Poster: Atoms of Style: Identifying the Authors of Program Binaries

Chris McKnight, Kritika Iyer and Ian Goldberg
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario
Canada N2L 3G1
{cmcknigh,k4iyer,iang}@uwaterloo.ca

***Abstract*—Being able to identify the author of a program has many applications in both academic and commercial environments. In most use cases, the source code is readily available, and this is reflected in the literature, as previous work has mostly focused on source code analyses. In contrast, scant research has been carried out on identifying the authors of executable program binaries. This would be most applicable to the analysis of malware, but it also has applications in other areas, such as the evaluation of code obfuscators.**

Our research builds on and extends the work of previous studies on source code analysis to the realm of executable binaries. We take the approach of initially reverse engineering the executable code to extract a source code representation, before applying deanonymization techniques and analyzing the results. Our reasons for reverse engineering are to use the results to better understand whether elements of style are preserved through compilation, and of those, which are instrumental in identifying authors.

Our preliminary results suggest that some logical stylistic features remain after compilation, however there is far less variety in these features due in part to the loss of information after compiling, which goes beyond loss of indentation, comments and naming conventions. However, the results provide some encouragement that accuracy can be improved given a more detailed investigation and experimentation with different feature sets.

I. Introduction

A programmer might reasonably assume their anonymity is secured post compilation. After all, the compiler (if used without debug flags) applies its own optimizations - reordering statements and stripping out much identifiable content for the sake of performance. However, this is not the case, and previous work has shown that some aspects of programmer style are preserved after compilation [1]. Discovering what aspects of style are preserved after compilation would be of benefit to programmers desiring anonymity and would enable the development of better obfuscation tools. So far, only limited research has been published into the aspects of style that are preserved after compilation.

Consider the iconic "hello, world" [2] program. Even this simplest of programs can be written in a variety of ways, depending on the language used. If there are several idioms for solving a problem as trivial as *hello, world* (in fact there are many subtle variations of *hello, world*), then the number of different ways a more complex problem can be solved is orders of magnitude greater. Programmers are creatures of habit [3], who will attempt to solve similar looking problems in familiar ways. If different logical styles in a high-level programming language map to differences at the processor instruction set level, then identification should be possible.

II. Contributions

In our work we are looking to extend the ideas of code stylometry as used by Caliskan-Islam et al. [4] into the realm of binary program author attribution, checking whether the same techniques apply and to what extent the compiler acts to obfuscate the programmer's logical style. We have tested the hypothesis that some elements of style are retained after compilation and confirmed our belief that indeed some of these aspects are retained, but that the variety and distinguishability of the features are diminished, resulting in lower overall success. Moreover, we have managed to apply some of the techniques of code stylometry relating to control flow graphs and abstract syntax trees to the realm of binary program author attribution and identified many areas for future research in this relatively unexplored area.

III. Approach

A. Overview

The initial approach we took was to try and build on the work of previous researchers, by following a similar path, with comparable ground truth, then repeating the process but with the added intermediate step of compilation and decompilation. This provided us with a control group and allowed us to isolate only those features applicable/not applicable after compilation. We experimented with different classification and pattern matching techniques to evaluate which delivered superior results.

B. Obtaining Source Code

We were able to obtain a large corpus of source code from the Google Code Jam 2015 competition via a website[1] that maintains a database of entrants and solutions by programming language, represented country and round number. We used

[1] http://www.go-hero.net/jam/15

the Scrapy Python library[2] to crawl the site and download solutions in the C programming language.

C. Parsing

After compiling and decompiling our source code corpus, we were left with two data sets – one containing the original source code files and the other containing the decompiled code. In order to extract meaningful features from these files we employed the Joern[3] fuzzy parser and analysis tool, as proposed and developed by Yamaguchi et al. [5]. Joern produces *code property graphs*, or CPGs, which are logical graph structures that represent a combination of several program metrics in an Abstract Syntax Tree (AST).

D. Extracting Feature Sets

Once each source and decompiled program file had been parsed by Joern, it was ready for feature extraction. We began with the frequency of node types in the graph, along with a total node count. We also included a first/last seen metric based on location and the maximum child number (i.e. most siblings) seen in each graph, providing a primitive "breadth" property. We would like to explore extracting features based on the relationships between nodes as future work.

There was a noticeable difference in the number of attributes produced by the original source files (108) compared with the decompiled binary files (80). The mapping from high- to low-level is not one to one, therefore when reversing this process there will be some loss of information.

E. Classification

We used an approach similar to Caliskan-Islam et al. [4] for classifying the decompiled source code, using the machine learning platform Weka to run our tests. We experimented with multiple machine learning algorithms for classifying both the original and decompiled source code dataset, namely *k*-NN, Random Forest and Neural Network.

1) Dataset Specifications: The initial dataset consisted of 1,378 authors with a collective 2,356 solutions. We limited our study to only those authors with 4 or more solutions to ensure we had sufficient training data and an unseen sample for testing. This reduced the dataset to a total 332 collective solutions by 56 authors. Next, we grouped the 332 instances into 3 subsets of authors with a minimum of 4, 5 and 6 solutions, respectively, assuming the accuracy would increase with more training samples.

IV. Results

We present the results for the most accurate classification algorithm we utilized: Random Forest. We used a 10-fold cross validation, with 100-tree forests, for both the original and decompiled source code. We ran separate tests for each of the 3 data subsets previously mentioned, and for the original source code we were able to attain an approximate accuracy of 30-45% for the three datasets using Random Forest. For the decompiled code we achieved accuracies of 14-39%, as shown in Figure 1.

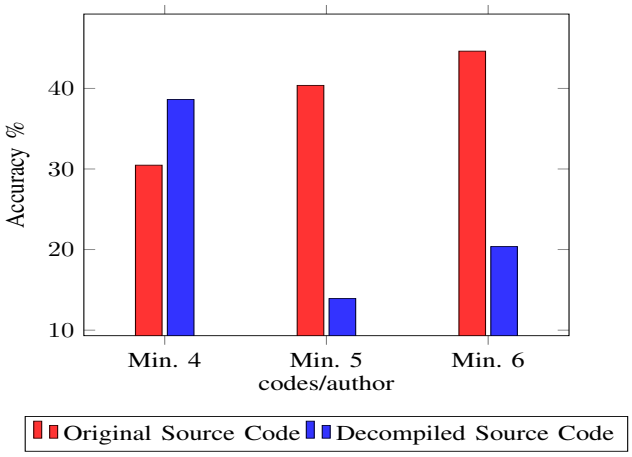

Fig. 1. Random Forest Classifier Results

In most of our tests, the decompiled dataset was classified with lower accuracy than the original source code. All our tests were conducted using the same parameters for both these sets. As mentioned previously, the attribute sets varied between the two datasets.

V. Future Work

We have identified a number of avenues for future work to explore this topic in greater depth. The preliminary results presented in this paper suggest there is truth to the hypothesis that logical style is preserved after compilation, however we realize that deriving the best feature set requires further refinement. We therefore would like to continue our studies and attempt to extract a more strongly identifying feature set from the property graphs produced by Joern, in particular the relationships between nodes. Examining the relationships provides context to the graph.

We would also like to experiment with different decompilers to see the effect this has on classification accuracy. Finally, it remains as future work to document the actual elements of style that are distinguishing.

[2]http://scrapy.org

[3]http://mlsec.org/joern/

References

[1] N. Rosenblum, X. Zhu, and B. P. Miller, "Who wrote this code? identifying the authors of program binaries," in *Proceedings of the 16th European Conference on Research in Computer Security*, ESORICS'11, (Berlin, Heidelberg), pp. 172–189, Springer-Verlag, 2011.

[2] B. Kernighan, *Programming in C: A Tutorial*. Bell Laboratories, Murray Hill, N. J., 1974.

[3] J. H. Hayes and J. Offutt, "Recognizing authors: an examination of the consistent programmer hypothesis," *Software Testing, Verification and Reliability*, vol. 20, no. 4, pp. 329–356, 2010.

[4] A. Caliskan-Islam, R. Harang, A. Liu, A. Narayanan, C. Voss, F. Yamaguchi, and R. Greenstadt, "De-anonymizing programmers via code stylometry," in *24th USENIX Security Symposium (USENIX Security 15)*, (Washington, D.C.), pp. 255–270, USENIX Association, August 2015.

[5] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Security and Privacy (SP), 2014 IEEE Symposium on*, pp. 590–604, IEEE, 2014.