MISSOURI S&T | CURTIS LAWS WILSON LIBRARY
MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY

Scholars' Mine

Masters Theses

Student Research & Creative Works

Summer 2014

# Evolving decision trees for the categorization of software

Jasenko Hosic

Follow this and additional works at: http://scholarsmine.mst.edu/masters_theses

Part of the Computer Sciences Commons

**Department:**

Recommended Citation

Hosic, Jasenko, "Evolving decision trees for the categorization of software" (2014). *Masters Theses*. 7308.
http://scholarsmine.mst.edu/masters_theses/7308

This Thesis - Open Access is brought to you for free and open access by Scholars' Mine. It has been accepted for inclusion in Masters Theses by an authorized administrator of Scholars' Mine. This work is protected by U. S. Copyright Law. Unauthorized use including reproduction for redistribution requires the permission of the copyright holder. For more information, please contact scholarsmine@mst.edu.

# EVOLVING DECISION TREES FOR THE CATEGORIZATION OF SOFTWARE

by

JASENKO HOSIC

A THESIS

Presented to the Faculty of the Graduate School of the

MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY

In Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE IN COMPUTER SCIENCE

2014

Approved by

Dr. Daniel Tauritz, Advisor
Dr. Samuel Mulder
Dr. Sriram Chellappan

Sandia National Laboratories

U.S. DEPARTMENT OF ENERGY

Copyright 2014

JASENKO HOSIC

All Rights Reserved

# ABSTRACT

Current manual techniques of static reverse engineering are inefficient at providing semantic program understanding. An automated method to categorize applications was developed in order to quickly determine pertinent characteristics. Prior work in this area has had some success, but a major strength of the approach detailed in this thesis is that it produces heuristics that can be reused for quick analysis of new data. The method relies on a genetic programming algorithm to evolve decision trees which can be used to categorize software. The terminals, or leaf nodes, within the trees each contain values based on selected features from one of several attributes: system calls, byte $N$-grams, opcode $N$-grams, registers, opcode collocation, cyclomatic complexity, and bonding. The evolved decision trees are reusable and achieve average accuracies above 90% when categorizing programs based on compiler origin, authorship, and versions. Developing new decision trees simply requires more labeled datasets and potentially different feature selection algorithms for other attributes, depending on the data being classified. The genetic programming algorithm used to evolve the decision trees was compared against C4.5, a classic decision tree technique. In all experiments, the genetic programming approach outperformed C4.5.

This thesis is an extension and expansion of the work published in the Computer Forensics in Software Engineering workshop at COMPSAC 2014 - the Annual 38th IEEE International Conference on Computer Software and Applications [1]. This thesis is also being prepared as a journal article to be submitted for publication.

## ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Dr. Daniel Tauritz. Not only is he responsible for delivering the passionate lectures that sparked my interests in the field of artificial intelligence, but without his dedicated involvement in every step of the process, this thesis would not exist. I would like to thank you for your understanding, patience, and guidance over the last six years.

I would also like to extend my gratidue to Dr. Samuel Mulder. He has been an instrumental and invaluable tool in helping me complete this thesis. I could not ask for a more knowledgeable mentor. His wisdom has guided this thesis to completion. Dr. Sriram Chellappan helped me greatly through his classes. His lectures aim not only to teach, but to inspire new ideas. His teaching style and enthusiasm for research left a lasting impression on me from the very first class I attended. I would like to thank Sandia National Laboratories for providing my funding through their Critical Skills Master's Program that made my graduate studies possible thus far. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energys National Nuclear Security Administration under contract DE-AC04-94AL85000.

Completing my thesis required more than professional and academic assistance. I would like to thank my brother, Jasmin Hosic, and my good friend, Ryan Birmingham. I cannot express how much I truly value their support, and often, tolerance of me as I worked to complete my thesis. Lastly, and most importantly, I wish to thank my parents, Mine and Mirsad, for raising me to appreciate and pursue education. Their love and countless sacrifices throughout my life are responsible for getting me where I am today. To them I dedicate this thesis.

# TABLE OF CONTENTS

Page

ABSTRACT . . . . . iii

ACKNOWLEDGMENTS . . . . . iv

LIST OF ILLUSTRATIONS . . . . . vii

LIST OF TABLES . . . . . ix

**SECTION**

1 INTRODUCTION . . . . . 1

2 BACKGROUND . . . . . 4

2.1 EVOLUTIONARY COMPUTATION . . . . . 4

2.1.1 EC Cycle . . . . . 4

2.1.2 Genetic Programming . . . . . 6

3 RELATED WORK . . . . . 8

4 FEATURE SELECTION . . . . . 14

4.1 LOW-LEVEL FEATURES . . . . . 15

4.1.1 Cyclomatic Complexity . . . . . 15

4.1.2 Byte N-Grams . . . . . 17

4.1.3 Bonding . . . . . 19

4.2 HIGH-LEVEL FEATURES . . . . . 20

4.2.1 System Calls . . . . . 20

4.2.2 Opcode N-Grams . . . . . 22

4.2.3 Opcode Collocation . . . . . 23
4.2.4 Registers . . . . . 25
4.3 INDIVIDUAL ATTRIBUTE PERFORMANCE . . . . . 25

5 GENETIC PROGRAMMING METHODOLOGY . . . . . 28

6 EXPERIMENTAL SETUP . . . . . 30

7 RESULTS . . . . . 32
7.1 FUNCTIONAL CLASSES EXPERIMENT . . . . . 37
7.2 C4.5 COMPARISON . . . . . 39

8 DISCUSSION . . . . . 44

9 FUTURE WORK . . . . . 48

10 CONTRIBUTIONS . . . . . 50

11 CONCLUSIONS . . . . . 51

BIBLIOGRAPHY . . . . . 52

VITA . . . . . 56

# LIST OF ILLUSTRATIONS


Figure Page

2.1 General EC cycle . . . . . 5

4.1 A) edges=9; nodes=8; P=1; C=9-8+2(1)=3 B) edges=10; nodes=8; P=1; C=10-8+2(1)=4 . . . . . 15

4.2 Individual feature tests run on the versions and compiler data sets, averaged over 30 runs. 80% of the data was dedicated to the training set, and the rest comprised the testing set. . . . . 26

4.3 Individual feature tests run on the authorship data set, averaged over 30 runs. 80% of the data was dedicated to the training set, and the rest comprised the testing set. . . . . 27

5.1 A fuzzy logic example showing that using a threshold, such as $\geq .5$ signifying true for a boolean operation, does not distinguish categories well. Both of the trees in this example would evaluate to true. Using the max/min rules, however, Tree A evaluates to .9987 while Tree B evaluates to .5141. This is used to prioritize matches. . . . . 29

7.1 Fitness progression across the evaluations for the versions data set . . 32

7.2 Fitness progression across the evaluations for the compilers data set . 33

7.3 Fitness progression across the evaluations for the authors data set . . 33

7.4 Percent match with GP decision trees – versions max = 100%, versions min = 89.47%; compilers max = 100%, compilers min = 82.69%; authors max = 100%, authors min = 66.66% . . . . . 34

7.5 Compilers testing set percent match plotted against evaluations . . . 34

7.6 Individual feature tests run on the functional classes data set, averaged over 30 runs. . . . . 38

7.7 Percent match with GP decision trees on functional classes – max = 28.57%, min = 7.14% . . . . . 39

7.8 Percent match with C4.5 decision trees – versions max = 100%, versions min = 84.21%; compilers max = 76.92%, compilers min = 53.85%; authors max = 83.33%, authors min = 33.33% . . . . . 40

7.9 Compilers - GCC, No optimization . . . . . 43

7.10 Versions - Pidgin . . . . . 43

7.11 Versions - Nestopia . . . . . 43

7.12 Versions - BaculaTrayMonitor . . . . . 43

# LIST OF TABLES

Table Page


4.1 Byte $N$-gram performance with varied $N$ values, averaged over 30 runs. Standard deviation values are shown in parentheses. . . . . 18

4.2 Opcode $N$-gram performance with varied $N$ values, averaged over 30 runs. Standard deviation values are shown in parentheses. . . . . 23

6.1 GP parameters . . . . . 31

7.1 $F$-Test: Two-Sample For Variances - Versions . . . . . 35

7.2 $F$-Test: Two-Sample For Variances - Compilers . . . . . 36

7.3 $F$-Test: Two-Sample For Variances - Authors . . . . . 36

7.4 $t$-Test: Two-Sample Assuming Unequal Variance - Versions . . . . . 36

7.5 $t$-Test: Two-Sample Assuming Equal Variance - Compilers . . . . . 37

7.6 $t$-Test: Two-Sample Assuming equal Variance - Authors . . . . . 37

7.7 $F$-Test: Two-Sample For Variances - Versions . . . . . 40

7.8 $F$-Test: Two-Sample For Variances - Compilers . . . . . 41

7.9 $F$-Test: Two-Sample For Variances - Authors . . . . . 41

7.10 $t$-Test: Two-Sample Assuming Equal Variance - Versions . . . . . 41

7.11 $t$-Test: Two-Sample Assuming Equal Variance - Compilers . . . . . 42

7.12 $t$-Test: Two-Sample Assuming Equal Variance - Authors . . . . . 42

# 1. INTRODUCTION

Software classification is the process of sorting applications into categories of similar software based on specific criteria. For instance, software can be placed into categories of 32-bit programs versus 64-bit programs, or applications that run on Windows versus applications that run on OS X. Providing a heuristic for these categorizations ranges in difficulty. The aforementioned examples are trivial, as all of the information necessary to achieve the classification lies in the application header data. Other criteria for categorizing software, however, require a thorough analysis in order to make the correct distinctions.

The problem of software classification has been previously examined, and its use for authorship [2], quality [3], and content attribution [4] is apparent. This field, however, can have far-reaching implications on new problem spaces as well. For instance, system administrators maintaining critical systems often need to quickly determine various information about new software appearances. In terms of digital forensics, recognizing more detailed semantic qualities of applications is essential. Current methods for determining vital semantic data require deep dynamic analysis or manual reverse engineering [5, 6]. While a thorough understanding of the instruction sequences of an application will most likely require some human expertise and manual analysis, the process can be assisted with some basic categorical knowledge. Is the application obvious malware? Is the program a mutation or new version of a known, preexisting program? Is the software packed or not? It is for these reasons that a means of rapidly classifying software into categories through evolved decision trees is proposed in this thesis.

A focus was placed on performing three primary experiments. First, an experiment was executed wherein software was categorized based on the compiler and

optimization flags used during development. Next, multiple versions of the same software were categorized. The third goal was to classify a set of software based on the author that wrote it. Although the versioning, authorship, and compiler identification problems have been tackled in previous research [2, 6, 7, 8], these experiments are an initial demonstration of the overall approach. The same algorithm was used to categorize software based on a variety of criteria. Furthermore, due to the absence of any dynamic analysis in the algorithm, it performs more efficiently than many previous approaches. Over 90% accuracy was achieved when matching test programs to categories in all three experiments, and the resulting decision heuristics that are derived can quickly be reused to categorize more software without requiring thorough binary analysis. A fourth experiment was also performed to classify software based on functionality, but the features used to perform this experiment were not tailor-made for that problem set, and the ultimate accuracy suffered as a result.

Many algorithms exist for developing decision trees, such as Chi-squared Automatic Interaction Detection (CHAID) [9, 10], Iterative Dichotomiser 3 (ID3) [11], and its successor, C4.5 [12]. However, due to some known performance issues with ID3 and a lower success rate on prediction problems when using CHAID [13, 14], a different technique was employed. C4.5 does suffer from many of these drawbacks, and it was therefore used as a basis for comparison in these experiments.

An approach utilizing Genetic Programming (GP) [15] was used to generate the decision trees. A GP algorithm is a population-based evolutionary algorithm that uses natural selection to reach a solution. GP solutions are often encoded with a tree structure. This structure is ideal as the tree representation translates directly into the decision trees that are employed to categorize applications. Although this is a reinforcement learning technique, a training set was still applied to assist the learning process. GP algorithms usually have a long runtime due to the many evolutionary cycles they must perform before converging to a satisfactory solution, and while this is

true of the GP employed in this decision tree approach, applying the evolved decision trees on new software takes only seconds. The performance of the resultant decision trees was compared to the matching accuracy of C4.5 using the same feature sets. In every case, the GP method outperformed C4.5.

The remainder of this thesis is structured in the following way: Section 3 contains related work in the areas of software classification, decision trees, and heuristic development is presented, followed by a description of the features used and the methodology of the GP approach in Section 4 and Section 5, respectively. Section 6 contains detailed explanations of the experiments performed, and Section 7 shows the results. The thesis concludes with a discussion of the results in Section 8 and the possible applications of this categorization technique in Section 9.

# 2. BACKGROUND

## 2.1. EVOLUTIONARY COMPUTATION

Evolutionary Computation (EC) is a subfield of computational intelligence. There are a large number of computational intelligence algorithms that address very different problem classes. Supervised learning algorithms use known, labeled data to train a model in order to make predictions on unlabeled data. Unsupervised learning algorithms use no training data and have no knowledge of the problem domain, but make predictions based on a pattern of incoming data. Reinforcement algorithms are an intermediate option wherein training data is not required, but a knowledge of the problem domain allows the algorithm to evaluate the quality of attempted solutions [16].

EC is a population-based, iterative reinforcement learning technique inspired by neo-darwinian evolution theory and Mendelian genetics, often used in optimization problems [17]. The use of a population offers a parallel approach to finding a global solution in the search space. Searching for a solution with a reinforcement algorithm can be simplified into two basic components: exploration and exploitation. Exploration means searching unexplored areas of the search space in order to gain new knowledge. Exploitation refers to the use of known information to produce more optimal solutions [18]. EC performs performs a mix of both exploration and exploitation to arrive at a solution [16].

**2.1.1. EC Cycle.** There are many kinds of evolutionary algorithms, but they all follow a cyclic process. The general cycle of EC is illustrated in Figure 2.1, and a general overview of the steps follows.

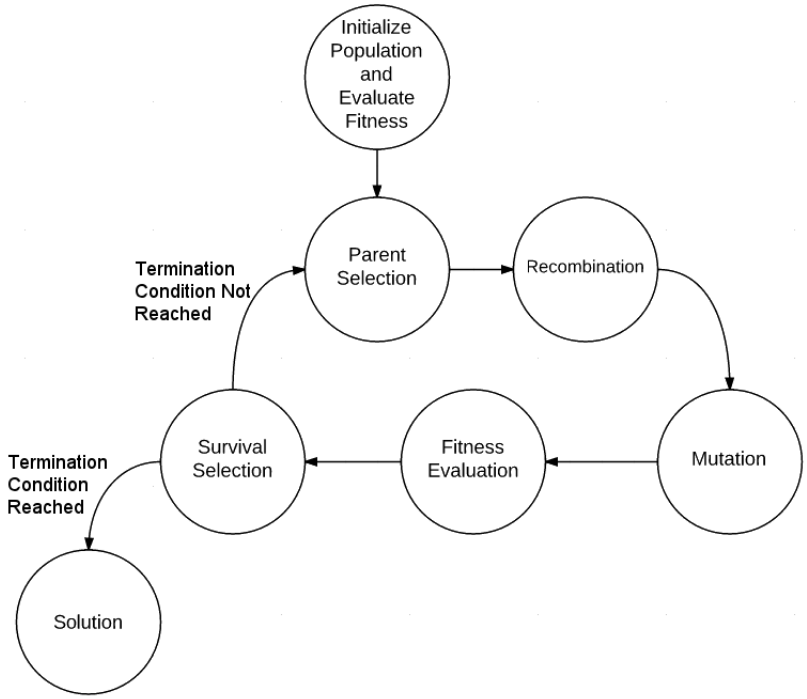


Figure 2.1: General EC cycle

**Initialization.** During this phase of the algorithm, a population of size $\mu$ is generated as a starting point for the evolutionary process. The starting population is typically a set of initial, explorative guesses at the solution.

**Fitness Evaluation.** The fitness evaluation determines a numerical value calculated to represent the quality of the solution as it pertains to the ultimate goal of the evolution. The fitness evaluation varies from problem to problem. It makes up the reinforcement phase of the learning process.

**Parent Selection.** Parent selection is the algorithm that pairs up the two individuals that will mate to produce new solutions. An example of a parent selection is $k$-tournament, where $k$ randomly-selected individuals are compared and the winner is selected. Two tournament winners are paired together to recombine.

**Recombination.** Recombination refers to the way in which the population mates to produce new solutions, or offspring. Existing solutions are typically paired

up with a parent selection algorithm, broken into fragments, and recombined to produce $\lambda$ offspring. The offspring contain components from both parent solutions. The possible recombination methods depend on the representation used for the individuals. Recombination exploits previously-explored portions of the search space to reach better solutions.

**Mutation.** Mutation is a way of manipulating existing solutions to introduce new possibilities into the gene pool. For instance, if the individuals are represented as binary strings, a viable mutation option is to randomly flip a bit. Even if both parents of a solution do not contain a certain component of a solution, mutation offers the chance to explore that option. Mutation, unlike recombination, is more of an exploration phase. It is an essential component in EC as it allows the algorithm to escape local optima in order to reach a global optimum.

**Survival Selection.** This step of the cycle is a major component of the "natural selection", as an algorithm is used to determine which solutions in the population should remain in order to reproduce and which should be discarded. Survival selection can be performed a number of ways, each with varying effects on the overall selective pressure on the system. An example of a survival selection algorithm is truncation, where the top $\mu$ solutions remain to make up the solution in the next cycle and the rest are discarded.

**Termination.** The termination represents the conditions under which the cycle is broken and the solutions are accepted as final. Termination could be determined with a multitude of criteria, such as reaching $n$ number of fitness evaluations, reaching a maximum fitness value, or running for a certain amount of time.

**2.1.2. Genetic Programming.** GP is a population-based meta-heuristic EC technique [15]. In classic GP, each solution, encoded in an individual in the population, is represented with a tree wherein the leaf nodes take the form of some terminal value while the internal nodes perform functionality that relates the terminals.

The trees can represent many things, such as mathematical formulas, complex programming sequences, or, as shown later in this paper, decision making processes [19].

GP with a tree representation has its own set of recombination and mutation operators. Single-point crossover, for example, is a recombination method that selects a random point on each parent's tree and swaps the sub-trees beneath that point. Sub-tree mutation is a classic mutation operation that stochastically generates a sub-tree and places it at a random location in an individual's tree [15].

This tree structure, if kept unchecked, can cause solutions to grow very quickly. A maximum tree depth provides a hard cap for solutions, but if set too small, can limit the quality of the solutions that evolve from the algorithm. Parsimony pressure [20] can also be applied, which punishes solutions that bloat too much in order to favor smaller, more elegant solutions.

# 3. RELATED WORK

Significant contributions have been made to the fields of GP as it relates to developing new heuristics [21, 22, 23, 24], software classification [3, 19], version matching [6, 7, 25], author identification [2], and compiler attribution [8]. The research presented in this thesis takes inspiration from those ideas with distinct differences for the purposes of providing fast semantic categorization. Methods used in former papers provide solid results but either tend to focus on solving one kind of problem or require long running times. As a result, a hybrid of some of these concepts is proposed with the design of the GP-evolved decision trees.

The results of the GP are compared against the results of C4.5. C4.5 performs well in many domains, and it does not suffer from some of the drawbacks of its predecessors. J. Sun et al. [13] discuss several decision tree methods and cite numerous potential weaknesses. They note that the Iterative Dichotomiser 3 (ID3) algorithm, a precursor to C4.5, tends to overfit in noisy domains, inappropriately grow the size of decision trees, and is particularly impaired when handling continuous, numeric values. The attributes used in the decision trees are extracted from a highly noisy domain, and therefore this algorithm does not seem suitable for the same sort of problem. In [26], J. R. Quinlan addresses some of the shortcommings in ID3 and earlier releases of his C4.5 algorithm. A solution is offered in release 8 of the algorithm, wherein the Minimum Description Length (MDL) principle is used to remedy the issues. The MDL principle is based on a sender and receiver scenario. The sender encodes a possible classification solution, called a theory, into bits and sends the data to the receiver. Small theories are likely inaccurate, but require the transmission of fewer bits. More accurate and complex theories need more bits. The improved C4.5 algorithm applies this MDL concept in order to maximize accuracy while attempting

to minimize theory bit length. The effects of this approach improve the performance of the algorithm significantly when tested against older releases. This idea, however, is captured within GP as well. Selecting appropriate parsimony pressure can produce the same results as MDL, as smaller, less complex solutions are favored if accuracy remains acceptable [20]. C4.5 is the ideal decision tree algorithm to use as a basis for comparison.

Other decision tree algorithms exist that perform well in noisy domains, but are not as appropriately suited for the problems outlined in this thesis. CHAID, for instance, is an algorithm for developing a set of induction rules which can be used for classification [9]. CHAID makes use of a unique branching technique. A statistical test is used to group attributes within the data that are very similar. This allows the algorithm to determine how many branches are required. The trees produced allow for non-binary classifiers with one node branching off into many others. Due to the use of the Chi-squared test during the execution of the algorithm, noisy data is collapsed and post-order pruning is not required, such as in CART [10]. In [14], P. Lewicki et al. discuss the situations that best suit each decision tree algorithm. They suggest that CHAID is best in accomplishing analysis, particularly in marketing scenarios, while C4.5 is better at making predictions. All of the categorization experiments require accurate predictions, therefore C4.5 provides the most apt comparison for the evolved GP heuristics.

Using GP to generate heuristics is not a new concept. It is often chosen because, in scenarios such as software categorization, determining the appropariate formulas by which programs can be compared and classified is not inherently obvious. Since the optimal metrics could be far too complex and difficult to construct by hand, GP can be used to evolve a heuristic without the need for hand-tuned trial and error. According to [21, 22], the tree structure of a GP and its ability to mutate in order to escape local optima are ideal for evolving heuristics where little is known about

the possible final result. With appropriate selective pressure, the algorithm is able to explore a wide variety of options before converging to a final solution.

A survey by P. G. Espejo et al. [19] shows that there are multiple ways in which GP can be used in the classification field. The flexibility of GP representations allows for many kinds of models, which has led to numerous applications. Conditional operators, as well as other comparison operators, can easily be implemented in tree representations to generate classifiers. If the need for multiple classifiers to be used in conjunction with each other exists, ensemble classifiers can be evolved with GP. In instances where achieving conflicting goals is desired, multi-objective GP classifiers have been developed. No matter the type of classifier that is required, the quality must be measured through the use of a fitness function. There are many ways to model a problem using GP, and as such, there are many ways to assess a solution's fitness. Common fitness function metrics for classification accuracy are precision, confidence, sensitivity, and specificity.

While GP is a solid method for building a classifier, other techniques have certainly been explored. K. Gao et al. [3] developed a classification system that uses several different feature selection algorithms, such as automatic hybrid search (AHS), with a combination of five different classifiers, including instance-based learning ($k$-nearest neighbor), multilayer perceptrons, support vector machines, naïve bayes, and logistic regression. They derived data by applying feature selection algorithms to numerous attributes. Metrics such as the number of loop constructs, the number of lines of code, and the span of variables have been considered. The results found with this method suggest two things. First, it is essential to remove metrics that are irrelevant and unnecessary for accomplishing the end goal so as to avoid over-saturating the possible terminals during evolution. Second, while the success of these techniques on a difficult problem like software quality classification is impressive,

there is a dependence on having access to the source code. Without the source code, certain metrics used in this work are difficult to extrapolate.

Software classification clearly has its merits, but it is not the only way to potentially solve the compiler and versions problems. V. Nagarajan et al. [7] attempt to tackle the version comparison problem, but take a drastically different approach than the categorization schemes considered in other papers. They specifically aim to detect version differences by using a similarity metric to match the call graphs and control flow of two programs. Their research attempts to identify situations in which two programs are functionally equivalent, but one is written in a more obfuscated manner. The applications are dynamically analyzed and every executed instruction is stored in tables of execution histories. An application is broken down into intraprocedural calls, interprocedural calls, and singular instructions. Due to the sporadic nature of multiple calls in potentially obfuscated code, V. Nagarajan et al. dynamically construct call graphs and perform flattening techniques so that the obfuscated code can accurately be matched to the execution history traces of another program. Although it performs well in many instances, some false-positives are found with this technique. In order to reduce the error, whenever one program's calls or instructions match multiple sections of another program, a confidence measure is applied to each match. These confidence measures are calculated using the reachability of each node, found via the control flow structure, and prioritized in order to produce a successful error correction mechanism. The accuracy of this technique when applied to applications that have undergone common obfuscation techniques is high. When comparing many applications at once, the number of control flow comparisons that would be required to achieve this success rate ($n^2$) could become unmanageable. The work proposed in this thesis attempts to provide a more efficient method, requiring fewer comparisons and no dynamic analysis to produce accurate results when attempting to identify multiple versions of the same application.

Software can be altered to produce new versions of the same or similar functionality in a multitude of ways. The work of D. Bruschi et al. [6] attempts to find similarities in multiple malware applications wherein any variations occurred due to self-mutation. Their approach also involves dynamically tracing the execution of the malware. Identical instructions between malware instances are paired, and any deviations are transformed until a match is found so as to account for programming variations or optimization differences [6]. While this idea shows promise, the running times are long due to the need to dynamically analyze all calls and instructions. The amount of time it would take to consider all essential flow paths and execute all the necessary code transformations could be staggering in certain cases where application sizes are large. The techniques presented are invaluable in scenarios where analyzing malware is commonplace and detailed analysis is prioritized over quick semantic attribution. Only self-mutating malware is considered, and as a result, the solution does not seem ideal for categorizing large numbers of purely benign applications. As a contrast to this work, the decision trees evolved with the GP algorithm primarily consider mundane, non-malicious software in order to identify version differences.

Compiler attribution has been examined in much the same way as version classification. N. E. Rosenblum et al. [8] have performed deep analysis of program binaries to determine compiler origin, even if multiple compilers were used (such as when statically linked library code is included). Their work analyzes idioms, or simply put, opcode trigrams along with their respective operands. These sequences of three instructions at function entry points allow for pattern recognition that hints at compiler origin [5]. Furthermore, gaps between functions as well as intraprocedural branches are also used to model compiler behavior. Whether multiple compilers were used or not, the compiler matching accuracy reaches as high as 90%.

Their work extends beyond just compiler provenance. One of the biggest inspirations for this thesis comes from the ideas presented in [2]. N. E. Rosenblum et

al. use various features, such as $N$-grams, idioms, graphlets (several basic blocks retrieved from control flow for pattern matching), and super graphlets (graphlets spanning larger distances with some collapsed control flow) to determine code authorship from compiler binaries. Using a statistical feature selection technique, essential features are extracted which aid in the categorization of software based on programmer. $N$-grams and idioms assist in detecting patterns that comprise an author's signature, allowing for high accuracy during classification. This thesis attempts to utilize some similar concepts in order to produce quick, reliable results for the version and compiler categorization problems as well as the authorship problem.

# 4. FEATURE SELECTION

In order to accurately develop a system of distinguishing and categorizing software, a heuristic for the decision making process of what software belongs in which group must be developed. However, many different criteria exist with which applications can be measured and grouped. For instance, programs that are simply version revisions of each other should intuitively have similar functionality while programs that are entirely different most likely contain very different instructions or code. There are a vast number of categories which software can be placed into, and a massive set of potential features that can resolve acceptance or denial into a particular group. Even when appropriate attributes are chosen, such as using byte $N$-grams to differentiate the programs, proper feature selection must be performed to obtain solid results [3]. For this reason, several different attributes to were considered, each with their own feature selection process. The attributes were chosen for their potential as distinguishing factors for the versions, authorship, and compiler problems. A data set for the version problem, composed of nineteen different applications with four to six versions each, totalling 90 programs, was used during experimentation. For the compilers problem, 61 programs compiled with GCC and Visual Studio, each with two different optimization levels for a total of 244 programs, were used. We ran the authorship experiment on a set of programs that were created by three different programmers. Each programmer independently solved the first ten challenges from Project Euler * in the same language and compiled them in the same manner. The following is an explanation of each attribute and its feature selection scheme.

*Colin Hughes. Project euler. https://projecteuler.net/, 2014.

## 4.1. LOW-LEVEL FEATURES

**4.1.1. Cyclomatic Complexity.** Cyclomatic complexity is a software metric that measures the logical complexity of an application. It was selected as a potentially useful feature as it was expected to aid in both the versions and compiler experiments. Programs that are merely different versions of each other likely have the same complexity if their functionality did not change much. Likewise, there is a possibility that complexity could be a factor in distinguishing compiler optimization. Higher amounts of optimization could decrease a program's complexity, which would allow for some insights into compiler origin.

Cyclomatic complexity, in this case, is defined by the following equation:

$$C = edges - nodes + 2P \tag{4.1}$$

where $C$ is the complexity, *edges* is the number of edges in the control flow graph, *nodes* is the number of nodes in the control flow graph, and $P$ is the number of exit nodes in the control flow graph. Figure 4.1 illustrates this equation in practice.

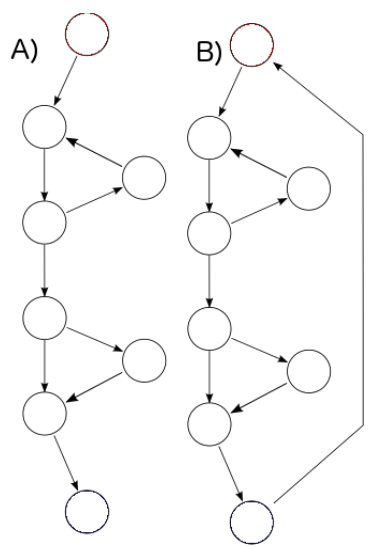


Figure 4.1: A) edges=9; nodes=8; P=1; C=9-8+2(1)=3 B) edges=10; nodes=8; P=1; C=10-8+2(1)=4

It is slightly higher-level metric than examining pure byte data, but it is based entirely on the control flow graph of an application. It ignores all of the truly high-level instruction information from disassembly. In the experiments, this equation is applied to an application on a per-function basis. The results of each function's computed cyclomatic complexity are then averaged to produce the cyclomatic complexity for an entire program. The mean of all the cyclomatic complexity averages is calculated to produce a category's complexity baseline. Test programs are then matched to categories based on smallest difference between a category's complexity and their own complexity. Algorithm 1 shows the psuedocode for the feature extraction and matching algorithms.

**Algorithm 1** Cyclomatic complexity feature extraction algorithm

```
function getComplexity(program)
    CTotal ← 0
    for all functions ∈ program do
        C ← edges − nodes + 2P
        CTotal ← CTotal + C
    CTotal ← CTotal/count(functions)
return CTotal

function getCategoryComplexity(programs)
    finalC ← 0
    for all program ∈ programs do
        finalC ← finalC + getComplexity(program)
    finalC ← finalC/count(programs)
return finalC

function matchPrograms(testPrograms, categories)
    for all testProgram ∈ testPrograms do
        minDiff ← ∞
        for all category ∈ categories do
            C ← getComplexity(testProgram)
            categoryComplexity ← getCategoryComplexity(category)
            diff ←| categoryComplexity − C |
            if diff ≤ minDiff then
                minDiff ← diff
                testProgramCategory ← category
```

**4.1.2. Byte N-Grams.** $N$-Grams are a method of partitioning data into $N$-sized chunks. These partitions can be used in many contexts, such as in textual data for classifying written documents [27, 28]. The information captured within an $N$-gram could provide insight into the contents of a document. For instance, the 3-grams, or trigrams, containing "win" and "ine" would very useful in identifying articles about fine wines. On the other hand, the trigram "the" is likely to show up in all English language articles. Determining which $N$-grams are useful and which are noise requires proper feature selection.

$N$-Grams can be used to partition software by bytes as well. Rosenblum et al. [2] had great success in using $N$-grams to classify software by author. The numerous chunks of byte data key in on essential programming patterns which allow for the identification of the author. This same concept can applied to idenitifying other information about the software. In identifying compiler origin, the $N$-grams could reveal essential header data or the compiler's allocation of registers. The case for using $N$-grams is most compelling when attempting to identify mutliple versions of the same software. It is expected for the applications to be very similar in terms of header data, functionality, and size. There are likely only marginal changes to the instructions. As such, the $N$-gram spread of two applications should be quite similar if they are merely different versions of each other.

The byte $N$-grams attribute, for the purposes of all experiments, contained only trigrams. Initial experiments were performed with bigrams and quadgrams as well, but their results were either virtually identical or significantly worse than the accuracies achieved with trigrams. Table 4.1 shows the results of the $N$-grams feature selection with $N$ values of two, three, and four. The results were obtained from using 80% of each data set as the training set and 20% as the testing set.

Table 4.1: Byte $N$-gram performance with varied $N$ values, averaged over 30 runs. Standard deviation values are shown in parentheses.

| $N$ | **Versions** | **Compilers** | **Authors** |
|---|---|---|---|
| 2 | 95.26% (1.61%) | 59.55% (9.25%) | 72.22% (13.37%) |
| 3 | 95.09% (1.34%) | 92.69% (3.48%) | 81.41% (19.04%) |
| 4 | 94.21% (1.61%) | 92.88% (4.20%) | 81.11% (17.90%) |

Using different values of $N$ can have an impact on more than just accuracy. As $N$ increases, so do the number of possible $N$-grams that can exist, increasing both the processing time and storage resources required to compute the feature selection algorithm. There are 256 possible values for a byte, therefore the number of possible $N$-grams with $N$-byte sized fragments is $256^N$. As this is an exponential relationship, smaller values of $N$ are favored. The difference in cost of using trigrams versus bigrams is marginal in both processing time and resources, however, as each is a one-time cost per program, which is miniscule in comparison to the disassembly costs of higher-level features. As a result, trigrams were chosen for their superior performance.

The feature selection was performed by first placing trigrams of the programs in each category into histograms. The intersection of all of the histograms within a category comprised the feature set. It is not necessarily the most common $N$-grams that carry the most weight in correctly categorizing applications, but instead, a particular $N$-gram may carry immense weight with only a few appearances. By intersecting not only on the $N$-grams that all programs in a category have in common, but also on the frequency with which they appear, some of those scenarios are captured. With this scheme, a particular trigram appearing the same number of times in each of the programs of the category is added to the feature set. Once a set of $N$-grams is selected for a category, testing programs are introduced. Testing programs are matched to the category with which they have the highest number of histogram values in common. A psuedocode representation is shown in Algorithm 2.

**Algorithm 2** Byte $N$-grams feature extraction algorithm

```
function getCategoryNGrams(programs)
    for all program ∈ programs do
        nGrams ← getNGramHistogram(program)
    for all program ∈ programs do
        for all nGram ∈ nGrams do
            if nGram ∈ otherProgs then
                if frequency(nGram) = frequency(otherProgs[nGram]) then
                    categoryNGrams ← categoryNGrams + nGram
    return categoryNGrams

function matchPrograms(testPrograms, categories)
    for all testProgram ∈ testPrograms do
        testNGrams ← getNGramHistogram(testProgram)
        for all category ∈ categories do
            percentMatch[category] ← 0
            categoryNGrams ← getCategoryNGrams(category)
            for all nGram ∈ categoryNGrams do
                if nGram ∈ testNGrams then
                    if frequency(nGram) = frequency(testNGram) then
                        percentMatch[category] ← percentMatch[category] + 1
            numNGrams ← count(categoryNGrams)
            matchPercent[cateogry] ← matchPercent[category]/numNGrams
        testProgramCategory ← category(max(matchPercent))
```

**4.1.3. Bonding.** Bonding is a concept presented in [29]. This graph metric has been useful in distinguishing many graph types, such as social graphs, and it was included in these experiments in an attempt to investigate its applicability in distinguishing control flow graphs. Bonding is calculated with a formula that takes the following form:

$$B = \frac{6 \cdot \#triangles}{\#length_two_paths} \tag{4.2}$$

Bonding refers to a ratio of triangles within the control flow to the number of potential triangles (2-paths). It takes a maximum value of one if a graph is complete, and zero if a graph contains no triangular subgraphs. In social graphs, triangular subgraphs identify situations where two people know a third person and also know

each other. A high bonding ratio in a social graph suggests that this occurs more frequently than situations where two people know a third but not each other. In a control flow graph, however, these subgraphs represent something very different. A triangular structure can represent many things, such as a single occurrence of a conditional sequence, a loop structure, or grouping of function calls.

The bonding was expected to be similar in applications that are version variations of each other. It was hypothesized that the control flow structure of an application would not change drastically if a program is simply a new version of existing software. In the case of compiler identification, the optimization level can have a large impact on the structure of a program's control flow. Optimization for size, compile time, and efficiency produce different control flow graphs, which in turn affects the bonding value. The third experiment, author identification, had the potential to benefit from this metric as well. The way in which a person programs may affect the number of triangular subgraphs that result after compilation. It is also possible that the minor differences in coding habits do not have a large enough affect on the overall subgraph structure of the whole application. This feature was included in order to investigate these ideas.

As with complexity, the bonding values are calculated on a per-function basis from only the control flow graph data. The function bonding values are averaged to find the program values, and a category average is again evaluated for each category. Test programs are matched to categories based on the smallest difference between their bonding values and category averages.

## 4.2. HIGH-LEVEL FEATURES

**4.2.1. System Calls.** Extracting system calls from an application requires a higher-level look at the content of the software. System calls can take many different

forms. Allocating memory, exception handling, network connections, and registry manipulation are all examples of procedures that can be performed by making the right system calls. Examining the different calls a function makes elucidates a large portion of its functionality. As with $N$-grams, noise exists in this domain as well. An application's primary purpose may be to establish a network connection, which could require a handful of different system calls. These system calls, while vital to the functionality, may appear only once. Other calls, however, such as exit procedures and exception handling, could be called numerous times for various conditions.

System calls were identified as possible discriminating features because different versions of the same program are likely to make the same types of calls due to their related functionality. They are likely less useful in distinguishing compilers and authors. Compiling the same application with a different compiler or optimization level would not have any effect on the system calls the programmer chose to make. In regard to authorship, a programmer may have a greater likelihood to make certain system calls that another programmer chooses not to use, but these distinctions are marginal at best.

When determining the proper way to select features with system calls, a balance needed to be achieved between the focus placed on the number of times each call was made within a category or program, versus the impact each call would have as a distinguishing factor. In order to create this balance, histograms of the system calls within each application were first created. The histograms were treated as vectors wherein the system calls denoted dimensions and the quantities determined the magnitudes of the vector. The vectors were then reduced to direction vectors in order to negate the impact of program size. An average direction vector was calculated for each category, and the resultant vector's direction was then compared to the direcion vector of test programs. The difference in direction from test program vectors and

category vectors was used to match programs to categories. Algorithm 3 shows a pseudocode implementation.

**Algorithm 3** System Calls feature extraction algorithm

```
function makeDirectionVector(systemCalls)
    sumFreqs ← 0
    for all systemCall ∈ systemCalls do
        sumFreqs ← sumFreqs + FREQUENCY(systemCall)
    for all systemCall ∈ systemCalls do
        directionVector[systemCall] ← FREQUENCY(systemCall)/sumFreqs
    return directionVector

function getCategorySystemCalls(programs)
    for all program ∈ programs do
        directionVector[program] ← MAKEDIRECTIONVECTOR(callHistogram)
    return AVERAGEVECTORS(directionVector, programs)

function matchPrograms(testPrograms, categories)
    for all testProgram ∈ testPrograms do
        testCalls ← MAKEDIRECTIONVECTOR(testProgram)
        for all category ∈ categories do
            sysCalls[category] ← GETCATEGORYSYSTEMCALLS(category)
            diff[category] ← DIRECTIONDIFF(sysCalls[category], testCalls)
        testProgramCategory ← category(min(diff))
```

**4.2.2. Opcode N-Grams.** Our opcode $N$-gram attribute is a higher-level version of the byte $N$-grams feature. Opcode $N$-grams are gathered by producing $N$-grams from the instruction opcodes (excluding operands) of a disassembled application. This type of attribute is very similar to the idioms proposed in [2], and R. K. Shahzad et al. [30] have had great success in detecting adware through the use of opcode $N$-grams. By using a higher-level data repository for the $N$-grams, the feature extraction is able to key in on more semantic abstractions of an application's content.

The versions experiment can benefit greatly from using opcode $N$-grams, as a majority of the functionality and instructions should remain unchanged. Compiler optimization can influence opcode use as certain instruction sequences can be reduced

to common equivalent instructions. While this is helpful for the compiler identification experiment, it does muddle the benefits to the authorship experiment. Even though a programmer determines the functionality of the program, the compiler's optimizations can influence groups of opcodes more.

Again, as with the byte-based attribute, trigrams were used exclusively in all experiments. Using bigrams or quadgrams did not significantly improve the results. Table 4.2 shows the comparisons of each $N$ value. The histogram intersection technique used with byte $N$-grams, as presumed, did not produce viable results in inital tests. Instead, the opcode $N$-grams for a category of application were placed into a unified histogram. The histogram represents the frequency of each opcode $N$-gram as it occurs among all programs in a category. The top 50% of the $N$-grams are used as essential features for that category. Test programs are matched to categories that have the most opcode $N$-grams in common. The percentage threshold value of 50% was determined through experimentation as it produced the most optimal results in the data sets.

Table 4.2: Opcode $N$-gram performance with varied $N$ values, averaged over 30 runs. Standard deviation values are shown in parentheses.

| $N$ | **Versions** | **Compilers** | **Authors** |
|---|---|---|---|
| 2 | 33.68% (3.27%) | 59.55% (8.23%) | 33.33% (4.37%) |
| 3 | 58.95% (4.23%) | 71.09% (4.96%) | 35.90% (9.06%) |
| 4 | 59.29% (4.35%) | 71.67% (5.02%) | 36.11% (18.61%) |

**4.2.3. Opcode Collocation.** Although a viable number of feature selection algorithms had been created to assist in the compiler and versions classifications, few of the aforementioned algorithms were sufficient in distinguishing authorship. It is difficult to capture programming style in low-level features. In order to accurately capture some semantic characteristics about the manner in which a programmer writes code, high-level features that highlight the relationship between instruction

sequences are required in some form. In this feature selection algorithm, opcode collocations were used to extract information about programming style.

Collocations show the relationships of objects, such as words, and express how they are conventionally grouped. For instance, in linguistics, the words "tall" and "tree" are used together more often than the words "high" and "tree". These words are therefore strongly connected. Determining these strong connections can be done via a mutual information calculation [31]. The formula for mutual information compares the probability of two objects occurring together if they are independent and the probability of their actual occurrence together. Equation 4.3 shows the mutual information formula.

$$I = P(XY) \cdot \log \frac{P(XY)}{P(X)P(Y)} \tag{4.3}$$

$I$ is the mutual information value and $X$ and $Y$ are the two objects being compared. This concept was applied to the opcodes of each application to potentially extract information about programming style from the authorship data set. The authorship experiment considered only applications that were compiled in the exact same way to negate the influence of compiler bias on the opcode occurrence. The way in which a programmer codes an application could cause a stronger collocation of certain opcode combinations in much the same way that a writer has a greater prepensity for putting together certain word combinations.

Such as with the system calls feature, the collocation feature selection technique considered the set of collocations to be a vector with each opcode pair signifying a dimension. An average direction vector was calculated for a category, and the difference in direction from a test program's direction vector determined the ultimate value of this feature.

**4.2.4. Registers.** Register allocation is the process of assigning a large number of variables to a limited number of available processor registers. Registers can then be accessed to retrieve allocated data. The occurrence of register use, whether for loading or saving, was the focus of this feature selection algorithm.

The compiler choice has the greatest influence in deciding which registers are used more frequently. When using the same compiler and optimization flags, however, the way in which a programmer chooses to design an application could lead to a greater allocation or reference of variables, which in turn, affects the register allocation. The register use frequency was extracted statically from a linear disassembly of each file. In the experiment, the register use was expressed with a histogram, which was again used in a vector direction calculation. The individual registers symbolized dimensions while their frequency of use in an application determined the magnitude. An average category direction vector was calculated, and the difference in direction of test program vectors was used to categorize the data set.

## 4.3. INDIVIDUAL ATTRIBUTE PERFORMANCE

Before implementing all of the features as primitives in a GP algorithm, each attribute's performance in categorizing the data sets was first examined. In each experiment, 80% of the data was used as the training set for determining the pertinent features and 20% was used as the testing set. Each feature was individually tested 30 times over both datasets with randomized training and testing groups for each run. The accuracy of each feature was independently gathered, and the results confirmed many of the assumptions about which experiments would benefit from each feature selection method. The averages of the results for the versions and compiler experiments are shown in Figure 4.2.

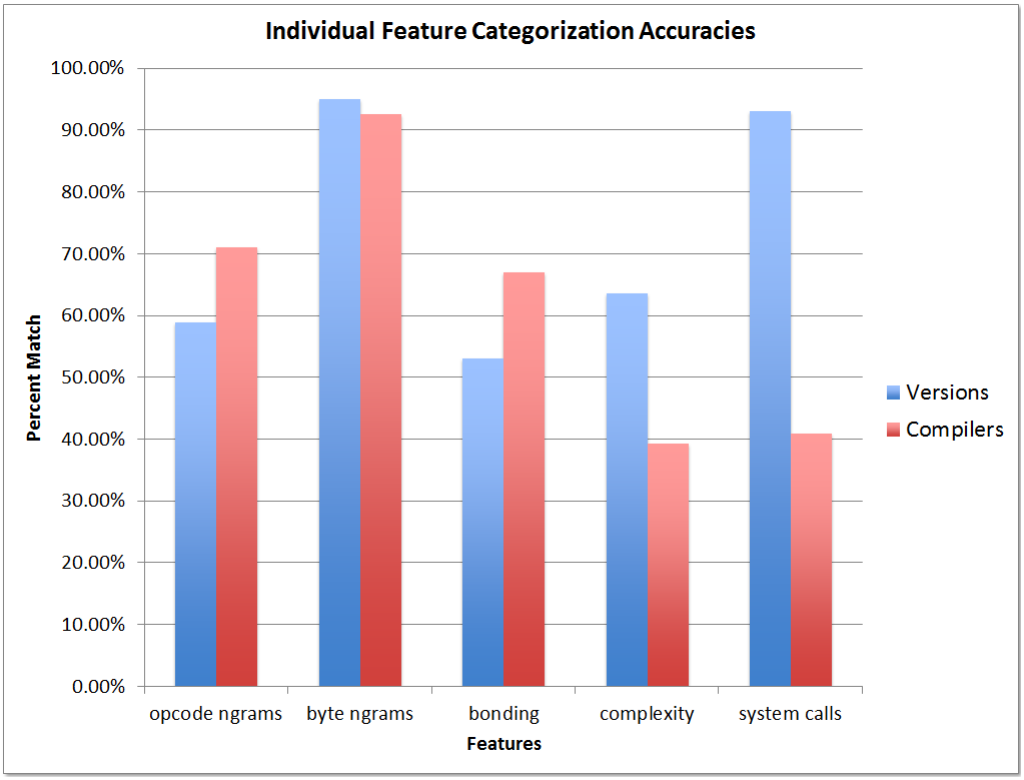


Figure 4.2: Individual feature tests run on the versions and compiler data sets, averaged over 30 runs. 80% of the data was dedicated to the training set, and the rest comprised the testing set.

It is evident that while both $N$-gram features produced performed very well, they were not perfect for all cases. The other features showed signs of promise in certain setups.

Initial experiments suggested that this same combination of features would not be sufficient in distinguishing the authors data set. As a result, two additional features, register vectors and opcode collocation, were introduced to assist in categorizing the data. It was clear that the additional high-level features were needed to represent more of the programming styles in the authorship data set.

Figure 4.3 shows the accuracy of all features when individually applied to the authorship data set. The results are averaged across 30 runs.

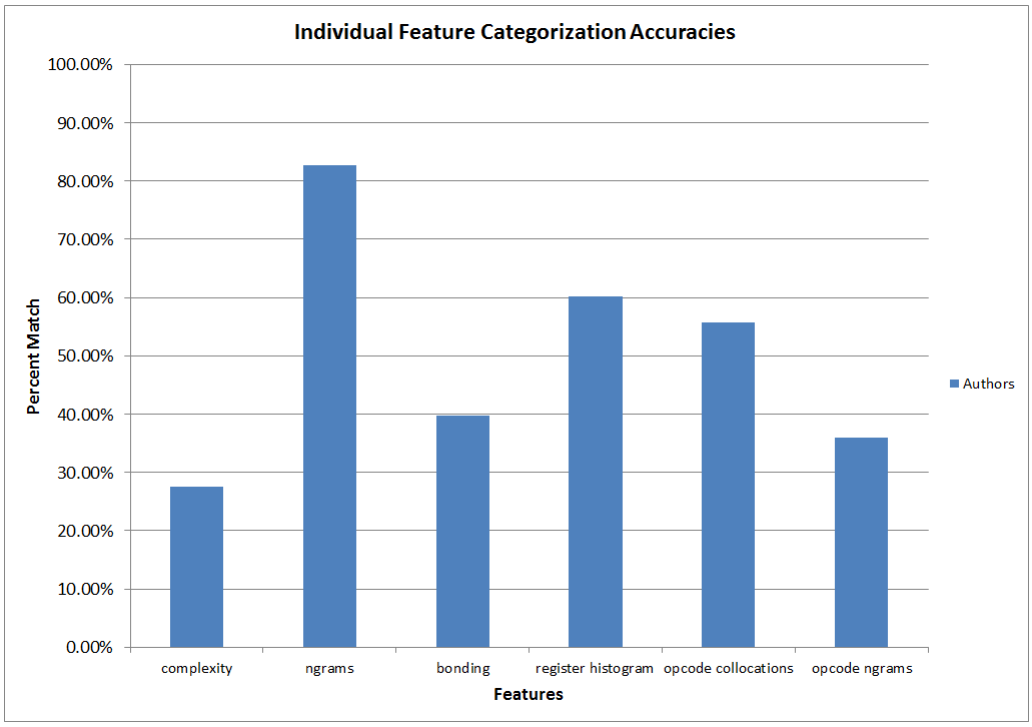


Figure 4.3: Individual feature tests run on the authorship data set, averaged over 30 runs. 80% of the data was dedicated to the training set, and the rest comprised the testing set.

The ultimate goal was to use some combination of these features to achieve fast, consistent results with high accuracy and outperform all single features. The method used in accomplishing this goal is explained in the following sections.

# 5. GENETIC PROGRAMMING METHODOLOGY

A GP algorithm was developed to evolve decision trees that would represent each individual category. In this way, a program could be matched to multiple categories (such as a program that belongs to a certain version group, and has been compiled by a particular compiler). The terminals in the GP trees contain the feature selection data presented in the previous section. Every category has its own values for each attribute, normalized between zero and one, and the terminals contain the relational data for a program being examined for acceptance into a category. For instance, a byte $N$-gram terminal would contain the percent match between the $N$-gram histogram of the program in question and the essential $N$-gram histogram features selected for a category. For cyclomatic complexity and bonding, the difference value is subtracted from one when normalized so that a high value in that terminal denotes a closer match. A linear disassembler was used for any features requiring disassembly.

The functional operators used in the non-leaf nodes of the GP trees are the binary operators AND, OR, and XOR. Due to the normalization of the terminal data, the binary operators must use fuzzy logic operators to be evaluated appropriately. Fuzzy logic dictates that the union of two values (OR) is equivalent to the maximum of both values, while an intersection (AND) is the same as a minimum of the values. A negation is equal to one minus the value. Extrapolating this further, an XOR can be represented as the AND of a higher value and the negation of a lower value.

Testing showed that if a hard threshold is imposed for the binary evaluations within the trees, such as evaluating anything greater than .5 as true, the category matching is far less precise. By using the fuzzy logic binary operators, a best match

can be evaluated because each tree receives a numerical value as opposed to a true or false evaluation. Figure 5.1 illustrates this concept.

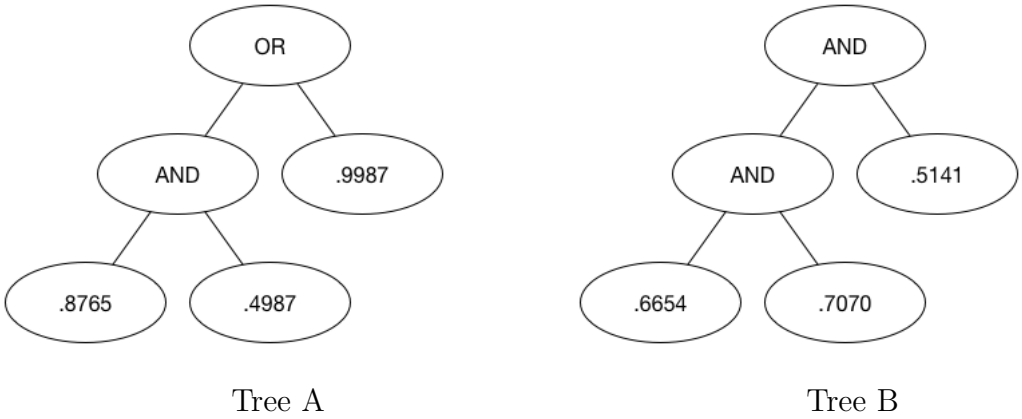


Figure 5.1: A fuzzy logic example showing that using a threshold, such as $\geq .5$ signifying true for a boolean operation, does not distinguish categories well. Both of the trees in this example would evaluate to true. Using the max/min rules, however, Tree A evaluates to .9987 while Tree B evaluates to .5141. This is used to prioritize matches.

In order for the GP algorithm to assess the quality of each tree in the population, a fitness function is needed. The fitness function does make use of a threshold (.5) to denote a match during the training phase. When a decision tree is evaluated for fitness, each program that makes up a category in the training set, known as the category set, is evaluated to determine if it would be accepted into the category using the current tree. An equal number of programs outside of the category set are evaluated to guide the decision tree in properly filtering out known mismatches. This set of programs, called the helper set, reduces the number of false-positives and ensures that the trees do not evolve to accept every program. A program in the helper set is not accepted by a category if the tree returns a value lower than the lowest match from the category set. Essentially, the lowest matching member of the category set becomes the new threshold for acceptance. The helper set carries the same weight as the category set to preserve fairness in acceptance and filtration.

# 6. EXPERIMENTAL SETUP

When testing the validity of this approach, only Windows executables were considered. Most software is developed for Windows, and it allows for an easy collection of data. The software in the versions and compiler data sets consisted of a large variety, ranging from video game emulators, system monitoring tools, and compression applications, to single-algorithm programs such as basic bubble sort and AVL tree implementations.

A GP generally requires a large number of parameters, all of which benefit greatly from tuning. However, in practical applications where a system administrator or security expert may need to evolve decision trees to categorize software, parameter tuning would be both too time-consuming, and out of the scope of their knowledge base. The algorithm should perform well enough under conditions where parameter tuning will not be performed. As such, some reasonable, but mostly arbitrary parameters were used in these experiments. A high tree depth was not necessary for either the versions or the compilers problem since optimal convergence came quickly with small values, and testing showed that higher values produced redundant logic. The authorship data set consistenly performed with three to ten percent higher average accuracies when a slightly higher tree depth was allowed. These concepts are easy to grasp, where as determining optimal terminal conditions and population sizes are not. Throughout the algorithm's runs, parsimony pressure was added to the trees to encourage smaller tree sizes and prevent unnecessary growth. The algorithm was trained on 80% of the data and tested on the remaining 20%. The intent of this option was to make sure that the evolved formula was not over-specialized to the

dataset. 30 runs were performed with the category, helper, and testing sets randomized each time. Table 6.1 shows the parameters applied to every category's GP in each problem.

Table 6.1: GP parameters

| **Parameter** | **Versions** | **Compilers** | **Authors** |
|---|---|---|---|
| $\mu$ | 100 | 100 | 100 |
| $\lambda$ | 20 | 20 | 20 |
| max depth | 2 | 2 | 3 |
| selection | $k$-tournament | $k$-tournament | $k$-tournament |
| survival | $k$-tournament | $k$-tournament | $k$-tournament |
| $k$ | 7 | 7 | 7 |
| crossover | single-point | single-point | single-point |
| mutation | sub-tree | sub-tree | sub-tree |
| mutation rate | .1 | .2 | .2 |
| termination | 5000 generations | 5000 generations | 5000 generations |

The results of these configurations are shown and discussed in the subsequent sections.

# 7. RESULTS

It is always important to ensure that a GP algorithm does not converge too early. During the all phases of these experiments, it is very clear what the maximum value should be. When all of the category applications are matched correctly and all of the helper set applications are filtered out during training, a 100% accuracy rate is achieved. In all of the experiments we performed, The average fitness of each GP population achieved the maximum value by the end of the evaluations. Figures 7.1, 7.2, and 7.3 show the average fitness progression of the populations (converted to percent accuracies in order to preserve an identical scale) across the evaluations for each experiment. It is clear from these figures that the convergence was appropriate.

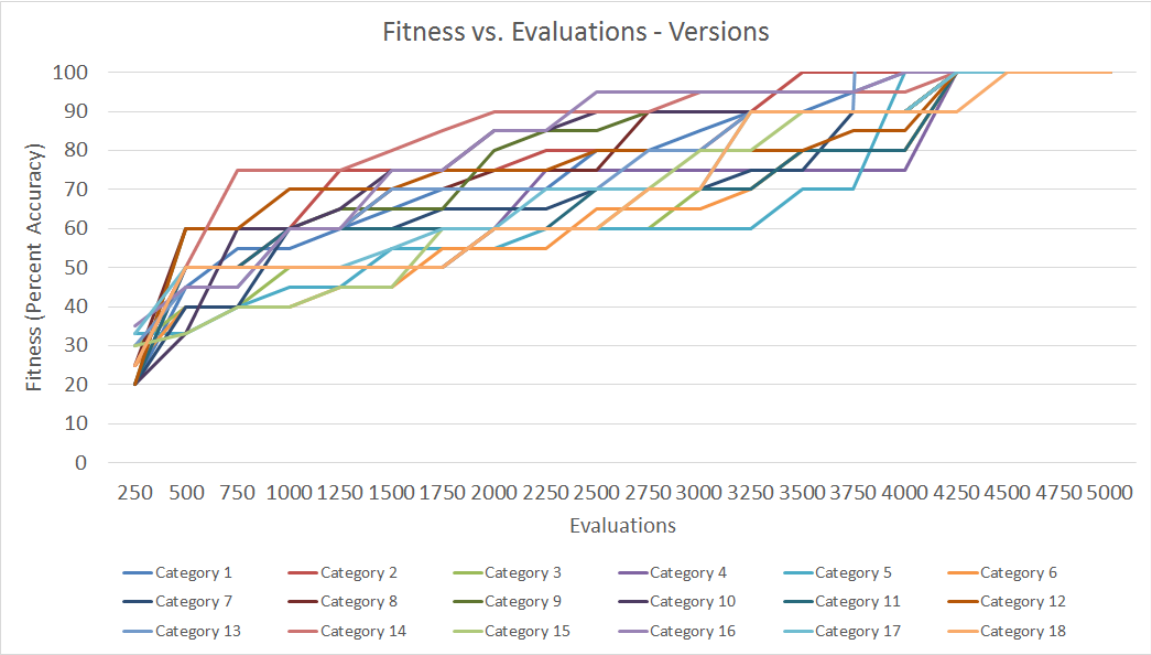


Figure 7.1: Fitness progression across the evaluations for the versions data set

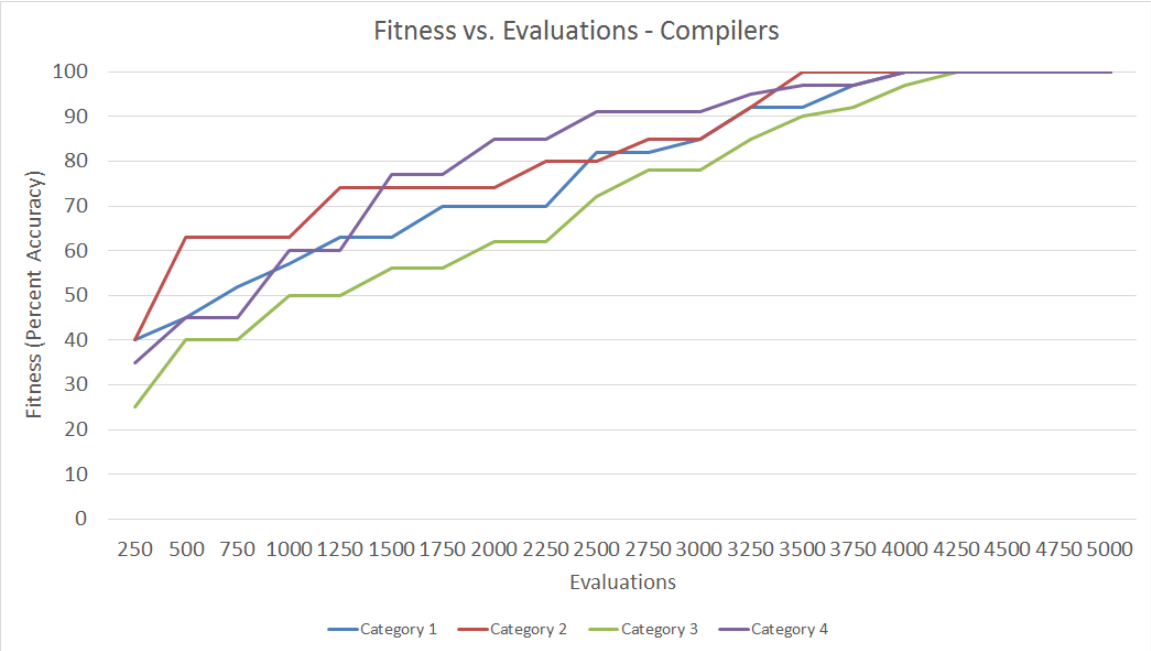


Figure 7.2: Fitness progression across the evaluations for the compilers data set

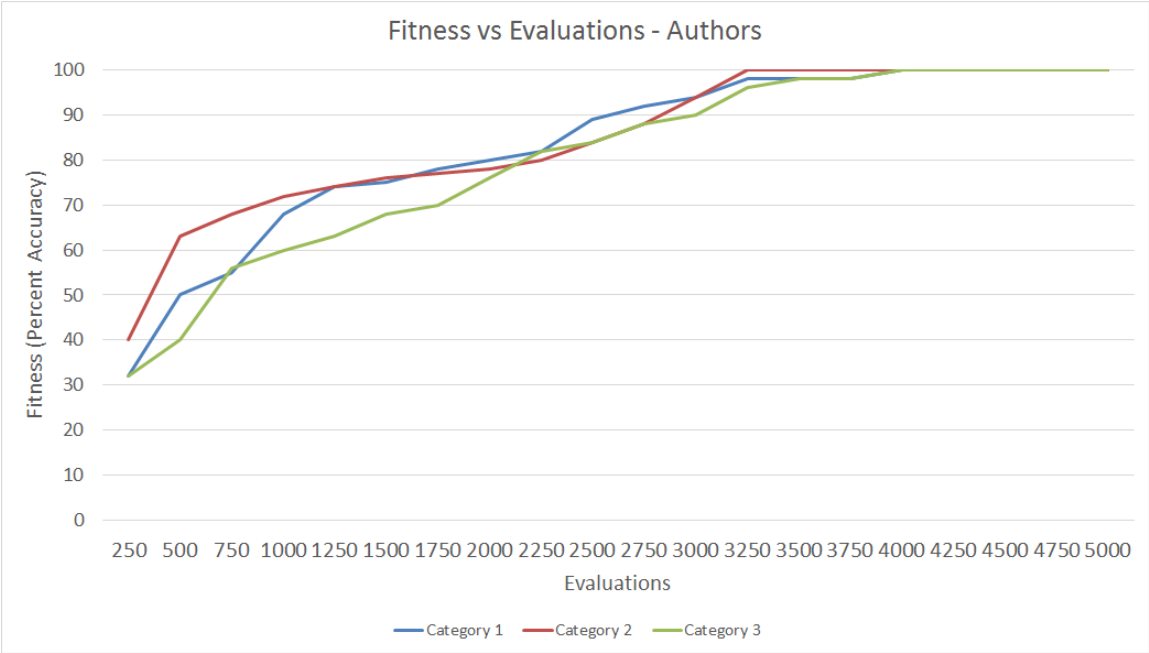


Figure 7.3: Fitness progression across the evaluations for the authors data set

Figure 7.4 shows the matching accuracy of the GP method for all three testing datasets, averaged over 30 runs.

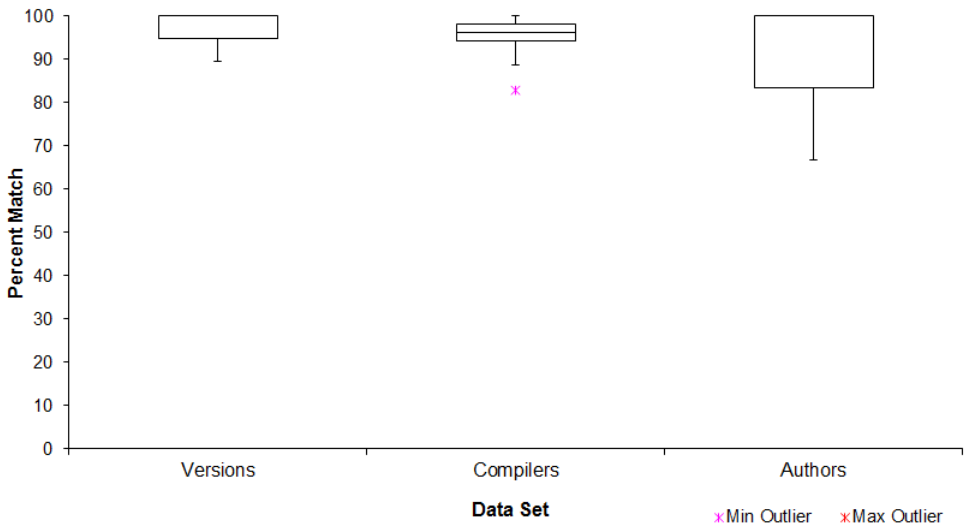


Figure 7.4: Percent match with GP decision trees – versions max = 100%, versions min = 89.47%; compilers max = 100%, compilers min = 82.69%; authors max = 100%, authors min = 66.66%

Due to the sufficiently large size of the compiler data set, an extra plot was produced to visualize potential overfitting in that experiment. The testing data was applied to every new generation of decision trees during the evolution of the training phase. The percent accuracy of categorizing the testing set was plotted against the evaluations. The plot is shown Figure 7.5.

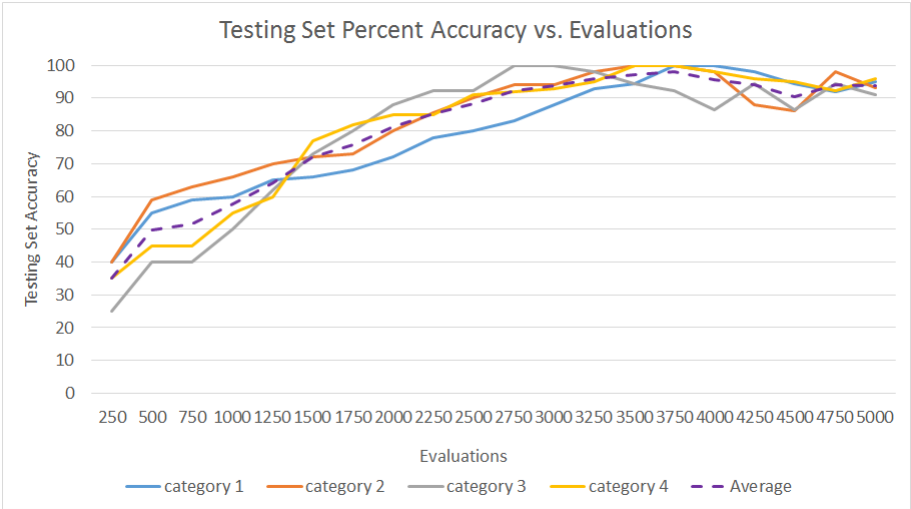


Figure 7.5: Compilers testing set percent match plotted against evaluations

The average accuracy reaches its highest percentage earlier in the evolution cycle and wavers at slightly lower values at the end. This dip in accuracy is an indication of the GP trees overfitting to the training data. While this is an undesired effect, the drop in accuracy is only a few percent, and the results were achieved without parameter tuning. Parameter tuning may allow for a more generalized solution which caps the evolution when the testing set's matching accuracy peaks, but such a small gain may not be worth the rigor required for true parameter optimization.

The average accuracies of the GP trees in every experiment were higher than those of any single feature. The $N$-grams performed only slightly worse, however. A set of $t$-tests were used to show that this method is a significant improvement over running only the $N$-grams feature. $F$-tests showed that unequal variance should be assumed for the versions data, but not the compilers and authors data. In all three $t$-tests, $|t$ Stat$|$ was greater than $t$ Critical Two-Tail. This means that the improvement gained from using the GP method is statistically significant. The results are summarized in Tables 7.1-7.6.

Table 7.1: $F$-Test: Two-Sample For Variances - Versions

| **Parameter** | **Decision Trees** | **$N$-Grams** |
|---|---|---|
| Mean | 97.19298246 | 95.0877193 |
| Variance | 12.863374 | 1.783042 |
| df | 29 | 29 |
| $F$ | 7.21428571 | |
| P($F \leq f$) One-Tail | 4.2497E-07 | |
| $F$ Critical One-Tail | 1.86081144 | |

Table 7.2: $F$-Test: Two-Sample For Variances - Compilers

| **Parameter** | **Decision Trees** | **$N$-Grams** |
|---|---|---|
| Mean | 95.25641026 | 92.69230769 |
| Variance | 17.020336 | 12.08937 |
| df | 29 | 29 |
| $F$ | 1.407876231 | |
| P($F \leq f$) One-Tail | 0.181150886 | |
| $F$ Critical One-Tail | 1.860811435 | |

Table 7.3: $F$-Test: Two-Sample For Variances - Authors

| **Parameter** | **Decision Trees** | **$N$-Grams** |
|---|---|---|
| Mean | 90.5555555555 | 81.66666667 |
| Variance | 89.7190293744 | 351.532567 |
| df | 29 | 29 |
| $F$ | .25522252497799 | |
| P($F \leq f$) One-Tail | .00021729 | |
| $F$ Critical One-Tail | .537399965 | |

Table 7.4: $t$-Test: Two-Sample Assuming Unequal Variance - Versions

| **Parameter** | **Decision Trees** | **$N$-Grams** |
|---|---|---|
| Mean | 97.19298246 | 95.0877193 |
| Variance | 12.863374 | 1.783042 |
| Hypoth. Mean Dif. | 0 | |
| df | 37 | |
| $t$ Stat | 3.0130152454 | |
| P($T \leq t$) Two-Tail | 0.0046475888 | |
| $t$ Critical Two-Tail | 2.026192463 | |

Table 7.5: $t$-Test: Two-Sample Assuming Equal Variance - Compilers

| **Parameter** | **Decision Trees** | **$N$-Grams** |
|---|---|---|
| Mean | 95.25641026 | 92.69230769 |
| Variance | 17.020336 | 12.08937 |
| Pooled Variance | 14.5548528 | |
| Hypoth. Mean Dif. | 0 | |
| df | 58 | |
| $t$ Stat | 2.6030176593 | |
| P($T \leq t$) Two-Tail | 0.011713133 | |
| $t$ Critical Two-Tail | 2.0017174841 | |

Table 7.6: $t$-Test: Two-Sample Assuming equal Variance - Authors

| **Parameter** | **Decision Trees** | **$N$-Grams** |
|---|---|---|
| Mean | 90.5555555555 | 81.66666667 |
| Variance | 89.7190293744 | 351.532567 |
| Pooled Variance | 220.625798 | |
| Hypoth. Mean Dif. | 0 | |
| df | 58 | |
| $t$ Stat | 2.3177413 | |
| P($T \leq t$) Two-Tail | .02401221 | |
| $t$ Critical Two-Tail | 2.00171748 | |

The performance of the GP trees was enhanced by the other features. Using only $N$-grams achieves a certain baseline, and the use of additional features increases the accuracy as more data is presented when categorizing the software each experiment. When $N$-grams cause false-positives to occur due to low matching percentages for certain test programs, the other features in the tree minimize the error as better matches are prioritized with the binary operators.

## 7.1. FUNCTIONAL CLASSES EXPERIMENT

A fourth experiment was also performed wherein the intent was to categorize software based on functionality using all of the same features detailed in Section 4. A set of 50 programs spanning thirteen categories of manually-labeled functional

classes were used. The GP was unable to achieve results that were any better than a random search. This is a much harder problem, and the features used in the other three experiments were not ideal in distinguishing these functional classes. It was hypothesized that at least the system calls attribute would be able to produce some correct matches, given that it could select features that elucidate some functionality. Unfortuantely, the feature selection technique was not enough. It may be possible to solve this problem using the GP decision tree method if other feature selection algorithms were developed to extract higher-level information about the program functionalities. For instance, developing some algorithm detection features would likely result in higher accuracies. Figure 7.6 shows the poor performance of each of the feature selection techniques. The same setup was used as in the other experiments. 80% of the data was used for training, and the rest was used for testing. The results are averaged over 30 runs.

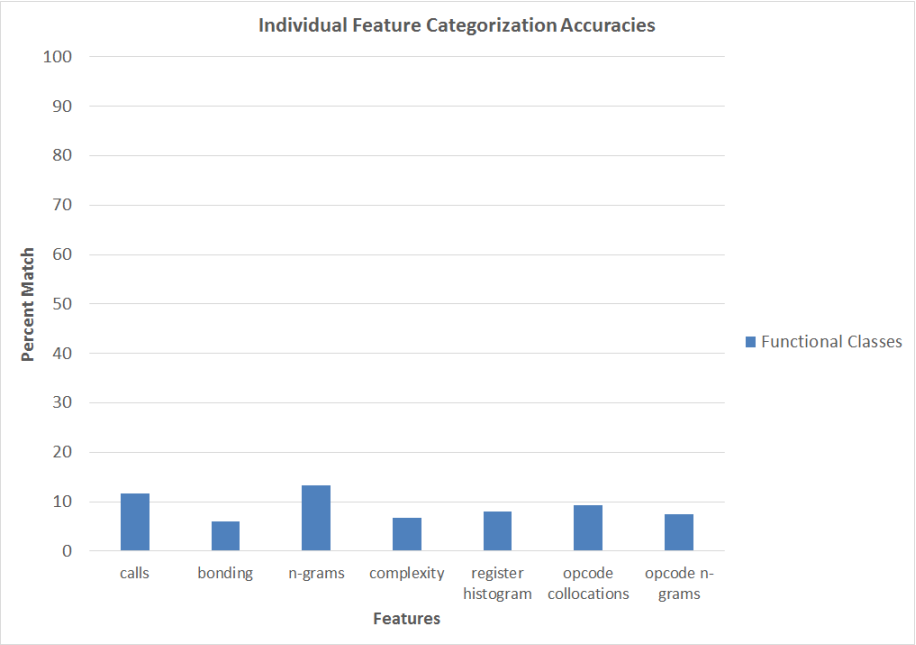


Figure 7.6: Individual feature tests run on the functional classes data set, averaged over 30 runs.

Given the poor perforamnce of the single features, the GP algorithm was not expected to perform well either. The box plot in Figure 7.7 shows the results of the GP. A single max outlier exists where the algorithm achieved 28.57% accuracy.

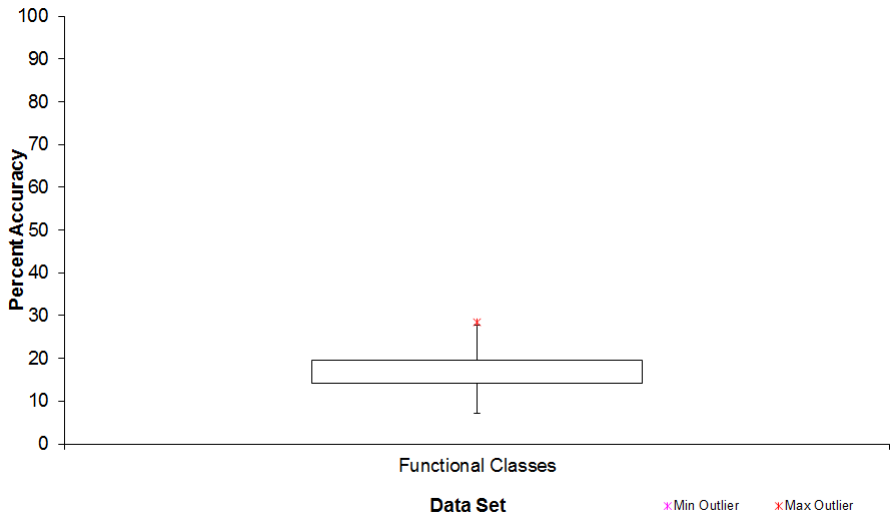


Figure 7.7: Percent match with GP decision trees on functional classes – max = 28.57%, min = 7.14%

## 7.2. C4.5 COMPARISON

The results of the GP decision trees were compared to the classic statistical decision tree classifier C4.5, which employs a table of data to make a decision tree classifier with rules at each node of the tree. The nodes of the tree are based on the table attributes that best split the data in order to achieve a correct categorization of a labeled data set. This tree can then be used on a testing set, similar to the GP approach. The same attributes and feature selection methods were used in both the GP trees and the C4.5 algorithm's data table. 80% of each category was again combined to create the training set, and the rest was used for testing. All of the feature selection values extracted for an application were normalized values, relative to values of each category. As such, the table that was created for C4.5 had relational data representing how much each program matched to each category for every

attribute. The training and testing sets were randomized for 30 total runs. A box plot containing the percent accuracy of the C4.5 algorithm on our data sets is shown in Figure 7.8.

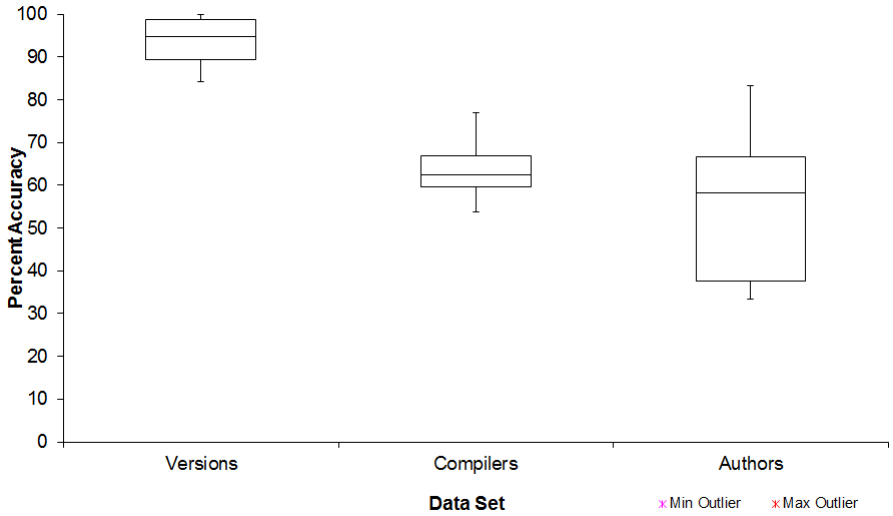


Figure 7.8: Percent match with C4.5 decision trees – versions max = 100%, versions min = 84.21%; compilers max = 76.92%, compilers min = 53.85%; authors max = 83.33%, authors min = 33.33%

The average accuracy in each experiment was lower for C4.5 than the GP method. A set of $t$-tests were performed again to show that the results of using GP were significantly better than those of C4.5. $F$-tests showed that two-tailed $t$-tests with equal variance were required for all experiments. The results of all tests are shown in Tables 7.7-7.12.

Table 7.7: $F$-Test: Two-Sample For Variances - Versions

| **Parameter** | **C4.5** | **GP** |
|---|---|---|
| Mean | 93.68421053 | 97.19298 |
| Variance | 27.50979081 | 12.86337 |
| df | 29 | 29 |
| $F$ | 2.138613861 | |
| P($F \leq f$) One-Tail | 0.022440709 | |
| $F$ Critical One-Tail | 1.86081144 | |

Table 7.8: $F$-Test: Two-Sample For Variances - Compilers

| **Parameter** | **C4.5** | **GP** |
|---|---|---|
| Mean | 64.42307692 | 95.25641 |
| Variance | 47.12048562 | 17.02034 |
| df | 29 | 29 |
| $F$ | 2.768481518 | |
| P($F \leq f$) One-Tail | 003869137 | |
| $F$ Critical One-Tail | 1.86081144 | |

Table 7.9: $F$-Test: Two-Sample For Variances - Authors

| **Parameter** | **C4.5** | **GP** |
|---|---|---|
| Mean | 56.111111111 | 90.5555555555 |
| Variance | 296.6155811 | 89.71903 |
| df | 29 | 29 |
| $F$ | 3.306049822 | |
| P($F \leq f$) One-Tail | 0.000952421 | |
| $F$ Critical One-Tail | 1.8608114355 | |

Table 7.10: $t$-Test: Two-Sample Assuming Equal Variance - Versions

| **Parameter** | **C4.5** | **GP** |
|---|---|---|
| Mean | 93.68421053 | 97.19298 |
| Variance | 27.50979081 | 12.86337 |
| Pooled Variance | 20.18658261 | |
| Hypoth. Mean Dif. | 0 | |
| df | 58 | |
| $t$ Stat | -3.02460989 | |
| P($T \leq t$) Two-Tail | 0.00370572 | |
| $t$ Critical Two-Tail | 2.001717484 | |

Table 7.11: $t$-Test: Two-Sample Assuming Equal Variance - Compilers

| **Parameter** | **C4.5** | **GP** |
|---|---|---|
| Mean | 64.42307692 | 95.25641 |
| Variance | 47.12048562 | 17.02034 |
| Pooled Variance | 37.0704108 | |
| Hypoth. Mean Dif. | 0 | |
| df | 58 | |
| $t$ Stat | -21.0869538 | |
| P($T \leq t$) Two-Tail | 7.0227E-29 | |
| $t$ Critical Two-Tail | 2.001717484 | |

Table 7.12: $t$-Test: Two-Sample Assuming Equal Variance - Authors

| **Parameter** | **C4.5** | **GP** |
|---|---|---|
| Mean | 56.111111111 | 90.5555555555 |
| Variance | 296.6155811 | 89.71903 |
| Pooled Variance | 193.1673052 | |
| Hypoth. Mean Dif. | 0 | |
| df | 58 | |
| $t$ Stat | -9.598381406 | |
| P($T \leq t$) Two-Tail | 1.38639E-13 | |
| $t$ Critical Two-Tail | 2.001717484 | |

In every experiment, the GP method's higher accuracies were statistically significant. The number of false-positives was far higher with C4.5. It was likely the addition of the helper set that enabled the GP algorithm to perform so well. The helper set trained the decision trees to filter out programs that do not belong to their categories. Without the helper set, the GP trees would evolve to accept all applications, and C4.5 would be far superior in its ability to classify software.

Figures 7.9, 7.10, 7.11, and 7.12 contain some of the GP-evolved trees with the highest fitness values for certain categories.

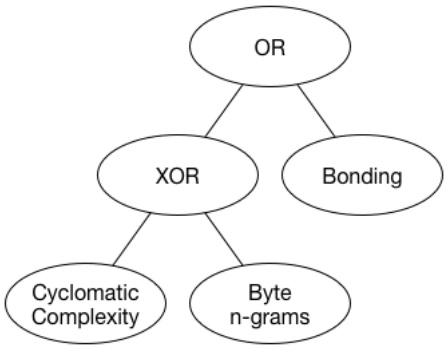


Figure 7.9: Compilers - GCC, No optimization

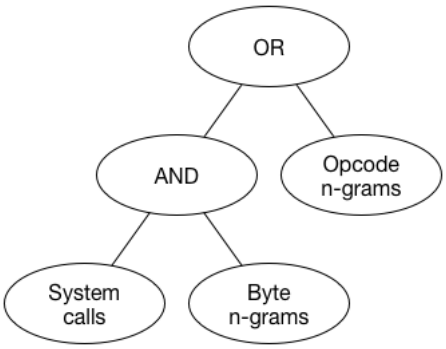


Figure 7.10: Versions - Pidgin

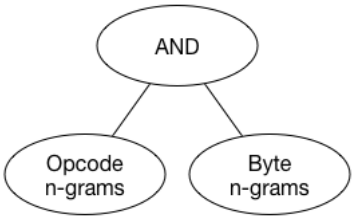


Figure 7.11: Versions - Nestopia

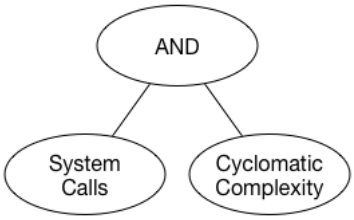


Figure 7.12: Versions - BaculaTrayMonitor

These figures illustrate how simplistic, yet powerful the trees can be. Figure 7.10, for instance, contains two high-level features and a low-level feature. If the $N$-grams and system calls match well, the minimum value of the two in comparison to the established category values (derived from the training set) is passed up the tree. The maximum of this value and the opcode $N$-grams feature is used as the ultimate value signifying how closely the test program matches the category. If the value produced by this tree is higher than that of any other tree, the test program belongs to this category.

# 8. DISCUSSION

The advantages of using GP to produce decision trees for program classification are quite clear. In all three test cases, the decision tree heuristics produced by the GP performed very well. For the versions problem, the solution produces an average accuracy of 97.2% while the compilers problem reached a slightly lower average of 95.3%. The authorship data set was the most difficult to categorize correctly, but the average accuracy of this method was still 90.6%. The majority of the compiler mismatches came from incorrectly guessing the optimization flags, not the actual source compiler. Version mismatches encountered a far different hurdle. Some of the programs in the versions dataset had very minor changes from one version to the next, and the evolved trees were able to make those connections. On more complicated version differences, if two very different versions of the same program were not both in the category set, some false matches occurred. One of the programs in the dataset had a size increase of 43% between versions. The evolved heuristics were unable to make this big leap and relate the versions. Ultimately it is a question of semantics, but a case can be made that the large variation from one version to the next resulted in an altogether different application. After all, if an application remains the same in name but completely overhauls every aspect of its code, it is reasonable to expect a different categorization between those versions. The results of the author origin experiment did not suggest a pattern to the false-positives, though certain programs were mismatched more often than others. Based on visual inspection, the coding styles for those programs seem consistent with the authors that wrote them, but it is possible that compiler optimization muddled the distinctions.

All of the GP trees reached maximum fitness during the training phase. This suggests that the success of the decision trees in classifying software lies more in the

proper feature selection. The features need to be robust enough to match programs within a category and filter out applications that do not belong. Even though 100% accuracy was achieved in the training phase, false positives occurred in testing. The different tree variations produced by the evolution were likely a result of a varied training and helper set.

The C4.5 algorithm performed worse than the GP in all three experiments, reaching average accuracy rates of 93.68%, 64.42%, and 56.11% for the versions, compilers, and authors data sets, respectively. There are many possible reasons for the unsuccessful results. Traditionally, data tables used for C4.5 contain direct attribute values for an item, where as the data tables used in these experiments contained attribute values for programs as they related to certain established category values. As such, enough columns had to exist in the data table to represent a program's feature values in relation to every cateogry. The number of columns in the table can be computed with the following equation:

$$Columns = Features \cdot Categories + 1 \tag{8.1}$$

where the +1 denotes the column for labeling the correct category. When the resultant decision trees were examined, the tree for the versions problem seemed to overfit to the training data as there were many highly specific rules. Regardless, the versions experiment was most successful for the C4.5 decision tree. The compilers problem produced a far different tree, however. The tree resolves to being nearly equivalent to the $N$-grams feature alone.

It should be noted from the individual feature tests and a majority of the GP trees, that byte $n$-grams gathered with our feature selection method produced the best results. This is presumably due to the types of categories used in these experiments. In the versions and compiler cases, byte $N$-grams from the portable executable (PE)

header could be the primary features selected with this attribute. Since none of the applications were purposely obfuscated and had no reason to contain corrupted or misleading headers, this information most likely led to easy matches. It can be assumed that these types of features would not be as useful given different problem types, such as categorizing software by functional classes. The $N$-gram features were expected to perform well on the authorship data set as the common trigrams within a category could indicate programming style.

In these experiments, not all terminals were useful all of the time. Cyclomatic complexity was a highly used attribute for the versions problem tests, but not throughout the compiler identification runs (unless it was included in an XOR). It stands to reason that the complexity of an application would give little insight to the compiler used to create it, unless optimization flags caused an extreme difference in complexity. The system calls primitive experienced the same high frequency of occurrence in the versions problem tests, but was almost nonexistent when applied to the compilers dataset. Although bonding occasionally appeared in final solutions of the compilers problem, it was not a major discriminator. Despite some filtration qualities within a few decision trees, this metric is best suited for social graphs. The introduction of the opcode collocations and register features increased the average accuracy of the authorship data set by almost 10%. It is clear that such a data set is categorized with a higher accuracy if the features in the GP trees key in on a programmer's habits and patterns.

Categories based on a more semantic grouping require more semantic features. Program behavior is difficult to capture in low-level attributes such as $N$-grams and cyclomatic complexity. A fourth experiment was performed to show that categorization based on criteria such as functionality cannot be done with the features presented in Section 4. Using all of the attributes detailed in this thesis on the functional classes data set resulted in a mere 15% average accuracy for the decision trees. In order to

represent such characteristics, algorithm or functionality identification may need to be implented as a primitive for the GP trees. Extracting such information is difficult to do and requires longer run times. It is difficult to determine how many primitives of this nature would be required to achieve high accuracy results.

# 9. FUTURE WORK

The future work of this research can take many logical paths. The results achieved in this thesis have a high average accuracy, but they can still be improved. Added both low-level and high-level attributes could drive the average accuracy even higher. Removing false-positives is necessary for any practical applications of this approach. More robust feature selection methods would need to be investigated in order to make that a reality.

It would also be useful to perform proper parameter tuning for the GP algorithms. Although the methodology detailed in Section 5 is sufficient for practitioners not experienced with EC to achieve high accuracy results, Figure 7.5 shows that the performance can be increased further by avoiding overfitting and generalizing the decision trees through parameter tuning.

Most importantly, more difficult categorization problems should be attempted with these methods. For instance, categorizing software based on functionality would have far-reaching impact on many fields in computer science, but as was shown in Section 7.1, that will require more semantic feature extraction. While research in the area of software quality exists, it would be useful to use this method to categorize poorly written software or software that is exploitable. Furthermore, it is likely that different universities or countries have different programming styles of their own. With the right data and a rich feature set, it would be interesting to categorize software by country of origin.

Although packing and obfuscation would increase the difficulty substantially, classifying benign versus malicious programs or categorizing different kinds of malware would be an essential accomplishment for the field of computer security. The true strength of the techniques discussed in this thesis lies in the richness of the

features that comprise the decision trees. Any research that extracts valuable semantic data about software can be easily combined with this method to further its categorization capabilities.

# 10. CONTRIBUTIONS

The work presented in this thesis contains three distinct contributions to the field of computer science. First, a novel and general approach to classifying software was developed. The method can be used to categorize software based on many different criteria, if the right feature selection methods are chosen. Furthermore, classifying more data into the same categories with this technique does not require running more EC algorithms or deep analytical comparisons. The evolved trees can be reused quickly after the initial evolution has completed. Second, an alternative option to developing decision trees was offered. GP was used to evolve a set of decision trees that employ fuzzy logic to determine the degree to which a certain application belongs to a category. While more experiments would need to be performed to verify the validity of this approach in other contexts, in this case, the decision tree method outperformed C4.5 on three different experiments. Third, and most important, several quality feature selection methods were provided and their usefulness in certain problem domains was elucidated. These features are the true foundations from which the GP trees draw their success. It is worth examining these contributions when considering problems in similar domains.

# 11. CONCLUSIONS

The need for quick methods of software classification is undeniable. Although techniques exist to analyze binaries in order to extract some semantic information about them, most require long running times and considerable computational resources. The ideas presented in this thesis aim to provide a means of quickly categorizing software based on a few key attributes. The notion of evolving decision trees through GP was explored. Due to the nature of the solutions being evolved by the GP, they can be reused to categorize more data. Applying a decision tree to a new set of data without executing the entire GP process requires only a few seconds.

This method can be applied to a large range of problems that require classification. All that is required is a training set to evolve initial decision trees, and the GP does the rest. When distinguishing programs by versions, compiler, or author origin, the decision trees achieve over 90% accuracy. These same methods can be utilized to categorize software using different criteria, though more applicable attributes may need to be mined for features because the particular attributes used in Section 4, as shown with the functional classes experiment, will not be sufficient discriminators of every kind of data set. In such instances, more semantic feature extraction methods may be required to key in on higher-level abstractions of application functionality.

The GP approach was compared to a classic decision tree algorithm called C4.5. Using the same feature data as was presented to the GP trees, the C4.5 algorithm performed significantly worse in the three primary experiments. This further validates our use of GP to generate decision trees for classifying software.

# BIBLIOGRAPHY


[1] Jasenko Hosic, Samuel A. Mulder, and Daniel R. Tauritz. Evolving Decision Trees for the Categorization of Software. In *Proceedings of COMPSAC 2014 - the 38th Annual IEEE International Computers, Software, and Applications Conference*, 2014.

[2] Nathan E. Rosenblum, Xiaojin Zhu, and Barton P. Miller. Who Wrote This Code? Identifying the Authors of Program Binaries. In *Proceedings of the 15th European Symposium on Research in Computer Security*, ESORICS '11, pages 172–189, September 2011.

[3] Kehan Gao, Taghi M. Khoshgoftaar, and Huanjing Wang. An Empirical Investigation of Filter Attribute Selection Techniques for Software Quality Classification. In *10th IEEE International Conference on Information Reuse and Integration*, pages 272–277, August 2009.

[4] James F. Bowring, James M Rehg, and Mary Jean Harrold. Active Learning for Automatic Classification of Software Behavior. In *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '04, pages 195–205, July 2004.

[5] Nathan E. Rosenblum, Xiaojin Zhu, Barton P. Miller, and Karen Hunt. Machine Learning-Assisted Binary Code Analysis. In *Workshop on Machine Learning in Adversarial Environments for Computer Security*, NIPS '07, December 2007.

[6] Bruschi Danilo, Lorenzo Martignoni, and Mattia Monga. Detecting Self-Mutating Malware Using Control-Flow Graph Matching. In *Proceedings of the Third international conference on Detection of Intrusions and Malware and Vulnerability Assessment*, DIMVA '06, pages 129–143, 2006.

[7] Vijay Nagarajan, Rajiv Gupta, Xiangyu Zhang, Matias Madou, and Bjorn De Sutter. Matching Control Flow of Program Versions. In *IEEE International Conference on Software Maintenance*, ICSM, pages 84–93, October 2007.

[8] Nathan E. Rosenblum, Barton P. Miller, and Xiaojin Zhu. Extracting Compiler Provenance from Program Binaries. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering*, PASTE '10, pages 21–28, June 2010.

[9] Gordon V. Kass. An Exploratory Technique for Investigating Large Quantities of Categorical Data. *Applied Statistics*, 29(2):119–127, 1980.

[10] Jay Fowdar, Keeley Crockett, Bandar Zuhair, and James O'Shea. On the Use of Fuzzy Trees for Solving Classification Problems with Numeric Outcomes. In *Proceedings of the Seventh International Conference on Machine Learning and Cybernetics*, volume 1, pages 12–17, July 2008.

[11] John Ross Quinlan. Induction of Decision Trees. In *Machine Learning*, volume 1, pages 81–106. Kluwer Academic Publishers, Boston, MA, USA, March 1986.

[12] John Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, USA, 1993.

[13] Juan Sun and Xi-Zhao Wang. An initial comparison on noise resisting between crisp and fuzzy decision trees. In *Proceedings of 2005 International Conference on Machine Learning and Cybernetics*, volume 4, pages 2545–2550, August 2005.

[14] Pawel Lewicki and Thomas Hill. CHAID Analysis. In *Statistics: Methods and Applications*, pages 84–86. StatSoft, Inc., Tulsa, OK, USA, November 2005.

[15] John R. Koza. Overview of Genetic Programming. In *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, pages 74–78. MIT PRESS, Cambridge, MA USA, 1992.

[16] Mario Bkassiny, Yang Li, and Sudharman K. Jayaweera. A Survey on Machine-Learning Techniques in Cognitive Radios. *IEEE Communications Surveys & Tutorials*, 15(3):1136–1159, July 2013.

[17] Jose L. Ribeiro Filho, Philip C. Treleaven, and Cesare Alippi. Genetic-Algorithm Programming Environments. *Computer*, 27(6):28–43, June 1994.

[18] Jih-Yiing Lin and Ying-Ping Chen. On the Effect of Population Size and Selection Mechanism from the Viewpoint of Collaboration between Exploration and Exploitation. In *2013 IEEE Workshop on Memetic Computing*, MC, pages 16–23, April 2013.

[19] Pedro G. Espejo, Sabastian Ventura, and Francisco Herrera. A Survey on the Application of Genetic Programming to Classification. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 40(2):942–958, March 2010.

[20] Riccardo Poli and Nicholas Freitag McPhee. Parsimony Pressure Made Easy: Solving the Problem of Bloat in GP. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, GECCO '08, pages 1267–1274, July 2008.

[21] Edmund K. Burke, Matthew R. Hyde, Graham Kendall, and John R. Woodward. A Genetic Programming Hyper-Heuristic Approach for Evolving 2-D Strip Packing Heuristics. *IEEE Transactions on Evolutionary Computation*, 14(6):942–958, December 2010.

[22] Edmund K. Burke, Matthew R. Hyde, Graham Kendall, Gabriela Ochoa, Ender Ozcan, and John R. Woodward. Exploring Hyper-heuristic Methodologies with Genetic Programming. In *Computational Intelligence: Collaboration, Fusion and Emergence*, pages 177–201. Springer, Berlin-Heidelberg, Germany, March 2009.

[23] Edmund K. Burke, Matthew R. Hyde, Graham Kendall, and John R. Woodward. Automatic Heuristic Generation with Genetic Programming: Evolving a Jack-of-all-Trades or a Master of One. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, GECCO '07, pages 1559–1565, 2007.

[24] Una-May O'Reilly. Using a Distance Metric on Genetic Programs to Understand Genetic Operators. In *IEEE International Conference on Systems, Man, and Cybernetics*, pages 4092–4097, October 1997.

[25] Silvio Cesare and Yang Xiang. Malware Variant Detection Using Similarity Search over Sets of Control Flow Graphs. In *IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*, TrustCom, pages 181–189, November 2011.

[26] John Ross Quinlan. Improved Use of Continuous Attributes in C4.5. *Journal of Artificial Intelligence Research*, 4(1):77–90, January 1996.

[27] Daniel Remy Tauritz. *Adaptive Information Filtering: concepts and algorithms*. Ph.D. dissertation, Leiden University, 2002.

[28] Imre Solti, Colin R. Cooke, Xia Fei, and Mark M. Wurfel. Automated Classification of Radiology Reports for Acute Lung Injury: Comparison of Keyword and Machine Learning Based Natural Language Processing Approaches. In *IEEE International Conference on Bioinformatics and Biomedicine Workshop*, BIBMW 2009, pages 314–319, November 2009.

[29] Owen Macindoe and Whitman Richards. Graph Comparison Using Fine Structure Analysis. In *Proceedings of the 2010 IEEE Second International Conference on Social Computing*, SOCIALCOM '10, pages 193–200, August 2010.

[30] Raja Khurram Shahzad, Niklas Lavesson, and Henric Johnson. Accurate Adware Detection using Opcode Sequence Extraction. In *Sixth International Conference on Availability, Reliability and Security*, ARES, pages 189–195, August 2011.

[31] Jian-Fang Lin, Sheng Li, and Yuhan Cai. A new collocation extraction method combining multiple association measures. In *Proceedings of the Seventh International Conference on Machine Learning and Cybernetics*, volume 1, pages 12–17, July 2008.

[32] Supreeth Burji, Kathy J. Liszka, and C. C. Chan. Malware Analysis Using Reverse Engineering and Data Mining Tools. In *International Conference on System Science and Engineering*, ICSSE, pages 619–624, July 2010.

[33] Avi Pfeffer, Catherine Call, Arun Lakhotia, John Bay, and Robert Hall. Malware Analysis and Attribution Using Genetic Information. In *7th International Conference on Malicious and Unwanted Software*, MALWARE, pages 39–45, October 2012.

[34] Sharif Monirul, Andrea Lanzi, Jonathon Griffin, and Wenke Lee. Automatic Reverse Engineering of Malware Emulators. In *30th Symposium on Security and Privacy*, pages 94–109, May 2009.

[35] Martin Apel, Christian Bockerman, and Michael Meier. Measuring Similarity of Malware Behavior. In *IEEE 34th Conference on Local Computer Networks*, pages 891–898, October 2009.

[36] Nicholas Sherlock and Andrew Trotman. Id - Dynamic Views on Static and Dynamic Disassembly Listings. In *Proceedings of the 14th Australasian Document Computing Symposium*, pages 19–26, December 2009.

# VITA

Jasenko Hosic was born on October 24th 1989 in Bosnia and Herzegovina. His family moved to Germany in 1992 to escape the Bosnian genocide. In 1997, his family moved to the United States. He spent the remainder of his childhood in Saint Louis, Missouri. He was ranked in the top ten of his high school, Rockwood Summit, with a GPA of 4.33. During his high school years, he became a National AP Scholar, National Merit Commended student, John M. Kasner award recipient, and Falcon Flight award winner. From fall of 2008 to Spring of 2014, he attended Missouri University of Science and Technology to earn Bachelor of Science degrees in Computer Science and Computer Engineering with the help of Bright Flight and Missouri S&T Chancellor's scholarships. From Spring 2009 to Spring of 2010, Jasenko served as Campus Ambassador and President of Missouri S&T's OSUM organization, an open-source initiative funded by Sun Microsystems. During the spring and summer of 2010 and the summer of 2011, he worked as a developer for Nucor-Yamato Steel. Throughout his years at Missouri S&T, he was an active member of ACM SIG-Security. In fall of 2012, he was the organization's Special Projects Officer. In the summer of 2012, Jasenko worked for Sandia National Laboratories as a technical intern in the Center of Cyber Defenders. He was then accepted into Sandia's Critical Skills Master's Program that lasted from the summer of 2013 to Spring of 2014. With funding from Sandia's Critical Skills Master's Program, Jasenko earned his Master of Science degree in Computer Science from Missouri University of Science and Technology in August of 2014 and performed the research upon which this thesis was based.