De-anonymizing Programmers via Code Stylometry

Aylin Caliskan-Islam
Drexel University

Richard Harang
U.S. Army Research Laboratory

Andrew Liu
University of Maryland

Arvind Narayanan
Princeton University

Clare Voss
U.S. Army Research Laboratory

Fabian Yamaguchi
University of Goettingen

Rachel Greenstadt
Drexel University

Abstract

Source code authorship attribution is a significant privacy threat to anonymous code contributors. However, it may also enable attribution of successful attacks from code left behind on an infected system, or aid in resolving copyright, copyleft, and plagiarism issues in the programming fields. In this work, we investigate machine learning methods to de-anonymize source code authors of C/C++ using coding style. Our Code Stylometry Feature Set is a novel representation of coding style found in source code that reflects coding style from properties derived from abstract syntax trees.

Our random forest and abstract syntax tree-based approach attributes more authors (1,600 and 250) with significantly higher accuracy (94% and 98%) on a larger data set (Google Code Jam) than has been previously achieved. Furthermore, these novel features are robust, difficult to obfuscate, and can be used in other programming languages, such as Python. We also find that (i) the code resulting from difficult programming tasks is easier to attribute than easier tasks and (ii) skilled programmers (who can complete the more difficult tasks) are easier to attribute than less skilled programmers.

1 Introduction

Do programmers leave fingerprints in their source code? That is, does each programmer have a distinctive "coding style"? Perhaps a programmer has a preference for spaces over tabs, or `while` loops over `for` loops, or, more subtly, modular rather than monolithic code.

These questions have strong privacy and security implications. Contributors to open-source projects may hide their identity whether they are Bitcoin's creator or just a programmer who does not want her employer to know about her side activities. They may live in a regime that prohibits certain types of software, such as censorship circumvention tools. For example, an Iranian programmer was sentenced to death in 2012 for developing photo sharing software that was used on pornographic websites [31].

The flip side of this scenario is that code attribution may be helpful in a forensic context, such as detection of ghostwriting, a form of plagiarism, and investigation of copyright disputes. It might also give us clues about the identity of malware authors. A careful adversary may only leave binaries, but others may leave behind code written in a scripting language or source code downloaded into the breached system for compilation.

While this problem has been studied previously, our work represents a qualitative advance over the state of the art by showing that Abstract Syntax Trees (ASTs) carry authorial 'fingerprints.' The highest accuracy achieved in the literature is 97%, but this is achieved on a set of only 30 programmers and furthermore relies on using programmer comments and larger amounts of training data [12, 14]. We match this accuracy on small programmer sets without this limitation. The largest scale experiments in the literature use 46 programmers and achieve 67.2% accuracy [10]. We are able to handle orders of magnitude more programmers (1,600) while using less training data with 92.83% accuracy. Furthermore, the features we are using are not trivial to obfuscate. We are able to maintain high accuracy while using commercial obfuscators. While abstract syntax trees can be obfuscated to an extent, doing so incurs significant overhead and maintenance costs.

Contributions. First, we use *syntactic features* for code stylometry. Extracting such features requires parsing of incomplete source code using a *fuzzy parser* to generate an *abstract syntax tree*. These features add a component to code stylometry that has so far remained almost completely unexplored. We provide evidence that these features are more fundamental and harder to obfuscate. Our complete feature set consists of a comprehensive set of around 120,000 layout-based, lexical, and syntactic features. With this complete feature set we are

able to achieve a significant increase in accuracy compared to previous work. Second, we show that we can scale our method to 1,600 programmers without losing much accuracy. Third, this method is not specific to C or C++, and can be applied to any programming language.

We collected C++ source of thousands of contestants from the annual international competition "Google Code Jam". A bagging (portmanteau of "bootstrap aggregating") classifier - random forest was used to attribute programmers to source code. Our classifiers reach 98% accuracy in a 250-class closed world task, 93% accuracy in a 1,600-class closed world task, 100% accuracy on average in a two-class task. Finally, we analyze various attributes of programmers, types of programming tasks, and types of features that appear to influence the success of attribution. We identified the most important 928 features out of 120,000; 44% of them are syntactic, 1% are layout-based and the rest of the features are lexical. 8 training files with an average of 70 lines of code is sufficient for training when using the lexical, layout and syntactic features. We also observe that programmers with a greater skill set are more easily identifiable compared to less advanced programmers and that a programmer's coding style is more distinctive in implementations of difficult tasks as opposed to easier tasks.

The remainder of this paper is structured as follows. We begin by introducing applications of source code authorship attribution considered throughout this paper in Section 2, and present our AST-based approach in Section 3. We proceed to give a detailed overview of the experiments conducted to evaluate our method in Section 4 and discuss the insights they provide in Section 5. Section 6 presents related work, and Section 7 concludes.

2 Motivation

Throughout this work, we consider an analyst interested in determining the programmer of an anonymous fragment of source code purely based on its style. To do so, the analyst only has access to labeled samples from a set of candidate programmers, as well as from zero or more unrelated programmers.

The analyst addresses this problem by converting each labeled sample into a numerical feature vector, in order to train a machine learning classifier, that can subsequently be used to determine the code's programmer. In particular, this abstract problem formulation captures the following five settings and corresponding applications (see Table 1). The experimental formulations are presented in Section 4.2.

We emphasize that while these applications motivate our work, we have not directly studied them. Rather, we formulate them as variants of a machine-learning (classification) problem. Our data comes from the Google Code Jam competition, as we discuss in Section 4.1. Doubtless there will be additional challenges in using our techniques for digital forensics or any of the other real-world applications. We describe some known limitations in Section 5.

Programmer De-anonymization. In this scenario, the analyst is interested in determining the identity of an anonymous programmer. For example, if she has a set of programmers who she suspects might be Bitcoin's creator, Satoshi, and samples of source code from each of these programmers, she could use the initial versions of Bitcoin's source code to try to determine Satoshi's identity. Of course, this assumes that Satoshi did not make any attempts to obfuscate his or her coding style. Given a set of probable programmers, this is considered a closed-world machine learning task with multiple classes where anonymous source code is attributed to a programmer. This is a threat to privacy for open source contributors who wish to remain anonymous.

Ghostwriting Detection. Ghostwriting detection is related to but different from traditional plagiarism detection. We are given a suspicious piece of code and one or more candidate pieces of code that the suspicious code may have been plagiarized from. This is a well-studied problem, typically solved using code similarity metrics, as implemented by widely used tools such as MOSS [6], JPlag [25], and Sherlock [24].

For example, a professor may want to determine whether a student's programming assignment has been written by a student who has previously taken the class. Unfortunately, even though submissions of the previous year are available, the assignments may have changed considerably, rendering code-similarity based methods ineffective. Luckily, stylometry can be applied in this setting—we find the most stylistically similar piece of code from the previous year's corpus and bring both students in for gentle questioning. Given the limited set of students, this can be considered a closed-world machine learning problem.

Software Forensics. In software forensics, the analyst assembles a set of candidate programmers based on previously collected malware samples or online code repositories. Unfortunately, she cannot be sure that the anonymous programmer is one of the candidates, making this an *open world* classification problem as the test sample might not belong to any known category.

Copyright Investigation. Theft of code often leads to copyright disputes. Informal arrangements of hired programming labor are very common, and in the absence of a written contract, someone might claim a piece of code was her own after it was developed for hire and delivered. A dispute between two parties is thus a two-class classification problem; we assume that labeled code from both programmers is available to the forensic expert.

Authorship Verification. Finally, we may suspect that a piece of code was not written by the claimed programmer, but have no leads on who the actual programmer might be. This is the authorship verification problem. In this work, we take the textbook approach and model it as a two-class problem where positive examples come from previous works of the claimed programmer and negative examples come from randomly selected unrelated programmers. Alternatively, anomaly detection could be employed in this setting, e.g., using a one-class support vector machine [see 30].

As an example, a recent investigation conducted by Verizon [17] on a US company's anomalous virtual private network traffic, revealed an employee who was outsourcing her work to programmers in China. In such cases, training a classifier on employee's original code and that of random programmers, and subsequently testing pieces of recent code, could demonstrate if the employee was the actual programmer.

In each of these applications, the adversary may try to actively modify the program's coding style. In the software forensics application, the adversary tries to modify code written by her to hide her style. In the copyright and authorship verification applications, the adversary modifies code written by another programmer to match his own style. Finally, in the ghostwriting application, two of the parties may collaborate to modify the style of code written by one to match the other's style.

Application	Learner	Comments	Evaluation
De-anonymization	Multiclass	Closed world	Section 4.2.1
Ghostwriting detection	Multiclass	Closed world	Section 4.2.1
Software forensics	Multiclass	Open world	Section 4.2.2
Copyright investigation	Two-class	Closed world	Section 4.2.3
Authorship verification	Two/One-class	Open world	Section 4.2.4

Table 1: Overview of Applications for Code Stylometry

We emphasize that code stylometry that is robust to adversarial manipulation is largely left to future work. However, we hope that our demonstration of the power of features based on the abstract syntax tree will serve as the starting point for such research.

3 De-anonymizing Programmers

One of the goals of our research is to create a classifier that automatically determines the most likely author of a source file. Machine learning methods are an obvious choice to tackle this problem, however, their success crucially depends on the choice of a feature set that clearly represents programming style. To this end, we begin by parsing source code, thereby obtaining access to a wide range of possible features that reflect programming language use (Section 3.1). We then define a number of different features to represent both syntax and structure of program code (Section 3.2) and finally, we train a random forest classifier for classification of previously unseen source files (Section 3.3). In the following sections, we will discuss each of these steps in detail and outline design decisions along the way. The code for our approach is made available as open-source to allow other researchers to reproduce our results[1].

3.1 Fuzzy Abstract Syntax Trees

To date, methods for source code authorship attribution focus mostly on sequential feature representations of code such as byte-level and feature level n-grams [8, 13]. While these models are well suited to capture naming conventions and preference of keywords, they are entirely language agnostic and thus cannot model author characteristics that become apparent only in the composition of language constructs. For example, an author's tendency to create deeply nested code, unusually long functions or long chains of assignments cannot be modeled using n-grams alone.

Addressing these limitations requires source code to be parsed. Unfortunately, parsing C/C++ code using traditional compiler front-ends is only possible for *complete code*, i.e., source code where all identifiers can be resolved. This severely limits their applicability in the setting of authorship attribution as it prohibits analysis of lone functions or code fragments, as is possible with simple n-gram models.

As a compromise, we employ the fuzzy parser *Joern* that has been designed specifically with incomplete code in mind [32]. Where possible, the parser produces *abstract syntax trees* for code fragments while ignoring fragments that cannot be parsed without further information. The produced syntax trees form the basis for our feature extraction procedure. While they largely preserve the information required to create n-grams or bag-of-words representations, in addition, they allow a wealth of features to be extracted that encode programmer habits visible in the code's structure.

As an example, consider the function `foo` as shown in Figure 1, and a simplified version of its corresponding abstract syntax tree in Figure 2. The function contains a number of common language constructs found in many programming languages, such as if-statements (line 3 and 7), return-statements (line 4, 8 and 10), and function call expressions (line 6). For each of these constructs, the abstract syntax tree contains a corresponding node. While the leaves of the tree make classical syntactic features such as keywords, identifiers and operators accessible, inner nodes represent operations showing

[1] https://github.com/calaylin/CodeStylometry

```
int foo()
{
   if((x < 0) || x > MAX)
      return -1;

   int ret = bar(x);
   if(ret != 0)
      return -1;
   else
   return 1;
}
```

Figure 1: Sample Code Listing

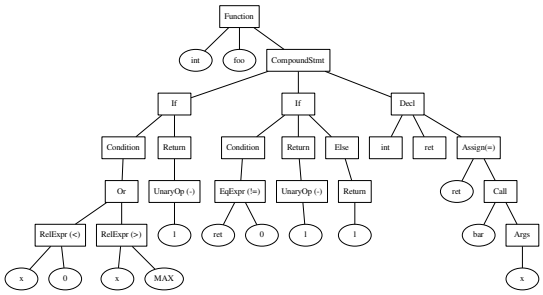

Figure 2: Corresponding Abstract Syntax Tree

how these basic elements are combined to form expressions and statements. In effect, the nesting of language constructs can also be analyzed to obtain a feature set representing the code's structure.

3.2 Feature Extraction

Analyzing coding style using machine learning approaches is not possible without a suitable representation of source code that clearly expresses program style. To address this problem, we present the *Code Stylometry Feature Set* (CSFS), a novel representation of source code developed specifically for code stylometry. Our feature set combines three types of features, namely *lexical features*, *layout features* and *syntactic features*. Lexical and layout features are obtained from source code while the syntactic features can only be obtained from ASTs. We now describe each of these feature types in detail.

3.2.1 Lexical and Layout Features

We begin by extracting numerical features from the source code that express preferences for certain identifiers and keywords, as well as some statistics on the use of functions or the nesting depth. Lexical and layout features can be calculated from the source code, without having access to a parser, with basic knowledge of the programming language in use. For example, we measure the number of functions per source line to determine the programmer's preference of longer over shorter functions. Furthermore, we tokenize the source file to obtain the number of occurrences of each token, so called *word unigrams*. Table 2 gives an overview of lexical features.

In addition, we consider *layout features* that represent code-indentation. For example, we determine whether the majority of indented lines begin with whitespace or tabulator characters, and we determine the ratio of whitespace to the file size. Table 3 gives a detailed description of these features.

Feature	Definition	Count
WordUnigramTF	Term frequency of word unigrams in source code	dynamic*
ln(num*keyword*/ length)	Log of the number of occurrences of *keyword* divided by file length in characters, where *keyword* is one of *do*, *else-if*, *if*, *else*, *switch*, *for* or *while*	7
ln(numTernary/ length)	Log of the number of ternary operators divided by file length in characters	1
ln(numTokens/ length)	Log of the number of word tokens divided by file length in characters	1
ln(numComments/ length)	Log of the number of comments divided by file length in characters	1
ln(numLiterals/ length)	Log of the number of string, character, and numeric literals divided by file length in characters	1
ln(numKeywords/ length)	Log of the number of unique keywords used divided by file length in characters	1
ln(numFunctions/ length)	Log of the number of functions divided by file length in characters	1
ln(numMacros/ length)	Log of the number of preprocessor directives divided by file length in characters	1
nestingDepth	Highest degree to which control statements and loops are nested within each other	1
branchingFactor	Branching factor of the tree formed by converting code blocks of files into nodes	1
avgParams	The average number of parameters among all functions	1
stdDevNumParams	The standard deviation of the number of parameters among all functions	1
avgLineLength	The average length of each line	1
stdDevLineLength	The standard deviation of the character lengths of each line	1
***About 55,000 for 250 authors with 9 files.**		

Table 2: Lexical Features

3.2.2 Syntactic Features

The syntactic feature set describes the properties of the language dependent abstract syntax tree, and keywords. Calculating these features requires access to an abstract syntax tree. All of these features are invariant to changes in source-code layout, as well as comments.

Table 4 gives an overview of our syntactic features. We obtain these features by preprocessing all C++ source files in the dataset to produce their abstract syntax trees.

Feature	Definition	Count
ln(numTabs/length)	Log of the number of tab characters divided by file length in characters	1
ln(numSpaces/length)	Log of the number of space characters divided by file length in characters	1
ln(numEmptyLines/length)	Log of the number of empty lines divided by file length in characters, excluding leading and trailing lines between lines of text	1
whiteSpaceRatio	The ratio between the number of whitespace characters (spaces, tabs, and newlines) and non-whitespace characters	1
newLineBeforeOpenBrace	A boolean representing whether the majority of code-block braces are preceded by a newline character	1
tabsLeadLines	A boolean representing whether the majority of indented lines begin with spaces or tabs	1

Table 3: Layout Features

An abstract syntax tree is created for each function in the code. There are 58 node types in the abstract syntax tree (see Appendix A) produced by *Joern* [33].

Feature	Definition	Count
MaxDepthASTNode	Maximum depth of an AST node	1
ASTNodeBigramsTF	Term frequency AST node bigrams	dynamic*
ASTNodeTypesTF	Term frequency of 58 possible AST node type excluding leaves	58
ASTNodeTypesTFIDF	Term frequency inverse document frequency of 58 possible AST node type excluding leaves	58
ASTNodeTypeAvgDep	Average depth of 58 possible AST node types excluding leaves	58
cppKeywords	Term frequency of 84 C++ keywords	84
CodeInASTLeavesTF	Term frequency of code unigrams in AST leaves	dynamic**
CodeInASTLeaves TFIDF	Term frequency inverse document frequency of code unigrams in AST leaves	dynamic**
CodeInASTLeaves AvgDep	Average depth of code unigrams in AST leaves	dynamic**
***About 45,000 for 250 authors with 9 files.**		
****About 7,000 for 250 authors with 9 files.**		
****About 4,000 for 150 authors with 6 files.**		
****About 2,000 for 25 authors with 9 files.**		

Table 4: Syntactic Features

The AST node bigrams are the most discriminating features of all. AST node bigrams are two AST nodes that are connected to each other. In most cases, when used alone, they provide similar classification results to using the entire feature set.

The term frequency (TF) is the raw frequency of a node found in the abstract syntax trees for each file. The term frequency inverse document frequency (TFIDF) of nodes is calculated by multiplying the term frequency of a node by inverse document frequency. The goal in using the inverse document frequency is normalizing the term frequency by the number of authors actually using that particular type of node. The inverse document frequency is calculated by dividing the number of authors in the dataset by the number of authors that use that particular node. Consequently, we are able to capture how rare of a node it is and weight it more according to its rarity.

The maximum depth of an abstract syntax tree reflects the deepest level a programmer nests a node in the solution. The average depth of the AST nodes shows how nested or deep a programmer tends to use particular structural pieces. And lastly, term frequency of each C++ keyword is calculated. Each of these features is written to a feature vector to represent the solution file of a specific author and these vectors are later used in training and testing by machine learning classifiers.

3.3 Classification

Using the feature set presented in the previous section, we can now express fragments of source code as numerical vectors, making them accessible to machine learning algorithms. We proceed to perform feature selection and train a random forest classifier capable of identifying the most likely author of a code fragment.

3.3.1 Feature Selection

Due to our heavy use of unigram term frequency and TF/IDF measures, and the diversity of individual terms in the code, our resulting feature vectors are extremely large and sparse, consisting of tens of thousands of features for hundreds of classes. The dynamic *Code stylometry feature set*, for example, produced close to 120,000 features for 250 authors with 9 solution files each.

In many cases, such feature vectors can lead to overfitting (where a rare term, by chance, uniquely identifies a particular author). Extremely sparse feature vectors can also damage the accuracy of random forest classifiers, as the sparsity may result in large numbers of zero-valued features being selected during the random subsampling of the features to select a best split. This reduces the number of 'useful' splits that can be obtained at any given node, leading to poorer fits and larger trees. Large, sparse feature vectors can also lead to slowdowns in model fitting and evaluation, and are often more difficult to interpret. By selecting a smaller number of more informative features, the sparsity in the feature vector can be greatly reduced, thus allowing the classifier to both produce more accurate results and fit the data faster.

We therefore employed a feature selection step using WEKA's information gain [26] criterion, which evaluates the difference between the entropy of the distribution of classes and the entropy of the conditional distribution of classes given a particular feature:

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

where A is the class corresponding to an author, H is Shannon entropy, and M_i is the i^{th} feature of the dataset. Intuitively, the information gain can be thought of as measuring the amount of information that the observation of the value of feature i gives about the class label associated with the example.

To reduce the total size and sparsity of the feature vector, we retained only those features that individually had non-zero information gain. (These features can be referred to as IG-CSFS throughout the rest of the paper.) Note that, as $H(A|M_i) \leq H(A)$, information gain is always non-negative. While the use of information gain on a variable-per-variable basis implicitly assumes independence between the features with respect to their impact on the class label, this conservative approach to feature selection means that we only use features that have demonstrable value in classification.

To validate this approach to feature selection, we applied this method to two distinct sets of source code files, and observed that sets of features with non-zero information gain were nearly identical between the two sets, and the ranking of features was substantially similar between the two. This suggests that the application of information gain to feature selection is producing a robust and consistent set of features (see Section 4 for further discussion). All the results are calculated by using CSFS and IG-CSFS. Using IG-CSFS on all experiments demonstrates how these features generalize to different datasets that are larger in magnitude. One other advantage of IG-CSFS is that it consists of a few hundred features that result in non-sparse feature vectors. Such a compact representation of coding style makes de-anonymizing thousands of programmers possible in minutes.

3.3.2 Random Forest Classification

We used the random forest ensemble classifier [7] as our classifier for authorship attribution. Random forests are ensemble learners built from collections of decision trees, each of which is grown by randomly sampling N training samples with replacement, where N is the number of instances in the dataset. To reduce correlation between trees, features are also subsampled; commonly $(logM)+1$ features are selected at random (without replacement) out of M, and the best split on these $(logM)+1$ features is used to split the tree nodes. The number of selected features represents one of the few tuning parameters in random forests: increasing the number of features increases the correlation between trees in the forest which can harm the accuracy of the overall ensemble, however increasing the number of features that can be chosen at each split increases the classification accuracy of each individual tree making them stronger classifiers with low error rates. The optimal range of number of features can be found using the out of bag (oob) error estimate, or the error estimate derived from those samples not selected for training on a given tree.

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 then aggregated. The most populous class can be selected as the output of the forest for simple classification, or classifications can be ranked according to the number of trees that 'voted' for a label when performing relaxed attribution (see Section 4.3.4).

We employed random forests with 300 trees, which empirically provided the best trade-off between accuracy and processing time. Examination of numerous oob 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 4), 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). 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. We report the average results across all iterations in the results, ensuring that they are not biased by improbably easy or difficult to classify subsets.

4 Evaluation

In the evaluation section, we present the results to the possible scenarios formulated in the problem statement and evaluate our method. The corpus section gives an overview of the data we collected. Then, we present the main results to programmer de-anonymization and how it scales to 1,600 programmers, which is an immediate privacy concern for open source contributors that prefer to remain anonymous. We then present the training data requirements and efficacy of types of features. The obfuscation section discusses a possible countermeasure to programmer de-anonymization. We then present possible machine learning formulations along with the verification section that extends the approach to an open world problem. We conclude the evaluation with generalizing the method to other programming languages and providing software engineering insights.

4.1 Corpus

One concern in source code authorship attribution is that we are actually identifying differences in coding style, rather than merely differences in functionality. Consider the case where Alice and Bob collaborate on an open source project. Bob writes user interface code whereas Alice works on the network interface and backend analytics. If we used a dataset derived from their project, we might differentiate differences between frontend and backend code rather than differences in style.

In order to minimize these effects, we evaluate our method on the source code of solutions to programming tasks from the international programming competition *Google Code Jam (GCJ)*, made public in 2008 [2]. The competition consists of algorithmic problems that need to be solved in a programming language of choice. In particular, this means that all programmers solve the same problems, and hence implement similar functionality, a property of the dataset crucial for code stylometry analysis.

The dataset contains solutions by professional programmers, students, academics, and hobbyists from 166 countries. Participation statistics are similar over the years. Moreover, it contains problems of different difficulty, as the contest takes place in several rounds. This allows us to assess whether coding style is related to programmer experience and problem difficulty.

The most commonly used programming language was C++, followed by Java, and Python. We chose to investigate source code stylometry on C++ and C because of their popularity in the competition and having a parser for C/C++ readily available [32]. We also conducted some preliminary experimentation on Python.

A validation dataset was created from 2012's GCJ competition. Some problems had two stages, where the second stage involved answering the same problem in a limited amount of time and for a larger input. The solution to the large input is essentially a solution for the small input but not vice versa. Therefore, collecting both of these solutions could result in duplicate and identical source code. In order to avoid multiple entries, we only collected the small input versions' solutions to be used in our dataset.

The programmers had up to 19 solution files in these datasets. Solution files have an average of 70 lines of code per programmer.

To create our experimental datasets that are discussed in further detail in the results section;
(i) We first partitioned the corpus of files by year of competition. The "main" dataset includes files drawn from 2014 (250 programmers). The "validation" dataset files come from 2012, and the "multi-year" dataset files come from years 2008 through 2014 (1,600 programmers).
(ii) Within each year, we ordered the corpus files by the round in which they were written, and by the problem within a round, as all competitors proceed through the same sequence of rounds in that year. As a result, we performed stratified cross validation on each program file by the year it was written, by the round in which the program was written, by the problems solved in the round, and by the author's highest round completed in that year.

Some limitations of this dataset are that it does not allow us to assess the effect of style guidelines that may be imposed on a project or attributing code with multiple/mixed programmers. We leave these interesting questions for future work, but posit that our improved results with basic stylometry make them worthy of study.

4.2 Applications

In this section, we will go over machine learning task formulations representing five possible real-world applications presented in Section 2.

4.2.1 Multiclass Closed World Task

This section presents our main experiment—de-anonymizing 250 programmers in the difficult scenario where all programmers solved the same set of problems. The machine learning task formulation for de-anonymizing programmers also applies to ghostwriting detection. The biggest dataset formed from 2014's Google Code Jam Competition with 9 solution files to the same problem had 250 programmers. These were the easiest set of 9 problems, making the classification more challenging (see Section 4.3.6). We reached 91.78% accuracy in classifying 250 programmers with the *Code Stylometry Feature Set*. After applying information gain and using the features that had information gain, the accuracy was 95.08%.

We also took 250 programmers from different years and randomly selected 9 solution files for each one of them. We used the information gain features obtained from 2014's dataset to see how well they generalize. We reached 98.04% accuracy in classifying 250 programmers. This is 3% higher than the controlled large dataset's results. The accuracy might be increasing because of using a mixed set of Google Code Jam problems, which potentially contains the possible solutions' properties along with programmers' coding style and makes the code more distinct.

We wanted to evaluate our approach and validate our method and important features. We created a dataset from 2012's Google Code Jam Competition with 250 programmers who had the solutions to the same set of 9 problems. We extracted only the features that had positive information gain in 2014's dataset that was used as

the main dataset to implement the approach. The classification accuracy was 96.83%, which is higher than the 95.07% accuracy obtained in 2014's dataset.

The high accuracy of validation results in Table 5 show that we identified the important features of code stylometry and found a stable feature set. This feature set does not necessarily represent the exact features for all possible datasets. For a given dataset that has ground truth information on authorship, following the same approach should generate the most important features that represent coding style in that particular dataset.

A = #programmers, F = max #problems completed		
N = #problems included in dataset (N $\leq$ F)		
A = 250 from 2014	**A = 250 from 2012**	**A = 250 all years**
F = 9 from 2014	**F = 9 from 2014**	**F $\geq$ 9 all years**
N = 9	**N = 9**	**N = 9**
Average accuracy after 10 iterations with IG-CSFS features		
95.07%	**96.83%**	**98.04%**

Table 5: Validation Experiments

4.2.2 Mutliclass Open World Task

The experiments in this section can be used in software forensics to find out the programmer of a piece of malware. In software forensics, the analyst does not know if source code belongs to one of the programmers in the candidate set of programmers. In such cases, we can classify the anonymous source code, and if the majority number of votes of trees in the random forest is below a certain threshold, we can reject the classification considering the possibility that it might not belong to any of the classes in the training data. By doing so, we can scale our approach to an open world scenario, where we might not have encountered the suspect before. As long as we determine a confidence threshold based on training data [30], we can calculate the probability that an instance belongs to one of the programmers in the set and accordingly accept or reject the classification.

We performed 270 classifications in a 30-class problem using all the features to determine the confidence threshold based on the training data. The accuracy was 96.67%. There were 9 misclassifications and all of them were classified with less than 15% confidence by the classifier. The class probability or classification confidence that source code fragment C is of class i is calculated by taking the percentage of trees in the random forest that voted for that particular class, as follows2:

$$P(C_i) = \frac{\sum_j V_j(i)}{|T|_f} \tag{2}$$

Where $V_j(i) = 1$ if the j^{th} tree voted for class i and 0 otherwise, and $|T|_f$ denotes the total number of trees in forest f. Note that by construction, $\sum_i P(C_i) = 1$ and $P(C_i) \geq 0 \; \forall \; i$, allowing us to treat $P(C_i)$ as a probability measure.

There was one correct classification made with 13.7% confidence. This suggests that we can use a threshold between 13.7% and 15% confidence level for verification, and manually analyze the classifications that did not pass the confidence threshold or exclude them from results.

We picked an aggressive threshold of 15% and to validate it, we trained a random forest classifier on the same set of 30 programmers 270 code samples. We tested on 150 different files from the programmers in the training set. There were 6 classifications below the 15% threshold and two of them were misclassified. We took another set of 420 test files from 30 programmers that were not in the training set. All the files from the 30 programmers were attributed to one of the 30 programmers in the training set since this is a closed world classification task, however, the highest confidence level in these classifications was 14.7%. The 15% threshold catches all the instances that do not belong to the programmers in the suspect set, gets rid of 2 misclassifications and 4 correct classifications. Consequently, when we see a classification with less than a threshold value, we can reject the classification and attribute the test instance to an unknown suspect.

4.2.3 Two-class Closed World Task

Source code author identification could automatically deal with source code copyright disputes without requiring manual analysis by an objective code investigator. A copyright dispute on code ownership can be resolved by comparing the styles of both parties claiming to have generated the code. The style of the disputed code can be compared to both parties' other source code to aid in the investigation. To imitate such a scenario, we took 60 different pairs of programmers, each with 9 solution files. We used a random forest and 9-fold cross validation to classify two programmers' source code. The average classification accuracy using CSFS set is 100.00% and 100.00% with the information gain features.

4.2.4 Two-class/One-class Open World Task

Another two-class machine learning task can be formulated for authorship verification. We suspect Mallory of plagiarizing, so we mix in some code of hers with a large sample of other people, test, and see if the disputed code gets classified as hers or someone else's. If it gets classified as hers, then it was with high probability really written by her. If it is classified as someone else's, it really was someone else's code. This could be an open

world problem and the person that originally wrote the code could be a previously unknown programmer.

This is a two-class problem with classes Mallory and others. We train on Mallory's solutions to problems a, b, c, d, e, f, g, h. We also train on programmer A's solution to problem a, programmer B's solution to problem b, programmer C's solution to problem c, programmer D's solution to problem d, programmer E's solution to problem e, programmer F's solution to problem f, programmer G's solution to problem g, programmer H's solution to problem h and put them in one class called ABCDEFGH. We train a random forest classifier with 300 trees on classes Mallory and ABCDEFGH. We have 6 test instances from Mallory and 6 test instances from another programmer ZZZZZZ, who is not in the training set.

These experiments have been repeated in the exact same setting with 80 different sets of programmers ABCDEFGH, ZZZZZZ and Mallorys. The average classification accuracy for Mallory using the CSFS set is 100.00%. ZZZZZZ's test instances are classified as programmer ABCDEFGH 82.04% of the time, and classified as Mallory for the rest of the time while using the CSFS. Depending on the amount of false positives we are willing to accept, we can change the operating point on the ROC curve.

These results are also promising for use in cases where a piece of code is suspected to be plagiarized. Following the same approach, if the classification result of the piece of code is someone other than Mallory, that piece of code was with very high probability not written by Mallory.

4.3 Additional Insights

4.3.1 Scaling

We collected a larger dataset of 1,600 programmers from various years. Each of the programmers had 9 source code samples. We created 7 subsets of this large dataset in differing sizes, with 250, 500, 750, 1,000, 1,250, 1,500, and 1,600 programmers. These subsets are useful to understand how well our approach scales. We extracted the specific features that had information gain in the main 250 programmer dataset from this large dataset. In theory, we need to use more trees in the random forest as the number of classes increase to decrease variance, but we used fewer trees compared to smaller experiments. We used 300 trees in the random forest to run the experiments in a reasonable amount of time with a reasonable amount of memory. The accuracy did not decrease too much when increasing the number of programmers. This result shows that information gain features are robust against changes in class and are important properties of programmers' coding styles. The following Figure 3 demonstrates how well our method scales. We are able to de-anonymize 1,600 programmers using 32GB memory within one hour. Alternately, we can use 40 trees and get nearly the same accuracy (within 0.5%) in a few minutes.

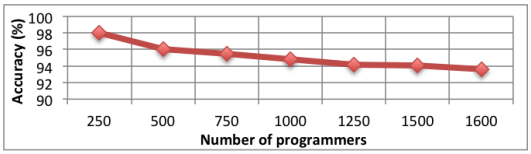

Figure 3: Large Scale De-anonymization

4.3.2 Training Data and Features

We selected different sets of 62 programmers that had F solution files, from 2 up to 14. Each dataset has the solutions to the same set of F problems by different sets of programmers. Each dataset consisted of programmers that were able to solve exactly F problems. Such an experimental setup makes it possible to investigate the effect of programmer skill set on coding style. The size of the datasets were limited to 62, because there were only 62 contestants with 14 files. There were a few contestants with up to 19 files but we had to exclude them since there were not enough programmers to compare them.

The same set of F problems were used to ensure that the coding style of the programmer is being classified and not the properties of possible solutions of the problem itself. We were able to capture personal programming style since all the programmers are coding the same functionality in their own ways.

Stratified F-fold cross validation was used by training on everyone's $(F-1)$ solutions and testing on the F^{th} problem that did not appear in the training set. As a result, the problems in the test files were encountered for the first time by the classifier.

We used a random forest with 300 trees and *(logM)+1* features with F-fold stratified cross validation, first with the *Code Stylometry Feature Set* (CSFS) and then with the CSFS's features that had information gain.

Figure 4 shows the accuracy from 13 different sets of 62 programmers with 2 to 14 solution files, and consequently 1 to 13 training files. The *CSFS* reaches an optimal training set size at 9 solution files, where the classifier trains on 8 $(F-1)$ solutions.

In the datasets we constructed, as the number of files increase and problems from more advanced rounds are included, the average line of code (LOC) per file also increases. The average lines of code per source code in the dataset is 70. Increased number of lines of code might have a positive effect on the accuracy but at the same time it reveals programmer's choice of program

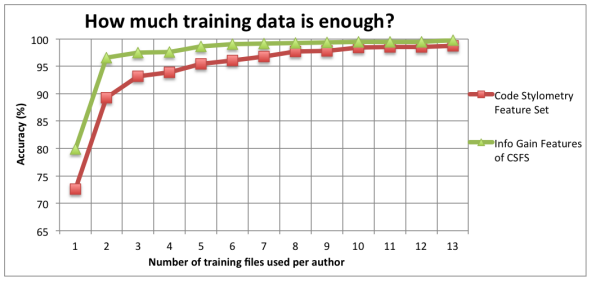

Figure 4: Training Data

length in implementing the same functionality. On the other hand, the average line of code of the 7 easier (76 LOC) or difficult problems (83 LOC) taken from contestants that were able to complete 14 problems, is higher than the average line of code (68) of contestants that were able to solve only 7 problems. This shows that programmers with better skills tend to write longer code to solve Google Code Jam problems. The mainstream idea is that better programmers write shorter and cleaner code which contradicts with line of code statistics in our datasets. Google Code Jam contestants are supposed to optimize their code to process large inputs with faster performance. This implementation strategy might be leading to advanced programmers implementing longer solutions for the sake of optimization.

We took the dataset with 62 programmers each with 9 solutions. We get 97.67% accuracy with all the features and 99.28% accuracy with the information gain features. We excluded all the syntactic features and the accuracy dropped to 88.89% with all the non-syntactic features and 88.35% with the information gain features of the non-syntactic feature set. We ran another experiment using only the syntactic features and obtained 96.06% with all the syntactic features and 96.96% with the information gain features of the syntactic feature set. Most of the classification power is preserved with the syntactic features, and using non-syntactic features leads to a significant decline in accuracy.

4.3.3 Obfuscation

We took a dataset with 9 solution files and 20 programmers and obfuscated the code using an off-the-shelf C++ obfuscator called stunnix [3]. The accuracy with the information gain code stylometry feature set on the obfuscated dataset is 98.89%. The accuracy on the same dataset when the code is not obfuscated is 100.00%. The obfuscator refactored function and variable names, as well as comments, and stripped all the spaces, preserving the functionality of code without changing the structure of the program. Obfuscating the data produced little detectable change in the performance of the classifier for this sample. The results are summarized in Table 6.

We took the maximum number of programmers, 20, that had solutions to 9 problems in C and obfuscated the code (see example in Appendix B) using a much more sophisticated open source obfuscator called Tigress [1]. In particular, Tigress implements *function virtualization*, an obfuscation technique that turns functions into interpreters and converts the original program into corresponding bytecode. After applying function virtualization, we were less able to effectively de-anonymize programmers, so it has potential as a countermeasure to programmer de-anonymization. However, this obfuscation comes at a cost. First of all, the obfuscated code is neither readable nor maintainable, and is thus unsuitable for an open source project. Second, the obfuscation adds significant overhead (9 times slower) to the runtime of the program, which is another disadvantage.

The accuracy with the information gain feature set on the obfuscated dataset is reduced to 67.22%. When we limit the feature set to AST node bigrams, we get 18.89% accuracy, which demonstrates the need for all feature types in certain scenarios. The accuracy on the same dataset when the code is not obfuscated is 95.91%.

Obfuscator	Programmers	Lang	Results w/o Obfuscation	Results w/ Obfuscation
Stunnix	20	C++	98.89%	100.00%
Stunnix	20	C++	98.89*%	98.89*%
Tigress	20	C	93.65%	58.33%
Tigress	20	C	95.91*%	67.22*%
*Information gain features				

Table 6: Effect of Obfuscation on De-anonymization

4.3.4 Relaxed Classification

The goal here is to determine whether it is possible to reduce the number of suspects using code stylometry. Reducing the set of suspects in challenging cases, such as having too many suspects, would reduce the effort required to manually find the actual programmer of the code.

In this section, we performed classification on the main 250 programmer dataset from 2014 using the information gain features. The classification was relaxed to a set of top R suspects instead of exact classification of the programmer. The relaxed factor R varied from 1 to 10. Instead of taking the highest majority vote of the decisions trees in the random forest, the highest R majority vote decisions were taken and the classification result was considered correct if the programmer was in the set of top R highest voted classes. The accuracy does not improve much after the relaxed factor is larger than 5.

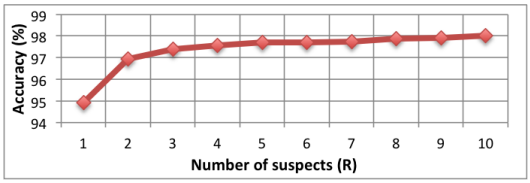

Figure 5: Relaxed Classification with 250 Programmers

4.3.5 Generalizing the Method

Features derived from ASTs can represent coding styles in various languages. These features are applicable in cases when lexical and layout features may be less discriminating due to formatting standards and reliance on whitespace and other 'lexical' features as syntax, such as Python's PEP8 formatting. To show that our method generalizes, we collected source code of 229 Python programmers from GCJ's 2014 competition. 229 programmers had exactly 9 solutions. **Using only the Python equivalents of syntactic features** listed in Table 4 and 9-fold cross-validation, the average accuracy is 53.91% for top-1 classification, 75.69% for top-5 relaxed attribution. The largest set of programmers to all work on the same set of 9 problems was 23 programmers. The average accuracy in identifying these 23 programmers is 87.93% for top-1 and 99.52% for top-5 relaxed attribution. The same classification tasks using the information gain features are also listed in Table 7. The overall accuracy in datasets composed of Python code are lower than C++ datasets. In Python datasets, we only used syntactic features from ASTs that were generated by a parser that was not fuzzy. The lack of quantity and specificity of features accounts for the decreased accuracy. The Python dataset's information gain features are significantly fewer in quantity, compared to C++ dataset's information gain features. Information gain only keeps features that have discriminative value all on their own. If two features only provide discriminative value when used together, then information gain will discard them. So if a lot of the features for the Python set are only jointly discriminative (and not individually discriminative), then the information gain criteria may be removing features that in combination could effectively discriminate between authors. This might account for the decrease when using information gain features. While in the context of other results in this paper the results in Table 7 appear lackluster, it is worth noting that even this preliminary test using only syntactic features has comparable performance to other prior work at a similar scale (see Section 6 and Table 9), demonstrating the utility of syntactic features and the relative ease of generating them for novel programming languages. Nevertheless, a CSFS equivalent feature set can be generated for other programming languages by implementing the layout and lexical features as well as using a fuzzy parser.

Lang.	Programmers	Classification	IG	Top-5	Top-5 IG
Python	23	87.93%	79.71%	99.52%	96.62
Python	229	53.91%	39.16%	75.69%	55.46

Table 7: Generalizing to Other Programming Languages

4.3.6 Software Engineering Insights

We wanted to investigate if programming style is consistent throughout years. We found the contestants that had the same username and country information both in 2012 and 2014. We assumed that these are the same people but there is a chance that they might be different people. In 2014, someone else might have picked up the same username from the same country and started using it. We are going to ignore such a ground truth problem for now and assume that they are the same people.

We took a set of 25 programmers from 2012 that were also contestants in 2014's competition. We took 8 files from their submissions in 2012 and trained a random forest classifier with 300 trees using CSFS. We had one instance from each one of the contestants from 2014. The correct classification of these test instances from 2014 is 96.00%. The accuracy dropped to 92.00% when using only information gain features, which might be due to the aggressive elimination of pairs of features that are jointly discriminative. These 25 programmers' 9 files from 2014 had a correct classification accuracy of 98.04%. These results indicate that coding style is preserved up to some degree throughout years.

To investigate problem difficulty's effect on coding style, we created two datasets from 62 programmers that had exactly 14 solution files. Table 8 summarizes the following results. A dataset with 7 of the easier problems out of 14 resulted in 95.62% accuracy. A dataset with 7 of the more difficult problems out of 14 resulted in 99.31% accuracy. This might imply that more difficult coding tasks have a more prevalent reflection of coding style. On the other hand, the dataset that had 62 programmers with exactly 7 of the easier problems resulted in 91.24% accuracy, which is a lot lower than the accuracy obtained from the dataset whose programmers were able to advance to solve 14 problems. 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 the dataset with

12 solution files and 62 programmers. A dataset with 6 of the easier problems out of 12 resulted in 91.39% accuracy. A dataset with 6 of the more difficult problems out of 12 resulted in 94.35% accuracy. These results are higher than the dataset whose programmers were only able to solve the easier 6 problems. The dataset that had 62 programmers with exactly 6 of the easier problems resulted in 90.05% accuracy.

A = #programmers, F = max #problems completed					
N = #problems included in dataset (N $\leq$ F)					
A = 62					
F = 14		**F = 7**	**F = 12**		**F = 6**
N = 7	**N = 7**	**N = 7**	**N = 6**	**N = 6**	**N = 6**
Average accuracy after 10 iterations while using CSFS					
99.31%	*95.62%*[2]	**91.24%**[1]	94.35%	*91.39%*[2]	**90.05%**[1]
Average accuracy after 10 iterations while using IG CSFS					
99.38%	*98.62%*[2]	**96.77%**[1]	96.69%	*95.43%*[2]	**94.89%**[1]
[1] **Drop in accuracy due to programmer skill set.**					
[2] **Coding style is more distinct in more difficult tasks.**					

Table 8: Effect of Problem Difficulty on Coding Style

5 Discussion

In this section, we discuss the conclusions we draw from the experiments outlined in the previous section, limitations, as well as questions raised by our results. In particular, we discuss the difficulty of the different settings considered, the effects of obfuscation, and limitations of our current approach.

Problem Difficulty. The experiment with random problems from random authors among seven years most closely resembles a real world scenario. In such an experimental setting, there is a chance that instead of only identifying authors we are also identifying the properties of a specific problem's solution, which results in a boost in accuracy.

In contrast, our main experimental setting where all authors have only answered the nine easiest problems is possibly the hardest scenario, since we are training on the same set of eight problems that all the authors have algorithmically solved and try to identify the authors from the test instances that are all solutions of the 9th problem. On the upside, these test instances help us precisely capture the differences between individual coding style that represent the same functionality. We also see that such a scenario is harder since the randomized dataset has higher accuracy.

Classifying authors that have implemented the solution to a set of difficult problems is easier than identifying authors with a set of easier problems. This shows that coding style is reflected more through difficult programming tasks. This might indicate that programmers come up with unique solutions and preserve their coding style more when problems get harder. On the other hand, programmers with a better skill set have a prevalent coding style which can be identified more easily compared to contestants who were not able to advance as far in the competition. This might indicate that as programmers become more advanced, they build a stronger coding style compared to novices. There is another possibility that maybe better programmers start out with a more unique coding style.

Effects of Obfuscation. A malware author or plagiarizing programmer might deliberately try to hide his source code by obfuscation. Our experiments indicate that our method is resistant to simple off-the-shelf obfuscators such as stunnix, that make code look cryptic while preserving functionality. The reason for this success is that the changes stunnix makes to the code have no effect on syntactic features, e.g., removal of comments, changing of names, and stripping of whitespace.

In contrast, sophisticated obfuscation techniques such as function virtualization hinder de-anonymization to some degree, however, at the cost of making code unreadable and introducing a significant performance penalty. Unfortunately, unreadability of code is not acceptable for open-source projects, while it is no problem for attackers interested in covering their tracks. Developing methods to automatically remove stylometric information from source code without sacrificing readability is therefore a promising direction for future research.

Limitations. We have not considered the case where a source file might be written by a different author than the stated contestant, which is a ground truth problem that we cannot control. Moreover, it is often the case that code fragments are the work of multiple authors. We plan to extend this work to study such datasets. To shed light on the feasibility of classifying such code, we are currently working with a dataset of git commits to open source projects. Our parser works on code fragments rather than complete code, consequently we believe this analysis will be possible.

Another fundamental problem for machine learning classifiers are mimicry attacks. For example, our classifier may be evaded by an adversary by adding extra dummy code to a file that closely resembles that of another programmer, albeit without affecting the program's behavior. This evasion is possible, but trivial to resolve when an analysts verifies the decision.

Finally, we cannot be sure whether the original author is actually a Google Code Jam contestant. In this case, we can detect those by a classify and then verify approach as explained in Stolerman et al.'s work [30]. Each classification could go through a verification step

to eliminate instances where the classifier's confidence is below a threshold. After the verification step, instances that do not belong to the set of known authors can be separated from the dataset to be excluded or for further manual analysis.

6 Related Work

Our work is inspired by the research done on authorship attribution of unstructured or semi-structured text [5, 22]. In this section, we discuss prior work on source code authorship attribution. In general, such work (Table 9) looks at smaller scale problems, does not use structural features, and achieves lower accuracies than our work.

The highest accuracies in the related work are achieved by Frantzeskou et al. [12, 14]. They used 1,500 7-grams to reach 97% accuracy with 30 programmers. They investigated the high-level features that contribute to source code authorship attribution in Java and Common Lisp. They determined the importance of each feature by iteratively excluding one of the features from the feature set. They showed that comments, layout features and naming patterns have a strong influence on the author classification accuracy. They used more training data (172 line of code on average) than us (70 lines of code). We replicated their experiments on a 30 programmer subset of our C++ data set, with eleven files containing 70 lines of code on average and no comments. We reach 76.67% accuracy with 6-grams, and 76.06% accuracy with 7-grams. When we used a 6 and 7-gram feature set on 250 programmers with 9 files, we got 63.42% accuracy. With our original feature set, we get 98% accuracy on 250 programmers.

The largest number of programmers studied in the related work was 46 programmers with 67.2% accuracy. Ding and Samadzadeh [10] use statistical methods for authorship attribution in Java. They show that among lexical, keyword and layout properties, layout metrics have a more important role than others which is not the case in our analysis.

There are also a number of smaller scale, lower accuracy approaches in the literature [9, 11, 18–21, 28], shown in Table 9, all of which we significantly outperform. These approaches use a combination of layout and lexical features.

The only other work to explore structural features is by Pellin [23], who used manually parsed abstract syntax trees with an SVM that has a tree based kernel to classify functions of two programmers. He obtains an average of 73% accuracy in a two class classification task. His approach explained in the white paper can be extended to our approach, so it is the closest to our work in the literature. This work demonstrates that it is non-trivial to use ASTs effectively. Our work is the first to use structural features to achieve higher accuracies at larger scales and the first to study how code obfuscation affects code stylometry.

There has also been some code stylometry work that focused on manual analysis and case studies. Spafford and Weeber [29] suggest that use of lexical features such as variable names, formatting and comments, as well as some syntactic features such as usage of keywords, scoping and presence of bugs could aid in source code attribution but they do not present results or a case study experiment with a formal approach. Gray et al. [15] identify three categories in code stylometry: the layout of the code, variable and function naming conventions, types of data structures being used and also the cyclomatic complexity of the code obtained from the control flow graph. They do not mention anything about the syntactic characteristics of code, which could potentially be a great marker of coding style that reveals the usage of programming language's grammar. Their case study is based on a manual analysis of three worms, rather than a statistical learning approach. Hayes and Offutt [16] examine coding style in source code by their consistent programmer hypothesis. They focused on lexical and layout features, such as the occurrence of semicolons, operators and constants. Their dataset consisted of 20 programmers and the analysis was not automated. They concluded that coding style exists through some of their features and professional programmers have a stronger programming style compared to students. In our results in Section 4.3.6, we also show that more advanced programmers have a more identifying coding style.

There is also a great deal of research on plagiarism detection which is carried out by identifying the similarities between different programs. For example, there is a widely used tool called Moss that originated from Stanford University for detecting software plagiarism. Moss [6] is able to analyze the similarities of code written by different programmers. Rosenblum et al. [27] present a novel program representation and techniques that automatically detect the stylistic features of binary code.

Related Work	# of Programmers	Results
Pellin [23]	2	73%
MacDonell et al.[21]	7	88.00%
Frantzeskou et al.[14]	8	100.0%
Burrows et al. [9]	10	76.78%
Elenbogen and Seliya [11]	12	74.70%
Kothari et al. [18]	12	76%
Lange and Mancoridis [20]	20	75%
Krsul and Spafford [19]	29	73%
Frantzeskou et al. [14]	30	96.9%
Ding and Samadzadeh [10]	46	67.2%
This work	**8**	**100.00%**
This work	**35**	**100.00%**
This work	**250**	**98.04%**
This work	**1,600**	**92.83%**

Table 9: Comparison to Previous Results

7 Conclusion and Future Work

Source code stylometry has direct applications for privacy, security, software forensics, plagiarism, copyright infringement disputes, and authorship verification. Source code stylometry is an immediate concern for programmers who want to contribute code anonymously because de-anonymization is quite possible. We introduce the first principled use of syntactic features along with lexical and layout features to investigate style in source code. We can reach 94% accuracy in classifying 1,600 authors and 98% accuracy in classifying 250 authors with eight training files per class. This is a significant increase in accuracy and scale in source code authorship attribution. In particular, it shows that source code authorship attribution with the *Code Stylometry Feature Set* scales even better than regular stylometric authorship attribution, as these methods can only identify individuals in sets of 50 authors with slightly over 90% accuracy [see 4]. Furthermore, this performance is achieved by training on only 550 lines of code or eight solution files, whereas classical stylometric analysis requires 5,000 words.

Additionally, our results raise a number of questions that motivate future research. First, as malicious code is often only available in binary format, it would be interesting to investigate whether syntactic features can be partially preserved in binaries. This may require our feature set to be improved in order to incorporate information obtained from control flow graphs.

Second, we would also like to see if classification accuracy can be further increased. For example, we would like to explore whether using features that have joint information gain alongside features that have information gain by themselves improve performance. Moreover, designing features that capture larger fragments of the abstract syntax tree could provide improvements. These changes (along with adding lexical and layout features) may provide significant improvements to the Python results and help generalize the approach further.

Finally, we would like to investigate whether code can be automatically normalized to remove stylistic information while preserving functionality and readability.

8 Acknowledgments

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. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the ARO, DFG, and AWS.

References

[1] The tigress diversifying c virtualizer, http://tigress.cs.arizona.edu.

[2] Google code jam, https://code.google.com/codejam, 2014.

[3] Stunnix, http://www.stunnix.com/prod/cxxo/, November 2014.

[4] ABBASI, A., AND CHEN, H. Writeprints: A stylometric approach to identity-level identification and similarity detection in cyberspace. *ACM Trans. Inf. Syst. 26*, 2 (2008), 1–29.

[5] AFROZ, S., BRENNAN, M., AND GREENSTADT, R. Detecting hoaxes, frauds, and deception in writing style online. In *Security and Privacy (SP), 2012 IEEE Symposium on* (2012), IEEE, pp. 461–475.

[6] AIKEN, A., ET AL. Moss: A system for detecting software plagiarism. *University of California–Berkeley. See www. cs. berkeley. edu/aiken/moss. html 9* (2005).

[7] BREIMAN, L. Random forests. *Machine Learning 45*, 1 (2001), 5–32.

[8] BURROWS, S., AND TAHAGHOGHI, S. M. Source code authorship attribution using n-grams. In *Proc. of the Australasian Document Computing Symposium* (2007).

[9] BURROWS, S., UITDENBOGERD, A. L., AND TURPIN, A. Application of information retrieval techniques for source code authorship attribution. In *Database Systems for Advanced Applications* (2009), Springer, pp. 699–713.

[10] DING, H., AND SAMADZADEH, M. H. Extraction of java program fingerprints for software authorship identification. *Journal of Systems and Software 72*, 1 (2004), 49–57.

[11] ELENBOGEN, B. S., AND SELIYA, N. Detecting outsourced student programming assignments. *Journal of Computing Sciences in Colleges 23*, 3 (2008), 50–57.

[12] FRANTZESKOU, G., MACDONELL, S., STAMATATOS, E., AND GRITZALIS, S. Examining the significance of high-level programming features in source code author classification. *Journal of Systems and Software 81*, 3 (2008), 447–460.

[13] FRANTZESKOU, G., STAMATATOS, E., GRITZALIS, S., CHASKI, C. E., AND HOWALD, B. S. Identifying authorship by byte-level n-grams: The source code author profile (scap) method. *International Journal of Digital Evidence 6*, 1 (2007), 1–18.

[14] FRANTZESKOU, G., STAMATATOS, E., GRITZALIS, S., AND KATSIKAS, S. Effective identification of source code authors using byte-level information. In *Proceedings of the 28th International Conference on Software Engineering* (2006), ACM, pp. 893–896.

[15] GRAY, A., SALLIS, P., AND MACDONELL, S. Software forensics: Extending authorship analysis techniques to computer programs.

[16] HAYES, J. H., AND OFFUTT, J. Recognizing authors: an examination of the consistent programmer hypothesis. *Software Testing, Verification and Reliability 20*, 4 (2010), 329–356.

[17] INOCENCIO, R. U.s. programmer outsources own job to china, surfs cat videos, January 2013.

[18] KOTHARI, J., SHEVERTALOV, M., STEHLE, E., AND MANCORIDIS, S. A probabilistic approach to source code authorship identification. In *Information Technology, 2007. ITNG'07. Fourth International Conference on* (2007), IEEE, pp. 243–248.

[19] KRSUL, I., AND SPAFFORD, E. H. Authorship analysis: Identifying the author of a program. *Computers & Security 16*, 3 (1997), 233–257.

[20] LANGE, R. C., AND MANCORIDIS, S. Using code metric histograms and genetic algorithms to perform author identification for software forensics. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation* (2007), ACM, pp. 2082–2089.

[21] MACDONELL, S. G., GRAY, A. R., MACLENNAN, G., AND SALLIS, P. J. Software forensics for discriminating between program authors using case-based reasoning, feedforward neural networks and multiple discriminant analysis. In *Neural Information Processing, 1999. Proceedings. ICONIP'99. 6th International Conference on* (1999), vol. 1, IEEE, pp. 66–71.

[22] NARAYANAN, A., PASKOV, H., GONG, N. Z., BETHENCOURT, J., STEFANOV, E., SHIN, E. C. R., AND SONG, D. On the feasibility of internet-scale author identification. In *Security and Privacy (SP), 2012 IEEE Symposium on* (2012), IEEE, pp. 300–314.

[23] PELLIN, B. N. Using classification techniques to determine source code authorship. *White Paper: Department of Computer Science, University of Wisconsin* (2000).

[24] PIKE, R. The sherlock plagiarism detector, 2011.

[25] PRECHELT, L., MALPOHL, G., AND PHILIPPSEN, M. Finding plagiarisms among a set of programs with jplag. *J. UCS 8*, 11 (2002), 1016.

[26] QUINLAN, J. Induction of decision trees. *Machine learning 1*, 1 (1986), 81–106.

[27] ROSENBLUM, N., ZHU, X., AND MILLER, B. Who wrote this code? identifying the authors of program binaries. *Computer Security–ESORICS 2011* (2011), 172–189.

[28] SHEVERTALOV, M., KOTHARI, J., STEHLE, E., AND MANCORIDIS, S. On the use of discretized source code metrics for author identification. In *Search Based Software Engineering, 2009 1st International Symposium on* (2009), IEEE, pp. 69–78.

[29] SPAFFORD, E. H., AND WEEBER, S. A. Software forensics: Can we track code to its authors? *Computers & Security 12*, 6 (1993), 585–595.

[30] STOLERMAN, A., OVERDORF, R., AFROZ, S., AND GREENSTADT, R. Classify, but verify: Breaking the closed-world assumption in stylometric authorship attribution. In *IFIP Working Group 11.9 on Digital Forensics* (2014), IFIP.

[31] WIKIPEDIA. Saeed Malekpour, 2014. [Online; accessed 04-November-2014].

[32] YAMAGUCHI, F., GOLDE, N., ARP, D., AND RIECK, K. Modeling and discovering vulnerabilities with code property graphs. In *Proc of IEEE Symposium on Security and Privacy (S&P)* (2014).

[33] YAMAGUCHI, F., WRESSNEGGER, C., GASCON, H., AND RIECK, K. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (2013), ACM, pp. 499–510.

A Appendix: Keywords and Node Types

AdditiveExpression	AndExpression	Argument
ArgumentList	ArrayIndexing	AssignmentExpr
BitAndExpression	BlockStarter	BreakStatement
Callee	CallExpression	CastExpression
CastTarget	CompoundStatement	Condition
ConditionalExpression	ContinueStatement	DoStatement
ElseStatement	EqualityExpression	ExclusiveOrExpression
Expression	ExpressionStatement	ForInit
ForStatement	FunctionDef	GotoStatement
Identifier	IdentifierDecl	IdentifierDeclStatement
IdentifierDeclType	IfStatement	IncDec
IncDecOp	InclusiveOrExpression	InitializerList
Label	MemberAccess	MultiplicativeExpression
OrExpression	Parameter	ParameterList
ParameterType	PrimaryExpression	PtrMemberAccess
RelationalExpression	ReturnStatement	ReturnType
ShiftExpression	Sizeof	SizeofExpr
SizeofOperand	Statement	SwitchStatement
UnaryExpression	UnaryOp	UnaryOperator
WhileStatement		

Table 10: Abstract syntax tree node types

Table 10 lists the AST node types generated by *Joern* that were incorporated to the feature set. Table 11 shows the C++ keywords used in the feature set.

alignas	alignof	and	and_eq	asm
auto	bitand	bitor	bool	break
case	catch	char	char16_t	char32_t
class	compl	const	constexpr	const_cast
continue	decltype	default	delete	do
double	dynamic_cast	else	enum	explicit
export	extern	false	float	for
friend	goto	if	inline	int
long	mutable	namespace	new	noexcept
not	not_eq	nullptr	operator	or
or_eq	private	protected	public	register
reinterpret_cast	return	short	signed	sizeof
static	static_assert	static_cast	struct	switch
template	this	thread_local	throw	true
try	typedef	typeid	typename	union
unsigned	using	virtual	void	volatile
wchar_t	while	xor	xor_eq	

Table 11: C++ keywords

B Appendix: Original vs Obfuscated Code

```c
#include<stdio.h>
int main()
{
  int T,test=1;
  double C,F,X,rate,time;
  scanf("%d",&T);
  while(T--)
    {
      scanf("%lf %lf %lf",&C,&F,&X);
      rate=2.0;
      time=0;
      while(X/rate>C/rate+X/(rate+F))
        {
          time+=C/rate;
          rate+=F;
        }
      time+=X/rate;
      printf("Case #%d: %lf\n",test++,time);
    }
  return 0;
}
```

Figure 6: A code sample X

Figure 6 shows a source code sample X from our dataset that is 21 lines long. After obfuscation with Tigress, sample X became 537 lines long. Figure 7 shows the first 13 lines of the obfuscated sample X.

```c
struct _IO_FILE;
struct timeval {
   long tv_sec ;
   long tv_usec ;
};
enum _1_main_$op {
    _1_main__string$value_LIT_0$result_REG_1__convert_void_star2void_star$
        result_STA_0$left_REG_0__local$result_STA_0$value_LIT_0__store_void_star$
        left_STA_0$right_STA_1__local$result_STA_0$
        value_LIT_0__convert_void_star2void_star$left_STA_0$result_REG_0__local$
        result_REG_0$value_LIT_1__convert_void_star2void_star$result_STA_0$
        left_REG_0__store_void_star$right_STA_0$left_REG_0 = 46,
    _1_main__local$result_REG_0$value_LIT_1__constant_int$result_STA_0$
        value_LIT_0__store_int$right_STA_0$left_REG_0__local$result_STA_0$
        value_LIT_0__convert_void_star2void_star$left_STA_0$result_REG_0__string$
        value_LIT_0$result_REG_1__convert_void_star2void_star$result_STA_0$
        left_REG_0__store_void_star$right_STA_0$left_REG_0__local$result_REG_0$
        value_LIT_1__convert_void_star2void_star$result_STA_0$left_REG_0 = 44,
    _1_main__convert_void_star2void_star$left_STA_0$result_REG_0__load_int$
        left_REG_0$result_REG_1__MinusA_int_int2int$result_REG_0$left_REG_1$
        right_REG_2__store_int$left_STA_0$right_REG_0__goto$label_LAB_0 = 161,
    _1_main__local$result_STA_0$value_LIT_0__local$result_REG_0$
        value_LIT_1__convert_void_star2void_star$result_STA_0$
        left_REG_0__load_double$left_STA_0$result_REG_0__local$result_REG_0$
        value_LIT_1__convert_void_star2void_star$result_STA_0$
        left_REG_0__load_double$left_STA_0$result_STA_0__convert_double2double$
        left_STA_0$result_REG_0__local$result_REG_0$
        value_LIT_1__convert_void_star2void_star$result_STA_0$left_REG_0 = 184,
    _1_main__local$result_REG_0$value_LIT_1__convert_void_star2void_star$
        result_STA_0$left_REG_0__local$result_STA_0$value_LIT_0__store_void_star$
        left_STA_0$right_STA_1__local$result_REG_0$value_LIT_1__local$result_STA_0$
        value_LIT_0__store_void_star$right_STA_0$left_REG_0 = 76,
    _1_main__local$result_STA_0$value_LIT_0__load_double$left_STA_0$
        result_REG_0__local$result_STA_0$value_LIT_0__load_double$left_STA_0$
        result_STA_0__Div_double_double2double$right_STA_0$left_REG_0$
        result_REG_1__local$result_STA_0$value_LIT_0__load_double$left_STA_0$
        result_REG_0__local$result_REG_0$value_LIT_1__convert_void_star2void_star$
        result_STA_0$left_REG_0__load_double$left_STA_0$result_STA_0 = 194,
    _1_main__PlusA_double_double2double$right_STA_0$result_STA_0$
        left_REG_0__Div_double_double2double$right_STA_0$left_REG_0$
        result_REG_1__convert_double2double$result_STA_0$
        left_REG_0__PlusA_double_double2double$right_STA_0$left_REG_0$
        result_REG_1__Gt_double_double2int$result_STA_0$right_REG_0$
        left_REG_1__branchIfTrue$expr_STA_0$label_LAB_0 = 199,
```

Figure 7: Code sample X after obfuscation