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

Aylin Caliskan-Islam*, Fabian Yamaguchi †,
Edwin Dauber ‡ Richard Harang§, Konrad Rieck †,
Rachel Greenstadt ‡, and Arvind Narayanan*
*Princeton University, aylinc@princeton.edu, arvindn@cs.princeton.edu
†University of Goettingen, fabian.yamaguchi@cs.uni-goettingen.de, konrad.rieck@cs.uni-goettingen.de
‡Drexel University, egd34@drexel.edu, greenie@cs.drexel.edu
§ U.S. Army Research Laboratory, richard.e.harang.civ@mail.mil

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

We examine executable binary authorship attribution from the standpoint of machine learning, using a novel set of features that include ones obtained by decompiling the executable binary to source code. We show that many syntactical features present in source code do in fact survive compilation and can be recovered from decompiled executable binary. This allows us to add a powerful set of techniques from the domain of source code authorship attribution to the existing ones used for binaries, resulting in significant improvements to accuracy and scalability. We demonstrate this improvement on data from the Google Code Jam, obtaining attribution accuracy of up to 96% with 20 candidate programmers. We also demonstrate that our approach is robust to a range of compiler optimization settings, and binaries that have been stripped of their symbol tables. Finally, for the first time we are aware of, we demonstrate that authorship attribution can be performed on real world code found "in the wild" by performing attribution on single-author GitHub repositories.

***Index Terms*—Authorship attribution; Decompilation; Machine Learning;**

I. Introduction

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

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

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

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

These innovations allow us to significantly improve the scale and accuracy of programmer de-anonymization compared to Rosenblum et al.'s work. We performed experiments with a controlled dataset collected from Google Code Jam, allowing a direct comparison since the same dataset was used in the previous work. The results of these experiments are discussed in detail in Section V. Specifically; for an equivalent accuracy we are able to distinguish between *five times* as many candidate programmers (100 vs. 20) while utilizing a smaller number of training samples per programmer. Alternately, holding the

number of programmers constant, our accuracy jumps to 78% compared to 61%, with a smaller number of training samples. The accuracy of our method degrades gracefully as the number of programmers increases, and we present experiments with as many as 600 programmers. Similarly, we are able to tolerate scarcity of training data: our accuracy for de-anonymizing sets of 20 candidate programmers with just a single training sample per programmer is over 75%.

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

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

All these experiments were carried out using the controlled Google Code Jam dataset; the availability of this dataset is a boon for research in this area since it allows us to develop and benchmark our results under controlled settings [10, 40]. Having done that, we present a case study with a real-world dataset collected from GitHub in Section VI-C. This data presents difficulties, particularly limited training data. However, we show that our approach classifies a set of programmers with only one training executable binary sample (and one testing executable binary sample) with 62% accuracy and that the accuracy improves greatly in cases where more executable binary training samples are available.

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

II. Problem Statement

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

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

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

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

Since we assume that a set of candidate programmers is known, we treat it as a closed-world, supervised machine learning task. It is a multi-class machine learning problem where the classifier calculates the most likely author for the anonymous executable binary sample among multiple authors.

Additional Assumptions. For our experiments, we assume that we know the compiler used for a given program binary. Previous work has shown that with only 20 executable binary samples per compiler as training data, it is possible to use a linear Conditional Random Field [26] to determine the compiler used with accuracy of 93% on average [41]. Other work has shown that by using pattern matching, library functions can be identified with precision and recall between 0.98 and 1.00 based on each of three criteria; compiler version, library version, and linux distribution [24].

In addition to knowing the compiler, we assume we know the optimization level used for compilation of the binary. Past work has shown that toolchain provenance, including compiler family, version, optimization, and source language, can be identified with a linear Conditional Random Field with accuracy of 99.9% for language, compiler family, and optimization and 91.9% for compiler version [39]. More recent work has looked at a stratified approach which, although having lower

accuracy, is designed to be used at the function level to enable preprocessing for further tasks including authorship attribution [37]. Due to the success of these techniques, we make the assumption that these techniques will be used to identify the toolchain provenance of the executable binaries of interest and that our method will be trained using the same toolchain.

III. Related Work

Any domain of creative expression allows authors or creators to develop a unique style, and we might expect that there are algorithmic techniques to identify authors based on their style. This class of techniques is called stylometry. Natural-language stylometry, in particular, is well over a century old [30]. Other domains such as source code and music also have linguistic features, especially grammar. Therefore stylometry is applicable to these domains as well, often using strikingly similar techniques [11, 43].

Linguistic stylometry. The state of the art in linguistic stylometry is dominated by machine-learning techniques [e.g., 7, 8, 31]. Linguistic stylometry has been applied successfully to security and privacy problems, for example Narayanan et al. used stylometry to identify anonymous bloggers in large datasets, exposing privacy issues [31]. On the other hand, stylometry has also been used for forensics in underground cyber forums. In these forums the text consists of a mixture of languages and information about underground forum products, which makes it more challenging to identify personal writing style. Not only have the forum users been de-anonymized but also their multiple identities across and within forums have also been linked through stylometric analysis [8].

Authors may deliberately try to obfuscate or anonymize their writing style [7, 14, 29]. Brennan et al. [14] show how stylometric authorship attribution can be evaded with adversarial stylometry. They present two ways for adversarial stylometry, namely obfuscating writing style and imitating someone else's writing style. Afroz et al. [7] identify the stylistic changes in a piece of writing that has been obfuscated while [29] present a method to make writing style modification recommendations to anonymize an undisputed document.

Source code stylometry. Several authors have applied similar techniques to identify programmers based on source code [e.g., 15, 18, 33]. It has applications in software forensics and plagiarism detection.[1]

The features used for machine learning in these works range from simple byte-level [22] and word-level n-grams [16, 17] to more evolved structural features obtained from abstract syntax trees [18, 33]. In particular, Burrows et al. [17] present an approach based on n-grams that reaches an accuracy of 76.8% in differentiating 10 different programmers.

Similarly, Kothari et al. [25] combine n-grams with lexical markers such as the line length, to build programmer profiles that allow them to identify 12 authors with an accuracy of 76%. Lange and Mancoridis [27] further show that metrics based on layout and lexical features along with a genetic algorithm allow an accuracy of 75% to be obtained for 20 authors. Finally, Caliskan-Islam et al. [18] incorporate abstract syntax tree based structural features to represent programmers' coding style. They reach 94% accuracy in identifying 1,600 programmers of the Google Code Jam data set.

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

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

Rosenblum et al. [40] use the Paradyn project's Parse API for parsing executable binaries to get the instruction sequences and control flow graphs whereas we use four different resources to parse executable binaries to generate a richer representation. Their dataset consists of Google Code Jam and homework assignment submissions. A linear support vector machine classifier [20] is trained on the numeric representations of varying numbers of executable binaries. Google Code Jam programmers have eight to sixteen files and students have four to 7 files. Students collaborated on the homework assignments and the skeleton code was available.

Malware attribution. While the analysis of malware is a well developed field, authorship attribution of malware has received much less attention. Stylometry may have a role in this application, and this is a ripe area for future work. The difficulty in obtaining ground truth labels for samples has led much work in this area to focus on clustering malware in some fashion, and the wide range of obfuscation techniques in common use have led many researchers to focus on dynamic analysis rather than the static features we consider. The work of Marquis-Boire et al. [28] examines several static features intended to provide credible links between executable malware binary produced by the same authors, however many of these features are specific to malware, such as command and control infrastructure and data exfiltration methods, and the authors note that many must be extracted by hand. In dynamic analysis, the work of Pfeffer et al. [34] examines information obtained via both static and dynamic analysis of malware samples to organize code samples into lineages that indicate the order in which samples are derived from each other. Bayer et al. [12] convert detailed execution traces from dynamic analysis into more general behavioral profiles, which are then used to cluster malware into groups with related functionality

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

and activity. Supervised methods (specifically a support vector machine) are used by Rieck et al. [38] to match new instances of malware with previously observed families, again on the basis of dynamic analysis.

IV. Approach

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

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

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

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

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

A. Feature extraction via disassembly

As a first step, we disassemble the executable binary to extract low-level features that have been shown to be suitable for authorship attribution in previous work. In particular, we follow the example set by Rosenblum et al. and extract raw instructions from the executable binary [40]. In addition to this, disassemblers commonly make available symbol information as well as strings referenced in the code, both of which greatly simplify manual reverse engineering. We augment the feature set with this information accordingly, to make use of it for executable binary authorship attribution. We obtain this information by querying the following two disassemblers.

- ***The Netwide Disassembler.*** We begin by exploring whether simple instruction decoding alone can already provide useful features for de-anonymization. To this end, we process each executable binary using the netwide disassembler (*ndisasm*) [42], a rudimentary disassembler that is capable of decoding instructions but is unaware of the executable's file format. Due to this limitation, it resorts to simply decoding the executable binary from start to end, skipping bytes when invalid instructions are encountered. A problem with this approach is that no distinction is made between bytes that represent data, and bytes that represent code. We explore this simplistic approach nonetheless as these inaccuracies may not be relevant given the statistical nature of machine learning approaches.
- ***The Radare2 Disassembler.*** We proceed to apply the *radare2* disassembler [32], a state-of-the-art open-source disassembler based on the capstone disassembly framework [36]. In contrast to the *ndisasm*, *radare2* understands the executable binary format, allowing it to process relocation and symbol information in particular. We make use of this to extract symbols from the dynamic (`.dynsym`) as well as the static symbol table (`.symtab`) where present, as well as any strings referenced in the code. In particular, our approach thus gains knowledge over functions of dynamic libraries used in the code.

For both disassemblers, we subsequently tokenize their output, and create token uni-grams and bi-grams, which serve as our disassembly features.

B. Feature extraction via decompilation

Decompilers are the second source of information that we consider for feature extraction in this work. In contrast to disassemblers, decompilers do not only uncover the program's machine code instructions, but additionally reconstruct higher level constructs in an attempt to translate an executable binary into equivalent source code. In particular, decompilers can reconstruct control structures such as different types of loops and branching constructs. We make use of these syntactical features of code as they have been shown to be valuable in the context of source code authorship attribution [18]. Similar to our experiments on features gained from disassemblers, we assess both the value of the data produced by a sophisticated state-of-the-art decompiler (Hex-Rays []), as well as a more simple open-source decompiler (Snowman [46]).

1) ***Obtaining features from Hex-Rays****:* Hex-Rays [1] is a commercial state-of-the-art decompiler. It converts executable programs into a human readable C-like pseudo code to be read by human analysts. It is noteworthy that this code is typically significantly longer than the original source code. For example, decompiling an executable binary generated from 70 lines of source code with Hex-Rays produces on average 900 lines

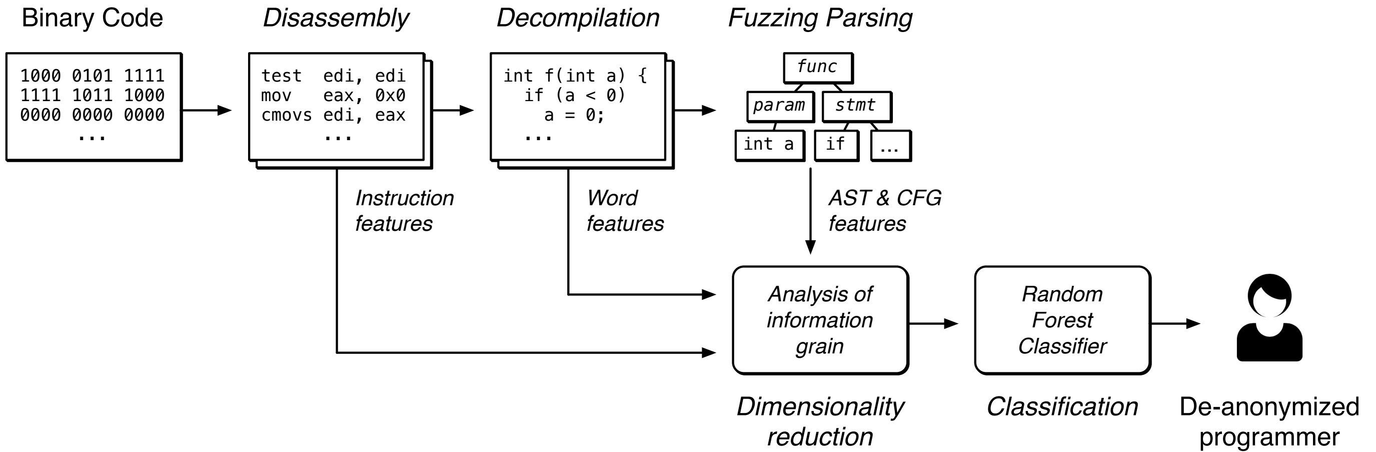

Figure 1. Overview of our method. Instructions, symbols, and strings are extracted using disassemblers (1), syntactical and control-flow features are obtained from decompilers (2). Dimensionality reduction is performed to obtain representative features (3). Finally, a random forest classifier is trained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

of decompiled code. We extract two types of features from this pseudo code: lexical features, and syntactical features. Lexical features are simply the word unigrams, which capture the integer types used in a program, names of library functions, and names of internal functions when symbol information is available. Syntactical features are obtained by passing the C-pseudo code to *joern* [45], a fuzzy parser for C that is capable of producing fuzzy abstract syntax trees (ASTs) from Hex-Rays pseudo code output. We derive syntactic features from the abstract syntax tree, which represent the grammatical structure of the program. Such features are AST node unigrams, labeled AST edges, AST node term frequency inverse document frequency (TFIDF), and AST node average depth. Previous work on source code authorship attribution [18, 44] shows that these features are highly effective in representing programming style. Fig 2 illustrates this process.

*2) **Obtaining features from Snowman**:* *Snowman* is an open source decompiler, which supports multiple different instruction sets and executable binary formats, including the Intel 32 bit instruction set and the ELF binary format considered in this work. In addition to decompiling code, *Snowman* constructs control flow graphs. The nodes of the control flow graph are the *basic blocks*, that is, sequences of statements that are known to always be executed in direct succession. These blocks provide a natural segmentation of a programs statements, which our method exploits. To this end, we normalize the code in basic blocks, and treat each basic block as a word. We then extract word uni-grams and bi-grams, that is, single basic blocks and sequences of two basic blocks, and apply the TF-IDF weighting scheme [] to obtain our final features.

C. Dimensionality Reduction via information gain

Our feature extraction process based on disassemblers and decompilers produces a large amount of features, resulting in sparse feature vectors with thousands of elements. However,

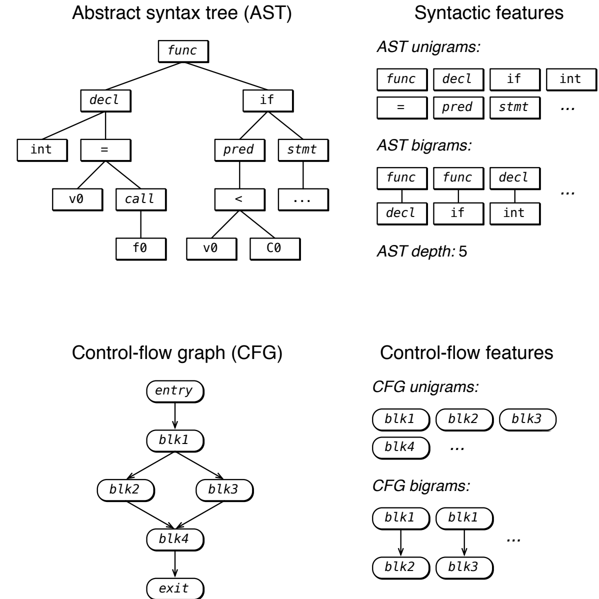

Figure 2. Feature extraction using decompilation and fuzzy parsing: the C-like pseudo code produced by hexrays is transformed into an abstract syntax tree and control-flow graph to obtain syntactic and control-flow features.

not all features are equally relevant to express a programmer's style. This makes it desirable to perform feature selection in order to obtain a more compact representation of the data that reduces the computational burden during classification. Moreover, sparse feature vectors may result in large number of zero-valued attributes being selected during random forest's random subsampling of the attributes to select a best split. Reducing the dimensions of the feature set is also important for avoiding overfitting. One example to overfitting would be a

rare assembly instruction uniquely identifying an author. For these reasons, we use information gain criteria to select the most informative attributes that represent each author as a class. This reduces vector size and sparsity while increasing accuracy and machine learning model training speed. For example, we get 200,000 features from the 900 executable binary samples of 100 programmers. If we use all of these features in classification, the accuracy is slightly above 20% because the random forest might be randomly selecting features with values of zero in the sparse feature vectors. Once the dimension of the feature vector is reduced, we get less than 500 information gain features. Extracting less than 500 features or training a machine learning model where each instance has less than 500 attributes is computationally efficient. On the other hand, no sparsity remains in the feature vectors after dimensionality reduction which is one reason for the performance benefits of dimensionality reduction. After dimensionality reduction, the correct classification accuracy of 100 programmers increases from 20% to close to 80%.

We employed the dimensionality reduction step using WEKA's [23] information gain [35] attribute selection 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 ith 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. We refer to these features as IG-features 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 that have demonstrable value in classification are included in our selected features.

D. Classification

We used the random forest ensemble classifier [13] as our classifier for programmer de-anonymization. Random forests are ensemble learners built from collections of decision trees, each of which is trained on a subsample of the data obtained by randomly sampling N training samples with replacement, where N is the number of instances in the dataset. 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 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-F).

We employed random forests with 500 trees, which empirically provided the best tradeoff between accuracy and processing time. Examination of numerous out of bag error values across multiple fits suggested that $(logM) + 1$ random features (where M denotes the total number of features) at each split of the decision trees was in fact optimal in all of the experiments listed in Section 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.

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

V. Experiments with Google Code Jam Data

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

A. Dataset

We evaluate our executable binary authorship attribution method on a dataset based on the annual programming competition *Google Code Jam* [5]. It is an annual contest that thousands of programmers take part in each year, including professionals, students, and hobbyists from all over the world. The contestants implement solutions to the same tasks in

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

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

B. Code Compilation

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

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

Feature	**Source**	**Number of Possible Features**	**Information Gain Features**
word unigrams	hex-rays decompiled code	29,686	102
AST node TF*	hex-rays decompiled code	14,663	24
Labeled AST edge TF*	decompiled code	25,941	88
AST node TFIDF**	decompiled code	14,663	8
AST node average depth	decompiled code	14,663	21
C++ keywords	decompiled code	73	5
radare2 disassembly unigrams	radare disassembly	12,629	45
radare2 disassembly bigrams	radare disassembly	33,919	75
ndisasm disassembly unigrams	ndisasm disassembly	532	8
ndisasm disassembly bigrams	ndisasm disassembly	4,570	25
CFG unigrams	Snowman CFG	11,503	5
CFG unigram TFIDF**	Snowman CFG	11,503	10
CFG bigrams	Snowman CFG	38,554	10
Total		201,396	426
TF = term frequency*			
*TFIDF** = term frequency inverse document frequency*			

Table I
PROGRAMMING STYLE FEATURES IN EXECUTABLE BINARIES

C. Dimensionality Reduction

We are interested in identifying features that represent coding style preserved in executable binaries. With the current approach, we extract more than 200,000 representations of code properties of 100 authors, but only a subset of these representations are the result of individual programming style. We are able to capture the features that represent each author's programming style that is preserved in executable binaries by applying information gain criteria to these 200,000 features. After applying information gain, we reduce the feature set size to 426 to effectively represent coding style properties that were preserved in executable binaries. Considering the fact that we are reaching such high accuracies on de-anonymizing 100 programmers with 900 executable binary samples (discussed below), these features are providing strong representation of style that survives compilation. We also see that all of our feature sources, namely disassembly, CFG, and decompiled code are capturing the preserved coding style.

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

This is the main experiment that demonstrates how de-anonymizing programmers from their executable binaries is possible. After preprocessing the dataset to generate the executable binaries without optimization, we further process the executable binaries to obtain the disassembly, control flow graphs, and decompiled source code. We then extract all the possible features detailed in Section IV. We take a set of 20 programmers who all have 14 executable binary samples. With 14-fold-cross-validation, the random forest classifier correctly classifies 280 test instances with 96.0% accuracy. With a set of 100 programmers who all have 9 executable binary samples, the random forest classifier correctly classifies 900 instances with 78.3%, which is significantly higher than the accuracies reached in previous work. Table II summarizes programmer de-anonymization results with different number of authors and samples.

There is an emphasis on the number of folds used in these experiments because each fold corresponds to the implementation of the same algorithmic function by all the programmers in the GCJ dataset (e.g. all samples in fold 1 may be attempts by the various authors to solve a list sorting problem). Since we know that each fold corresponds to the same Code Jam problem, by using stratified cross validation without randomization, we ensure that all training and test samples contain the same algorithmic functions implemented by all of the programmers. The classifier uses the excluded fold in the testing phase, which contains executable binary samples

that were generated from an algorithmic function that was *not* previously observed in the training set for that classifier. Consequently, the only distinction between the test instances is the coding style of the programmer, without the potenially confounding effect of identifying an algorithmic function.

Number of Programmers	Number of Training Samples	Cross Validation	Accuracy
20	5	5-fcv*	95.0%
20	13	13-fcv*	96.0%
20	8	8-fcv*	96.0%
100	8	8-fcv*	78.3%
**k-fcv refers to k-fold cross validation*			

Table II
PROGRAMMER DE-ANONYMIZATION

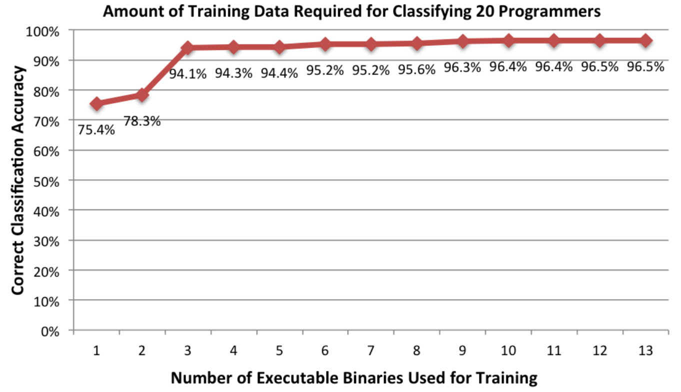

Figure 3. Required Amount of Training Data for De-anonymizing 20 Programmers

*E. **Even a single training sample per programmer is sufficient for de-anonymization.***

A drawback of supervised machine learning methods, which we employ, is that they require labeled examples to build a model. The ability of the model to accurately generalize is often strongly linked to the amount of data provided to it during the training phase, particularly for complex models. In domains such as executable binary authorship attribution, where samples may be rare and obtaining “ground truth” for labeling training samples may be costly or laborious, this can pose a significant challenge to the usefulness of the method.

We therefore devised an experiment to determine how much training data is required to reach a stable classification accuracy, as well as to explore the accuracy of our method with severely limited training data. As programmers produce a limited number of code samples per round of the GCJ competition, and programmers are eliminated in each successive round, the GCJ dataset has an upper bound in the number of code samples per author as well as a limited number of authors with a large number of samples. Accordingly, we identified a set of 20 programmers that had exactly 14 program samples each, and examined the ability of our method to correctly classify each author out of the candidate set of 20 authors when training on between 1 and 13 files per author.

As shown in Figure 3, the classifier is capable of correctly identifying the author of a code sample from a potential field of 20 with 75.4% accuracy on the basis of a single training sample. The classifier also reaches a point of dramatically diminishing returns with as few as three training samples, and obtains a stable accuracy by training on 6 samples. Given the complexity of the task, this combination of high accuracy with extremely low requirement on training data is remarkable, and suggests the robustness of our features and method. It should be noted, however (see Section V-K) that this set of programmers with a large number of files corresponds to more skilled programmers, as they were able to remain in the competition for a longer period of time and thus produce this large number of samples.

To examine the question of programmer skill, and to examine the analogue to a perhaps more realistic setting where a much larger field of candidate authors is available, we examine a larger set of 100 authors who were successful enough in the competition to produce 9 code samples. As in the previous experiment, we train the classifier on from 1 to 8 code samples per author, and ask it to correctly classify the authorship of a test sample out of the potential set of 100. Due to the larger set of class labels to choose from and the lower overall skill level of the authors represented in this set, we expect this task to be substantially more difficult. Indeed, Figure 4 shows that, while we are able to obtain relatively high classification accuracy even over a field of 100 programmers with a limited number of samples (64.4% accuracy using just 4 samples per author, compare to approximately 61% accuracy using 8 to 16 files per author in Rosenblum et al. [40]), the accuracy keeps increasing as more training data is utilized by the random forest model and does not display the sharp cutoff observed in Figure 3. In section V-F, we examine relaxed attribution methods for reducing the effective number of classes, allowing us to improve accuracy further, at the cost of a higher manual inspection burden on the analyst.

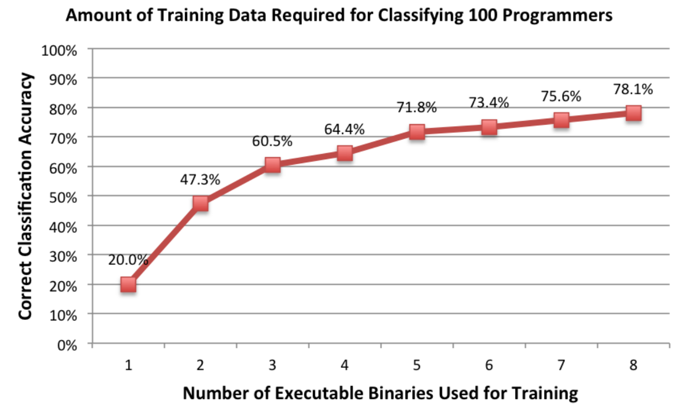

Figure 4. Required Amount of Training Data for De-anonymizing 100 Programmers

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

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

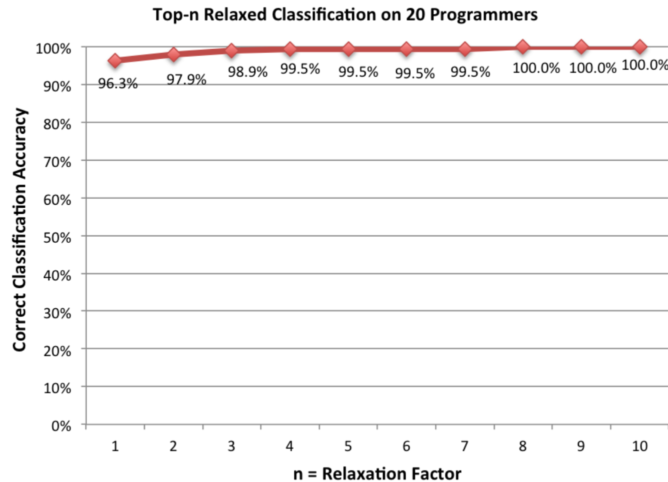

Figure 5. De-anonymizing 20 Programmers

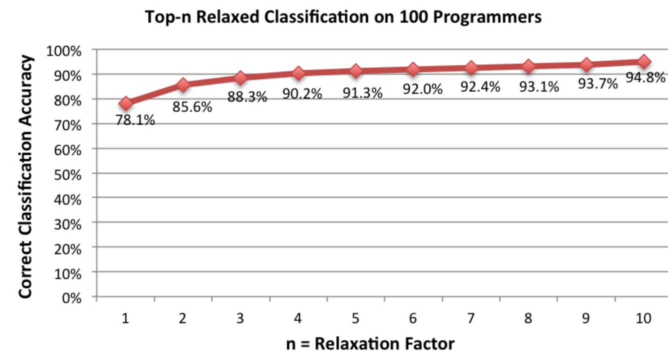

Figure 6. De-anonymizing 100 Programmers

It is important to note from Figure 3 that, by using only a single training sample in a 20-class classification task, the machine learning model can correctly classify new samples with 75.0% accuracy. This is of particular interest to an analyst or adversary who does not have a large amount of labeled samples in her suspect set. Figure 3 shows that an analyst or adversary can narrow down the suspect set size from 20 authors to 4 authors, guaranteed that the main suspect is among the 4, by using only one training sample from each of the 20 authors.

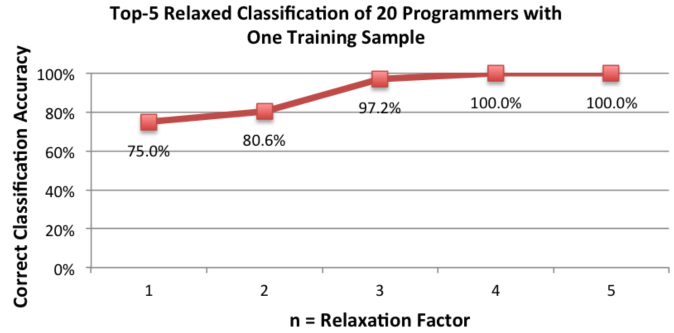

Figure 7. De-anonymizing 20 Programmers with One Training Sample

G. *The feature set selected via information gain works across different sets of programmers.*

In our earlier experiments, we trained the classifier on the same set of binaries that we used for selecting features via information gain. While this is a perfectly valid choice, we would like to compare it to the accuracy achieved if we had chosen different sets of programmers for computing information gain and for training the classifier. If we are able to reach accuracies similar to what we got earlier, we can conclude that these information gain features are not overfitting to the 100 programmers they were generated from. This also implies that the information gain features in general capture programmer style.

Recall that analyzing 900 executable binary samples of the 100 programmers resulted about 200,000 features, and after dimensionality reduction, we are left with 426 information gain features. We picked a different (non-overlapping) set of 100 programmers and re-did the de-anonymization experiment. This resulted in very similar accuracies: we obtained 78.3% accuracy with the first set of programmers, as reported above, compared to 77.9% with the validation set of programmers. This confirms that the information gain features obtained from the main set of 100 programmers do actually represent coding style in executable binaries and can be used across different datasets.

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

We would like to see how well our method scales up to 600 users. An analyst with a large set of labeled samples might be interested in performing large scale de-anonymization. For this experiment, we use 600 contestants from GCJ with 9 files. We only extract the information gain features from the 600 users. This reduces the amount of time required for feature

extraction. On the other hand, this experiment shows how well the information gain features represent overall programming style. The results of large scale programmer de-anonymization are in Figure 8.

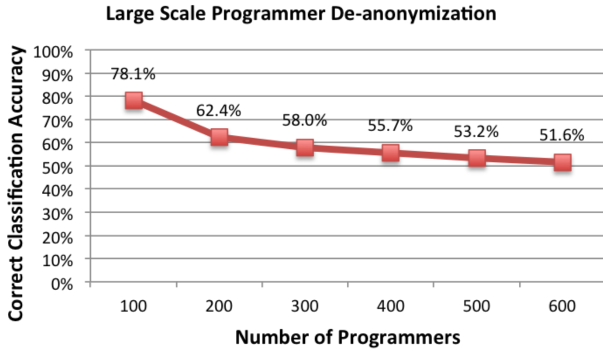

Figure 8. Large Scale Programmer De-anonymization

*I. **We advance the state of executable binary authorship attribution.***

Rosenblum et al. [40] present the largest scale evaluation of executable binary authorship attribution in related work. [40]'s largest dataset contains 191 programmers with at least 8 training samples per programmer. We compare our results with [40]'s and in Table III show how we advance the state of the art both in accuracy and on larger datasets. [40] use 1,900 coding style features to represent coding style whereas we use 426 features, which might suggest that our features are more powerful in representing coding style that is preserved in executable binaries.

Related Work	Number of Programmers	Number of Training Samples	Accuracy
Rosenblum et al.	**20**	8-16	77%
This work	**100**	8	78%
Rosenblum et al.	20	**8-16**	77%
This work	20	**2**	78%
This work	20	**6**	95%
This work	20	**8**	96%
Rosenblum et al.	100	8-16	**61%**
This work	100	8	**78%**
Rosenblum et al.	191	8-16	**51%**
This work	191	8	**63%**
This work	**600**	8	**52%**

Table III
COMPARISON TO PREVIOUS RESULTS

*J. **Programmer style is preserved in executable binaries.***

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

To see how well we can reconstruct the original source code features from decompiled executable binary features, we reconstructed the 900 instances with 150 features that represent the highest information gain features. We calculated the cosine similarity between the original 900 instances and the reconstructed instances after normalizing the features to unit distance. The cosine similarity for these instances is in Figure 9, where a cosine similarity of 1 means the two feature vectors are identical. The high values (average of 0.81) in cosine similarity suggest that the reconstructed features are similar to the original features. When we calculate the cosine distance between the feature vectors of the original source code and the corresponding decompiled code's feature vectors (*no predictions*), the average cosine distance is 0.35. This result suggests that the predicted features are much similar to original source code than the features extracted from decompiled code but decompiled code still preserves transformed forms of the original source code features well enough to reconstruct the original source code features.

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

In order to investigate the effect of programmer skill set on coding style that is preserved in executable binaries, we took two sets with 20 programmers. We considered the GCJ contestants who were able to advance to more difficult rounds as more advanced programmers as opposed to contestants that were eliminated in easier rounds. The programmers with more advanced skill sets were able to solve 14 problems and the programmers that had a less advanced skill set were only able to solve 7 problems. All of the 40 programmers had implemented the same 7 problems from the easiest rounds. We were able to identify the more advanced 20 programmers with 88.2% accuracy while we identified the less advanced 20 programmers with 80.7% accuracy. This might indicate that, programmers who are advanced enough to answer 14 problems likely have more unique coding styles compared to contestants that were only able to solve the first 7 problems.

To investigate the possibility that contestants who are able to

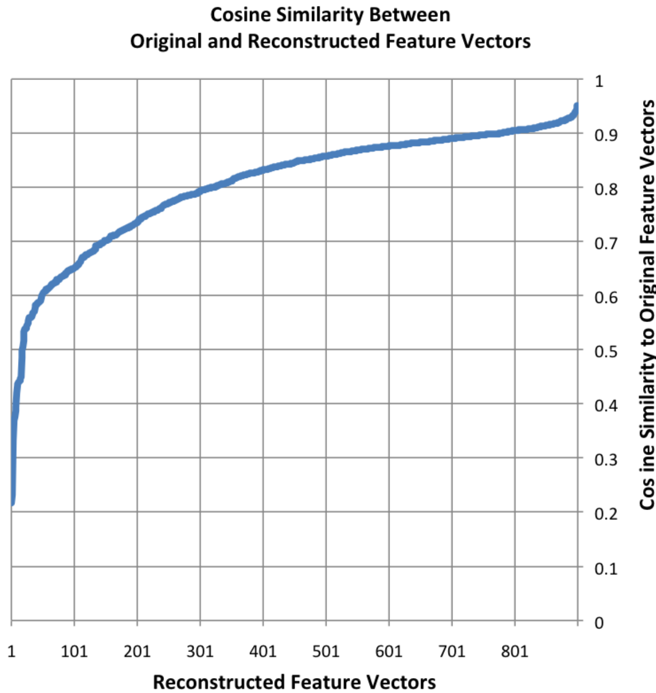

Figure 9. Feature Transformations: Each data point on the x-axis is a different executable binary sample. Each y-axis value is the cosine distance between the feature vector extracted from the original source code and the feature vector that tries to *predict* the original features. The average value of these 900 cosine distance measurements is 0.81.

advance further in the rounds have more unique coding styles, we performed a second round of experiments on comparable datasets. We took a dataset with 6 solution files and 20 authors and also a dataset that contains these 6 files but has 12 files in total and 20 programmers. We were able to identify the more advanced 20 programmers with 86.7% accuracy while we identified the less advanced 20 programmers with 78.1% accuracy. These results suggest that programmer skill set has an effect on coding style, and this effect on coding style is preserved in compilation.

A = #authors, F = max #problems completed			
N = #problems included in dataset ($N \leq F$)			
A = 20			
F = 14	**F = 7**	**F = 12**	**F = 6**
N = 7 easier	**N = 7**	**N = 6 easier**	**N = 6**
Average accuracy after 10 iterations			
88.2%	80.7%[1]	86.7%	78.1%[1]
[1]**Drop in accuracy due to programmer skill set.**			

Table IV
EFFECT OF PROGRAMMER SKILL SET ON CODING STYLE PRESERVED IN EXECUTABLE BINARIES

VI. EXPERIMENTS WITH REAL WORLD SCENARIOS

A. *Compiler Optimization: Programmers of optimized executable binaries can be de-anonymized.*

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

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

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

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

Number of Programmers	**Number of Training Samples**	**Compiler Optimization Level**	**Accuracy**
100	8	None	78.3%
100	8	1	64.2%
100	8	2	61.3%
100	8	3	60.1%

Table V
PROGRAMMER DE-ANONYMIZATION WITH COMPILER OPTIMIZATION

B. *Removing symbol information does not anonymize executable binaries.*

To investigate the relevance of symbol information for classification accuracy, we repeat our experiments with 100 authors presented in the previous section on *fully stripped*

executable binaries, that is, executable binaries where symbol information is missing completely. We obtain these executable binaries using the standard utility *GNU strip* on each executable binary sample prior to analysis. Upon removal of symbol information, without any optimizations, we notice a drop in classification accuracy by 12.4%, showing that stripping symbol information from executable binaries is not effective enough to anonymize an executable binary sample.

C. GitHub Programmer De-anonymization in the "Wild"

We developed our method and evaluated it on the Google Code Jam dataset, but collecting code from open source projects is another option for constructing a dataset. Open source projects do not guarantee ground truth on authorship. The feature vectors might capture topics of the project instead of programming style. As a result, open source code does not constitute the ideal data for authorship analysis; however, it allows us to better assess the applicability of programmer de-anonymization in the wild. We therefore present results from a dataset collected from the hosting platform GitHub, which we obtain by spidering the platform to collect C and C++ repositories. We collect two GitHub [6] datasets in C and C++. The first GitHub dataset consists of repositories that have been authored by multiple programmers, whereas the second GitHub dataset includes repositories authored by a single programmer. Table VI shows the statistics of the first dataset.

Type	Amount
Authors	118
Repositories	338
Files	4112
Repositories / Author	2 – 10
Files / Author	10 – 203

Table VI
MULTIPLE AUTHORED GITHUB REPOSITORIES

The conditions the files in Table VI satisfy are as follows:
Crawling:

- The repository is marked as C/C++ by Github.
- The repository is not a fork of another repository.
- The repository has at least 10 stars on Github.
- Moreover, some repository names are black-listed, such as repositories which are often mirrors or very large. The current blacklist is 'linux', 'kernel', 'osx', 'gcc', 'llvm', 'next'.

Cloning: The collected repositories are cloned from Github, such that all files are available for analysis. This enables further filtering of the data. Files are only included in the dataset if they satisfy the following conditions:

- The file name ends with a typical C/C suffix, such as 'c' and 'cpp'.
- The file contains at least 50 lines.
- The file has a main author that is "git-blamed" for 90% of the lines.
- The main author contributed at least 5 commits to the file.
- The commit messages do not contain 'signed-off'.

Indexing:
In the final step, the remaining files are indexed and further filtered using the identified authors. In particular, an author and her/his files are only included in the dataset if they satisfy the following conditions.

- The author has contributed to at least 2 repositories
- The author has written at least 10 different files

The second dataset has the statistics listed in Table VII. There have been two additions to the crawling method when compared to the dataset in Table VI. This dataset includes repositories if only one author has committed to the repository and the total number of lines of code is at least 500. While cloning, the file does not need to contain at least 50 lines. The one and only main author must be "git-blamed" for 100% of the lines. There is no limit on the number of different files that need to be authored by the main programmer.

Type	Amount
Authors	49
Repositories	117
Files	782
Repositories / Author	2 – 5
Files / Author	2 – 88

Table VII
SINGLE AUTHORED GITHUB REPOSITORIES

At this stage, we had a set of C and C++ source files divided into repositories for each author. In order to compile the code, we needed the entire repositories. Therefore, the final step in data collection was to download the zipped folder for each selected repository.

We can de-anonymize GitHub programmers.
Generating executable binaries from GitHub repositories requires manual effort. Furthermore, some repositories are unable to be compiled to 32-bit Intel 80386 Unix executable binaries. Therefore, we compiled a subset of the collected repositories to obtain 50 executable binary samples from 12 authors. The number of executable binary samples per author ranges from 2 to 11, with most authors having 2 or 3 samples. Some of the repositories had a single author and some had multiple authors. We extract the information gain features obtained from GCJ data from this GitHub dataset. Overall, we reach 62.0% accuracy in correctly identifying the author of an executable binary sample.

GitHub datasets are noisy for two reasons. First, most of the executable binaries are the result of a collaborative effort. However, in our dataset at least 90% of the source code used to generate the executable binary can be attributed to one main GitHub programmer. Second, the executable binary used in de-anonymization might contain properties from third party libraries and code. For these two reasons, it is more difficult to attribute authorship to anonymous executable binary samples from GitHub, but nevertheless we reach 62.0% accuracy in correctly classifying these programmers' executable binaries.

Another difficulty in this particular dataset is that there is not much training data to train an accurate random forest

classifier that models each programmer. For example, we can de-anonymize the two programmers with the most samples, one with 11 samples and one with 7, with 100% accuracy.

We perform the classification in different ways to see how the accuracy changes according to the number of folds. For classification, we use 50% of the samples for training a model and the remaining data for testing. We do this with all possible permutations to use all instances as both training and testing data. The accuracy ranges between 60% and 63%, and the average is 62%. We also take a random dataset from GCJ with 12 programmers that have the same number of files as the GitHub programmers in this case study. By using the same classification methods, the accuracy on a GCJ dataset with 12 programmers and 50 files is 68.0%. These results are summarized in Table VIII and show that the IG-features we obtain from GCJ are a robust representation of coding style preserved in executable binaries and also that our method performs similarly well on noisy datasets.

Being able to de-anonymize programmers in the wild by using less than 500 stylistic features obtained from our clean evaluation dataset is a promising step towards attacking more challenging real world de-anonymization problems.

Dataset	**Authors**	**Total Files**	**Accuracy**
GitHub	12	50	62.0%
GCJ	12	50	68.0%

Table VIII
GITHUB PROGRAMMER DE-ANONYMIZATION

VII. DISCUSSION

Our experiments are devised for a setting where the programmer is not trying to hide his coding style. Therefore we have not included any experiments based on identifying the authors of obfuscated executable binary samples. We focused on the general case of executable binary authorship attribution, which is a serious threat to privacy but at the same time an aid for forensic analysis.

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

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

Previous work shows that coding style is quite prevalent in source code. We were surprised to find out that coding style is preserved to a great degree even in compiled source code. We can de-anonymize programmers from compiled source code with great accuracy and furthermore we can de-anonymize programmers from source code compiled with optimization. Optimizations transform executable binaries further to improve performance or memory usage. In our experiments, we see that even though optimization or stripping symbols transforms executable binaries more than plain compilation, stylistic features are still preserved to a large degree. However, our experiments in these cases were done using the information-gain features determined from the unoptimized case with symbol tables intact. Future work that customizes the dimensionality reduction for these cases (for example, removing features from the trees that are no longer relevant) may be able to improve upon these numbers, especially since the dimensionality reduction was able to provide such a large boost in the unoptimized case.

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

Even though executable binaries look cryptic and difficult to analyze, by the help of available tools, we can extract many useful features from them. We extract features from disassembly, control flow graphs, and also decompiled code to identify features relevant to only programming style. We extract features from different feature spaces to obtain a rich representation of the binary. After dimensionality reduction with information gain, we see that each of the feature spaces provides programmer style information. All the feature spaces contain a total of more than 200,000 features for 900 executable binary samples of 100 authors. Approximately 400 features suffice to capture enough key features of coding style to enable robust authorship attribution. Even though 400 is a small number to represent hundreds of programmers' styles, these features are quite powerful in de-anonymizing programmers. We see that the information gain features are valid in different datasets with different programmers, including optimized programmers or GitHub programmers. Also, the information gain features are helpful in scaling up the programmer de-anonymization approach. While we can identify 100 programmers with 78% accuracy, we can de-anonymize 600 programmers with 52% accuracy using the same set of 400 features. 52% is a very high number for such a challenging de-anonymization task where the random chance of correctly identifying an author is 0.17%.

VIII. LIMITATIONS

Our experiments suggest that our method is able to assist in de-anonymizing programmers with significantly higher accu-

racy than state-of-the-art approaches. However, there are also assumptions that underlie the validity of our experiments as well as inherent limitations of our method that we discuss in the following paragraphs.

First, we assume that our ground truth is correct, but in reality programs in Google CodeJam or on GitHub might be written by programmers other than the stated programmer, or by multiple programmers. Such a ground truth problem would cause the classifier to train on noisy models which would lead to lower de-anonymization accuracy and a noisy representation of programming style.

Second, many source code samples from Google Code Jam contestants cannot be compiled. Consequently, we perform evaluation only on that subset of samples which can be compiled. This has two effects: first, we are performing attribution with fewer executable binary samples than the number of available source code samples. This is a limitation for our experiments but it is not a limitation for an attacker who first gets access to the executable binary instead of the source code. If the attacker gets access to the source code instead, she could perform regular source code authorship attribution. Second, we must assume that whether or not a code sample can be compiled does not correlate with the ease of attribution for that sample.

Third, we only consider C/C++ code compiled using the GNU compiler `gcc` in this work, and assume that the executable binary format is the Executable and Linking Format (ELF). This is important to note as dynamic symbols are typically present in ELF binary files even after stripping of symbols, which may ease the attribution task relative to other executable binary formats that may not contain this information. We defer the investigation of the impact that other compilers, languages, and executable binary formats might have on the attribution task to future work.

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

IX. Conclusion

De-anonymizing programmers has direct implications for privacy and security. Being able to attribute authorship to anonymous executable binary samples has applications in software forensics. Executable binary authorship attribution is an immediate concern for programmers that would like to remain anonymous. In this work, we de-anonymize 100 programmers from their executable binary samples with 78% accuracy. This is a significant advance over previous work and shows that coding style is preserved in compilation, contrary to the belief that compilation wipes away stylistic properties. We show that surprisingly, programmer style is embedded in executable binaries at a great degree, even when the executable binary has been generated with aggressive compiler optimizations or when the symbols of the executable binary samples have been stripped. These findings suggest that while compilation, optimizations, and stripping symbols do reduce the accuracy of stylistic analysis, they are not effective in anonymizing coding style. We are able to identify GitHub authors from their executable binary samples in the wild, even though GitHub authors' executable binary samples are noisy and products of collaborative efforts. We have shown that attribution is sometimes possible with only small amounts of training binaries, however, having more binaries to train on helps significantly and that advanced programmers (as measured by progression in the Google Code Jam contest) can be attributed more easily than their less skilled peers.

We scale up the machine learning problem of automated programmer de-anonymization to 600 programmers and we still achieve 52% accuracy. This is a significant success in correct classification accuracy for such a large dataset, especially when compared to previous work that was able to reach this accuracy on a dataset with at most 191 programmers. Additionally, we are able to achieve high de-anonymization success with less than 500 stylistic features, out of 200,000 executable binary properties that are embedded in executable binaries. These information gain features obtained from the Google Code Jam dataset effectively represent authors' coding style in optimized executable binary and GitHub executable binary samples. We achieve such a precise representation of coding style by incorporating two different disassemblers, control flow graphs, and an executable binary to source code decompiler.

Our results present a clear concern for people who would like to release binaries anonymously. In future work, we plan to investigate if stylistic properties can be completely stripped from binaries to render them anonymous. We also plan to look at different real world executable binary authorship attribution cases, such as identifying authors of malware, which go through a mixture of sophisticated obfuscation methods by combining polymorphism and encryption. Our results so far suggest that while stylistic analysis is unlikely to provide a “smoking gun” in the malware case, it may contribute significantly to attribution efforts.

References

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

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

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

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

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

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

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

[8] S. Afroz, A. Caliskan-Islam, A. Stolerman, R. Greenstadt, and D. McCoy. Doppelgänger finder: Taking stylometry to the underground. In *Proc. of IEEE Symposium on Security and Privacy*, 2014.

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

[10] S. Alrabaee, N. Saleem, S. Preda, L. Wang, and M. Debbabi. Oba2: an onion approach to binary code authorship attribution. *Digital Investigation*, 11, 2014.

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

[12] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, behavior-based malware clustering. In *NDSS*, volume 9, pages 8–11. Citeseer, 2009.

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

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

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

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

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

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

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

[20] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. Liblinear: A library for large linear classification. *Journal of Machine Learning Research (JMLR)*, 9, 2008.

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

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

[23] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *SIGKDD Explor. Newsl.*, 11, 2009.

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

[25] 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*. IEEE, 2007.

[26] J. Lafferty, A. McCallum, and F. C. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. 2001.

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

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

[29] A. W. McDonald, S. Afroz, A. Caliskan, A. Stolerman, and R. Greenstadt. Use fewer instances of the letter "i": Toward writing style anonymization. In *Privacy Enhancing Technologies*, pages 299–318. Springer Berlin Heidelberg, 2012.

[30] T. C. Mendenhall. The characteristic curves of composition. *Science*, pages 237–249, 1887.

[31] 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 *Proc. of IEEE Symposium on Security and Privacy*, 2012.

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

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

[34] A. Pfeffer, C. Call, J. Chamberlain, L. Kellogg, J. Ouellette, T. Patten, G. Zacharias, A. Lakhotia, S. Golconda, J. Bay, et al. Malware analysis and attribution using genetic information. In *Malicious and Unwanted Software (MALWARE), 2012 7th International Conference on*, pages 39–45. IEEE, 2012.

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

[36] N. A. Quynh. Capstone. http://www.capstone-engine.org/, visited, October 2015.

[37] A. Rahimian, P. Shirani, S. Alrbaee, L. Wang, and M. Debbabi. Bincomp: A stratified approach to compiler provenance attribution. *Digital Investigation*, 14:S146–S155, 2015.

[38] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov. Learning and classification of malware behavior. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 108–125. Springer, 2008.

[39] N. Rosenblum, B. P. Miller, and X. Zhu. Recovering the toolchain provenance of binary code. In *Proc. of the International Symposium on Software Testing and Analysis*. ACM, 2011.

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

[41] N. E. Rosenblum, B. P. Miller, and X. Zhu. Extracting compiler provenance from program binaries. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. ACM, 2010.

[42] S. Tatham and J. Hall. The netwide disassembler: NDISASM. http://www.nasm.us/doc/nasmdoca.html, visited, October 2015.

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

[44] W. Wisse and C. Veenman. Scripting dna: Identifying the javascript programmer. *Digital Investigation*, 2015.

[45] 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*, 2014.

[46] yegord. Snowman. https://github.com/yegord/snowman, 2013.