Malware analysis and attribution using genetic information.* Paper presented at the 7th International Conference on Malicious and Unwanted Software (MALWARE).

Pierluigi, P. (2013). 2013 – The Impact of Cybercrime. Retrieved from http://resources.infosecinstitute.com/2013-impact-cybercrime/

Pietrek, M. (1994). Peering inside the PE: a tour of the win32 (R) portable executable file format. *Microsoft Systems Journal-US Edition*, 15-38.

Popa, M. (2012). Binary Code Disassembly for Reverse Engineering. *Journal of Mobile, Embedded and Distributed Systems, 4*(4), 233-248.

Popov, I. V., Debray, S. K., & Andrews, G. R. (2007). *Binary Obfuscation Using Signals.* Paper presented at the Usenix Security.

Python Software Foundation. (2015). python. Retrieved from https://www.python.org/

Qiao, Y., Yang, Y., He, J., Tang, C., & Liu, Z. (2014). CBM: Free, Automatic Malware Analysis Framework Using API Call Sequences *Knowledge Engineering and Management* (pp. 225-236): Springer.

Qiu, D., Li, H., & Sun, J. (2013). *Measuring software similarity based on structure and property of class diagram.* Paper presented at the Sixth International Conference on Advanced Computational Intelligence (ICACI).

Qiu, H., & Osorio, F. C. (2013). *Static malware detection with Segmented Sandboxing.* Paper presented at the 8th International Conference on Malicious and Unwanted Software: The Americas (MALWARE).

Quick Unpack. (2017). GUnpacker 0.5 Download. Retrieved from http://qunpack.ahteam.org/?p=327

Quinlan, J. R. (1986). Induction of decision trees. *Machine learning, 1*(1), 81-106.

Rahimian, A., Shirani, P., Alrbaee, S., Wang, L., & Debbabi, M. (2015). BinComp: A stratified approach to compiler provenance Attribution. *Digital Investigation, 14*, S146-S155.

Rees, M. J. (1982). Automatic assessment aids for Pascal programs. *ACM Sigplan Notices, 17*(10), 33-42.

Rieck, K., Trinius, P., Willems, C., & Holz, T. (2011). Automatic analysis of malware behavior using machine learning. *Journal of Computer Security, 19*(4), 639-668.

Rodriguez, R. J. (2015). Qualitative and Quantitative Evaluation of Software Packers. Retrieved from http://webdiis.unizar.es/~ricardo/files/slides/industrial/slides_NcN-15.pdf

Rolles, R. (2009). *Unpacking virtualization obfuscators.* Paper presented at the 3rd USENIX Workshop on Offensive Technologies.(WOOT).

Rosenblum, N., Miller, B. P., & Zhu, X. (2010). *Extracting compiler provenance from program binaries.* Paper presented at the Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering.

Rosenblum, N., Miller, B. P., & Zhu, X. (2011). *Recovering the toolchain provenance of binary code.* Paper presented at the Proceedings of the 2011 International Symposium on Software Testing and Analysis.

Rosenblum, N., Zhu, X., Miller, B., & Hunt, K. (2007). *Machine learning-assisted binary code analysis.* Paper presented at the NIPS Workshop on Machine Learning in Adversarial Environments for Computer Security, Whistler, British Columbia, Canada.

Rosenblum, N., Zhu, X., & Miller, B. P. (2011). Who wrote this code? Identifying the authors of program binaries *Computer Security–ESORICS 2011* (pp. 172-189): Springer.

Rosenblum, N., Zhu, X., Miller, B. P., & Hunt, K. (2008). *Learning to Analyze Binary Computer Code.* Paper presented at the 23rd national conference on Artificial Intelligence (AAAI'08).

Roundy, K. A., & Miller, B. P. (2013). Binary-code obfuscations in prevalent packer tools. *ACM Computing Surveys (CSUR), 46*(1), 4.

Roy, C. K., Cordy, J. R., & Koschke, R. (2009). Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming, 74*(7), 470-495.

Royal, P., Halpin, M., Dagon, D., Edmonds, R., & Lee, W. (2006). *Polyunpack: Automating the hidden-code extraction of unpack-executing malware.* Paper presented at the 22nd Annual Computer Security Applications Conference. (ACSAC'06). .

Ruttenberg, B., Miles, C., Kellogg, L., Notani, V., Howard, M., LeDoux, C., . . . Pfeffer, A. (2014). Identifying Shared Software Components to Support Malware Forensics *Detection of Intrusions and Malware, and Vulnerability Assessment* (pp. 21-40): Springer.

Ryder, B. G. (1979). Constructing the call graph of a program. *IEEE Transactions on Software Engineering*(3), 216-226.

SA, H.-R. (2016). *Hex-rays Decompiler.* Retrieved from https://www.hex-rays.com/products/decompiler/index.shtml

Sæbjørnsen, A., Willcock, J., Panas, T., Quinlan, D., & Su, Z. (2009). *Detecting code clones in binary executables.* Paper presented at the Proceedings of the eighteenth international symposium on Software testing and analysis.

Sallis, P., Aakjaer, A., & MacDonell, S. (1996). *Software forensics: old methods for a new science.* Paper presented at the International Conference on Software Engineering: Education and Practice.

Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural networks, 61*, 85-117.

Schwartz, E. J., Avgerinos, T., & Brumley, D. (2010). *All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask).* Paper presented at the IEEE Symposium on Security and Privacy (SP).

Schwartz, E. J., Lee, J., Woo, M., & Brumley, D. (2013). *Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring.* Paper presented at the Proceedings of the USENIX Security Symposium.

Schwarz, B., Debray, S., & Andrews, G. (2002). *Disassembly of executable code revisited.* Paper presented at the Ninth working conference on Reverse engineering.

Sharif, M., Lanzi, A., Giffin, J., & Lee, W. (2009). *Automatic reverse engineering of malware emulators.* Paper presented at the 30th IEEE Symposium on Security and Privacy.

Sharif, M., Lanzi, A., Giffin, J. T., & Lee, W. (2008). *Impeding Malware Analysis Using Conditional Code Obfuscation.* Paper presented at the NDSS.

Sharif, M., Yegneswaran, V., Saidi, H., Porras, P., & Lee, W. (2008). Eureka: A framework for enabling static malware analysis *Computer security-ESORICS 2008* (pp. 481-500): Springer.

Shaw, V. (2016). Intellectual Property Theft Cases: Examining the Technology Industry. Retrieved from https://www.smartfile.com/blog/intellectual-property-theft-cases-examining-technology-industry/

Silicon Realms. (2017). Armadillo. Retrieved from https://www.bleepingcomputer.com/forums/t/80738/the-silicon-realms-toolworks-armadillo/

Singh, A., Walenstein, A., & Lakhotia, A. (2012). *Tracking concept drift in malware families.* Paper presented at the Proceedings of the 5th ACM workshop on Security and artificial intelligence.

Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M. G., . . . Saxena, P. (2008). BitBlaze: A new approach to computer security via binary analysis *Information Systems Security* (pp. 1-25): Springer.

Sourceforge. (2015). UPX - a powerful executable packer. Retrieved from http://sourceforge.net/projects/upx/files/upx/

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.

Stańczyk, U. (2013). *Establishing relevance of characteristic features for authorship attribution with ANN.* Paper presented at the Database and Expert Systems Applications.

Stojanović, S., Radivojević, Z., & Cvetanović, M. (2014). Approach for estimating similarity between procedures in differently compiled binaries. *Information and Software Technology*.

Sycurelab. (2016). GitHub - DECAF. Retrieved from https://github.com/sycurelab

Tanguy, L., Urieli, A., Calderone, B., Hathout, N., & Sajous, F. (2011). *A multitude of linguistically-rich features for authorship attribution.* Paper presented at the PAN Lab at CLEF.

TEMU. (2009). TEMU: The BitBlaze Dynamic Analysis Component. Retrieved from http://bitblaze.cs.berkeley.edu/temu.html

Tennyson, M. F. (2013). *A Replicated Comparative Study of Source Code Authorship Attribution.* Paper presented at the Replication in Empirical Software Engineering Research (RESER), 2013 3rd International Workshop on.

Tennyson, M. F., & Mitropoulos, F. J. (2014). A Bayesian Ensemble Classifier for Source Code Authorship Attribution *Similarity Search and Applications* (pp. 265-276): Springer.

Tennyson, M. F., & Mitropoulos, F. J. (2014). *Choosing a profile length in the SCAP method of source code authorship attribution.* Paper presented at the SOUTHEASTCON 2014, IEEE.

TheFreeDictionary.com. (2016). ground truth. Retrieved from http://www.thefreedictionary.com/ground+truth

Tian, Z., Zheng, Q., Liu, T., & Fan, M. (2013). *DKISB: Dynamic Key Instruction Sequence Birthmark for Software Plagiarism Detection.* Paper presented at the IEEE 10th International Conference on High Performance Computing and Communications.

Torri, S. A. (2009). *Generic reverse engineering architecture with compiler and compression classification components*: ProQuest.

TortoiseSVN. (2016). TortoiseSVN: the coolest interface to (Sub)version control. Retrieved from https://tortoisesvn.net/

Treude, C., Storey, M., & Salois, M. (2011). *An exploratory study of software reverse engineering in a security context.* Paper presented at the 18th Working Conference on Reverse Engineering (WCRE).

Trinius, P., Holz, T., Gobel, J., & Freiling, F. C. (2009). *Visual analysis of malware behavior using treemaps and thread graphs.* Paper presented at the 6th International Workshop on Visualization for Cyber Security (VizSec).

Trinius, P., Willems, C., Holz, T., & Rieck, K. (2009). A malware instruction set for behavior-based analysis.

Ubuntu. (2016). Download Ubuntu Desktop. Retrieved from http://www.ubuntu.com/download/desktop

Udupa, S. K., Debray, S. K., & Madou, M. (2005). *Deobfuscation: Reverse engineering obfuscated code.* Paper presented at the 12th Working Conference on Reverse Engineering.

Ugarte-Pedrero, X., Balzarotti, D., Santos, I., & Bringas, P. G. (2015). *SoK: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers.* Paper presented at the IEEE Symposium on Security and Privacy, San Jose, CA.

Ultra-Protect. (2016). UltraProtect Download. Retrieved from https://web.archive.org/web/20081216070813/http://www.ultraprotect.com/download.htm

Unger, S. H. (1964). Git—a heuristic program for testing pairs of directed line graphs for isomorphism. *Communications of the ACM, 7*(1), 26-34.

University of Waikato. (2016). Weka. Retrieved from http://www.cs.waikato.ac.nz/ml/weka/

unknown. (2016). UPolyX Download. Retrieved from http://www.leetupload.com/database/Win32/Compression - Cryptage/UPolyX v0.5.rar

Vapnik, V. N. (1999). The nature of statistical learning theory: Springer (New York).

Veeralakshmi, S., & Sindhuja, M. (2013). A Secure Environment for Unsafe Component Loading. *International Journal Of Engineering And Computer Science, 2*(4), 1111-1116.

Verco, K. L., & Wise, M. J. (1996). Software for detecting suspected plagiarism: Comparing structure and attribute-counting systems. *ACSE, 96*.

Vigna, G. (2007). Static disassembly and code analysis *Malware Detection* (pp. 19-41): Springer.

Vinciguerra, L., Wills, L., Kejriwal, N., Martino, P., & Vinciguerra, R. (2003). *An experimentation framework for evaluating disassembly and decompilation tools for C++ and Java.* Paper presented at the 20th Working Conference on Reverse Engineering (WCRE) (2003), Victoria, B.C., Canada.

Vlachos, V., Ilioudis, C., & Papanikolaou, A. (2012). On the Evolution of Malware Species *Global Security, Safety and Sustainability & e-Democracy* (pp. 54-61): Springer.

VMProtect Software. (2017). VMProtect. Retrieved from http://vmpsoft.com/

Walenstein, A., & Lakhotia, A. (2007). *The software similarity problem in malware analysis*: Internat. Begegnungs-und Forschungszentrum für Informatik.

Walenstein, A., Venable, M., Hayes, M., Thompson, C., & Lakhotia, A. (2007). *Exploiting similarity between variants to defeat malware.* Paper presented at the Proc. BlackHat DC Conf.

Wang, X., Jhi, Y.-C., Zhu, S., & Liu, P. (2009). *Detecting software theft via system call based birthmarks.* Paper presented at the Annual Computer Security Applications Conference (ACSAC).

Wang, Z., Ming, J., Jia, C., & Gao, D. (2011). Linear obfuscation to combat symbolic execution *Computer Security–ESORICS 2011* (pp. 210-226): Springer.

Wang, Z., Pierce, K., & McFarling, S. (1999). Bmat-a binary matching tool. *Feedback-Directed Optimization (FDO2)*.

Wartell, R., Zhou, Y., Hamlen, K. W., Kantarcioglu, M., & Thuraisingham, B. (2011). Differentiating code from data in x86 binaries *Machine Learning and Knowledge Discovery in Databases* (pp. 522-536): Springer.

Whale, G. (1990). Identification of program similarity in large populations. *The Computer Journal, 33*(2), 140-146.

WikiBooks. (2016). QEMU/Images. Retrieved from https://en.wikibooks.org/wiki/QEMU/Images

Wise, M. J. (1996). *YAP3: Improved detection of similarities in computer program and other texts.* Paper presented at the ACM Sigcse Bulletin.

Wisse, W., & Veenman, C. (2015). Scripting dna: Identifying the javascript programmer. *Digital Investigation, 15*, 61-71.

Witten, I. H., Frank, E., & Hall, M. A. (2011). *Data Mining: Practical Machine Learning Tools and Techniques: Practical Machine Learning Tools and Techniques*: Elsevier.

Wroblewski, G. (2002). *General Method of Program Code Obfuscation.* Institute of Engineering Cybernetics, Wroclaw University of Technology.

Wu, Y., Chiueh, T.-c., & Zhao, C. (2009). Efficient and automatic instrumentation for packed binaries *Advances in Information Security and Assurance* (pp. 307-316): Springer.

Wu, Z., Gianvecchio, S., Xie, M., & Wang, H. (2010). *Mimimorphism: A new approach to binary code obfuscation.* Paper presented at the Proceedings of the 17th ACM conference on Computer and communications security.

Xie, P. D., Li, M. J., Wang, Y. J., Su, J. S., & Lu, X. C. (2012). *Unpacking techniques and tools in malware analysis.* Paper presented at the Applied Mechanics and Materials.

Xu, M., Wu, L., Qi, S., Xu, J., Zhang, H., Ren, Y., & Zheng, N. (2013). A similarity metric method of obfuscated malware using function-call graph. *Journal of Computer Virology and Hacking Techniques, 9*(1), 35-47.

Yamaguchi, F. (2016). Joern - A Robust analysis platform for C/C++. Retrieved from http://www.mlsec.org/joern/

Yin, Z., Fu, J., Zhu, F., Su, F., Yao, H., & Liu, F. (2008). *Fingerprinting Executable Programs Based on Color Moments of a Novel Abstract Call Graph.* Paper presented at the 9th International Conference for Young Computer Scientists (ICYCS).

Yin, Z., Yu, X., & Niu, L. (2013). *Malicious Code Detection Based on Software Fingerprint.* Paper presented at the The International Conference on Artificial Intelligence and Software Engineering (ICAISE 2013).

Yoda. (2017). Yoda's Crypter 1.3. Retrieved from https://sourceforge.net/projects/yodap/files/Yoda Crypter/1.3/

Yu, S., Zhou, S., Liu, L., Yang, R., & Luo, J. (2010). *Malware variants identification based on byte frequency.* Paper presented at the Second International Conference on Networks Security Wireless Communications and Trusted Computing (NSWCTC).

Zeng, J., Fu, Y., Miller, K. A., Lin, Z., Zhang, X., & Xu, D. (2013). *Obfuscation resilient binary code reuse through trace-oriented programming.* Paper presented at the Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security.

Zhang, Q., & Reeves, D. S. (2007). *Metaaware: Identifying metamorphic malware.* Paper presented at the Twenty-Third Annual Computer Security Applications Conference (ACSAC).

Zhao, Y., & Zobel, J. (2007). *Searching with style: Authorship attribution in classic literature.* Paper presented at the Thirtieth Australasian conference on Computer science, Ballarat, Australia.

Zheng, R., Li, J., Chen, H., & Huang, Z. (2006). A framework for authorship identification of online messages: Writing‐style features and classification techniques. *Journal of the American Society for Information Science and Technology, 57*(3), 378-393.

Zou, D., Long, W.-J., & Ling, Z. (2010). *A cluster-based plagiarism detection method.* Paper presented at the Notebook Papers of CLEF 2010 LABs and Workshops, 22-23 September.

Zwanger, V., Gerhards-Padilla, E., & Meier, M. (2014). *Codescanner: Detecting (Hidden) x86/x64 code in arbitrary files.* Paper presented at the 9th International Conference on Malicious and Unwanted Software: The Americas (MALWARE).