De-anonymizing Programmers via Code Stylometry

Aylin Caliskan-Islam*, Richard Harang†, Andrew Liu‡, Arvind Narayanan§,
Clare Voss†, Fabian Yamaguchi¶, and Rachel Greenstadt*
*Drexel University, ac993@drexel.edu, greenie@cs.drexel.edu
†ARL, richard.e.harang.civ@mail.mil, clare.r.voss.civ@mail.mil
‡UMD, aliu1@umd.edu
§Princeton University, arvindn@cs.princeton.edu
¶ University of Goettingen, fabs@goesec.de

***Abstract*—Source code authorship attribution could provide proof of authorship in court, automate the process of finding a cyber criminal from the source code left in an infected system, or aid in resolving copyright, copyleft and plagiarism issues in the programming fields. In this work, we investigate methods to de-anonymize source code authors of C++ using coding style. We cast source code authorship attribution as a machine learning problem using natural language processing techniques to extract the necessary features. The 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. Such a unique representation of coding style has not been used before in code attribution.**

Our random forest and abstract syntax tree-based approach attributes more authors (250) with significantly higher accuracy (95%) on a larger data set (Google Code Jam) than has been previously attempted. Furthermore these novel features are more robust than previous approaches, and are still able to attribute authors even when code is run through commercial obfuscation with no significant change in accuracy. This analysis also produces interesting insights relevant to software engineering. We 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.

I. 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. Can elements of coding style be extracted computationally, and if so, what features are most informative? Can these extracted styles be used to distinguish one programmer's code from another, or to determine the author of an unattributed piece of code?

Affirmative answers to these questions would have at least two sets of applications. The first is software forensics, i.e., examining the artifacts on a system after an intrusion to obtain evidence for a criminal prosecution. Often, the adversary leaves behind code after an intrusion, either a backdoor or a payload. If we are able to identify the code's author — for example, by a stylistic comparison of the code with various authors in online code repositories — it may give us clues about the adversary's identity. A careful adversary may only leave binaries, but a less careful one may leave behind code written in a scripting language or source code downloaded into the breached system for compilation.

The second application is determining whether or not a piece of code was written by its claimed author. This takes various forms including detection of ghostwriting, a form of plagiarism, and investigation of copyright disputes. We elaborate in Section II.

We can formulate code stylometry as a machine-learning task consisting of feature extraction followed by classification. Prior works have used primarily two types of features: *layout features* such as whitespace that don't change the meaning of a program, and *lexical features* such as those that count various types of tokens in the language. The latter, roughly speaking, are features that can be obtained via lexical program analysis rather than parsing.

Contributions. Our first contribution is the use of *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 also provide evidence that these features are more fundamental and harder to obfuscate. Our complete feature set consists of a comprehensive set of layout-based, lexical, and syntactic features totalling around 20,000.

With our complete feature set we are able to achieve a dramatic increase in accuracy compared to previous work. Each year more than 15,000 contestants from all around the world compete in the international programming competition "Google Code Jam". We scraped the correct solutions of problems implemented in C++ with ground truth authorship information from the available competition data. A bagging (portmanteau of "bootstrap aggregating") classifier - random forest was used to attribute authors to source code by making use of the relevant features. Our classifiers reach 95% accuracy in a 250-class closed world task, 99% accuracy on average in a two-class task, and 93% accuracy in attributing the test instances of the training class in a two-class/one-class task. Finally, we analyze various attributes of authors, type of programming task, and types of features that appear to influence the success of stylometric identification. We identified the most important 137 features out of 20,000 and 18% of them are syntactic, 4% are layout-based and the rest of the features are lexical. 8 training files with an average of 70 lines of code provides enough data while using the lexical, layout and syntactic features. We have also observed that programmers with a greater skill set are more easily identifiable compared to less advanced programmers. Also, a programmer's coding style is more prevalent in implementations of functionality for difficult tasks as opposed to easier tasks.

Discussion. Our use of syntactic features opens the door to adapting techniques from linguistics and natural-language processing to code stylometry. Indeed, authorship attribution of textual documents has a rich history (see Section VII). For example, features derived from *edges* of the AST may prove useful, analogous to [24]. Exploring these and other feature classes such as the *control-flow graph* is left to future work.

Most of the application scenarios we have outlined are adversarial; it is therefore important to ask how easy it is for an adversary to change his coding style via manual or automated means. *We do not claim* that our feature set resists attempts at manipulating one's coding style. However, we do find that our syntactic feature set is impervious to off-the-shelf *code obfuscators* which only change layout and some lexical features (Section V-D). Our results show that we are able to maintain a high accuracy using *only* the syntactic feature set. Our syntactic-features-only approach attributes the correct author among 250 candidates with 44% accuracy and reaches 78% accuracy in attributing the correct author among 25 candidates.

Furthermore, we believe that if adversarial code stylometry can be achieved, it can only be through syntactic and structural features of code. We have taken the first steps in this direction. Finding more robust structural features is an important avenue for future work, as well as user studies that ask programmers to consciously modify their coding style (for an analogous study of modifying linguistic style, see [4]).

The fact that programmers possess consistent coding styles is of interest beyond the problem of inferring authorship. In software engineering, identifying coding features that are associated with a higher incidence of bugs can be of great practical impact. Another application is showing programmers how their coding style has evolved over time. Conversely, well-known metrics used in software engineering such as code cohesion might be useful as features for stylometry.

The possibility that elements of coding style may be preserved in compiled binaries is a fascinating prospect, and would greatly expand the realm of stylometry in software forensics. In particular, since the majority of malware spreads in binary form, stylometry may provide an additional avenue for tracking the origins of malware in addition to conventional malware analysis techniques. There is some preliminary evidence that binaries retain some stylistic features [29].

Finally, as with any forensic technology, code stylometry has privacy implications. Contributors to an open-source project may hide their identity for entirely innocuous reasons. The programmer may not want her employer to know about her side activities, or may live in an oppressive regime that prohibits certain types of software — an Iranian programmer was sentenced to death in 2012 for developing photo sharing software that was used on pornographic websites [33]. Sadly but predictably, there is virtually no technical difference between security-enhancing and privacy-infringing use cases. We hope that our work will lead to greater awareness among programmers wishing to distribute code anonymously that their code might be de-anonymized. Automated obfuscation of coding style is a useful direction for future work.

Roadmap: Now that we introduced code stylometry, we will give the formal problem statement with possible scenarios in section II. In section III, we will explain our novel approach that uses abstract syntax trees and the details of machine learning tasks along the way. Section IV will follow explaining the properties of our corpus, why we chose it and summarize the types of experimental datasets we generated. We will present results to various tasks in section V, and present our evaluation results with statistical significance tests in section VI to validate our approach. We finally summarize our results and elaborate on related work to make a comparison in section VII. Section VIII includes the major discussion points and section IX talks about the possible limitations. Section X lists work left to the future. We will conclude with section XI with the main take away from our source code stylometry research.

II. Problem Statement and Threat Model

We are given an anonymous piece of code whose author we'd like to determine based on coding style. We are also given labeled code from one or more candidate authors, and labeled code from zero or more unrelated authors. At a high level, the stylometric analyst converts each labeled sample into a numerical feature vector, uses it to train a classifier, and uses it to classify the feature vector extracted from the anonymous code. This formulation represents at least four different settings corresponding to different applications, summarized in Table I.

In software forensics, we assemble a set of candidate authors based on previously collected malware samples, online code repositories, etc. We can never be sure that the anonymous author is one of the candidates, so this represents an "open world" classification problem – the test sample might not belong to any of the known categories.

Ghostwriting detection is related to but different from traditional plagiarism detection. Traditionally, 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 (more generally, we are given N pieces of code and would like to find all pairs that may have been plagiarized from each other). This is a well-studied problem typically solved using code similarity metrics. MOSS, JPlag, and Sherlock are widely used tools [5], [27], [26].

Ghostwriting can be considered a type of plagiarism, but detecting it is a distinct problem. Perhaps we notice that a freshman student's performance on programming assignments suddenly shot up, and we suspect that someone else is writing his code, perhaps a sophomore who took the class the previous year. Unfortunately, even though we have the previous years'

Application	**Learner**	**Comments**
Software forensics	Multiclass	Open world
Detailed experiments on the specific problem left to future work.		
Stylometric plagiarism detection	Multiclass	Closed world
This is the predominant scenario for most of the experiments in section V.		
Copyright investigation	Two-class	
Results of experiments related to this scenario are in section V-F.		
Authorship verification	Two-class	One-class open-world and anomaly detection approaches also possible
Results of experiments related to this scenario are in section V-G.		

Table I
APPLICATIONS

submissions, the assignments are all different this year, which means that code similarity-based methods are inapplicable. Luckily, stylometry is still relevant — we decide to find the most stylistically similar piece of code from the previous year's corpus and bring both students in for gentle questioning.

Note that even though we do not know for sure that one of the labeled authors was the ghostwriter, the additional layer of human investigation means that from a machine-learning perspective, we can formulate it as a closed-world problem.

Next, 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 his 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 authors is available to the forensic expert.

Finally, we may suspect that a piece of code was not written by the claimed author, but have no leads on who the actual author 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 author and negative examples come from randomly selected unrelated authors. However, there is a school of thought that holds that the use of negative training examples is in fact harmful to classification accuracy, and that we should instead use a "one-class" formulation [32]. One-class SVMs were originally developed for cases where negative examples are unavailable. This formulation is similar to anomaly detection. Exploring alternative formulations is left to future work.

A forensic investigation conducted by Verizon on a US company's anomalous activity in VPN logs revealed that one of the employees 'Bob' was outsourcing his work to China. Bob, a talented programmer, physically mailed his RSA token to China so that the Chinese programmers could authenticate themselves on the VPN as this particular employee. Verizon investigators eventually noticed that the VPN connection from China was at least six months old, according to the oldest saved VPN logs. The VPN credentials belonged to employee Bob and unable to figure out the anomaly, the investigators took a forensic image of Bob's computer which revealed that Bob was surfing the Internet all day long and not doing any programming while outsourcing his programming tasks to China with one fifth of his salary. After this discovery, Bob got fired. In such cases, training on Bob's original code and some random authors' code and testing the recent pieces of source code using some confidence metrics would verify if Bob was the actual programmer.

In each of these applications, the adversary may try to actively modify the coding style of the anonymous code. In the software forensics application, the adversary tries to modify code written by him to hide his style. In the copyright and authorship verification applications, the adversary modifies code written by another author to match his own style. 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.

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.

III. APPROACH

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 III-A). We then define a number of different features to represent both syntax and structure of program code (section III-B) and finally, we train a random forest classifier for classification of previously unseen source files (section III-C). 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 available in our repository[1].

A. Generation of 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 [7, 14]. 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 frontends 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

[1]— Link removed for blind review —

```
1  int foo()
2  {
3    if((x < 0) || x > MAX)
4      return -1;
5
6    int ret = bar(x);
7    if(ret != 0)
8      return -1;
9    else
10     return 1;
11 }
```

Figure 1. Sample code listing

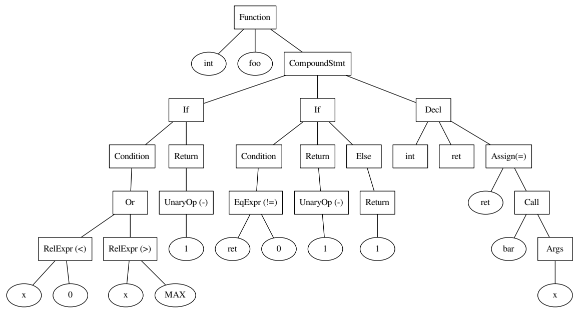

Figure 2. Corresponding abstract syntax tree

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 [35]. 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 syntactical features such as keywords, identifiers and operators accessible, inner nodes represent operations showing 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 that represents the code's structure. In the following sections, we describe the features extracted from the syntax tree in detail.

B. Feature Extraction

1) Syntactic Feature Set (SFS): The syntactic feature set consists of properties that can only be derived from the language dependent abstract syntax tree and keywords. Keywords are predefined reserved identifiers in a programming language that have special meanings to define structure. The syntactic features are independent of any lexical or layout features, or comments.

The syntactic feature set was created to capture properties of coding style that are completely independent from writing style. Features of writing style such as function or variable name choices, or comments can easily be changed. If an adversary takes a piece of code and tries to modify it without changing the functionality, the adversary will probably preserve the structural properties to avoid modifying the functionality. Nevertheless, the adversary can obfuscate all layout properties and lexical properties such as programmer's selection of identifiers, variable names, and comments. If obfuscation is carried through refactoring all names given by the author, removing or changing comments and layout properties, it would have no effect on the syntactic features. The syntactic features would remain the exact same after such an obfuscation.

Feature	Definition	Count
MaxDepthASTLeaf	Maximum depth of an AST node	1
ASTNodeTypesTF	Term frequency of 58 possible AST node type	58
ASTNodeTypesTFIDF	Term frequency inverse document frequency of 58 possible AST node type	58
ASTNodeTypeAvgDep	Average depth of 58 possible AST node types	58
cppKeywords	Term frequency of 84 C++ keywords	84

Table II
SYNTACTIC FEATURE SET (SFS)

The syntactic feature set is listed in Table II. All C++ source files in the dataset are preprocessed to produce their abstract syntax trees. An abstract syntax tree is created for each function including *main* in the code. There are 58 node types in the abstract syntax tree produced by *Joern* [34], which are listed in Table XV. Features are extracted for each source file and written to a feature vector to be used in author classification per file.

The maximum depth of an abstract syntax tree reflects the deepest level a programmer nests a node in the solution. 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 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.

2) Code Stylometry Feature Set (CSFS): Code Stylometry Feature Set uses a large set of syntactic, lexical and layout features to help classify authors in general cases where there are no adversaries trying to obfuscate or tamper code. This feature set includes the *Syntactic Feature Set* described in section III-B1 with one exception. The solution files are preprocessed to create their abstract syntax trees. The resulting trees' inner nodes consist of the 58 predefined node types, but the leaves contain code unigrams, such as *x*, *cout*, *endl*, ++, and *get*. All the features extracted from a solution file are written to a feature vector to be used by the random forest classifier.

C. Classification

1) Random Forest Classification: We used the random forest ensemble classifier [6] 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 between at each split also 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 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 V-B).

We employed random forests with 300 trees, which empirically provided the best tradeoff 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 V, 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, and average results across all iterations are reported, ensuring that results are not biased by improbably easy or difficult to classify subsets.

2) Attribute 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 attributes for hundreds of classes. The dynamic *Code stylometry feature set*, for example, produced close to 20,000 attributes 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 attributes being selected during the random subsampling of the attributes 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 attributes, 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 an attribute selection step using WEKA's information gain [28] 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:

$$IG(A, M_i) = H(A) - H(A|M_i) \tag{1}$$

where A is the class corresponding to an author, H is Shannon entropy, and M_i is the i^{th} attribute of the data set. Intuitively, the information gain can be thought of as measuring the amount of information that the observation of the value of attribute i 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. (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 only those features

Feature	Definition	Count
Syntactic Features		
MaxDepthASTNode	Maximum depth of an AST node	1
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
Lexical Features		
CodeInASTLeavesTF	Term frequency of code unigrams in AST leaves	dynamic*
CodeInASTLeavesTFIDF	Term frequency inverse document frequency of code unigrams in AST leaves	dynamic*
CodeInASTLeavesAvgDep	Average depth of code unigrams in AST leaves	dynamic*
ln(numDo/length)	Log of the number of *do-while* loops divided by file length in characters	1
ln(numElif/length)	Log of the number of *else if* statements divided by file length in characters	1
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 user-defined 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
ln(numIf/length)	Log of the number of *if* statements divided by file length in characters, excluding *else if* statements	1
ln(numElse/length)	Log of the number of *else* statements divided by file length in characters, excluding *else if* statements	1
ln(numSwitch/length)	Log of the number of *switch* statements divided by file length in characters	1
ln(numFor/length)	Log of the number of *for* loops divided by file length in characters	1
ln(numWhile/length)	Log of the number of *while* loops divided by file length in characters, excluding *do-while* loops	1
nestingDepth	The highest degree to which control statements and loops would be nested within each other	1
branchingFactor	The branching factor of the tree formed by converting code blocks of the files into nodes (Nesting loops or control statements makes children nodes rather than sibling nodes)	1
avgParams	The average number of parameters among all the user-defined functions	1
stdDevNumParams	The standard deviation of the number of parameters among all the user-defined functions	1
avgLineLength	The average length of each line	1
stdDevLineLength	The standard deviation of the character lengths of each line	1
***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.**		
Layout Features		
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 the 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 III
CODE STYLOMETRY FEATURE SET (CSFS)

that have demonstrable value in classification are included in our selected features.

In order 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 attributes 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 V for further discussion). We did not apply information gain to the smaller *Syntactic Feature Set*, since its feature vectors are not sparse and did not require reduction.

IV. CORPUS

Obtaining a representative data corpus for the evaluation of source code authorship attribution methods is challenging for several reasons. For example, while collecting code from open-source projects may initially seem like a viable option, these source files are often modified by several different authors over time, and even manual attribution might not be possible. Moreover, projects may impose easily recognizable style-guidelines, interface with characteristic APIs or they may simply be recognizable by the functionality they implement. To obtain meaningful evaluation results, we therefore have to ensure that classification does not hinge on these artifacts.

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)*. 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. Contestants in 2014 participated from 166 countries. The majority of them were from India, United States, China, Russia, Japan, Canada, Brasil, South Korea, France, Egypt, and Poland. The participation statistics are similar throughout years. GCJ is a good representation of programmers from all around the world, including professional programmers, students, academics, and hobbyists. GCJ dataset's nature was crucial for code stylometry analysis. All of these contestants are solving the same set of problems, consequently we were able to capture the differences in coding style on source code that essentially have the same functionality.

Programmers need to pass the qualification round within a 27 hour frame to become contestants and advance to the online rounds. 3,000 contestants from the first round that have the highest scores advance to the second round. The top-scoring 500 contestants in the second round advance to the third round. 25 of the top-scoring contestants in the third round advance to the onsite final round. As the round number increases, the set of problems become more difficult. For example, 26,470 contestants were able to pass the qualification round. 15,563 of these completed round-1 and 3,000 contestants with the top scores advanced to round-2. Only 2,599 contestants out of this set of skilled 3,000 contestants were able to complete round-2. 500 contestants with the highest scores from round-2 advanced to round-3 and only 393 of these highly skilled programmers were able to complete round-3.

In 2008, the correct solutions and information about the competitors became public on GCJ's website [1]. This presented the opportunity to build a source code corpus with thousands of programmers with ground truth information on authorship. We scraped the C++ code of thousands of contestants from 2008 to 2014 and the scraper is available in our repository [2]. Statistics such as usernames, round numbers, and problem numbers were obtained from a website[3] dedicated to keeping statistics about the competition. Datasets were formed by limiting the authors to the years to avoid ground truth problems. For example, a user in 2008 might come back to the competition in 2014 with a different username, and there is no way to tell whether they are the same people.

The most commonly used programming language was C++, followed by Java, Python, C#, C, Ruby, and Haskell. A total of 92 programming languages were used. We chose to investigate source code stylometry on C++ because of its popularity in the competition and having a parser to generate abstract syntax trees for C/C++ readily available [35]. A dataset of 17,433 contestants were used as the main dataset. A validation dataset was created from 2012's GCJ competition. Some problems had two stages. The first stage had to find a solution to a small input file. The second stage had to answer the same problem in a limited amount of time and the input was larger. The solution to the large input is essentially a solution for the small input but not vice versa. Therefore, scraping both of these solutions could result in duplicate and identical source code. In order to avoid multiple entries, we only scraped the small input versions' solutions to be used in our dataset.

The authors had up to 19 solution files in these datasets. Solution files have an average of 70 lines of code per author. The same set of problems were used in datasets to investigate the different styles in implementing the same functionality. The set of problems depends on the author size and problem we are investigating. The only exception to this is the randomized large scale experiment where the authors are from random years, and the instances are randomly picked.

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. In this paper, when we refer to the "main" dataset, these are the files drawn from 2014. When we refer to the "validation" dataset, the files come from 2012, and for the "multi-year" dataset, the files comes from years 2008 through 2014.
(ii) within each year, we ordered the corpus files by the round in which they were written, since 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, and by its author's highest round completed in that year.

[2]— Link removed for blind review —
[3]http://www.go-hero.net/jam/

V. Results

A. Required amount of training data

We selected different sets of 62 authors 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 authors. Each dataset consisted of authors that were able to solve exactly F problems and 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 authors to make logical comparison.

The same set of F problems were used to make sure that the coding style of the author is being classified and not the properties of possible solutions of the problem itself are being attributed. We were able to capture personal programming style since all the authors 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 problem of the test files was 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 *Syntactic Feature Set*, then with the *Code Stylometry Feature Set* (CSFS) and finally with the CSFS's features that had information gain.

Figure 3 shows the accuracy from 13 different sets of 62 authors with 2 to 14 solution files, and consequently 1 to 13 training files. In the case where the *Syntactic Feature Set* is used, the accuracy seems to be monotonically increasing and an optimal training set size is not established, which suggests the need for *relaxed attribution* explained in section V-B. The *Code Stylometry Feature Set* leads to an optimal training set size with 9 solution files, where the classifier trains on 8 $(F-1)$ solutions. While the difference is not statistically significant between $F = 13$ and $F = 14$ for either the code stylometry feature set or syntactic feature set, the visually apparent jump led us to investigate this specific case in section .

In the datasets we constructed, as the number of files increase and problems from more advanced rounds are included, the average line of code per file also increases, which is shown in Table IV. 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 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

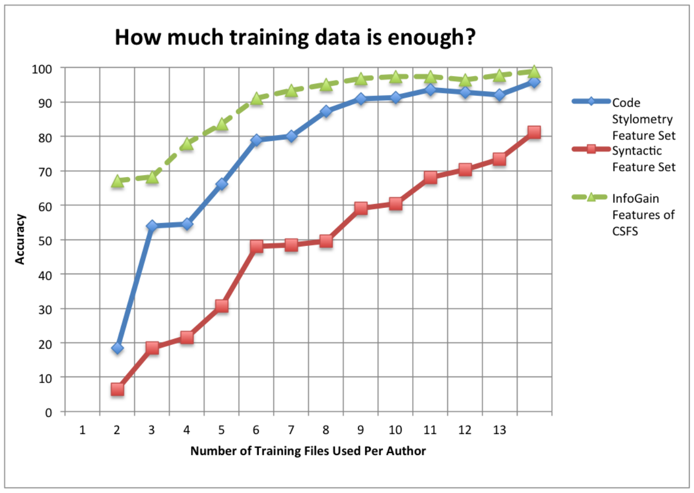

Figure 3. Training Data

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.

Authors	Files	Average LOC*
62 authors	2 files	55 LOC
62 authors	3 files	60 LOC
62 authors	4 files	60 LOC
62 authors	5 files	61 LOC
62 authors	6 files	71 LOC
62 authors	6 easy	78 LOC
62 authors	6 difficult	86 LOC
62 authors	7 files	68 LOC
62 authors	7 easy	76 LOC
62 authors	7 difficult	83 LOC
62 authors	8 files	71 LOC
62 authors	9 files	77 LOC
62 authors	10 files	72 LOC
62 authors	11 files	79 LOC
62 authors	12 files	82 LOC
62 authors	13 files	86 LOC
62 authors	14 files	80 LOC
***Line of code**		

Table IV
AVERAGE LINES OF CODE IN DATASETS

B. *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 obfuscated code, would reduce the effort required to manually find the actual author of the code. This method is especially helpful while using the *Syntactic Feature Set* explained in section III-B1 since its accuracy is lower when there are very few training files but many authors.

The classification was relaxed to a set of top R suspects instead of exact classification of the author. 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 author was in the set of top R highest votes.

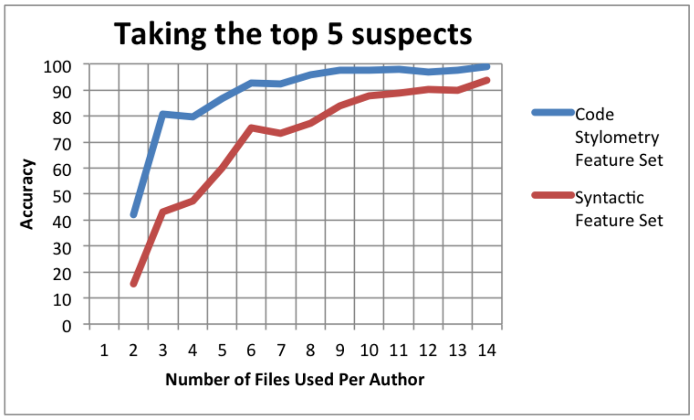

Figure 4. Accuracy with top 5 suspects

Relaxing the suspect set size to $R = 5$ hits the 90% accuracy bar with 9 solution files but keeps increasing steadily which is shown in Figure 4. Relaxing the suspect set size to $R = 10$ seems, in Figure 5, to result in an optimal training set size of 8 $(F - 1)$ solution files. Consequently, in larger scale experiments we will use $F = 9$ to see how the method scales and what the important features are. Nevertheless our method shows promising results with $F < 9$ and without any attribute selection which further improves the accuracy.

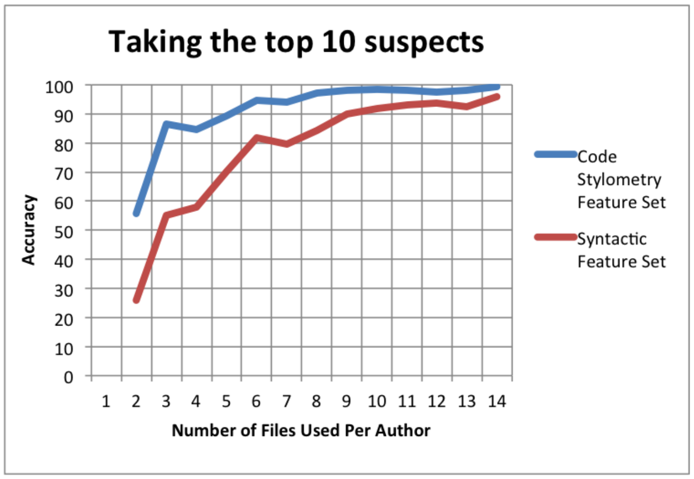

Figure 5. Accuracy with top 10 suspects

C. *The effect of problem difficulty*

We took the dataset with 14 solution files and 62 authors. We excluded the most advanced problem and the classification accuracy using the *Code Stylometry Feature Set* with the exact same method in section V-A was 92.12%. Excluding the most difficult problem had a noticeable negative effect on the accuracy.

We took the exact same dataset with the 14 solution files and 62 authors. This time we excluded the easiest problem in the dataset and the accuracy increased to 95.66%. To further investigate the effect of problem difficulty's effect on coding style we created two more datasets with 62 authors that had exactly 14 solution files. Table V summarizes the following results. A dataset with 7 of the easier problems out of 14 resulted in 89.88% accuracy with all the features and 67.97% accuracy with the syntactic features. A dataset with 7 of the more difficult problems out of 14 resulted in 94.84% accuracy with all the features and 74.88% with the syntactic features. This might infer that more difficult coding tasks have a more prevalent reflection of coding style. On the other hand, the dataset that had 62 authors with exactly 7 of the easier problems resulted in 80.09% accuracy with all the features and 48.34% with the syntactic features, which is a lot lower than the accuracy obtained from the dataset whose authors 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 authors. A dataset with 6 of the easier problems out of 12 resulted in 84.07% accuracy with all the features and 50.54% with the syntactic features. A dataset with 6 of the more difficult problems out of 12 resulted in 87.63% accuracy with all the features and 57.26% with the syntactic features. These results are quite higher than the dataset whose authors were only able to solve the easier 6 problems. The dataset that had 62 authors with exactly 6 of the easier problems resulted in 78.87% accuracy with all the features and 48.12% with the syntactic features.

A = #authors, 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					
89.88%	*94.84%*^{2}	**80.09%**^{1}	84.07%	*87.64%*^{2}	**78.87%**^{1}
Average accuracy after 10 iterations while using SFS					
67.97%	*74.88%*^{2}	**48.34%**^{1}	50.54%	*57.26%*^{2}	**48.12%**^{1}
^{1}**Drop in accuracy due to programmer skill set.**					
^{2}**Coding style is more distinct in more difficult tasks.**					

Table V
EFFECT OF PROBLEM DIFFICULTY ON CODING STYLE

D. Obfuscation

We took a dataset with 9 solution files and 25 authors and obfuscated the code using an off the shelf C++ obfuscator stunnix [2]. The accuracy with the syntactic feature set is 77.78%. The accuracy on the same dataset when the code is not obfuscated is 77.33%. The accuracy with the code stylometry feature set on the obfuscated dataset is 96.71%. The accuracy on the same dataset when the code is not obfuscated is 96.44%. There's zero significance to the difference between the accuracies obtained from the original source code and obfuscated code. Obfuscating the data produced no detectable change in the performance of the classifier for this sample. The results are summarized in Table VI.

A = #authors, F = max #problems completed		
N = #problems included in dataset (N $\leq$ F)		
A = 25		
F = 9		
N = 9	**N = 9 obfuscated files**	**Comparison**
Average accuracy after 10 iterations with CSFS		**p-value**
96.44%	96.71%	4.89×10^{-1}
Average accuracy after 10 iterations with SFS		**p-value**
77.33%	77.78%	2.89×10^{-1}

Table VI
EFFECT OF OBFUSCATION ON AUTHORSHIP ATTRIBUTION

E. Simple two class classification

We took 20 different pairs of authors, each with 9 solution files. We used a random forest and 9-fold cross validation to classify two authors. The average classification accuracy using the code stylometry feature set is 98.89%. The average area under curve for 40 classes is 0.9999. The average classification accuracy using the syntactic feature set is 92.92% The average area under curve for 40 classes is 9.67×10^{-1} using the syntactic feature set.

F. Copyright Dispute - Two class classification

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 author.

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 author A's solution to problem a, author B's solution to problem b, author C's solution to problem c, author D's solution to problem d, author E's solution to problem e, author F's solution to problem f, author G's solution to problem g, author 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, both with the syntactic and code stylometry feature set. We have 6 test instances from Mallory and 6 test instances from another author ZZZZZZ, who is not in the training set.

These experiments have been repeated in the exact same setting with 80 different sets of authors ABCDEFGH, ZZZZZZ and Mallorys. The average classification accuracy for Mallory using the code stylometry feature set is 93.19% and the average area under curve for Mallory's class is 1.00. The average classification accuracy for Mallory using the syntactic feature set is 87.79% and the average area under curve for

Mallory's class is 1.00. Area under curve has been calculated by using the Mann Whitney statistic. ZZZZZZ's test instances are classified as author ABCDEFGH 84.35% of the time, and classified as Mallory for the rest of the time while using the code stylometry feature set. ZZZZZZ's test instances are classified as author ABCDEFGH 75.58% of the time, and classified as Mallory for the rest of the time while using the syntactic feature set.

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.

G. Verification

We performed 503 classifications in a two-class problem and the false positive rate with the code stylometry feature set was 0.02% which includes two classifications where the probability of classifying it as $class1$ or $class2$ was exactly 50.00%. If we exclude those from the false positives, the rate decreases to 0.16%. The class probability or classification confidence ($P(C)_i$) is calculated by taking the percentage of trees (T) in the random forest (f) that voted for that particular class (V_i), which can be seen in equation 2. This could be used as a method for verifying a classification. Since this is a two class problem, the confidence level to choose the classification needs to be at least 50.0%.

$$P(C)_i = \frac{\sum V_i}{\sum T_f} \tag{2}$$

All the false positive classifications were made by at most 53.30% confidence, except one that had 54.30% confidence. All the $class1$ classifications made with more than 54.30% confidence were correct and 2 of them with exactly 54.30% class probability were also correct. This suggests that we can use the 54.40% confidence level as a threshold for verification for $class1$, and manually analyze the classifications that did not pass the confidence threshold or exclude them from results.

There were 18 false positive cases out of 503 while using the syntactic feature set that make up 0.04% of the instances and all the rest were attributed correctly. All the $class2$ classifications made with less than 55.90% confidence were wrong. Classifications with less than 55.90% confidence could go through more vigorous manual inspection to investigate if they really belong to $class2$ or such classifications could simply be rejected.

H. Larger scale experiments - Same Problems

The biggest dataset formed from 2014's Google Code Jam Competition with 9 solution files to the same problem had 250 authors. These were the easiest set of 9 problems, making the classification more challenging, since the more identifying difficult problems are missing.

We reached 80.98% accuracy in classifying 250 authors with the *Code Stylometry Feature Set*. The accuracy with the *Syntactic Feature Set* was 44.29%. After applying information gain and using the 137 features that had positive information gain the accuracy was 91.73%.

I. Larger scale experiments with a randomized dataset

We took 250 authors from different years and randomly selected 9 solution files for each one of them. We reached 84.19% accuracy in classifying 250 authors with the *Code Stylometry Feature Set*, which is 3% higher than the controlled large dataset's results. The accuracy with the *Syntactic Feature Set* was 54.74%. The randomized dataset leads to 11% higher correct classification accuracy than the controlled main dataset while using the syntactic feature set. After information gain reduction to the feature set and using the 137 features that had positive information gain, the accuracy was 95.33%.

J. The effect of dataset size

We took the main dataset with 250 authors each with 9 solution files to the same set of problems. We created 5 datasets of differing sizes, with 25 authors, 50 authors, 100 authors, 150 authors and 200 authors. We calculated the accuracy with the *Code Stylometry Feature Set* and *Syntactic Feature Set* to observe the effect of dataset size on classification accuracy which can be seen in Figure 6. We applied information gain to CSFS and observed that the accuracy did not decrease too much when increasing the number of authors. The slight drop in accuracy when classifying 250 authors compared to 25 authors might be due to the fact that we always used 300 trees and larger datasets might require more trees in the random forest. This result shows that information gain features are robust against changes in class size as long as the number of training documents is the same.

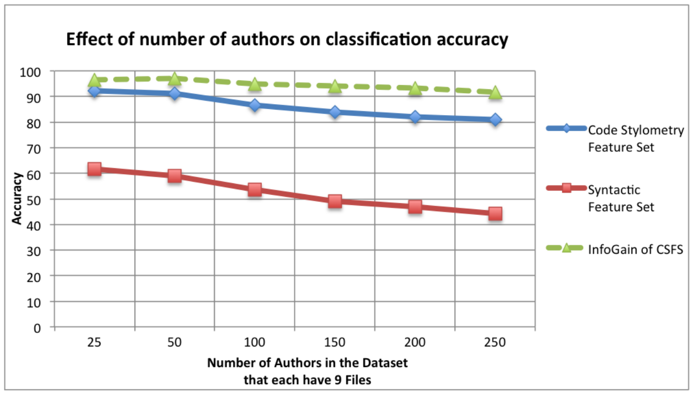

Figure 6. Change in accuracy as the dataset size changes

K. Consistency of programming style throughout years

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 authors 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 the code stylometry feature set. We had one instance from each one of the contestants from 2014. The correct classification of these test instances from 2014 is 88.00%. In section V-J, the experiment of 25 authors with 9 files within 2014 had a correct classification accuracy of 92.36%. These results indicate that coding style is reserved up to some degree throughout years.

L. Feature Selection

Features that had positive information gain are listed in Table VII. These features are used in the validation experiments to make sure that we are not overfitting to the main dataset. Surprisingly, the information gain features when using a different set of 173 authors with 6 files from 2014, another set of 173 authors with 6 files from 2008, and the validation set from 2012 of 250 authors with 9 files were all complete subsets of the following feature set, which shows that these actually are the most important features for identifying authors programming in C++. The nonzero information gain attributes of the syntactic feature set are shown in Table VIII to show the most important syntactic features. These specific subset of syntactic features were not tested separately since the syntactic feature set does not result in sparse feature vectors and is already small enough for fast performance.

M. Statistical Significance Testing

All tests were two-tailed, done using either Fisher's exact test [12], or Liddell's exact test [22], depending on whether observations were paired by known author or not. As we applied k-fold validation, resulting in highly correlated predictors, pooling of results to calculate significance across all data would be inappropriate (see [9] for detailed discussion). Standard practice would designate selecting a random test set for evaluation of significance [9], however this would not allow us to make full use of the available data, and due to the relatively small test set sizes for some experiments, run a risk of either type I or II errors.

In order to use all available data in determining significance, we do not pool results across folds, but instead test each fold individually and then report the minimum or maximum p-value across all folds, depending on the claim being made. When we make a claim of no statistical significance, we report the single *most* significant (read: minimum) p-value across all folds; thus no other fold indicated more statistical significance than our reported value, and all folds indicated statistical significance. Similarly, when we make a claim of statistical significance, we report the maximum observed p-value (indicating the highest probability of a result due to random chance, or least statistical significance) across all folds; thus all other folds indicated (a higher degree of) statistical significance. While this approach is highly conservative, it does allow us to

IG*	**Feature**	**IG***	**Feature**
2.21	ASTNodeAvgDep[t]	2.16	ASTNodesTF[T]
2.14	ASTNodeAvgDep[T]	1.58	ASTNodesTF[t]
1.41	**cppKeyword[typedef]**	1.39	ASTNodeAvgDep[in]
1.38	ASTNodeAvgDep[Case]	1.36	*ln(numSpaces/length)*
1.28	ASTNodeAvgDep[printf]	1.20	ASTNodeAvgDep[out]
1.18	ln(numMacros/length)	1.17	ASTNodeAvgDep[scanf]
1.14	ASTNodeAvgDep[cin]	1.10	ASTNodesTF[stdin]
1.10	ASTNodesTFIDF[stdin]	1.07	*ln(numEmptyLines/length)*
1.07	ASTNodeAvgDep[cout]	1.07	**ASTNodeAvgDep[UnaryExpression]**
1.06	ASTNodesTF[stdout]	1.06	ASTNodesTFIDF[stdout]
1.04	ASTNodeAvgDep[w]	1.04	*ln(numTabs/length)*
1.03	ASTNodesTFIDF[freopen]	1.03	ASTNodesTF[freopen]
0.99	ASTNodeAvgDep[d]	0.96	ASTNodeAvgDep[freopen]
0.94	ASTNodeAvgDep[txt]	0.86	ASTNodesTFIDF[solve]
0.86	ASTNodesTF[solve]	0.84	ASTNodeAvgDep[stdin]
0.84	ASTNodeAvgDep[stdout]	0.80	**ASTNodesTF[UnaryExpression]**
0.79	ASTNodesTF[scanf]	0.76	ASTNodesTF[printf]
0.75	ASTNodeAvgDep[small]	0.74	**ASTNodesTF[ShiftExpression]**
0.73	ASTNodesTFIDF[cin]	0.73	ASTNodesTF[cin]
0.71	ASTNodeAvgDep[test]	0.71	*tabsLeadLines*
0.70	ASTNodeAvgDep[output]	0.70	ASTNodeAvgDep[input]
0.69	**cppKeyword[case]**	0.69	**cppKeyword[template]**
0.69	ln(numFor/length)	0.69	ASTNodesTF[endl]
0.69	ASTNodesTFIDF[endl]	0.68	ASTNodesTFIDF[cout]
0.68	ASTNodesTF[cout]	0.67	ASTNodeAvgDep[solve]
0.66	ASTNodeAvgDep[r]	0.63	ASTNodeAvgDep[tt]
0.62	ASTNodesTFIDF[argv]	0.62	ASTNodesTF[argv]
0.62	**ASTNodeAvgDep[ShiftExpression]**	0.61	ASTNodesTF[test]
0.61	ASTNodesTFIDF[test]	0.61	**ASTNodeAvgDep[IncDecOp]**
0.60	ASTNodesTF[tt]	0.59	ASTNodesTF[argc]
0.59	ASTNodesTFIDF[argc]	0.58	ln(numIf/length)
0.57	**ASTNodesTF[IncDecOp]**	0.57	**cppKeyword[const]**
0.57	ASTNodeAvgDep[tc]	0.56	**cppKeyword[signed]**
0.56	*whiteSpaceRatio*	0.55	ASTNodeAvgDep[argc]
0.55	**cppKeyword[unsigned]**	0.55	ASTNodesTF[tc]
0.55	ASTNodesTFIDF[tc]	0.54	ASTNodeAvgDep[argv]
0.51	**cppKeyword[class]**	0.50	ASTNodeAvgDep[const]
0.45	**ASTNodeAvgDep[ForInit]**	0.44	**cppKeyword[typename]**
0.44	**ASTNodesTF[ForInit]**	0.43	**cppKeyword[inline]**
0.43	**ASTNodeAvgDep[ForStatement]**	0.42	ASTNodeAvgDep[cas]
0.42	**ASTNodesTF[ForStatement]**	0.42	ASTNodesTF[cas]
0.42	ASTNodesTFIDF[cas]	0.40	ASTNodeAvgDep[std]
0.39	ASTNodesTF[in]	0.39	**cppKeyword[operator]**
0.39	ASTNodeAvgDep[fin]	0.35	ASTNodeAvgDep[fout]
0.35	ASTNodeAvgDep[ifstream]	0.35	ASTNodesTF[fin]
0.35	ASTNodesTFIDF[fin]	0.35	ASTNodesTF[fout]
0.35	ASTNodesTFIDF[fout]	0.34	**cppKeyword[static]**
0.32	ASTNodesTFIDF[ifstream]	0.32	ASTNodesTF[ifstream]
0.32	ASTNodeAvgDep[cerr]	0.31	ASTNodeAvgDep[ofstream]
0.30	ASTNodeAvgDep[cases]	0.30	ASTNodesTFIDF[cases]
0.30	ASTNodesTF[cases]	0.30	**cppKeyword[compl]**
0.29	ASTNodesTFIDF[ofstream]	0.29	ASTNodesTF[ofstream]
0.28	ASTNodesTFIDF[cerr]	0.28	ASTNodesTF[cerr]
0.28	ASTNodeAvgDep[inline]	0.27	ASTNodeAvgDep[FOR]
0.27	ASTNodesTF[FOR]	0.27	ASTNodesTFIDF[FOR]
0.27	ASTNodeAvgDep[sync_with_stdio]	0.26	**cppKeyword[auto]**
0.26	**cppKeyword[namespace]**	0.26	ASTNodesTFIDF[REP]
0.26	ASTNodesTF[REP]	0.26	ASTNodesTFIDF[fopen]
0.26	ASTNodesTF[fopen]	0.25	ASTNodeAvgDep[open]
0.25	ASTNodeAvgDep[REP]	0.25	**cppKeyword[using]**
0.25	ASTNodesTFIDF[out]	0.25	ASTNodesTF[out]
0.25	ASTNodesTFIDF[open]	0.25	ASTNodesTF[open]
0.24	ASTNodesTF[size_t]	0.24	ASTNodesTFIDF[size_t]
0.22	ASTNodeAvgDep[fopen]	0.22	ASTNodeAvgDep[size_t]
0.21	ASTNodesTFIDF[fprintf]	0.21	ASTNodesTF[fprintf]
0.20	ASTNodesTF[close]	0.20	ASTNodesTFIDF[close]
0.18	*newLineBeforeOpeningBrace*	0.18	ASTNodeAvgDep[close]
0.18	ASTNodeAvgDep[forn]	0.18	ASTNodesTFIDF[forn]
0.18	ASTNodesTF[forn]	**IG* = information gain**	
Syntactic features are in bold.			
Layout features are italicized.			
Lexical features are in regular font.			

Table VII
INFORMATION GAIN IN CODE STYLOMETRY FEATURE SET

IG*	Feature	IG*	Feature
1.41	cppKeyword[typedef]	1.07	ASTNodeAvgDep[UnaryExpression]
0.80	ASTNodeTF[UnaryExpression]	0.74	ASTNodeTF[ShiftExpression]
0.69	cppKeyword[case]	0.69	cppKeyword[template]
0.62	ASTNodeAvgDep[ShiftExpression]	0.61	ASTNodeAvgDep[IncDecOp]
0.58	ASTNodeTF[Statement]	0.57	ASTNodeTF[IncDecOp]
0.57	cppKeyword[const]	0.56	cppKeyword[signed]
0.55	cppKeyword[unsigned]	0.54	ASTNodeAvgDep[Statement]
0.51	cppKeyword[class]	0.44	cppKeyword[typename]
0.43	cppKeyword[inline]	0.43	ASTNodeAvgDep[ForInit]
0.43	ASTNodeAvgDep[ForStatement]	0.42	ASTNodeTF[ForInit]
0.42	ASTNodeTF[ForStatement]	0.39	cppKeyword[operator]
0.34	cppKeyword[static]	0.30	cppKeyword[compl]
0.26	cppKeyword[auto]	0.26	cppKeyword[namespace]
0.25	cppKeyword[using]	**IG* = information gain**	

Table VIII
INFORMATION GAIN IN SYNTACTIC FEATURE SET

determine significance while mitigating the small sample risk inherent in several of our tests.

Result1	Result2	p-value*
Is CSFS[1] better than SFS[2]		
Each dataset has 62 authors and specified number of files.		
2files_CSFS	2files_SFS	1.95×10^{-2}
3files_CSFS	3files_SFS	8.12×10^{-7}
4files_CSFS	4files_SFS	1.11×10^{-4}
5files_CSFS	5files_SFS	5.48×10^{-6}
6files_CSFS	6files_SFS	2.09×10^{-3}
7files_CSFS	7files_SFS	1.37×10^{-4}
8files_CSFS	8files_SFS	1.05×10^{-5}
9files_CSFS	9files_SFS	1.03×10^{-3}
10files_CSFS	10files_SFS	4.88×10^{-4}
11files_CSFS	11files_SFS	4.88×10^{-4}
12files_CSFS	12files_SFS	3.17×10^{-3}
13files_CSFS	13files_SFS	1.93×10^{-2}
14files_CSFS	14files_SFS	1.95×10^{-2}
*maximum/least significant p-values		
[1]: Code Stylometry Feature Set		
[2]: Syntactic Feature Set		

Table IX
CODE STYLOMETRY FEATURE SET VS SYNTACTIC FEATURE SET

In all cases in Table IX the CSFS outperforms SFS. In these cases the original source code has been sustained and there has been no obfuscation or tampering.

Result1	Result2	p-value*
Does obfuscation matter?		
25 authors 9Files_CSFS	25 authors 9Files_CSFS	4.90×10^{-1}
25 authors 9Files_SFS	25 aut. obfuscated-9Files_SFS	2.89×10^{-1}
*maximum/least significant p-values		

Table X
ORIGINAL SOURCE CODE VS OBFUSCATED DATA

We cannot distinguish between the classification accuracy for obfuscated and non-obfuscated source code for either set shown in Table X.

VI. EVALUATION

A. Validation Experiments

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 authors 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 89.87%. To investigate the slight decrease in accuracy compared to 91.73% obtained from 2014's main dataset, we extracted the *Code Stylometry Feature Set* and also the *Syntactic Feature Set* from 2012's dataset. The accuracy with all the features was 78.17% and 43.91% with the syntactic features. These are also a few percent lower than 2014's results of 80.98% and 44.29% respectively. In this case, we can attribute the slight drop in accuracy to noise and the different nature of 2012's dataset.

B. Information Gain in Validation Dataset

The information gain features from 2012 are a proper subset of the information gain features of 2014. Adding some dummy features to 2012's information gain to increase M to 200 and consequently increasing $logM + 1$, lead to 91.11% accuracy. Increasing the number of features in the random forest also increased the randomness in the decision process and eventually lead to slightly increased classification accuracy which is almost the same with the results from the main dataset. We have tried the same approach on a few datasets created from 2008, 2012, and 2014 with at most 250 files and 9 files per author. The information gain features always ended up being a proper subset of the information gain features generated from 2014 and the accuracy was in the same range with the results reported in the paper.

The validation results in Table XI 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 datasets. For a given dataset that has ground truth information, following the same approach should generate the most important features that represent coding style in that particular dataset.

A = #authors, 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 CSFS		
80.98%	78.17%	84.19%
Average accuracy after 10 iterations with SFS		
44.29%	43.91%	54.74%
Average accuracy after 10 iterations with IG-CSFS features		
91.73%	91.11%	95.33%

Table XI
SUMMARY OF LARGE SCALE EXPERIMENTS

C. Statistical Significance on Information Gain Reduction

While the p-value is low enough to be perhaps suggestive, we cannot distinguish between the accuracy obtained using the two feature sets in Table XII at the 0.05 level of significance.

Information gain features emphatically outperform the CSFS and SFS features as stated in Table XIII.

Result1	Result2	p-value*
Can we use 2014 features to classify 2012 files?		
250 Authors from 2012		
IG-CSFS-2014	IG-CSFS-2012	1.37×10^{-1}
*maximum/least significant p-values		

Table XII
Information Gain Features from 2014 and 2012

Result1	Result2	p-value*
Do infogain features outperform CSFS/SFS features?		
Each dataset has 250 authors. The feature set is in parentheses.		
2012-(IG-CSFS-2012)	2012-(CSFS)	1.51×10^{-2}
2012-(IG-CSFS-2012)	2012-(SFS)	2.73×10^{-25}
2014-(IG-CSFS-2014)	2014-(CSFS)	1.27×10^{-2}
2014-(IG-CSFS-2014)	2014-(SFS)	1.91×10^{-8}
*maximum/least significant p-values		

Table XIII
IG-CSFS vs CSFS and SFS

VII. Related Work

There has been a great amount of work done on authorship attribution of unstructured or semi-structured text. In this research, we are interested in structured text, source code in particular which has been explored up to an extent.

Burrows and Tahaghoghi [7] classified source code authors by looking at n-grams. Burrows et al. [8] also investigated C code of 1,597 programming assignments written by 100 students. Their most successful approach reached 76.78% classification accuracy in a 10 class problem, which we outperform in this work. Rosenblum et al. [29] present a novel program representation and techniques that automatically detect the stylistic features of binary code. Frantzeskou et al. [15] 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. 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 [5] is able to analyze the similarities of code written by different authors.

Kothari et al. [19] used lexical markers in source code such as line length and character 4-grams to build programmer profiles. Their highest accuracy in identifying 12 open source software developers was 76%, which we are significantly outperforming in this work. Shevertalov et al. [30] take discretized lexical features from source code, namely number of characters, words, and lines of text to improve authorship classification accuracy.

Spafford and Weeber [31] 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. Krsul and Spafford [20] try to find characteristics that represent coding style. They suggest that within a closed environment and with a specific number of programmers, identifying programming style should be possible and they focus on source code in C. They use layout features such as indentation, curly bracket usage, and lexical metrics, such as name lengths, comments, ratio of variables to lines of code and some syntactic features such as the usage of keywords, and some software engineering metrics. Their correct classification rate with multivariate discriminant analysis [18] is 73% with 29 authors.

Gray et al.[16] 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. Lange and Mancoridis [21] use 18 metrics that are based on layout and lexical features along with a genetic algorithm to obtain 75% correct classification accuracy among 20 authors. Pellin [25] uses abstract syntax trees with an SVM that has a tree based kernel to classify functions of two authors. 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 literature.

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.

Hayes and Offutt [17] examine coding style in source code by their consistent programmer hypothesis. They focused on lexical and layout features, such as the occurrence of semi colons, operators and constants. Their dataset consisted of 20 authors 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 V-C, we also show that more advanced programmers have a more identifying coding style.

A results summary of some of the related work and some of our results are listed in Table XIV for ease of comparison.

VIII. Discussion

The experiment that had random problems from random authors among seven years might sound closest to 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.

A malware author or plagiarizing programmer might deliberately try to hide his source code by obfuscation. Our

Related Work	Author Size	Instances	Avg. LOC	Lang	Method	Result
MacDonell et al.[23]	7	351	148	C++	Case-based reasoning	88.00%
Frantzeskou et al.[13]	8	107	145	Java	Nearest neighbor	100.0%
Elenbogen and Seliya [11]	12	83	100	C++	C4.5 Decision Tree	74.70%
Frantzeskou et al. [13]	30	333	172	Java	Nearest neighbor	96.9%
Ding and Samadzadeh [10]	46	225	N/A	Java	Nearest neighbor	67.2%
Ours	**35**	**245**	**68**	**C++**	**Random forest**	**98.37%**
Ours	**250**	**2250**	**77**	**C++**	**Random forest**	**95.33%**

Table XIV
COMPARISON OF OUR RESULTS TO PREVIOUS RESULTS

method is resistant to most off-the-shelf obfuscators that are used to make code look cryptic while preserving functionality. Obfuscation commonly happens through refactoring all names, removing comments and stripping space to make code look difficult to understand. Such changes do not have any effect on the syntactic features that will be extracted from the source code. Syntactic features extracted from the original code and the obfuscated code will be equivalent. Consequently, such an obfuscation method is not effective in hiding coding style. If the person trying to hide his coding style takes the effort to change the structure of the code after obfuscation, that would require as much effort as implementing the functionality from scratch which cannot really be considered plagiarism. On the other hand the commercial obfuscator experiments suggest that the code stylometry feature set might be the way to go even if obfuscation is suspected, as results while using CSFS are significantly higher than using syntactic only features. Our method is resilient to obfuscation in this scope and with this particular off-the-shelf obfuscator.

Our main experimental setting where everyone has only answered the easiest 9 problems is possibly the hardest the scenario can get, since we are training on the same set of 8 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 other hand, 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 though 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 newbies. There is another possibility that maybe better programmers start out with a more unique coding style. It is hard to say if good programmers are born or made.

If a feature is able to help the classification by itself, it will show up in the information gain feature set. Feature reduction through information gain eliminates features that have joint information gain. When joint information gain is present, excluding those feature pairs will affect the accuracy adversely. In our main dataset, the number of features was reduced to 137 after applying information gain to 20,000 *Code Stylometry Feature Set* features. Nevertheless, excluding feature pairs with joint information gain makes the information gain feature set smaller, which is quicker to extract from the source code and the classifier performs faster with a smaller number of features. Joint information gain can always be calculated in cases where an increase in accuracy is essential. On the other hand, since the syntactic feature set is not sparse we did not use its information gain to generate more results. We show features that have information gain in the syntactic feature set in Table VIII to show which syntactic features are more relevant to coding style.

In cases where the original author of the code is not actually the Google Code Jam contestant himself, we can detect those by a classify and then verify approach as explained in Stolerman et al.'s work [32]. 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.

IX. 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.

Adding extra dummy code to a file without affecting the functionality will represent a different coding style as long as the features from the extra code are extracted. The adversary can evade the classifier by this method but detecting such an addition to code is trivial through manual analysis.

X. FUTURE WORK

We would like to apply our approach to code in different languages and this is straightforward as long as we have the fuzzy abstract syntax tree parser for programming language in use. We can directly apply our methods to source code written in C, since Joern is able to parse both C and C++. Google Code Jam provides large datasets in numerous languages. Applying code stylometry to different programming languages will show which languages are more prone to revealing coding style. This might also show that some languages give the freedom to programmers to use individual coding style while implementing the same functionality. This might be an indication of a programming language's richness.

We have not tried to find the optimal classifier in these experiments, instead we focused on showing that coding style exists and it is possible to identify programmers in sets of hundreds of suspects. We picked a random forest for its good performance and to avoid overfitting. In future work, we would like to maximize our results and optimize our approach by figuring out the best performing classifier.

We would also like to improve our feature set by using more structural features. For example, we could use edges in the abstract syntax tree to connect two nodes and represent parts of speech bigrams. Narayanan et al. [24] show improved accuracy in large scale authorship attribution by the use of tree based features. In future work, we would like to incorporate more structural properties from abstract syntax tree leaves as well. We would also like to make use of control flow graphs in a similar manner to extracting syntactic features and come up with a principled approach for generating stylistic control flow representations.

We can try increasing the classification accuracy by using features that have joint information gain alongside features that have information gain by themselves. Calculating the joint information gain is quadratic so we left this piece to future work. A pair of features, three features,..., up to M features together might have positive information gain where M is the total number of extracted features. We can analyze why these features are only helpful in the presence of their pairs and under what conditions these pairs change.

We would like to test other off the shelf and preferably open source obfuscators that change the structure of source code and test our method on different types of obfuscation strategies.

XI. Conclusion

Source code stylometry has direct applications for software forensics, plagiarism, copyright infringement disputes, authorship verification, and privacy. We introduce the first principled use of syntactic features along with lexical and layout features to investigate style in source code. We can reach 92% accuracy in classifying more than 170 authors even with just five training files per author and reach 95% accuracy in classifying 250 authors with eight training files per class. This is a breakthrough in accuracy and scale in source code authorship attribution.

State-of-the-art natural language stylometry methods can identify individuals in sets of 50 authors with over 90% accuracy with the *Writeprints Feature Set* as shown in Abbasi and Chen's work [3]. *Code Stylometry Feature Set* scales even better than regular stylometric authorship attribution and requires less text. Authorship attribution requires 5,000 words of author's writing for training while source code authorship attribution requires on average 550 lines of code or eight solution files to train on.

Programmers who are more advanced and are able to solve more difficult tasks have more distinct coding styles than programmers that are not as advanced. Programmers with a larger skill set can be identified more easily and with higher confidence. Programmers reveal more individual coding style in more challenging programming tasks.

Syntactic features are preserved in binaries up to a degree among with some other features. This gives a promising direction for identifying authors of binaries and further extending this approach to malware classification.

Appendix

A. 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 XV
Abstract Syntax Tree Node Types

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 XVI
C++ Keywords

References

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

[2] November 2014. URL http://www.stunnix.com/prod/cxxo/.

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

[4] S. Afroz, M. Brennan, and R. Greenstadt. Detecting hoaxes, frauds, and deception in writing style online. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 461–475. IEEE, 2012.

[5] 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.

[6] L. Breiman. Random forests. *Machine Learning*, 45(1): 5–32, 2001.

[7] S. Burrows and S. M. Tahaghoghi. Source code authorship attribution using n-grams. In *Proc. of the Australasian Document Computing Symposium*, 2007.

[8] S. Burrows, A. L. Uitdenbogerd, and A. Turpin. Application of information retrieval techniques for source code authorship attribution. In *Database Systems for Advanced Applications*, pages 699–713. Springer, 2009.

[9] T. G. Dietterich. Approximate statistical tests for comparing supervised classification learning algorithms. *Neural Computation*, 10(7):1895–1923, 1998.

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

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

[12] R. A. Fisher. On the interpretation of $\chi 2$ from contingency tables, and the calculation of p. *Journal of the Royal Statistical Society*, pages 87–94, 1922.

[13] G. Frantzeskou, E. Stamatatos, S. Gritzalis, and S. Katsikas. Effective identification of source code authors using byte-level information. In *Proceedings of the 28th International Conference on Software Engineering*, pages 893–896. ACM, 2006.

[14] G. Frantzeskou, E. Stamatatos, S. Gritzalis, C. E. Chaski, and B. S. Howald. Identifying authorship by byte-level n-grams: The source code author profile (scap) method. *International Journal of Digital Evidence*, 6(1):1–18, 2007.

[15] G. Frantzeskou, S. MacDonell, E. Stamatatos, and S. Gritzalis. Examining the significance of high-level programming features in source code author classification. *Journal of Systems and Software*, 81(3):447–460, 2008.

[16] A. Gray, P. Sallis, and S. MacDonell. Software forensics: Extending authorship analysis techniques to computer programs. 1997.

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

[18] R. A. Johnson, D. W. Wichern, and P. Education. *Applied multivariate statistical analysis*, volume 4. Prentice hall Englewood Cliffs, NJ, 1992.

[19] J. Kothari, M. Shevertalov, E. Stehle, and S. Mancoridis. A probabilistic approach to source code authorship identification. In *Information Technology, 2007. ITNG'07. Fourth International Conference on*, pages 243–248. IEEE, 2007.

[20] I. Krsul and E. H. Spafford. Authorship analysis: Identifying the author of a program. *Computers & Security*, 16(3):233–257, 1997.

[21] R. C. Lange and S. Mancoridis. 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*, pages 2082–2089. ACM, 2007.

[22] F. Liddell. Simplified exact analysis of case-referent studies: matched pairs; dichotomous exposure. *Journal of Epidemiology and Community Health*, 37(1):82–84, 1983.

[23] S. G. MacDonell, A. R. Gray, G. MacLennan, and P. J. Sallis. 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*, volume 1, pages 66–71. IEEE, 1999.

[24] A. Narayanan, H. Paskov, N. Z. Gong, J. Bethencourt, E. Stefanov, E. C. R. Shin, and D. Song. On the feasibility of internet-scale author identification. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 300–314. IEEE, 2012.

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

[26] R. Pike. The sherlock plagiarism detector, 2011.

[27] L. Prechelt, G. Malpohl, and M. Philippsen. Finding plagiarisms among a set of programs with jplag. *J. UCS*, 8(11):1016, 2002.

[28] J. Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[29] N. Rosenblum, X. Zhu, and B. Miller. Who wrote this code? identifying the authors of program binaries. *Computer Security–ESORICS 2011*, pages 172–189, 2011.

[30] M. Shevertalov, J. Kothari, E. Stehle, and S. Mancoridis. On the use of discretized source code metrics for author identification. In *Search Based Software Engineering, 2009 1st International Symposium on*, pages 69–78. IEEE, 2009.

[31] E. H. Spafford and S. A. Weeber. Software forensics: Can we track code to its authors? *Computers & Security*, 12 (6):585–595, 1993.

[32] A. Stolerman, R. Overdorf, S. Afroz, and R. Greenstadt. Classify, but verify: Breaking the closed-world assumption in stylometric authorship attribution. In *IFIP Working Group 11.9 on Digital Forensics*. IFIP, 2014.

[33] Wikipedia. Saeed Malekpour, 2014. URL http://en. wikipedia.org/wiki/Saeed_Malekpour. [Online; accessed

04-November-2014].

[34] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, pages 499–510. ACM, 2013.

[35] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. In *Proc of IEEE Symposium on Security and Privacy (S&P)*, 2014.