arXiv:1512.08546v2 [cs.CR] 1 Mar 2016

When Coding Style Survives Compilation: De-anonymizing Programmers from Executable Binaries

Aylin Caliskan-Islam
Princeton University

Fabian Yamaguchi
University of Goettingen

Edwin Dauber
Drexel University

Richard Harang
U.S. Army Research Laboratory

Konrad Rieck
University of Goettingen

Rachel Greenstadt
Drexel University

Arvind Narayanan
Princeton University

Abstract

The ability to identify authors of computer programs based on their coding style is a direct threat to the privacy and anonymity of programmers. Previous work has examined attribution of authors from both source code and compiled binaries, and found that while source code can be attributed with very high accuracy, the attribution of executable binary appears to be much more difficult. Many potentially distinguishing features present in source code, e.g. variable names, are removed in the compilation process, and compiler optimization may alter the structure of a program, further obscuring features that are known to be useful in determining authorship.

We examine executable binary authorship attribution from the standpoint of machine learning, using a novel set of features that include ones obtained by decompiling the executable binary to source code. We show that many syntactical features present in source code do in fact survive compilation and can be recovered from decompiled executable binary. This allows us to add a powerful set of techniques from the domain of source code authorship attribution to the existing ones used for binaries, resulting in significant improvements to accuracy and scalability. We demonstrate this improvement on data from the Google Code Jam, obtaining attribution accuracy of up to 92% with 100 candidate programmers. We also demonstrate that our approach is robust to basic obfuscations, a range of compiler optimization settings, and binaries that have been stripped of their symbol tables. Finally, for the first time we are aware of, we demonstrate that authorship attribution can be performed on both obfuscated binaries, and real world code found "in the wild" by performing attribution on single-author GitHub repositories.

1 Introduction

If we encounter an executable binary sample in the wild, what can we learn from it? In this work, we show that the programmer's stylistic fingerprint, or coding style, is preserved in the compilation process and can be extracted from the executable binary. This means that it may be possible to infer the programmer's identity if we have a set of known potential candidate programmers, along with executable binary samples (or source code) known to be authored by these candidates.

Besides its intrinsic interest, programmer de-anonymization from executable binaries has implications for privacy and anonymity. Perhaps the creator of a censorship circumvention tool distributes it anonymously, fearing repression. Our work shows that such a programmer might be de-anonymized. Further, there are applications for software forensics, for example to help adjudicate cases of disputed authorship or copyright.

Rosenblum et al. studied this problem and presented encouraging early results [40]. We build on their work and make several advances to the state of the art, detailed in Section 4. First, whereas Rosenblum et al. extract structures such as control-flow graphs directly from the executable binaries, our work is the first to show that *automated decompilation* of executable binaries gives additional categories of useful features. Specifically, we generate *abstract syntax trees* of decompiled source code. Abstract syntax trees have been shown to greatly improve author attribution of source code [18]. We find that properties of these trees–including frequencies of different types of nodes, edges, and average depth of different types of nodes–also improve the accuracy of executable binary attribution techniques.

Second, we demonstrate that using multiple tools for disassembly and decompilation in parallel increases the accuracy of de-anonymization, possibly because different tools generate different representations of code that capture different aspects of the programmer's style. We present a machine-learning framework based on information gain for dimensionality reduction, followed by random-forest classification, that allows us to effectively use these disparate types of features in conjunction.

These innovations allow us to significantly improve the scale and accuracy of programmer de-anonymization

compared to Rosenblum et al.'s work. We performed experiments with a controlled dataset collected from Google Code Jam, allowing a direct comparison since the same dataset was used in the previous work. The results of these experiments are discussed in detail in Section 5. Specifically; for an equivalent accuracy we are able to distinguish between *thirty times* as many candidate programmers (600 vs. 20) while utilizing a smaller number of training samples per programmer. The accuracy of our method degrades gracefully as the number of programmers increases, and we present experiments with as many as 600 programmers.

Third, we find that traditional binary obfuscation, enabling compiler optimizations, or stripping debugging symbols in executable binaries results in only a modest decrease in classification accuracy. These results, described in Section 6, are an important step toward establishing the practical significance of the method.

The fact that coding style survives compilation is unintuitive, and may leave the reader wanting a "sanity check" or an explanation for why this is possible. In Section 5.9, we present several experiments that help illuminate this mystery. First, we show that decompiled source code *isn't* necessarily similar to the original source code in terms of the features that we use; rather, the feature vector obtained from disassembly and decompilation can be used to *predict*, using machine learning, the features in the original source code. Even if no individual feature is well preserved, there is enough information in the vector as a whole to enable this prediction. On average, the cosine similarity between the original feature vector and the reconstructed vector is over 80%. Further, we investigate factors that are correlated with coding style being well-preserved, and find that more skilled programmers are more fingerprintable. This suggests that programmers gradually acquire their own unique style as they gain experience.

All these experiments were carried out using the controlled Google Code Jam dataset; the availability of this dataset is a boon for research in this area since it allows us to develop and benchmark our results under controlled settings [10, 40]. Having done that, we present a case study with a real-world dataset collected from GitHub in Section 6.4. This data presents difficulties, particularly noise in ground truth because of library and code reuse. However, we show that we can handle a noisy dataset of 50 programmers found in the wild with 61% accuracy and further extend our method to tackle open world scenarios.

We emphasize that research challenges remain before programmer de-anonymization from executable binaries is fully ready for practical use. Many programs are authored by multiple programmers and may have gone through encryption. We have not yet performed experiments that model these scenarios. Also, while identifying the authors of executable malware binaries is an exciting potential application, attribution techniques will need to deal with obfuscated malware. Nonetheless, we believe that our results have significantly advanced the state of the art, and present immediate concerns for privacy and anonymity.

2 Problem Statement

In this work, we consider an analyst interested in determining the author of an executable binary purely based on its style. Moreover, we assume that the analyst only has access to executable binary samples each assigned to one of a set of candidate programmers.

Depending on the context, the analyst's goal might be defensive or offensive in nature. For example, the analyst might be trying to identify a misbehaving employee that violates the non-compete clause in his company by launching an application related to what he does at work. Similarly, a malware analyst might be interested in finding the author or authors of a malicious executable binary.

By contrast, the analyst might belong to a surveillance agency in an oppressive regime who tries to unmask anonymous programmers. The regime might have made it unlawful for its citizens to use certain types of programs, such as censorship-circumvention tools, and might want to punish the programmers of any such tools. If executable binary stylometry is possible, it means that compiling code is not a way of anonymization. Because of its potential dual use, executable binary stylometry is of interest to both security and privacy researchers.

In either (defensive or offensive) case, the analyst (or adversary) will seek to obtain labeled executable binary samples from each of these programmers who may have potentially authored the anonymous executable binary. The analyst proceeds by converting each labeled sample into a numerical feature vector, and subsequently deriving a classifier from these vectors using machine learning techniques. This classifier can then be used to attribute the anonymous executable binary to the most likely programmer.

Since we assume that a set of candidate programmers is known, we treat our main problem as a closed-world, supervised machine learning task. It is a multi-class machine learning problem where the classifier calculates the most likely author for the anonymous executable binary sample among multiple authors. We briefly present initial experiments on an open-world scenario in Section 6.5.

Additional Assumptions. For our experiments, we assume that we know the compiler used for a given program binary. Previous work has shown that with only 20 executable binary samples per compiler as training data,

it is possible to use a linear Conditional Random Field [27] to determine the compiler used with accuracy of 93% on average [42]. Other work has shown that by using pattern matching, library functions can be identified with precision and recall between 0.98 and 1.00 based on each of three criteria; compiler version, library version, and linux distribution [24].

In addition to knowing the compiler, we assume we know the optimization level used for compilation of the binary. Past work has shown that toolchain provenance, including compiler family, version, optimization, and source language, can be identified with a linear Conditional Random Field with accuracy of 99.9% for language, compiler family, and optimization and 91.9% for compiler version [41]. More recent work has looked at a stratified approach which, although having lower accuracy, is designed to be used at the function level to enable preprocessing for further tasks including authorship attribution [38]. Due to the success of these techniques, we make the assumption that these techniques will be used to identify the toolchain provenance of the executable binaries of interest and that our method will be trained using the same toolchain.

3 Related Work

Any domain of creative expression allows authors or creators to develop a unique style, and we might expect that there are algorithmic techniques to identify authors based on their style. This class of techniques is called stylometry. Natural-language stylometry, in particular, is well over a century old [31]. Other domains such as source code and music also have stylistic features, especially grammar. Therefore stylometry is applicable to these domains as well, often using strikingly similar techniques [11, 44].

Linguistic stylometry. The state of the art in linguistic stylometry is dominated by machine-learning techniques [e.g., 7, 8, 32]. Linguistic stylometry has been applied successfully to security and privacy problems, for example Narayanan et al. used stylometry to identify anonymous bloggers in large datasets, exposing privacy issues [32]. On the other hand, stylometry has also been used for forensics in underground cyber forums. In these forums the text consists of a mixture of languages and information about underground forum products, which makes it more challenging to identify personal writing style. Not only have the forum users been de-anonymized but also their multiple identities across and within forums have also been linked through stylometric analysis [8].

Authors may deliberately try to obfuscate or anonymize their writing style [7, 14, 30]. Brennan et al. [14] show how stylometric authorship attribution can be evaded with adversarial stylometry. They present two ways for adversarial stylometry, namely obfuscating writing style and imitating someone else's writing style. Afroz et al. [7] identify the stylistic changes in a piece of writing that has been obfuscated while [30] present a method to make writing style modification recommendations to anonymize an undisputed document.

Source code stylometry. Several authors have applied similar techniques to identify programmers based on source code [e.g., 15, 18, 34]. It has applications in software forensics and plagiarism detection.[1]

The features used for machine learning in these works range from simple byte-level [22] and word-level n-grams [16, 17] to more evolved structural features obtained from abstract syntax trees [18, 34]. In particular, Burrows et al. [17] present an approach based on n-grams that reaches an accuracy of 76.8% in differentiating 10 different programmers.

Similarly, Kothari et al. [26] combine n-grams with lexical markers such as the line length, to build programmer profiles that allow them to identify 12 authors with an accuracy of 76%. Lange and Mancoridis [28] further show that metrics based on layout and lexical features along with a genetic algorithm allow an accuracy of 75% to be obtained for 20 authors. Finally, Caliskan-Islam et al. [18] incorporate abstract syntax tree based structural features to represent programmers' coding style. They reach 94% accuracy in identifying 1,600 programmers of the Google Code Jam (GCJ) data set.

Executable binary stylometry. In contrast, identifying programmers from compiled code is considerably more difficult and has received little attention to date. Code compilation results in a loss of information and obstructs stylistic features. We are aware of only two prior works: Rosenblum et al. [40] and Alrabaee et al. [10], which we over perform in this work. Both [40] and [10] perform their evaluation and experiments on controlled corpora that are not noisy, such as the GCJ dataset and student homework assignments.

Rosenblum et al. [40] present two main machine learning tasks based on programmer de-anonymization. One is based on supervised classification to identify the authors of compiled code. The second machine learning approach they use is based on clustering to group together programs written by the same programmers. They incorporate a distance based similarity metric to differentiate between features related to programmer style to increase the clustering accuracy.

Rosenblum et al. [40] use the Paradyn project's Parse API for parsing executable binaries to get the instruction sequences and control flow graphs whereas we use four different resources to parse executable binaries to gen-

[1]Note that popular plagiarism-detection tools such as Moss [9] are not based on stylometry; rather they detect code that may have been copied, possibly with modifications. This is an orthogonal problem.

erate a richer representation. Their dataset consists of GCJ and homework assignment submissions. A support vector machine classifier [20] is trained on the numeric representations of varying numbers of executable binaries. GCJ programmers have eight to sixteen files and students have four to 7 files. Students collaborated on the homework assignments and the skeleton code was available.

Malware attribution. While the analysis of malware is a well developed field, authorship attribution of malware has received much less attention. Stylometry may have a role in this application, and this is a ripe area for future work. The difficulty in obtaining ground truth labels for samples has led much work in this area to focus on clustering malware in some fashion, and the wide range of obfuscation techniques in common use have led many researchers to focus on dynamic analysis rather than the static features we consider. The work of Marquis-Boire et al. [29] examines several static features intended to provide credible links between executable malware binary produced by the same authors, however many of these features are specific to malware, such as command and control infrastructure and data exfiltration methods, and the authors note that many must be extracted by hand. In dynamic analysis, the work of Pfeffer et al. [35] examines information obtained via both static and dynamic analysis of malware samples to organize code samples into lineages that indicate the order in which samples are derived from each other. Bayer et al. [12] convert detailed execution traces from dynamic analysis into more general behavioral profiles, which are then used to cluster malware into groups with related functionality and activity. Supervised methods are used by Rieck et al. [39] to match new instances of malware with previously observed families, again on the basis of dynamic analysis.

4 Approach

Our ultimate goal is to automatically recognize programmers of compiled code. We approach this problem using supervised machine learning, that is, we generate a classifier from training data of sample executable binaries with known authors. The advantage of such learning-based methods over techniques based on manually specified rules is that the approach is easily retargetable to any set of programmers for which sample executable binaries exist. A drawback is that the method is inoperable if samples are not available or too few in number. We study the amount of sample data necessary for successful classification in Section 5.

Data representation is critical to the success of machine learning. Accordingly, we design a feature set for executable binary authorship attribution with the goal of faithfully representing properties of executable binaries relevant for programmer style. We obtain this feature set by augmenting lower-level features extractable from disassemblers with additional string and symbol information, and, most importantly, incorporating higher-level syntactical features obtained from decompilers.

In summary, this results in a method consisting of the following four steps (see Figure 1).

- **Disassembly.** We begin by disassembling the program to obtain features based on machine code instructions, referenced strings, symbol information, and control flow graphs (Section 4.1).
- **Decompilation.** We proceed to translate the program into C-like pseudo code via decompilation. By subsequently passing the code to a fuzzy parser for C, we thus obtain abstract syntax trees from which syntactical features and n-grams can be extracted (Section 4.2).
- **Dimensionality Reduction.** With features from disassemblers and decompilers at hand, we select those among them that are particularly useful for classification by employing a standard feature selection technique based on information gain (Section 4.3).
- **Classification.** Finally, a random-forest classifier is trained on the corresponding feature vectors to yield a program that can be used for automatic executable binary authorship attribution (Section 4.4).

In the following, we describe these steps in greater detail and provide background information on static code analysis and machine learning where necessary.

4.1 Feature extraction via disassembly

As a first step, we disassemble the executable binary to extract low-level features that have been shown to be suitable for authorship attribution in previous work. In particular, we follow the example set by Rosenblum et al. and extract raw instruction traces from the executable binary [40]. In addition to this, disassemblers commonly make available symbol information as well as strings referenced in the code, both of which greatly simplify manual reverse engineering. We augment the feature set accordingly. Finally, we can obtain control flow graphs of functions from disassemblers, providing features based on program basic blocks. The required information neccessary to construct our feature set is obtained from the following two disassemblers.

- ***The Netwide Disassembler.*** We begin by exploring whether simple instruction decoding alone can already provide useful features for de-anonymization.

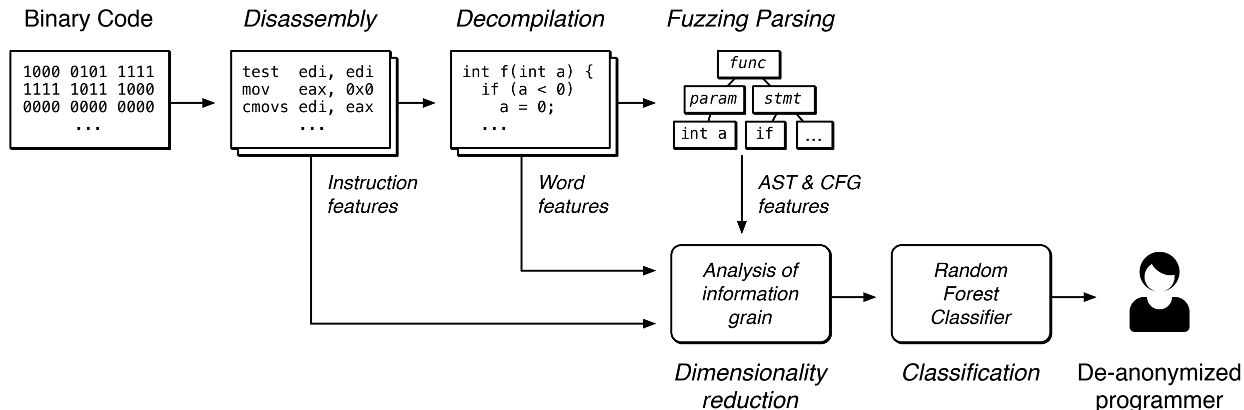

Figure 1: Overview of our method. Instructions, symbols, and strings are extracted using disassemblers (1), syntactical and control-flow features are obtained from decompilers (2). Dimensionality reduction is performed to obtain representative features (3). Finally, a random forest classifier is trained to de-anonymize programmers (4).

To this end, we process each executable binary using the netwide disassembler (*ndisasm*) [43], a rudimentary disassembler that is capable of decoding instructions but is unaware of the executable's file format. Due to this limitation, it resorts to simply decoding the executable binary from start to end, skipping bytes when invalid instructions are encountered. A problem with this approach is that no distinction is made between bytes that represent data, and bytes that represent code. We explore this simplistic approach nonetheless as these inaccuracies may not be relevant given the statistical nature of machine learning.

- ***The Radare2 Disassembler.*** We proceed to apply *radare2* [33], a state-of-the-art open-source disassembler based on the capstone disassembly framework [37]. In contrast to the *ndisasm*, *radare2* understands the executable binary format, allowing it to process relocation and symbol information in particular. This allows us to extract symbols from the dynamic (`.dynsym`) as well as the static symbol table (`.symtab`) where present, as well as any strings referenced in the code. Our approach thus gains knowledge over functions of dynamic libraries used in the code. Finally, *radare2* attemps to identify functions in code and generates corresponding control flow graphs.

The information provided by the two disassemblers is combined to obtain our disassembly feature set as follows: we tokenize the instruction traces of both disassemblers and extract token uni-grams, bi-grams, and tri-grams within a single line of assembly, and 6-grams, which span two consecutive lines of assembly. In addition, we extract single basic blocks of *radare2*'s control flow graphs, as well as pairs of basic blocks connected by control flow.

4.2 Feature extraction via decompilation

Decompilers are the second source of information that we consider for feature extraction in this work. In contrast to disassemblers, decompilers do not only uncover the program's machine code instructions, but additionally reconstruct higher level constructs in an attempt to translate an executable binary into equivalent source code. In particular, decompilers can reconstruct control structures such as different types of loops and branching constructs. We make use of these syntactical features of code as they have been shown to be valuable in the context of source code authorship attribution [18]. For decompilation, we employ the Hex-Rays decompiler [1].

Hex-Rays is a commercial state-of-the-art decompiler. It converts executable programs into a human readable C-like pseudo code to be read by human analysts. It is noteworthy that this code is typically significantly longer than the original source code. For example, decompiling an executable binary generated from 70 lines of source code with Hex-Rays produces on average 900 lines of decompiled code. We extract two types of features from this pseudo code: lexical features, and syntactical features. Lexical features are simply the word unigrams, which capture the integer types used in a program, names of library functions, and names of internal functions when symbol information is available. Syntactical features are obtained by passing the C-pseudo code to *joern* [46], a fuzzy parser for C that is capable of producing fuzzy abstract syntax trees (ASTs) from Hex-Rays pseudo code output. We derive syntactic features from the abstract syntax tree, which represent the grammatical structure of the program. Such features are (illustrated in Figure 2) AST node unigrams, labeled AST edges, AST node term frequency inverse document frequency, and AST node average depth. Previous work on source code authorship attribution [18, 45] shows that these features are highly effective in representing programming style.

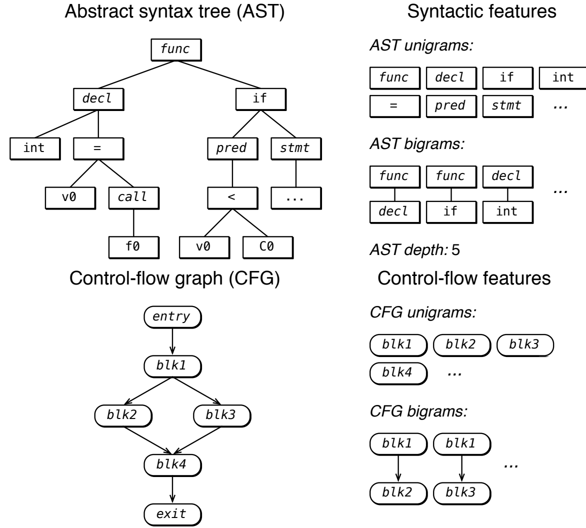

Figure 2: Feature extraction via decompilation and fuzzy parsing: the C-like pseudo code produced by Hexrays is transformed into an abstract syntax tree and control-flow graph to obtain syntactic and control-flow features.

4.3 Dimensionality Reduction via information gain

Our feature extraction process produces a large amount of features, resulting in sparse feature vectors with thousands of elements. However, not all features are equally relevant to express a programmer's style. This makes it desirable to perform feature selection in order to obtain a more compact representation of the data that reduces the computational burden during classification. Moreover, sparse feature vectors may result in a large number of zero-valued attributes being selected during random forest's random subsampling of the attributes to select a best split. Reducing the dimensions of the feature set is also important for avoiding overfitting. One example to overfitting would be a rare assembly instruction uniquely identifying an author. For these reasons, we use information gain criteria to select the most informative attributes that represent each author as a class. This reduces vector size and sparsity while increasing accuracy and machine learning model training speed. For example, we get 750,000 features from the 900 executable binary samples of 100 programmers. If we use all of these features in classification, the accuracy is slightly above 30% because the random forest might be randomly selecting features with values of zero in the sparse feature vectors. Once the dimension of the feature vector is reduced, we get less than 2,000 information gain features. Extracting less than 2,000 features or training a machine learning model where each instance has fewer than 2,000 attributes is computationally efficient. On the other hand, no sparsity remains in the feature vectors after dimensionality reduction which is one reason for the performance benefits of dimensionality reduction. After dimensionality reduction, the correct classification accuracy of 100 programmers increases from 30% to 90%.

We employed the dimensionality reduction step using WEKA's [23] information gain [36] attribute selection criterion, which evaluates the difference between the entropy of the distribution of classes and the Shannon entropy of the conditional distribution of classes given a particular feature. Information gain can be thought of as measuring the amount of information that the observation of the value of an attribute gives about the class label associated with the example.

In order to reduce the total size and sparsity of the feature vector, we retained only those features that individually had non-zero information gain.

4.4 Classification

For classification, we used a random forest ensemble classifier [13]. Random forests are ensemble learners built from collections of decision trees, where each tree is trained on a subsample of the data obtained by randomly sampling N training samples with replacement, where N is the number of instances in the dataset.

During classification, each test example is classified via each of the trained decision trees by following the binary decisions made at each node until a leaf is reached, and the results are aggregated. The most populous class can be selected as the output of the forest for simple classification, or several possible classifications can be ranked according to the number of trees that 'voted' for the label in question when performing relaxed attribution (see Section 5.5).

We employed random forests with 500 trees, which empirically provided the best tradeoff between accuracy and processing time. Examination of numerous out of bag error values across multiple fits suggested that $(logM)+1$ random features (where M denotes the total number of features) at each split of the decision trees was in fact optimal in all of the experiments listed in Section 5, and was used throughout. Node splits were selected based on the information gain criteria, and all trees were grown to the largest extent possible, without pruning.

The data was analyzed via k-fold cross-validation, where the data was split into training and test sets stratified by author (ensuring that the number of code samples per author in the training and test sets was identical across authors). The parameter k varies according to datasets and is equal to the number of instances present from each author. The cross-validation procedure was repeated 10 times, each with a different random seed, and average results across all iterations are reported, ensuring that results are not biased by improbably easy or difficult

to classify subsets.

Following previous work [10, 40] in this area, we report our classification results in terms of accuracy. As programmer de-anonymization is a multi-class classification problem accuracy, or the true positive rate, represents the correct classification rate in the most meaningful way.

5 Experiments with Google Code Jam

In this section, we go over the details of the various experiments we performed to address the research question formulated in Section 2.

5.1 Dataset

We evaluate our executable binary authorship attribution method on a dataset based on the annual programming competition *Google Code Jam* [5]. It is an annual contest that thousands of programmers take part in each year, including professionals, students, and hobbyists from all over the world. The contestants implement solutions to the same tasks in a limited amount of time in a programming language of their choice. Accordingly, all the correct solutions have the same algorithmic functionality. There are two main reasons for choosing Google Code Jam competition solutions as an evaluation corpus. First, it enables us to directly compare our results to previous work on executable binary authorship attribution as both Alrabaee et al. [10] and Rosenblum et al. [40] evaluate their approaches on data from Google Code Jam (GCJ). Second, we eliminate the potential confounding effect of identifying programming task rather than programmer by identifying functionality properties instead of stylistic properties. GCJ is a less noisy and clean dataset known definitely to be single authored. GCJ solutions do not have significant dependencies outside of the standard library and contain few or no third party libraries.

We focus our analysis on compiled C++ code, the most popular programming language used in the competition. We collect the solutions from the years 2008 to 2014 along with author names and problem identifiers.

5.2 Code Compilation

To create our experimental datasets, we first compiled the source code with GNU Compiler Collection's gcc or g++ without any optimization to Executable and Linkable Format (ELF) 32-bit, Intel 80386 Unix binaries.

Next, to measure the effect of different compilation options, such as compiler optimization flags, we additionally compiled the source code with level-1, level-2, and level-3 optimizations, namely the O1, O2, and O3 flags. The compiler attempts to improve the performance and/or code size when the compiler flags are turned on. Optimization has the expense of increasing compilation time and complicating program debugging.

5.3 Dimensionality Reduction

We are interested in identifying features that represent coding style preserved in executable binaries. With the current approach, we extract 700,000 representations of code properties of 100 authors, but only a subset of these representations are the result of individual programming style. We are able to capture the features that represent each author's programming style that is preserved in executable binaries by applying information gain criteria to these 700,000 features. After applying information gain, we reduce the feature set size to approximately 1,600 to effectively represent coding style properties that were preserved in executable binaries. Considering the fact that we are reaching such high accuracies on de-anonymizing 100 programmers with 900 executable binary samples (discussed below), these features are providing strong representation of style that survives compilation. We also see that all of our feature sources, namely disassembly, CFG, and decompiled code are capturing the preserved coding style. To examine the potential for overfitting, we also consider the ability of this feature set to generalize to a different set of programmers (see Section 5.6), and show that it does so, further supporting our belief that these features effectively capture coding style.

5.4 We can de-anonymize programmers from their executable binaries.

This is the main experiment that demonstrates how de-anonymizing programmers from their executable binaries is possible. After preprocessing the dataset to generate the executable binaries without optimization, we further process the executable binaries to obtain the disassembly, control flow graphs, and decompiled source code. We then extract all the possible features detailed in Section 4. We take a set of 100 programmers who all have 9 executable binary samples. With 9-fold-cross-validation, the random forest classifier correctly classifies 900 test instances with 89.8% accuracy, which is significantly higher than the accuracies reached in previous work.

There is an emphasis on the number of folds used in these experiments because each fold corresponds to the implementation of the same algorithmic function by all the programmers in the GCJ dataset (e.g. all samples in fold 1 may be attempts by the various authors to solve a list sorting problem). Since we know that each fold corresponds to the same Code Jam problem, by using stratified cross validation without randomization, we ensure that all training and test samples contain the same algorithmic functions implemented by all of the programmers. The classifier uses the excluded fold in the testing phase, which contains executable binary samples that were generated from an algorithmic function that was *not*

previously observed in the training set for that classifier. Consequently, the only distinction between the test instances is the coding style of the programmer, without the potentially confounding effect of identifying an algorithmic function.

5.5 Relaxed Classification: In difficult scenarios, the classification task can be narrowed down to a small suspect set.

In Section 5.1, the previously unseen anonymous executable binary sample is classified such that it belongs to the most likely author's class. In cases where we have too many classes or the classification accuracy is lower than expected, we can relax the classification to *top–n* classification. In *top–n* relaxed classification, if the test instance belongs to one of the most likely *n* classes, the classification is considered correct. This can be useful in cases when an analyst or adversary is interested in finding a suspect set of *n* authors, instead of a direct *top–1* classification. Being able to scale down an authorship investigation for an executable binary sample of interest to a reasonable sized set of suspect authors among hundreds of authors greatly reduces the manual effort required by an analyst or adversary. Once the suspect set size is reduced, the analyst or adversary could adhere to content based dynamic approaches and reverse engineering to identify the author of the executable binary sample. Figure 3 shows how correct classification accuracies approach 100% as the classification is relaxed to top-10 in binaries that have been optimized at different levels.

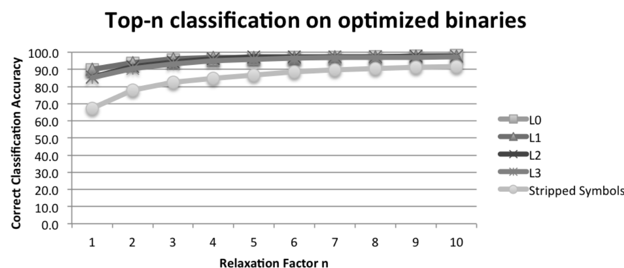

Figure 3: De-anonymizing 100 Programmers

5.6 The feature set selected via information gain works and is validated across different sets of programmers.

In our earlier experiments, we trained the classifier on the same set of executable binaries that we used for selecting features via information gain. The high number of starting features from which we select our final feature set via information gain does raise the potential concern of overfitting. To examine this, we applied this main feature set to a different set of programmers and executable binaries. If we are able to reach accuracies similar to what we got earlier, we can conclude that these information gain features do generalize to other programmers and problems, and therefore are not overfitting to the 100 programmers they were generated from. This also suggests that the information gain features in general capture programmer style.

Recall that analyzing 900 executable binary samples of the 100 programmers resulted in about 700,000 features, and after dimensionality reduction, we are left with 1,600 information gain features. We picked a different (non-overlapping) set of 100 programmers and performed another de-anonymization experiment in which the feature selection step was omitted, using instead the information gain features obtained from the original experiment. This resulted in very similar accuracies: we obtained 92,2% accuracy in de-anonymizing the validation set using features selected via the main development set, compared to the 89.8% we achieve with the main development set. The ability of the information gain features to generalize beyond the dataset which guided their selection strongly supports the assertion that these features obtained from the main set of 100 programmers are not overfitting, do actually represent coding style in executable binaries, and can be used across different datasets.

5.7 Large Scale De-anonymization: We can de-anonymize 600 programmers from their executable binaries.

We would like to see how well our method scales up to 600 users. An analyst with a large set of labeled samples might be interested in performing large scale de-anonymization. For this experiment, we use 600 contestants from GCJ with 9 files. We only extract the information gain features from the 600 users. This reduces the amount of time required for feature extraction. On the other hand, this experiment shows how well the information gain features represent overall programming style. The results of large scale programmer de-anonymization in Figure 4, show that our method can scale to larger datasets with the initial set of features with a surprisingly small drop on accuracy.

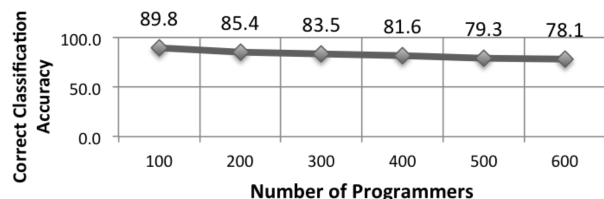

Figure 4: Large Scale Programmer De-anonymization

5.8 We advance the state of executable binary authorship attribution.

Rosenblum et al. [40] present the largest scale evaluation of executable binary authorship attribution in related work. [40]'s largest dataset contains 191 programmers with at least 8 training samples per programmer. We compare our results with [40]'s and in Table 1 show how we advance the state of the art both in accuracy and on larger datasets. [40] use 1,900 coding style features to represent coding style whereas we use 1,600 features, which might suggest that our features are more powerful in representing coding style that is preserved in executable binaries.

Related Work	Number of Programmers	Number of Training Samples	Accuracy
Rosenblum et al.	**20**	8-16	77%
This work	**600**	8	78%
Rosenblum et al.	20	**8-16**	77%
This work	20	**8**	94%
Rosenblum et al.	100	8-16	**61%**
This work	100	8	**92%**
Rosenblum et al.	191	8-16	**51%**
This work	191	8	**86%**
This work	**600**	8	**78%**

Table 1: Comparison to Previous Results

5.9 Programmer style is preserved in executable binaries.

We show throughout the results that it is possible to de-anonymize programmers from their executable binaries with a high accuracy. To quantify how stylistic features are preserved in executable binaries, we calculated the correlation of stylistic source code features and decompiled code features. We used the stylistic source code features from previous work [18] on de-anonymizing programmers from their source code. We took the most important 150 features in coding style that consist of AST node frequency, AST node average depth, AST node bigram frequency, AST node TFIDF, word unigram recency, and C++ keyword frequency. For each executable binary sample, we have the corresponding source code sample. We extract 150 information gain features from the original source code. We extract decompiled source code features from the decompiled executable binaries. For each executable binary instance, we set one corresponding information gain feature as the class to predict and we calculate the correlation between the decompiled executable binary features and the class value. A random forest classifier with 500 trees predicts the class value of each instance, and then Pearson's correlation coefficient is calculated between the predicted and original values. The correlation has a mean of 0.32 and ranges from -0.12 to 0.69 for the most important 150 features.

To see how well we can reconstruct the original source code features from decompiled executable binary features, we reconstructed the 900 instances with 150 features that represent the highest information gain features. We calculated the cosine similarity between the original 900 instances and the reconstructed instances after normalizing the features to unit distance. The cosine similarity for these instances is in Figure 5, where a cosine similarity of 1 means the two feature vectors are identical. The high values (average of 0.81) in cosine similarity suggest that the reconstructed features are similar to the original features. When we calculate the cosine distance between the feature vectors of the original source code and the corresponding decompiled code's feature vectors (*no predictions*), the average cosine distance is 0.35. This result suggests that the predicted features are much similar to original source code than the features extracted from decompiled code but decompiled code still preserves transformed forms of the original source code features well enough to reconstruct the original source code features.

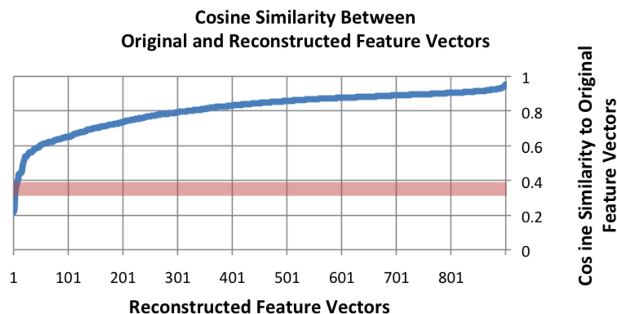

Figure 5: Feature Transformations: Each data point on the x-axis is a different executable binary sample. Each y-axis value is the cosine distance between the feature vector extracted from the original source code and the feature vector that tries to *predict* the original features. The average value of these 900 cosine distance measurements is 0.81.

5.10 Programmer skill set has an effect on coding style that is preserved in executable binaries.

In order to investigate the effect of programmer skill set on coding style that is preserved in executable binaries, we took two sets with 20 programmers. We considered the GCJ contestants who were able to advance to more difficult rounds as more advanced programmers as opposed to contestants that were eliminated in easier rounds. The subset of 20 programmers used for this ex-

periment is not as large compared to the other ones used during evaluation because there were a limited number of people who were able to advance to the more difficult rounds. The programmers with more advanced skill sets were able to solve 14 problems and the programmers that had a less advanced skill set were only able to solve 7 problems. All of the 40 programmers had implemented the same 7 problems from the easiest rounds. We were able to identify the more advanced 20 programmers with 95.8% accuracy while we identified the less advanced 20 programmers with 90.1% accuracy. This might indicate that, programmers who are advanced enough to answer 14 problems likely have more unique coding styles compared to contestants that were only able to solve the first 7 problems.

To investigate the possibility that contestants who are able to advance further in the rounds have more unique coding styles, we performed a second round of experiments on comparable datasets. We took a dataset with 6 solution files and 20 authors and also a dataset that contains these 6 files but has 12 files in total and 20 programmers. We were able to identify the more advanced 20 programmers with 92.5% accuracy while we identified the less advanced 20 programmers with 88.6% accuracy. These results suggest that programmer skill set has an effect on coding style, and this effect on coding style is preserved in compilation.

A = #authors, F = max #problems completed			
N = #problems included in dataset (N $\leq$ F)			
A = 20			
F = 14	**F = 7**	**F = 12**	**F = 6**
N = 7 easier	**N = 7**	**N = 6 easier**	**N = 6**
Average accuracy after 10 iterations			
95.8	90.1[1]	92.5	88.6[1]
[1]**Drop in accuracy due to programmer skill set.**			

Table 2: Effect of Programmer Skill Set on Coding Style Preserved in Executable Binaries

6 Experiments with Real World Scenarios

6.1 Programmers of optimized executable binaries can be de-anonymized.

In Section 5, we discussed how we evaluated our approach on a controlled and clean real world dataset. Section 5 shows how we advance over all the previous methods that were all evaluated with clean datasets such as GCJ or homework assignments. In this section, we investigate a more complicated dataset which has been optimized during compilation, where the executable binary samples have been normalized further during compilation.

Compiling with optimization tries to minimize or maximize some attributes of an executable computer program. The goal of optimization is to minimize the time it takes to execute a program or to minimize the amount of memory a program occupies. The compiler applies optimizing transformations which are algorithms that take a program and transform it to a semantically equivalent program that uses fewer resources.

GCC has predefined optimization levels that turn on sets of optimization flags. Compilation with optimization level-1, tries to reduce code size and execution time, takes more time and much more memory for large functions than compilation with no optimizations. Compilation with optimization level-2 optimizes more than level-1 optimization, uses all level-1 optimization flags and more. Level-2 optimization performs all optimizations that do not involve a space-speed tradeoff. Level-2 optimization increases compilation time and performance of the generated code when compared to optimization with level-1. Level-3 optimization yet optimizes more than both level-1 and level-2.

This work shows that programming style features survive compilation without any optimizations. As compilation with optimizations transforms code further, we investigate how much programming style is preserved in executable binaries that have gone through compilation with optimization. Our results summarized in Table 3 show that programming style is preserved to a great extent even in the most aggressive level-3 optimization. This shows that programmers of optimized executable binaries can be de-anonymized and optimization is not a highly effective code anonymization method.

Number of Programmers	**Number of Training Samples**	**Compiler Optimization Level**	**Accuracy**
100	8	None	89.8%
100	8	1	90.1%
100	8	2	86.2%
100	8	3	85.7%

Table 3: Programmer De-anonymization with Compiler Optimization

6.2 Removing symbol information does not anonymize executable binaries.

To investigate the relevance of symbol information for classification accuracy, we repeat our experiments with 100 authors presented in the previous section on *fully stripped executable binaries*, that is, executable binaries where symbol information is missing completely. We obtain these executable binaries using the standard utility *GNU strip* on each executable binary sample prior to analysis. Upon removal of symbol information, without any optimizations, we notice a drop in classification accuracy by 23%, showing that stripping symbol information from executable binaries is not effective enough to anonymize an executable binary sample.

6.3 We can de-anonymize programmers from obfuscated binaries.

We are furthermore interested in finding out whether our method is capable of dealing with simple binary obfuscation techniques as implemented by tools such as Obfuscator-LLVM [25]. These obfuscators substitute instructions by other semantically equivalent instructions, they introduce bogus control flow, and can even completely flatten control flow graphs.

For this experiment, we consider a set of 100 programmers from the GCJ data set, who all have 9 executable binary samples. This is the same data set as considered in our main experiment (see Section 5.4), however, we now apply all three obfuscation techniques implemented by Obfuscator-LLVM to the samples prior to learning and classification.

We proceed to train a classifier on obfuscated samples. This approach is feasible in practice as an analyst who has only non-obfuscated samples available can easily obfuscate them to obtain the necessary obfuscated samples for classifier training. Using the same information gain features as in Section 5.4, we obtain an accuracy of 86.2% in correctly classifying authors, which is only a mild drop in comparison to the 89.8% accuracy observed without obfuscation.

6.4 De-anonymization *in the Wild*

To better assess the applicability of our programmer de-anonymization approach *in the wild*, we extend our experiments to code collected from real open-source programs as opposed to solutions for programming competitions. To this end, we automatically collected source files from the popular open-source collaboration platform GitHub [6]. Starting from a seed set of popular repositories, we traversed the platform to obtain C/C++ repositories that meet the following criteria. Only one author has committed to the repository. The repository is popular as indicated by the presence of at least 5 *stars*, a measure of popularity for repositories on GitHub. Moreover, it is sufficiently large, containing a total of 200 lines at least. The repository is not a fork of another repository, nor is it named 'linux', 'kernel', 'osx', 'gcc', 'llvm', 'next', as these repositories are typically copies of the so-named projects.

We cloned the repositories meeting these criteria and collect only C/C++ files for which the main author has contributed at least 5 commits and the commit messages do not contain the word 'signed-off', a message that typically indicates that the code is written by another person. An author and her files are included in the dataset only if she has written at least 10 different files. In the final step, we manually verified ground truth on authorship for the selected files to make sure that they do not show any clear signs of code reuse from other projects. Table 4 shows the statistics of the resulting dataset.

Type	Amount
Authors	161
Repositories	439
Files	3,438
Repositories / Author	2 – 8
Files / Author	2 – 344

Table 4: Single Authored Github Repositories

We subsequently compile the collected projects to obtain object files for each of the selected source files. We perform our experiment on object files as opposed to entire binaries as these are the binary representations of the source files we can clearly attribute to one author.

For different reasons, compiling code may not be possible for a project, e.g., the code may not be in a compilable state, it may not be compilable for our target platform (32 bit Intel, Linux), or the necessary files to setup a working build environment can no longer be obtained. Despite these difficulties, we are able to generate 1,075 object files from 90 different authors, where the number of object files per author ranges from 2 to 24, with most authors having at least 9 samples. We used 50 of these authors that have 6 to 15 files to perform a machine learning experiment with more balanced class sizes.

We extract the information gain features selected from GCJ data from this GitHub dataset. The corresponding classifier reaches an accuracy of 60.7% in correctly identifying the authors of executable binary samples.

Being able to de-anonymize programmers in the wild by using fewer than 2,000 stylistic features obtained from our clean evaluation dataset is a promising step towards attacking more challenging real world de-anonymization problems.

6.5 Have I seen this programmer before?

While attempting to de-anonymize programmers in real world settings, we cannot be certain that we have formerly encountered code samples from the programmers in the test set. As a mechanism to check whether an anonymous test file belongs to one of the candidate programmers in the training set, we extend our method to an open world setting by incorporating classification confidence thresholds. In random forests, the class probability or classification confidence that executable binary B is of class i is calculated by taking the percentage of trees in the random forest that voted for that particular class, as follows:

$$P(B_i) = \frac{\sum_j V_j(i)}{|T|_f} \tag{1}$$

There are multiple ways to assess classifier confidence and we devise a method that calculates the classification

confidence by using classification margins. In this setting, the classification margin of a single instance is the difference between the highest and second highest $P(B_i)$. The first step towards attacking an open world classification task is identifying the confidence threshold of the classifier for classification verification. As long as we determine a confidence threshold based on training data, we can calculate the probability that an instance belongs to one of the programmers in the training set and accordingly accept or reject the classification.

We performed 900 classifications in a 100-class problem to determine the confidence threshold based on the training data. The accuracy was 92.7%. There were 66 misclassifications whose mean classification margin was 0.15 and it ranged from 0 to 0.4, except the 4 outliers that can be seen in Figure 6. By picking 0.4 as an aggressive threshold, we can reject 62 false positives out of 66 and manually inspect classifications that are under the confidence threshold with the knowledge that most of the misclassifications lie in that range.

Now that we picked 0.4 as our classification margin threshold, which can be adjusted according to false positive and false negative tolerance, we can check how effective this method is in an open world setting. We take another set of 100 programmers with 900 samples. We classify these 900 samples with the closed world classifier that was trained in the first step on samples from a disjoint set of programmers. All of the 900 programmers are attributed to a programmer in the closed world classifier with a mean classification margin of 0.17. By using the verification threshold of 0.4, we can reject 811 of these classifications for manual inspection to see if they are false positives or if they do not belong to a programmer in the training set. These results are encouraging for extending our programmer de-anonymization method to open world settings where an analyst deals with many uncertainties under varying fault tolerance levels.

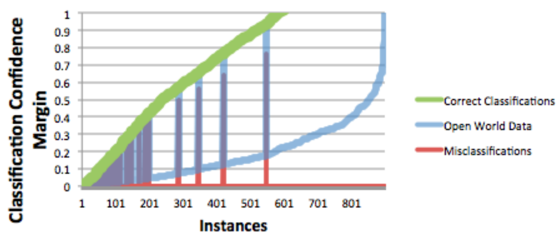

Figure 6: Confidence Thresholds from Classification Margin Curves for Classification Verification

7 Discussion

We consider two data sets: the Google Code Jam (GCJ) dataset, and a dataset based on GitHub repositories. Using the GitHub dataset, we show that we can perform programmer de-anonymization with executable binary authorship attribution in the wild. We de-anonymize GitHub programmers by using stylistic features obtained from the GCJ dataset. This supports the supposition that, in addition to its other useful properties for scientific analysis of attribution tasks, the GCJ dataset is a valid and useful proxy for real-world authorship attribution tasks.

The advantage of using the GCJ dataset is that we can perform the experiments in a strictly controlled environment where the most distinguishing difference between programmers' solutions is their programming style. Every contestant implements the same functionality, in a limited amount of time while at each round problems get more difficult. This provides the opportunity to control the difficulty level of the samples and the skill set of the programmers in the dataset. In contrast, GitHub offers a noisy dataset due to the collaborative nature of the samples. However, our results show that in cases where enough training data is available, high accuracies are still achievable.

Previous work shows that coding style is quite prevalent in source code. We were surprised to find that it is also preserved to a great degree in compiled source code. We can de-anonymize programmers from compiled source code with great accuracy, and furthermore, we can de-anonymize programmers from source code compiled with optimization. Optimizations transform executable binaries further to improve performance or memory usage. In our experiments, we see that even though basic obfuscation, optimization, or stripping symbols transforms executable binaries more than plain compilation, stylistic features are still preserved to a large degree.

In source code authorship attribution [18], programmers who can implement more sophisticated functionality have a more distinct programming style. We observe the same pattern in executable binary samples and gain some software engineering insights by analyzing stylistic properties of executable binaries.

Even though executable binaries look cryptic and difficult to analyze, we can still extract many useful features from them. We extract features from disassembly, control flow graphs, and also decompiled code to identify features relevant to only programming style. After dimensionality reduction with information gain, we see that each of the feature spaces provides programmer style information. All the feature spaces contain a total of more than 700,000 features for 900 executable binary samples of 100 authors. Approximately 1,600 features suffice to capture enough key features of coding style to enable robust authorship attribution. We see that the information gain features are valid in different datasets with different programmers, including optimized or obfuscated programmers. Also, the information gain features are helpful in scaling up the programmer

de-anonymization approach. While we can identify 100 programmers with 92% accuracy, we can de-anonymize 600 programmers with 78% accuracy using the same set of 1,600 features. 78% is a very high number for such a challenging de-anonymization task where the random chance of correctly identifying an author is 0.17%.

8 Limitations

Our experiments suggest that our method is able to assist in de-anonymizing programmers with significantly higher accuracy than state-of-the-art approaches. However, there are also assumptions that underlie the validity of our experiments as well as inherent limitations of our method that we discuss in the following paragraphs. First, we assume that our ground truth is correct, but in reality programs in GCJ or on GitHub might be written by programmers other than the stated programmer, or by multiple programmers. Such a ground truth problem would cause the classifier to train on noisy models which would lead to lower de-anonymization accuracy and a noisy representation of programming style. Second, many source code samples from GCJ contestants cannot be compiled. Consequently, we perform evaluation only on the subset of samples which can be compiled. This has two effects: first, we are performing attribution with fewer executable binary samples than the number of available source code samples. This is a limitation for our experiments but it is not a limitation for an attacker who first gets access to the executable binary instead of the source code. If the attacker gets access to the source code instead, she could perform regular source code authorship attribution. Second, we must assume that whether or not a code sample can be compiled does not correlate with the ease of attribution for that sample. Third, we only consider C/C++ code compiled using the GNU compiler `gcc` in this work, and assume that the executable binary format is the Executable and Linking Format. This is important to note as dynamic symbols are typically present in ELF binary files even after stripping of symbols, which may ease the attribution task relative to other executable binary formats that may not contain this information. We defer the investigation of the impact that other compilers, languages, and binary formats might have on the attribution task to future work.

Finally, while we show that our method is capable of dealing with simple binary obfuscation techniques, we do not consider executable binaries that are heavily obfuscated to hinder reverse engineering. While simple systems, such as packers [2] or encryption stubs that merely restore the original executable binary into memory during execution may be analyzed by simply recovering the unpacked or decrypted executable binary from memory, more complex approaches are becoming increasingly commonplace, particularly in malware. A wide range of anti-forensic techniques exist [21], including methods that are designed specifically to prevent easy access to the original bytecode in memory via such techniques as modifying the process environment block or triggering decryption on the fly via guard pages. Other techniques such as virtualization [3, 4] transform the original bytecode to emulated bytecode running on virtual machines, making decompilation both labor-intensive and error-prone. Finally, the use of specialized compilers that lack decompilers and produce nonstandard machine code – see [19] for an extreme but illustrative example – may likewise hinder our approach, particularly if the compiler is not generally available and cannot be fingerprinted. We leave the examination of these techniques, both with respect to their impact on authorship attribution and to possible mitigations, to future work.

9 Conclusion

De-anonymizing programmers has direct implications for privacy and security. The ability to attribute authorship to anonymous executable binary samples has applications in software forensics, and is an immediate concern for programmers that would like to remain anonymous. We de-anonymize 100 programmers from their executable binary samples with 92% accuracy, and 600 programmers with 78% accuracy. Our work is a significant advance over previous work and shows that coding style is preserved in compilation, contrary to the belief that compilation wipes away stylistic properties.

We discover 1,600 features that effectively represent coding style in executable binaries. We obtain this precise representation of coding style via two different disassemblers, control flow graphs, and a decompiler. With this comprehensive representation, we are able to re-identify GitHub authors from their executable binary samples in the wild, where we reach an accuracy of 61% for 50 programmers, even though these samples are noisy and products of collaborative efforts.

We show that programmer style is embedded in executable binaries to a surprising degree, even when it is obfuscated, generated with aggressive compiler optimizations, or when symbols are stripped. While compilation, basic binary obfuscation, optimization, and stripping of symbols reduce the accuracy of stylistic analysis, they are not effective in anonymizing coding style. In future work, we plan to investigate if stylistic properties can be completely stripped from binaries to render them anonymous. We also plan to look at different real world executable binary authorship attribution cases, such as identifying authors of malware, which go through a mixture of sophisticated obfuscation methods by combining polymorphism and encryption. Our results so far suggest that while stylistic analysis is unlikely to provide a "smoking gun" in the malware case, it may contribute significantly to attribution efforts.

Acknowledgment

This material is based on work supported by the ARO (U.S. Army Research Office) Grant W911NF-15-2-0055 and AWS in Education Research Grant award. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notice herein. This material is based on work supported by the ARO (U.S. Army Research Office) Grant W911NF-14-1- 0444, the DFG (German Research Foundation) under the project DEVIL (RI 2469/1-1), and AWS in Education Research Grant award. This research was supported in part by the Center for Information Technology and Policy at Princeton University.

References

[1] Hex-rays decompiler, November 2015. URL `https://www.hex-rays.com/`.

[2] Upx: the ultimate packer for executables, November 2015. URL `http://upx.sourceforge.net/`.

[3] November 2015. URL `http://vmpsoft.com/`.

[4] Oreans technology: Code virtualizer, 2015 November. URL `http://www.oreans.com/codevirtualizer.php`.

[5] The Google Code Jam Programming Competition. `https://code.google.com/codejam`, visited, November 2015.

[6] The github repository hosting service. `http://www.github.com`, visited, November 2015.

[7] Sadia Afroz, Michael Brennan, and Rachel Greenstadt. Detecting hoaxes, frauds, and deception in writing style online. In *Proc. of IEEE Symposium on Security and Privacy*. IEEE, 2012.

[8] Sadia Afroz, Aylin Caliskan-Islam, Ariel Stolerman, Rachel Greenstadt, and Damon McCoy. Doppelgänger finder: Taking stylometry to the underground. In *Proc. of IEEE Symposium on Security and Privacy*, 2014.

[9] A. Aiken et al. Moss: A system for detecting software plagiarism. *University of California–Berkeley. See www. cs. berkeley. edu/aiken/moss. html*, 9, 2005.

[10] Saed Alrabaee, Noman Saleem, Stere Preda, Lingyu Wang, and Mourad Debbabi. Oba2: an onion approach to binary code authorship attribution. *Digital Investigation*, 11, 2014.

[11] Eric Backer and Peter van Kranenburg. On musical stylometry—a pattern recognition approach. *Pattern Recognition Letters*, 26 (3):299–309, 2005.

[12] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. Scalable, behavior-based malware clustering. In *NDSS*, volume 9, pages 8–11. Citeseer, 2009.

[13] Leo Breiman. Random forests. *Machine Learning*, 45(1), 2001.

[14] Michael Brennan, Sadia Afroz, and Rachel Greenstadt. Adversarial stylometry: Circumventing authorship recognition to preserve privacy and anonymity. *ACM Transactions on Information and System Security (TISSEC)*, 15(3):12–1, 2012.

[15] S. Burrows. Source code authorship attribution. 2010.

[16] Steven Burrows and Seyed MM Tahaghoghi. Source code authorship attribution using n-grams. In *Proc. of the Australasian Document Computing Symposium*, 2007.

[17] Steven Burrows, Alexandra L Uitdenbogerd, and Andrew Turpin. Application of information retrieval techniques for source code authorship attribution. In *Database Systems for Advanced Applications*, 2009.

[18] A. Caliskan-Islam, R. Harang, A. Liu, A. Narayanan, C. Voss, F. Yamaguchi, and R. Greenstadt. De-anonymizing programmers via code stylometry. In *Proc. of the USENIX Security Symposium*, 2015.

[19] Chris Domas. M/o/vfuscator, November 2015. URL `https://github.com/xoreaxeaxeax/movfuscator`.

[20] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. Liblinear: A library for large linear classification. *Journal of Machine Learning Research (JMLR)*, 9, 2008.

[21] Peter. Ferrie. Anti-unpacker tricks–part one. Virus Bulletin (2008): 4.

[22] Georgia Frantzeskou, Efstathios Stamatatos, Stefanos Gritzalis, and Sokratis Katsikas. Effective identification of source code authors using byte-level information. In *Proc. of the International Conference on Software Engineering*. ACM, 2006.

[23] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The weka data mining software: an update. *SIGKDD Explor. Newsl.*, 11, 2009.

[24] Emily R Jacobson, Nathan Rosenblum, and Barton P Miller. Labeling library functions in stripped binaries. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*, pages 1–8. ACM, 2011.

[25] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-LLVM – software protection for the masses. In *Proc. of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15*, 2015.

[26] Jay Kothari, Maxim Shevertalov, Edward Stehle, and Spiros Mancoridis. A probabilistic approach to source code authorship identification. In *Information Technology, 2007. ITNG'07. Fourth International Conference on*. IEEE, 2007.

[27] John Lafferty, Andrew McCallum, and Fernando CN Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. 2001.

[28] Robert Charles Lange and Spiros Mancoridis. Using code metric histograms and genetic algorithms to perform author identification for software forensics. In *Proceedings of the Annual Conference on Genetic and Evolutionary Computation*. ACM, 2007.

[29] Morgan Marquis-Boire, Marion Marschalek, and Claudio Guarnieri. Big game hunting: The peculiarities in nation-state malware research. In *Proc. of Black Hat USA*, 2015.

[30] Andrew WE McDonald, Sadia Afroz, Aylin Caliskan, Ariel Stolerman, and Rachel Greenstadt. Use fewer instances of the letter "i": Toward writing style anonymization. In *Privacy Enhancing Technologies*, pages 299–318. Springer Berlin Heidelberg, 2012.

[31] Thomas Corwin Mendenhall. The characteristic curves of composition. *Science*, pages 237–249, 1887.

[32] Arun Narayanan, Hristo Paskov, Neil Zhenqiang Gong, John Bethencourt, Emil Stefanov, Eui Chul Richard Shin, and Dong Song. On the feasibility of internet-scale author identification. In *Proc. of IEEE Symposium on Security and Privacy*, 2012.

[33] pancake. Radare. `http://www.radare.org/`, visited, October 2015.

[34] Brian N Pellin. Using classification techniques to determine source code authorship. *White Paper: Department of Computer Science, University of Wisconsin*, 2000.

[35] Avi Pfeffer, Catherine Call, John Chamberlain, Lee Kellogg, Jacob Ouellette, Terry Patten, Greg Zacharias, Arun Lakhotia, Suresh Golconda, John Bay, et al. Malware analysis and attribution using genetic information. In *Malicious and Unwanted Software (MALWARE), 2012 7th International Conference on*, pages 39–45. IEEE, 2012.

[36] J.R. Quinlan. Induction of decision trees. *Machine learning*, 1 (1), 1986.

[37] Nguyen Anh Quynh. Capstone. `http://www.capstone-engine.org/`, visited, October 2015.

[38] Ashkan Rahimian, Paria Shirani, Saed Alrbaee, Lingyu Wang, and Mourad Debbabi. Bincomp: A stratified approach to compiler provenance attribution. *Digital Investigation*, 14:S146–S155, 2015.

[39] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Düssel, and Pavel Laskov. Learning and classification of malware behavior. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 108–125. Springer, 2008.

[40] N. Rosenblum, X. Zhu, and B. Miller. Who wrote this code? Identifying the authors of program binaries. *Computer Security–ESORICS 2011*, 2011.

[41] Nathan Rosenblum, Barton P Miller, and Xiaojin Zhu. Recovering the toolchain provenance of binary code. In *Proc. of the International Symposium on Software Testing and Analysis*. ACM, 2011.

[42] 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*. ACM, 2010.

[43] Simon Tatham and Julian Hall. The netwide disassembler: NDISASM. `http://www.nasm.us/doc/nasmdoca.html`, visited, October 2015.

[44] Peter van Kranenburg. Composer attribution by quantifying compositional strategies. In *ISMIR*, pages 375–376, 2006.

[45] Wilco Wisse and Cor Veenman. Scripting dna: Identifying the javascript programmer. *Digital Investigation*, 2015.

[46] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *Proc. of IEEE Symposium on Security and Privacy*, 2014.

Feature	Source	Number of Possible Features	Information Gain Features
word unigrams	hex-rays decompiled code	29,278	120
AST node TF*	hex-rays decompiled code	5,278	43
Labeled AST edge TF*	decompiled code	26,783	80
AST node TFIDF**	decompiled code	5,278	37
AST node average depth	decompiled code	5,278	39
C++ keywords	decompiled code	73	4
radare2 disassembly unigrams	radare disassembly	21,206	86
radare2 disassembly bigrams	radare disassembly	39,506	113
radare2 disassembly trigrams	radare disassembly	112,913	105
radare2 disassembly 6-grams	ndisasm disassembly	260,265	81
radare2 CFG node unigrams	radare disassembly	5,297	13
radare2 CFG edges	radare disassembly	10,246	5
ndisasm disassembly unigrams	ndisasm disassembly	5,383	64
ndisasm disassembly bigrams	ndisasm disassembly	14,305	138
ndisasm disassembly trigrams	ndisasm disassembly	5,237	90
ndisasm disassembly 6-grams	ndisasm disassembly	159,142	469
Total		705,468	1,655
TF = term frequency*			
*TFIDF** = term frequency inverse document frequency*			

Table 5: Programming Style Features in Executable Binaries