SAND2014-3929C

Evolving Decision Trees for the Categorization of Software

Jasenko Hosic, Daniel R. Tauritz
Natural Computation Laboratory
Department of Computer Science
Missouri University of Science and Technology
Rolla, Missouri 65409-0350, USA
Email: jhosic@gmail.com, dtauritz@acm.org

Samuel A. Mulder
Sandia National Laboratories
Albuquerque, New Mexico 87185, USA
Email: samulde@sandia.gov

Abstract—**Current manual techniques of static reverse engineering are inefficient at providing semantic program understanding. We have developed an automated method to categorize applications in order to quickly determine pertinent characteristics. Prior work in this area has had some success, but a major strength of our approach is that it produces heuristics that can be reused for quick analysis of new data. Our 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, cyclomatic complexity, and bonding. The evolved decision trees are reusable and achieve average accuracies above 90% when categorizing programs based on compiler origin 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.**

I. Introduction

The problem of software classification has been previously examined, and its use for authorship [14], quality [8], and content attribution [1] is apparent. This field, however, can have far-reaching implications on new problem spaces as well. For instance, system administrators maintaining 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 [6], [15]. 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 we propose a means of rapidly classifying software into categories through evolved decision trees.

In this paper, we focus on performing two distinct experiments: categorizing software based on the compiler and optimization flags used during development, and categorizing multiple versions of the same software. Although the versioning and compiler identification problems have been tackled in previous research [6], [11], [13], we performed these experiments as an initial demonstration of our approach. We are able to categorize software using a variety of criteria with the same algorithm. Furthermore, due to the absence of any dynamic analysis in our algorithm, it performs more efficiently than many previous approaches. We achieve over 90% accuracy when matching test programs to categories in both experiments, and the resulting decision heuristics that are derived can quickly be reused to categorize more software without requiring thorough binary analysis.

Due to the known limitations and problems of classic decision tree methods [16], an approach utilizing genetic programming (GP) [9] was used to develop the decision trees. GP is a population-based meta-heuristic technique that searches for a solution by making multiple initial guesses. These guesses are then broken into fragments and recombined until a termination condition is met. With the use of a fitness function, a numerical value can be calculated to represent the quality of a solution. Natural selection determines which solutions should continue to reproduce and which should be discarded. 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. This structure is ideal as it translates directly into the decision trees we use to categorize applications. GP algorithms usually have a long runtime, and while this is true of our GP, applying the evolved decision trees on new software takes only seconds.

II. Related Work

Significant contributions have been made to the fields of GP as it relates to developing new heuristics [2]–[4], [12], software classification [7], [8], version matching [5], [6], [11], and compiler attribution [13]. The research presented in this paper 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, we propose a hybrid of some of these concepts in the design of our GP evolution of decision trees.

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 [2], [4], 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.

V. Nagarajan et al. [11] 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. 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. 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 and used as a prioritization field. 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 paper 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.

Compiler attribution has been examined in much the same way as version classification. Rosenblum et al. [13] 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 [15]. 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 paper comes from the ideas presented in [14]. 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. Idioms and n-grams assist in detecting patterns that comprise an author's signature, allowing for high accuracy during classification. This paper attempts to utilize some similar concepts in order to produce quick, reliable results for the version and compiler categorization problems.

III. Methodology

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 [8]. For this reason, we have chosen six different attributes to consider, each with their own feature selection process. The attributes were chosen for their potential as distinguishing factors for the versions and compiler problems. A testing 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. 80% of the data was used as the training set for determining the pertinent features and 20% was used as the testing set. The following is an explanation of each attribute and its feature selection scheme.

A. System calls

We identified system calls 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. When determining the proper way to select features with system calls, we had to strike a balance between focusing on the number of times each call was made within a category or program versus the weight each call would have as a distinguishing factor. In order to achieve this balance, we first created histograms of the system calls within each application. 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.

B. Cyclomatic complexity

Cyclomatic complexity was selected as a potentially useful feature as it was expected to aid in both 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. Cyclomatic complexity, in this case, is defined by the following equation:

$$C = edges - nodes + 2P \tag{1}$$

where P is the number of exit nodes and C is the cyclomatic complexity. This equation is applied to an application on a per-function basis. The results are then averaged to produce the cyclomatic complexity for an entire program. The mean of all the cyclomatic complexity averages is calculated to produce the categories complexity baseline. Test programs are matched based on smallest difference between a category's complexity and their own average complexity.

C. Bonding

Bonding is a concept presented in [10]. This metric has been useful in distinguishing many graph types, such as social graphs, and we wanted to investigate its applicability in distinguishing control flow graphs. Bonding is calculated with a formula that takes the following form:

$$B = \frac{6 * \#triangles}{\#length\_two\_paths} \tag{2}$$

As with complexity, the bonding values are calculated on a per-function basis. 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. The function bonding values are averaged to find the program values, and a category average is again evaluated for each. Test programs are matched based on smallest difference in bonding values.

D. Byte n-grams

The byte n-grams attribute, for the purposes of our experiments, used 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 I shows the results of the n-grams feature selection with n values of two, three, and four. Standard deviation values are shown in parentheses. The results were obtained from using 80% of each data set as the training set and 20% as the testing set.

TABLE I. BYTE n-GRAM PERFORMANCE WITH VARIED n VALUES, AVERAGED OVER 30 RUNS.

n	**Versions**	**Compilers**
2	95.26% (1.61)	59.55% (9.25)
3	95.09% (1.34)	92.69% (3.48)
4	94.21% (1.61)	92.88% (4.20)

Trigrams of the programs in each category were placed 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, we attempt to capture some of those scenarios. 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. Testing programs are matched to the category with which they have the highest number of histogram values in common.

E. Opcode n-grams

Opcode n-grams are gathered by producing n-grams from the instruction opcodes (excluding operands) of a disassembled application. Again, trigrams were used exclusively in our experiments. Using bigrams or quadgrams did not significantly improve the results. Table II shows the comparisons of each n value. Standard deviation values are shown in parentheses. The histogram intersection technique used with byte n-grams, as we presumed, did not produce viable results. Instead, the opcode n-grams 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 was determined through experimentation as it produced the most optimal results in our dataset.

TABLE II. OPCODE n-GRAM PERFORMANCE WITH VARIED n VALUES, AVERAGED OVER 30 RUNS.

n	**Versions**	**Compilers**
2	33.68% (3.27)	59.55% (8.23)
3	58.95% (4.23)	71.09% (4.96)
4	59.29% (4.35)	71.67% (5.02)

F. Individual attribute performance

Each feature was individually tested 30 times over both datasets with randomized training and testing groups for each run. The averages of the results are shown in Figure 1.

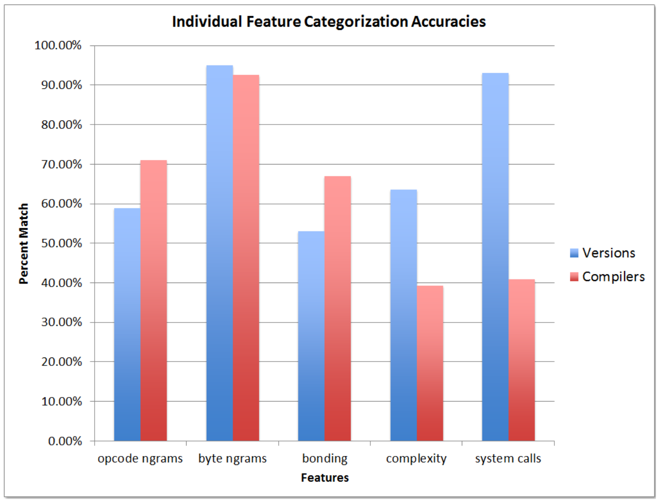

Fig. 1. Individual feature tests run on both data sets, averaged over 30 runs. 80% of the data was dedicated to the training set, and the rest comprised the testing set

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. Our goal was to use some combination of these features to achieve fast, consistent results with high accuracy.

G. GP algorithm

A GP algorithm was developed to evolve decision trees that 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 above. 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 contains 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.

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 2 illustrates this concept.

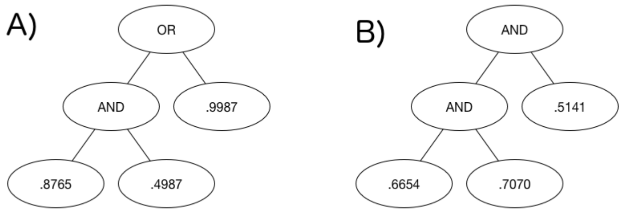

Fig. 2. A fuzzy logic example showing that using a threshold, such as $\geq .5$ signifying true for a boolean operation, does not distinguish categories well. Both of the trees in this example would evaluate to true. Using the max/min rules, however, Tree A evaluates to .9987 while Tree B evaluates to .5141. This is used to prioritize matches.

A fitness function is needed in order for the GP algorithm to assess the quality of each tree in the population. 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.

IV. Experimental Setup

A GP generally requires a lot of parameters, all of which benefit greatly from tuning. In this case, parameters were hand-tuned. 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. Throughout the GP evolution, parsimony pressure was applied to the trees to 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 III shows the parameters applied to the GP for each problem.

TABLE III. GP parameters

Parameter	**Versions**	**Compilers**
μ	100	100
λ	20	20
max depth	2	2
selection	k-tournament	k-tournament
survival	k-tournament	k-tournament
k	7	7
crossover	single-point	single-point
mutation	sub-tree	sub-tree
mutation rate	.1	.2
termination	5000 generations	5000 generations

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

V. Results

Figure 3 shows the matching accuracy of the GP method for both datasets, averaged over 30 runs.

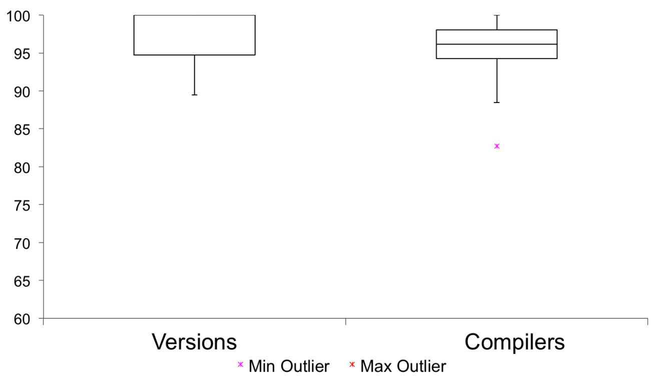

Fig. 3. Percent match with decision trees – versions max = 100%, versions min = 89.47%; compilers max = 100%, compilers min = 82.69%

The average accuracy of this method was higher than the accuracy of any single feature running alone. The n-grams performed only slightly worse, however. We used t-tests in order to show that this method is a significant improvement over using only the n-grams feature. F-tests showed that unequal variance should be assumed for the versions data, but not the compilers data. In both t-tests, t Stat was greater than t Critical Two-Tail. The results are summarized in the tables below.

TABLE IV. F-TEST: TWO-SAMPLE FOR VARIANCES - VERSIONS

Parameter	Decision Trees	n-Grams
Mean	0.9719298246	0.950877193
Variance	0.0012863374	0.0001783042
df	29	29
F	72142.8571	
P($F \leq f$) One-Tail	1.4186E-63	
F Critical One-Tail	1.86081144	

TABLE V. F-TEST: TWO-SAMPLE FOR VARIANCES - COMPILERS

Parameter	Decision Trees	n-Grams
Mean	0.9525641026	0.9269230769
Variance	0.0017020336	0.001208937
df	29	29
F	1.407876231	
P($F \leq f$) One-Tail	0.181150886	
F Critical One-Tail	1.860811435	

TABLE VI. t-TEST: TWO-SAMPLE ASSUMING UNEQUAL VARIANCE - VERSIONS

Parameter	Decision Trees	n-Grams
Mean	0.9719298246	0.950877193
Variance	0.0012863374	0.0001783042
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 VII. t-TEST: TWO-SAMPLE ASSUMING EQUAL VARIANCE - COMPILERS

Parameter	Decision Trees	n-Grams
Mean	0.9525641026	0.9269230769
Variance	0.0017020336	0.001208937
Pooled Variance	0.0014554853	
Hypoth. Mean Dif.	0	
df	58	
t Stat	2.6030176593	
P($T \leq t$) Two-Tail	0.011713133	
t Critical Two-Tail	2.0017174841	

Figures 4, 5, and 6 contain some of the trees with the highest fitness values for certain categories.

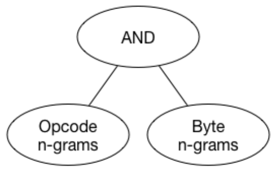

Fig. 4. Category: Versions - Nestopia

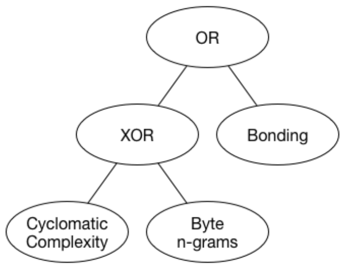

Fig. 5. Category: Compilers - GCC, No optimization

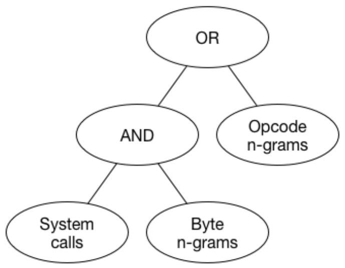

Fig. 6. Category: Versions - Pidgin

The trees vary in complexity based on the categories they represent. Although these small trees were all that was required to represent some of the categories within our datasets, larger trees with more intricate logic relationships are possible.

VI. DISCUSSION

The advantages of using a GP to produce distance heuristics for programs are quite clear. In both of the 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 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 GP evolution process executed as expected. Convergence was reached at a reasonable pace, and the best solution produced excellent results. The solutions produced by the algorithm all made intuitive sense based on the datasets used to produce them.

It should be noted from our individual feature tests and a majority of the GP trees, that byte n-grams in conjunction with the feature selection method listed previously produced the best results. This is presumably due to the types of categories used in this experiment. In both 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.

Even in this experiment, 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.

VII. Conclusion

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 paper aim to provide a means of quickly categorizing software based on a few key attributes. We explored the notion of evolving decision trees through GP. 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 or compiler 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 as the particular attributes used in this paper may not be sufficient discriminators of every kind of dataset.

The future work of this research can take many logical paths. 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. While viable solutions exist for authorship attribution, it would be useful to compare this method with prior works. The true strength of the techniques discussed in this paper 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.

References

[1] J. F. Bowring, J. M. Rehg, and M. J. 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.

[2] E. K. Burke, M. R. Hyde, G. Kendall, G. Ochoa, E. Ozcan, and J. 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.

[3] E. K. Burke, M. R. Hyde, G. Kendall, and J. 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.

[4] E. K. Burke, M. R. Hyde, G. Kendall, and J. 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.

[5] S. Cesare and Y. 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.

[6] B. Danilo, L. Martignoni, and M. 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] P. G. Espejo, S. Ventura, and F. 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.

[8] K. Gao, T. M. Khoshgoftaar, and H. 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.

[9] J. 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.

[10] O. Macindoe and W. 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.

[11] V. Nagarajan, R. Gupta, X. Zhang, M. Madou, and B. De Sutter. Matching Control Flow of Program Versions. In *IEEE International Conference on Software Maintenance*, ICSM, pages 84–93, October 2007.

[12] U.-M. 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.

[13] N. E. Rosenblum, B. P. Miller, and X. 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.

[14] N. E. Rosenblum, X. Zhu, and B. 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.

[15] N. E. Rosenblum, X. Zhu, B. P. Miller, and K. Hunt. Machine Learning-Assisted Binary Code Analysis. In *Workshop on Machine Learning in Adversarial Environments for Computer Security*, NIPS '07, December 2007.

[16] J. Sun and X.-Z. 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.