Who Wrote This Code? Identifying the Authors of Program Binaries

Nathan Rosenblum, Xiaojin Zhu, and Barton P. Miller

University of Wisconsin,
Madison, Wisconsin
{nater,jerryzhu,bart}@cs.wisc.edu

Abstract. Program authorship attribution—identifying a programmer based on stylistic characteristics of code—has practical implications for detecting software theft, digital forensics, and malware analysis. Authorship attribution is challenging in these domains where usually only binary code is available; existing source code-based approaches to attribution have left unclear whether and to what extend programmer style survives the compilation process. Casting authorship attribution as a machine learning problem, we present a novel program representation and techniques that automatically detect the *stylistic features* of binary code. We apply these techniques to two attribution problems: identifying the precise author of a program, and finding stylistic similarities between programs by unknown authors. Our experiments provide strong evidence that programmer style is preserved in program binaries.

1 Introduction

Program authorship attribution has immediate implications for the security community, particularly in its potential to significantly impact applications like plagiarism detection [17] and digital forensics [13]. The central thesis of authorship attribution is that authors imbue their works with an individual style; while attribution research has historically focused on literary documents [7], computer programs are no less the product of a creative process, one in which opportunities for stylistic expression abound. Previous studies of program authorship attribution have been limited to source code [6, 10], and rely on surface characteristics like spacing and variable naming, both of which reflect the essentially textual nature of program source. In many domains, such as analysis of commercial software or malware, source code is usually unavailable. Program binaries, however, retain none of the surface characteristics used in source code attribution; such details are stripped away in the compilation process. Adapting program authorship attribution to the binary domain—to identify known malware authors or detect new ones, e.g., or to discover theft of commercial software—requires new ways to recognize the style of individual authors.

We have developed novel authorship attribution techniques that automatically discover the stylistic characteristics of binary code. We adopt a *machine learning approach*, defining a large number of simple *candidate features* and using *training data* to automatically discover which features are indicative of programmer style. This approach avoids the problem of choosing good stylistic features *a priori*, which has been the focus of source code attribution [18], and which is the primary challenge for attribution in the binary domain. We apply our techniques to two related binary code authorship problems: identifying the author of a program out of a set of candidates, and grouping programs by stylistic similarity, respectively developing *classification* and *clustering* models that build on stylistic features of binary code.

In this paper, we explore various aspects of these previously unstudied problems, examining trade-offs in different program representations and several attribution scenarios. This study demonstrates that programmer style is reflected in binary code, and lays the groundwork for authorship attribution applications in a variety of domains. Our paper makes the following contributions:

– We introduce the problem of binary code authorship attribution and define a program representation in terms of *stylistic features* that differentiate different programmers; we provide an algorithm for automatically selecting stylistic features using a set of simple *feature templates* that cover a broad range of program details.

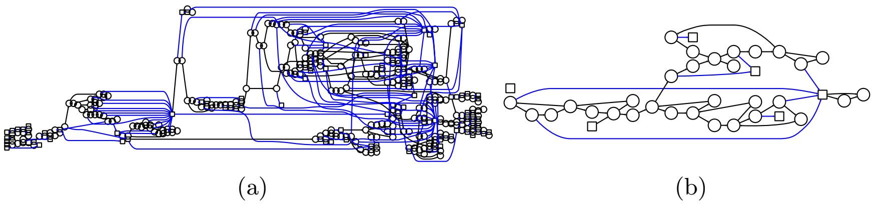
– We formulate two program authorship tasks: (1) discriminating between programs written by different authors (*authorship identification*), and (2) grouping together stylistically similar programs (*authorship clustering*). We use information derived from the authorship identification task to improve the performance of authorship clustering.

– We evaluate binary program authorship attribution on several large sets of programs from the Google Code Jam programming competition[1] and from student projects from an undergraduate operating systems course at the University of Wisconsin. Our results show that programmer style is preserved through the compilation process; a classifier trained on stylistic features can discriminate among programs written by ten different authors with 81% accuracy.

1.1 Overview

Our authorship attribution techniques are based on the hypothesis that programmer style is preserved throughout the compilation process, as suggested by the differences depicted in Figure 1 between implementations of the same functionality by two different programmers. Evaluating this hypothesis requires solving two problems: (1) choosing a program representation broad enough to capture any residual stylistic characteristics, and (2) selecting those representational elements that actually reflect programmer style. The second problem is particularly important for authorship clustering; author identity is just one property of many for a given program, and if the representation reflects more than

[1] `http://code.google.com/codejam/`



(a) (b)

Fig. 1. The control flow graphs for two implementations of the same program by different authors. Program (a) is implemented as many small subroutines and makes use of several C++ STL classes; program (b) is almost entirely implemented as a monolithic C function.

just stylistic characteristics, a clustering algorithm may group programs according to some other property, such as program functionality. Rather than design complicated features to capture specific facets of programmer style, we define a large number of simple features that capture local and global code details at the instruction and control flow level. We adopt a machine learning approach to the problem, letting the data determine the features which best capture authorship; this data-driven policy informs our high-level workflow:

1. We collect several large corpora of programs with known authorship; these programs provide a *ground truth*, which is used to discover the stylistically important features of binary code, as well as reference points on which to evaluate authorship attribution techniques.

2. Using existing software for recursive traversal parsing [14], we extract a control flow graph and the instruction sequence for each binary, which we use as a basis for the features we describe in the following section.

3. A subset of the features that correlate with programmer style is selected. We compute the *mutual information* between features and programmer identity on a *training set* of labeled programs, ranking features according to their correlation with particular programmers. This approach is heuristic and does not take into consideration the interaction between multiple features; the learning algorithms we apply to this feature representation are responsible for refining the stylistic importance of these features.

4. We use the training set of labeled programs to build an authorship classifier based on support vector machines [3]. The classifier chooses the most likely author of a program based on its stylistic feature representation.

5. Classification is not possible for collections of programs with no training data; instead, we use the k-means clustering algorithm [1] to group programs together by stylistic similarity. To avoid clustering according to the wrong property (e.g. program functionality), we *transfer* knowledge between a *supervised* domain (a set of programs with different authors) to this *unsupervised* domain: we use the *large margin nearest neighbors* algorithm [20] to learn a *distance metric* over a labeled set of programs, then used this metric to transform the unlabeled data prior to clustering.

In the following sections, we describe our binary code representation (2) and formally state the models and procedures we use for author classification (3) and clustering (4). We evaluate our techniques over several large program data sets (5), exploring several trade-offs inherent in binary authorship attribution. We conclude with a discussion of issues raised by this study and future directions for attribution research (6) and a review of the related literature (7).

2 Binary Code Representation

We base our binary code representation on instruction-level and structural characteristics of programs. The first step in obtaining this representation is to parse the program binary. We use the ParseAPI [14] library to extract instructions and build interprocedural *control flow graphs* from binaries, where a CFG is a directed graph $G = (V, E, \tau)$ defined by:

- the *basic block* nodes V comprising the executable code,
- the edges $E \subseteq V \times V$ representing control flow, and
- a labeling function $\tau : E \rightarrow \mathcal{T}$ corresponding to the type of the edge.

The control flow graph and underlying machine code form the basis for *feature templates*: patterns that instantiate into many concrete features of a particular binary. We first describe two feature templates, *idioms* and *graphlets*, used in our previous work on toolchain provenance [16], and then introduce new templates that capture additional properties of the binary. We stress that these features are not designed to capture any specific notion of programmer style, but rather to express many different characteristics of binary code; we use machine learning algorithms to pick out the stylistically significant features.

2.1 Idioms

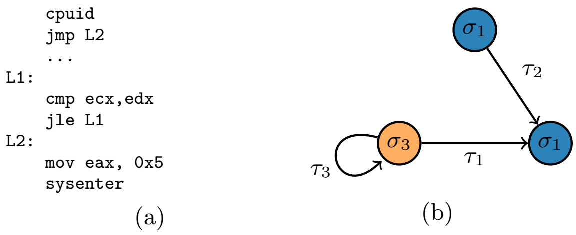
The idiom feature template captures low-level details of the instruction sequence underlying a program. Idioms are short sequences of instructions, possibly with wildcards, which we have previously used to recognize compiler-specific code patterns; for example, the idiom

$$u_1 = (\texttt{push ebp | * | mov esp,ebp})$$

describes a stack frame set-up operation. Idioms are an abstraction of the true instruction sequence, insofar as instruction details such as immediate operands and memory addresses are elided. The idiom template we use for authorship attribution describes all possible sequences of 1–3 instructions, and is intended to capture stylistic characteristics that are reflected in the order of instructions output by the compiler.

2.2 Graphlets

While idioms capture instruction-level details, graphlet features represent details of program structure. Graphlets are three-node subgraphs of the control flow



```
    cpuid
    jmp L2
    ...
L1:
    cmp ecx,edx
    jle L1
L2:
    mov eax, 0x5
    sysenter
```

(a)

(b)

Fig. 2. A code example and a corresponding graphlet. The node colors σ are determined by the instructions in each block (for example, both of the blocks represented by (●) nodes contain system instructions). Edge labels τ indicate control flow edge type (for example, τ_3 represents the `jle` conditional branch and τ_1 is its fall-through edge).

graph that reflect the *local* structure of the program. A graphlet feature template also defines a *coloring function* $\sigma : V \to \mathcal{C}$, where $\mathcal{C}$ is the set of possible colors for a particular graphlet template. For example, the *instruction summary graphlets* we use for toolchain provenance recovery (and which we adopt here) color nodes according the various classes of instructions occurring in a basic block, as illustrated in Figure 2. We refer the reader to our previous work for details of the instruction summary coloring function and algorithms for efficient graphlet matching [16].

In the current study, graphlet features are a bridge between the instruction-level representation (using colors based on instruction classes) and the program structure (the local control flow); however, these features may miss stylistic characteristics that are visible only in high-level program structure. We could attempt to capture such characteristics by defining graphlet-like features using larger subgraphs, but there is an essential tension between the expressiveness of such features and the computational complexity of the subgraph matching problem. Instead, we introduce two additional graphlet-based features, *supergraphlets* and *call graphlets*, that are defined over transformations of the original control flow graph.

2.3 Supergraphlets

Supergraphlets are analogous to instruction summary graphlets defined over a *collapsed* control flow graph, as illustrated in Figure 4. The graph collapse operation , specified in Figure 3, merges each node in the graph with a random neighbor. The edge set and color of the collapsed node represent the union of the edge sets and colors of the original nodes. A three-node graphlet instantiated from the collapsed graph is thus an approximate representation of six nodes in the original CFG. This process can be repeated recursively to obtain the desired long-range structural coverage. Note that because random neighbors are selected, we do not obtain all possible supergraphlets of the original graph; in keeping with our general approach to code representation, we rely on the vast number of features to capture any authorship characteristics in the program. Selecting neighbors to collapse at random avoids systematically biasing the collapse operation towards particular control flow structure.

```
function Collapse($G = (V, E, \tau, \sigma)$)
    $V' \leftarrow \emptyset$, $E' \leftarrow \emptyset$
    for all $v \in V$ do
        choose $n \in$ Neighbors($v$)                    $\triangleright$ arbitrary $n$
        $\sigma'(v') \leftarrow \sigma(v) \cup \sigma(n)$
        $V' \leftarrow V' \cup v'$
        $E' \leftarrow E' \cup \{(v', n') : (v, n') \in E \vee (n, n') \in E\}$
        $E' \leftarrow E' \cup \{(n', v') : (n', v) \in E \vee (n', n) \in E\}$
    return $(V', E', \tau, \sigma')$
```

Fig. 3. Graph collapse for supergraphlet features. While the node colors are merged in the collapsed graph, the edge degree and types are preserved.

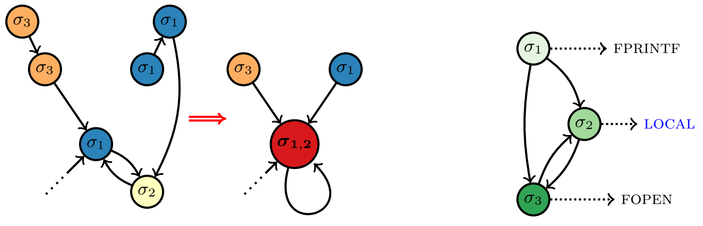
2.4 Call Graphlets

Recursively collapsing the control flow graph and extracting supergraphlet features only loosely approximates arbitrarily long-range program structure. Call graphlets are designed to directly capture both *interprocedural* control flow and a program's interaction with external libraries. Call graphlets are defined over a new graph G^c containing only those nodes that contain call instructions, with edges $E^c = \{(v, v') : v \rightsquigarrow v'\}$, where $\rightsquigarrow$ indicates the existence of a path in the original control flow graph. Call graphlets admit the coloring function $\sigma_c : V^c \rightarrow \{\mathcal{L}, \text{LOCAL}\}$, where $\mathcal{L}$ is a predefined set of *external library functions* and LOCAL is a special value meaning any internal function within the program binary. Internal functions receive a single, generic color because, unlike calls to external libraries, they are not comparable across different programs. While $\mathcal{L}$ could be restricted to a set of specific library functions, in practice we let it extend to the entire set of library routines called by programs in our corpus and rely on feature selection to eliminate irrelevant call graphlet features.

2.5 N-grams and External Interaction

To cast as wide a net as possible in our search for good authorship features, we define several more features that are relaxations of those described above. Byte *n-grams* are short strings of three or four bytes, and can be thought of as a relaxation of the idiom instruction abstractions: using the raw bytes, n-grams capture specific instruction opcodes and immediate and memory operands. *Library call* features simply count the number of invocations of the set of $\mathcal{L}$ external library functions used in the call graphlet features, eliminating structural characteristics.

Table 1 summarizes our binary code feature templates and the number of each instantiated in a typical corpus. Our algorithms automatically select a subset of these features based on the training data, as we describe in the following section.



(a) (b)

Fig. 4. Graphlet-based features of transformations of the control flow graph. Supergraphlets (a) represent control flow relationships in a graph where the neighbors of the middle three nodes have been *collapsed*; the color of one collapsed node (●) reflects the union of two nodes with different colors. Call graphlets (b) are defined over a graph reduced to blocks containing `call` instructions.

		Code Property		
Feature	#	Instruction	Control flow	External
N-grams	391,056	✓		
Idioms	54,705	✓		
Graphlets	37,358	✓	✓	
Supergraphlets	117,997	✓	✓	
Call graphlets	8,062		✓	✓
Library calls	152			✓

Table 1. The number of concrete features instantiated by each feature template for a representative corpus of 1,747 C and C++ binaries comprising 27MB of code. Each template captures one or more instruction-level, control-flow, or external library interaction properties of the code.

3 Author Classification

In author classification, we assume that there exists a known set of programmers of interest, and that training data are available in the form of samples of programs written by each programmer. We model program binaries as a collection of the features described in the previous section in order to discriminate between programs written by different authors. To be precise, given a known set of program authors $\mathcal{Y}$ and a set of M training programs $\mathcal{P}_1, \cdots, \mathcal{P}_M$ with author labels $y_1, \cdots, y_M$, the task of the classifier is to learn a decision function that assigns a label $y \in \mathcal{Y}$ to a new program, indicating the identity of the most likely author.

A program $\mathcal{P}_m$ is represented by a integral-valued *feature vector* $\mathbf{x}_m$ describing the features that occur in the program. Feature vectors summarize a set of *feature functions* $f \in \Phi$ that indicate the presence of that feature evaluated over a feature-specific domain in the binary. For example, the function

$$f_{\text{FPRINTF}}(\mathcal{P}_m, c_j) = \begin{cases} 1 & \text{if call site } c_j \text{ in } \mathcal{P}_m \text{ calls } \textsc{fprintf} \\ 0 & \textit{otherwise} \end{cases}$$

tests for a particular library call and is defined over the domain of *call sites* in the program; idiom feature functions

$$f_\iota(\mathcal{P}_m, a_j) = \begin{cases} 1 & \text{if idiom } \iota \text{ exists at instruction offset } a_j \text{ in } \mathcal{P}_m \\ 0 & \textit{otherwise} \end{cases}$$

are defined over the domain of instruction offsets in the binary. The feature vector $\mathbf{x}_m$ for a program counts up the $|\varPhi|$ features

$$\mathbf{x}_m = \begin{pmatrix} \sum_{Dom(f_1)} f_1(\mathcal{P}_m, \cdot) \\ \sum_{Dom(f_2)} f_2(\mathcal{P}_m, \cdot) \\ \cdots \\ \sum_{Dom(f_n)} f_{|\varPhi|}(\mathcal{P}_m, \cdot) \end{pmatrix}.$$

evaluated at every point in the domain $Dom(f_i)$ of the particular feature.

The number of feature functions in $\varPhi$ is quite large; using feature vectors that summarize all possible features would increase both training cost and the risk that the learned parameters would *overfit* the data—that is, that the resulting classifier would fail to generalize to new programs. Because our feature templates are not designed to highlight particular stylistic characteristics, we expect that many features will be of little value for authorship attribution. We therefore perform a simple form of feature selection, ranking features by the *mutual information* between the feature and the true author label. More precisely, we compute

$$I(\varPhi, \mathcal{Y}) = \sum_{f \in \varPhi} \sum_{y \in \mathcal{Y}} p(f, y) \log \left(\frac{p(f, y)}{p(f)p(y)} \right)$$

on the training set, where $p(f)$ and $p(y)$ are the empirical expectations of features and author labels, respectively, and $p(f, y)$ is the co-occurrence of these variables. Mutual information measures the decrease in uncertainty about one variable as a function of the other; features that are positively correlated with only a single programmer will score high under this criterion, while features that are distributed uniformly over programs by all authors will have low mutual information. The number of features to retain is chosen through cross-validation: we split the training data into ten *folds*, reserving one fold as a *tuning set*, then train a classifier on the remaining folds and evaluate its accuracy on the tuning set. By performing cross-validation on data represented by varying numbers of the features ranked highest by mutual information, we automatically select a subset of features that produce good authorship classifiers.

There are many different models that can be used to classify data such as ours. We use linear support vector machines (SVMs) [3], which scale well with high-dimensional data and have shown good performance in our experience with other classification tasks for binary programs. Two-class SVMs are usually formulated with labels $y \in \{-1, +1\}$, and compute a weight vector $\mathbf{w}$ that solves the following optimization problem:

$$\min_{\mathbf{w},\xi,b} \frac{1}{2} \parallel \mathbf{w} \parallel^2 + C \sum_{i}^{n} \xi_i \qquad \text{s.t. } y_i(\mathbf{w}^T\mathbf{x} - b) \geq 1 - \xi_i, \;\; \xi_i \geq 0.$$

Such binary SVMs can be easily extended to the case of K classes by training K different binary classifiers with weight vectors $\mathbf{w}_1, \cdots, \mathbf{w}_K$; the classifier assigns a new example the label $k \in [1, K]$ that leads to the largest margin, i.e.

$$\operatorname*{argmax}_k \mathbf{w}_k^T \mathbf{x}.$$

We use the LIBLINEAR support vector machine implementation [4] for authorship classification. We scale the feature vectors to the interval $[0, 1]$; scaling prevents frequently occurring features from drowning the contribution of rarer ones, while preserving the sparsity of the feature vectors. In our evaluation section, we examine the contribution of each feature template to overall classifier performance.

4 Author Clustering

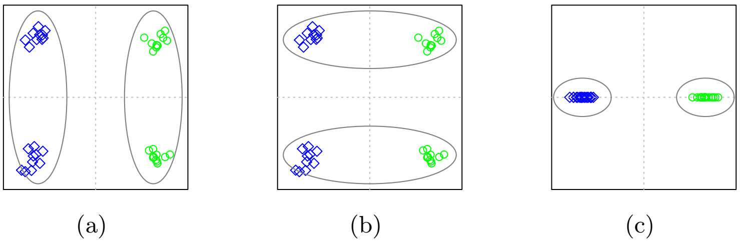
Clustering is an unsupervised learning technique that groups data by similarity. For authorship attribution, clustering corresponds to the task of finding stylistically similar programs without assuming particular authors are present. In many ways, clustering is harder than classification: without training data, it is generally not possible to tell whether particular features are more or less useful for relating the data, which leads to the possibility that clustering algorithms will arrive at clusters that reflect a different property than what was desired. This issue is particularly challenging for authorship clustering, where we have a large number of features and no assurance that they reflect only programmer style and not, for example, program functionality.

One way to encourage the formation of authorship clusters is to transform the feature space such that stylistically similar programs are closer to one another; equivalently, we can define a $d \times d$ *distance metric* A such that the *Mahalanobis distance* [12] between two feature vectors $\mathbf{x}_a, \mathbf{x}_b$ in $\mathbb{R}^d$ is

$$D_A(\mathbf{x}_a, \mathbf{x}_b) = \sqrt{(\mathbf{x}_a - \mathbf{x}_b)^T A (\mathbf{x}_a - \mathbf{x}_b)}.$$

If a particular metric can be found such that stylistically similar programs are close under that metric, then clustering techniques will do better at forming authorship clusters. Figure 5 illustrates this solution with a simple example.

We observe that stylistic features, if they are general, can be learned from *any* set of authors; although the programs to be clustered may have no training data, we can derive a metric from a different collection of programs with



(a) (b) (c)

Fig. 5. The hazards of unsupervised clustering. Assuming that the data belong to true classes y_1 (◇) and y_2 (○) and two clusters are formed, the correct cluster partition (a) is no more likely than the alternative (b). Using the distance metric $\left(\begin{smallmatrix}1&0\\0&0\end{smallmatrix}\right)$ is equivalent to transforming the data as in (c), where the clustering decision is unambiguous.

author labels. More precisely, consider two sets of programs $\{\mathcal{P}_1, \cdots, \mathcal{P}_\ell\}$ and $\{\mathcal{P}_{\ell+1}, \cdots, \mathcal{P}_{\ell+u}\}$, with known author labels $\{y_1, \cdots, y_\ell\}$; the authors for the unlabeled programs may or may not coincide with those of the labeled programs. Both sets of programs are represented using the feature vectors we describe in the previous section. We define a two part algorithm for transferring stylistic knowledge from the labeled data to the unlabeled data:

1. Learn a metric A over ℓ labeled programs $\mathcal{P}_1, \cdots, \mathcal{P}_\ell$ such that the distance in the feature space between two programs with the same author is always less than the distance between two programs with different authors.
2. Cluster u unlabeled programs $\mathcal{P}_{\ell+1}, \cdots, \mathcal{P}_{\ell+u}$ using the distance function D_A.

We use the *large margin nearest neighbors* (LMNN) algorithm [20] to learn the style metric. LMNN learns the metric by optimizing the margin for nearby programs in the feature space, making it complementary to the k-means algorithm we use for clustering.

5 Evaluation

We investigate several aspects of authorship attribution in our evaluation: (1) the extent to which our techniques recover author style in program binaries, (2) the trade-offs involved in imprecise classification (i.e., tolerating some *false positives*), and (3) whether and how much stylistic clustering of one set of programs can be improved by using information derived from another, unrelated set. Our evaluation shows that:

- The binary code features we introduce effectively capture programmer style. Our classifier achieves accuracies of 81% for ten distinct authors (10% accuracy is expected for labels selected by random chance) and 51% when discriminating among almost 200 authors (0.5% for random chance). These results show that a strong author style signal survives the compilation process.

- The authorship classifier offers practical attribution with good accuracy, if a few false positives can be tolerated. The correct author is among the top five 95% of the time for a data set of 20 authors, and 81% of the time when 100 authors are represented.
- Stylistic knowledge derived from supervised authorship classification can be transferred to authorship clustering, improving cluster quality. The cluster assignments improve by 20% when clustering uses a stylistic metric.

5.1 Evaluation Data Set

Obtaining appropriate data sets is a fundamentally limiting problem for the development of authorship attribution techniques. At the minimum, the collection of programs (the *corpus*) used to develop an authorship model must have author labels. A *parallel* corpus—one in which each author has written programs with the same functionality—helps to control for confounding variables in the evaluation. Previous program authorship studies have obtained parallel corpora by eliciting program implementations from a small number of human subjects [6, 10]. We have constructed substantially larger corpora from programs written for the Google Code Jam programming competition[2] and from student projects from an operating systems course at the University of Wisconsin.

Google Code Jam The Code Jam competition is an annual world-wide programming contest conducted over multiple rounds and representing the efforts of thousands of contestants. Each round of the competition involves writing a program to solve a small number (usually 3-6) of problems; contestants who pass a threshold are advanced to later rounds. The Code Jam website provides access to all correct solutions for each contestant, meeting both of our criteria for a high-quality authorship attribution data. We use the 2009 and 2010 contest data in our evaluation.

Code Jam solutions are not required to be written in any particular programming language. We restrict our attention to those solutions that were written in C or C++ and that could be compiled with GCC 4.5. We further filter the data set to only include programs by authors who submitted solutions to eight or more problems over the course of a given year's competition. Importantly, we note that each contestant submits *two* solutions for a given problem: one that is tested against a "short" test case and one that is run against a (presumably harder) "long" test case. In almost all cases the second solution is identical to the first (or has minor bugfixes); we therefore eliminate the second solution from our data set to avoid artificially inflating our evaluation accuracy by including many identical programs.

Class Projects Our second corpus consists of programs written by students enrolled in an undergraduate operating systems course (CS537) at the University of Wisconsin. The students completed seven programming assignments during

[2] http://code.google.com/codejam/

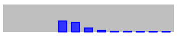
Corpus	Authors	Program Types	Binaries	Prog./Author Dist.
				4 16
Code Jam 2010	191	23	1,747	
Code Jam 2009	93	22	834	
CS537 Fall 2009	32	7	203	

Table 2. Corpora used for model training and evaluation. Each binary is the implementation by a particular author of one of the program types for a given corpus.

the course, including a shell implementation, several memory and scheduling libraries, and a simple web server. This data set has several properties that make it more difficult to use than the Code Jam database. Students were allowed to work on programs alone or with partners, so a program's authorship can be ambiguous in some cases; also, in many assignments skeleton code or fully implemented modules were provided to the students. These uncertainties in the ground truth may impact our evaluation. We attempt to mitigate the former possibility by eliminating one author out of every partnership. The latter is more difficult to address; we analyzed the original assignments by hand and removed some fully implemented code that was provided, but this approach is imperfect and likely missed a great deal of skeleton code. We discuss the implications of using such noisy data below.

Table 2 summarizes our experimental corpora. The two different data sets differ significantly, both in the distribution over program sizes and in that of the number of different programs written by each author. The fewer programs per author in the CS537 corpus increase the difficulty of the learning problem.

5.2 Methodology

To create a data representation suitable for learning and evaluation, we process the binaries in each corpus with the ParseAPI parsing library to obtain control flow graphs and the underlying instructions. We eliminate statically linked library functions and other known binary code snippets that are unrelated to the program author.[3] We then exhaustively enumerate all of the features described in Section 2, using the occurrence of these features along with the known authorship labels to compute the mutual information for each feature. We select a subset of features using the cross-validation procedure described in Section 3. We use the top 1,900 features for modeling and evaluation of the Code Jam data; 1,700 features are used for CS537.

Our evaluation methodology involves both standard ten-fold cross-validation and random subset testing, depending on the experiment:

- For classification of the entire data set (e.g. 191-way classification for the Code Jam 2010 data), we use ten-fold cross-validation.

[3] Our data preparation procedure is fully described in our supplementary materials at `http://pages.cs.wisc.edu/~nater/esorics-supp/`.

– When evaluating how classification accuracy behaves as a function of the number of authors represented in the data, we randomly draw a subset $\mathcal{Y}_s \subseteq \mathcal{Y}$ of authors and use their programs in the test. We cannot test all possible combinations of $|\mathcal{Y}_s|$ authors; instead, we repeat the test 20 times and expect relatively high variance for small sets of authors. We approach the clustering evaluation similarly.

5.3 Classification

We evaluate authorship classification to determine (1) how much each feature template contributes to attribution, and (2) how accurately the identity of a particular author can be inferred using a model based on our feature templates. For the former question, our experience led us to expect that simple feature templates that instantiate large numbers of features (e.g., idioms) would be more useful in authorship modeling. For the latter question, we hypothesized that discriminating among authors would become increasingly difficult with larger author populations.

Figure 6a depicts the cross-validation accuracy of models trained with varying numbers of the best features (by mutual information) derived from each template. Our intuition is borne out by these results: the individual contributions of simple idiom and n-gram features exceed those of the other templates. The best classifier uses a combination of all of the feature templates, achieving 51% accuracy on the full Code Jam data set.

Experiments confirm our hypothesis that author classification becomes harder for larger populations. Figure 6b depicts classifier performance as a function of the number of authors included in a subset of the data; classifier accuracy decreases as the author population size grows. In cases where precise author identification is infeasible, predicting a small set of likely authors can help to focus further investigation and analysis. In Figure 6b, this relaxed accuracy measure is plotted for a classifier that returns the top five most likely authors.

Table 3 lists exact and relaxed cross-validation accuracy for authorship classification on each corpus. The CS537 data present a significantly harder challenge for authorship attribution, due to two factors. First, there are fewer programs per author (4–7) than in the other data sets (8–16), making this a fundamentally harder learning problem. More importantly, the programs in this data set do not reflect only the work of individual programmers; students in the course were often provided with substantial amounts of partially implemented skeleton

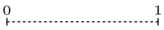
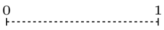
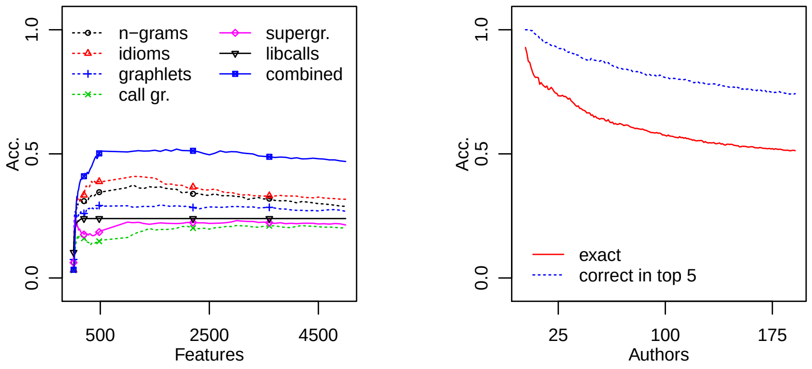
	Code Jam 2009		Code Jam 2010		CS537	
	Acc.	spread	Acc.	spread	Acc.	spread
						
Exact	.778		.768		.384	
Top 5	.947		.937		.843	

Table 3. Classification results averaged over 20 randomly selected subsets of 20 authors.



(a) (b)

Fig. 6. Evaluation of authorship classification on the Code Jam 2010 data set. In Figure (a) we show cross-validation accuracy over all 191 authors for classifiers trained using individual feature templates, as well as the combined classifier. Figure (b) depicts accuracy using the best combination of features as the true number of authors in the data set is increased for both the exact (—) and relaxed (···) evaluations.

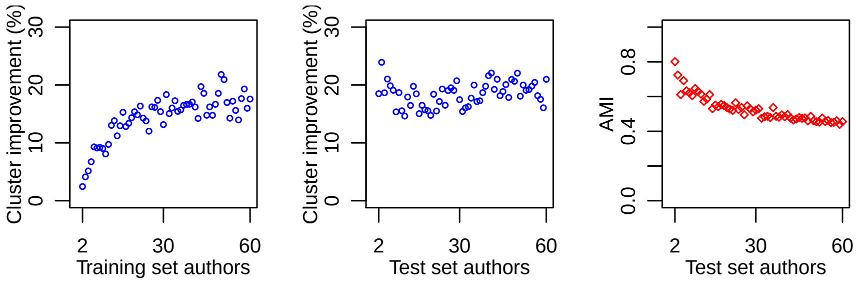
code, and also worked closely with the course professor follow an often rigid specification at the sub-module level. Despite these challenges, our attribution techniques recover significant stylistic characteristics in this data set.

5.4 Clustering

We evaluated authorship clustering to determine (1) how well the clusters reflect the ground truth program authorship, and (2) whether stylistic characteristics learned from one set of authors can improve the clustering of programs written by different authors (i.e., how well stylistic knowledge generalizes). Unlike classifiers, clustering algorithms have no notion of candidate labels, so cluster assignments are evaluated against the ground truth authors with measures based on cluster *agreement*: whether (a) programs by the same author are assigned to the same cluster, and (b) programs by different authors are assigned to different clusters. We computed several common measures of cluster agreement, including Adjusted Mutual Information (AMI), Normalized Mutual Information (NMI), and the Adjusted Rand Index (ARI); we prefer AMI because it is stable across different numbers of clusters, easing comparison of different data sets [19]. All of the measures we use take values in the range $[0, 1]$, where higher scores indicate better cluster agreement.

We performed several experiments to evaluate authorship clustering:

1. We randomly selected N authors from the Code Jam 2010 corpus and used LMNN to learn a distance metric over the feature space. We then randomly selected 30 different authors and clustered their programs using k-means with and without transforming the data with the learned metric. Since there are multiple sources of randomness in this experiment (both in selecting the



(a) (b) (c)

Fig. 7. Clustering with metric learning. The improvement over the original clustering is illustrated as a function of the number of training authors (a) and the true number of testing authors (b). The cluster score in (c) uses a metric learned from 30 training authors.

data sets and in the k-means clustering algorithm), we repeated the experiment 20 times and computed the average AMI. Figure 7a depicts clustering improvement over the un-transformed data as a function of N. As expected, using more training authors to compute a metric leads a greater improvement. We conclude that stylistic information derived from one set of authors can be transferred to improve clustering of programs written by a different set of authors.

2. We performed a similar set of experiments with the number of authors used to compute the metric fixed at 30 to evaluate whether the clustering improvement is affected by the number of test set authors. Figure 7b shows that that the improvement due to incorporation of the stylistic metric is nearly invariant for a range of test set sizes.
3. Figure 7c depicts the AMI measure for metric-transformed cluster assignments as a function of test set sizes, with the metric training set fixed at 30 authors; clustering 60 authors using the learned metric achieves an average AMI of .456. Cluster evaluation measures can be difficult to interpret. For comparison, the labels predicted by a 60-author supervised classifier that achieves 64% accuracy receive an AMI of .503.

Table 4 compares the results of clustering 10 authors' programs with and without metric transformation. The cluster quality measures we compute are highly variable, due to the random nature of training and test set selection and the inherent randomness in the clustering algorithm; nonetheless, the improvement offered by the learned metric is significant at a 95% confidence level for all measures.

6 Discussion

Our evaluation shows that programmer style is preserved in program binaries, and can be recovered using techniques that automatically select stylistic code

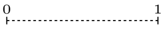
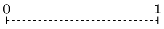
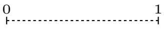
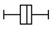
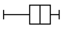
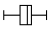
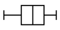
	AMI	AMI spread	NMI	spread	ARI	spread
						
no transformation	.510		.637		.406	
learned metric	.606		.723		.480	

Table 4. Cluster evaluation measures for 10 test authors, using metrics learned from 30 different authors.

features with which to model program authorship. The SVM-based classifier we introduce can identify the correct author out of tens of candidates with good accuracy, though discriminating among a large number of authors is likely to be more limited. Nonetheless, we argue that our techniques offer a practical solution to program author identification: when discriminating among programs written by 100 authors, the correct author is ranked among the top five most likely 81% of the time, reducing the number of candidates by 95%. Moreover, our evaluation of unsupervised author clustering using stylistic metrics derived from the classification problem shows that programs can be effectively clustered by programmer style even when no training data are available for the authors in question.

The conclusions we draw are subject to limitations inherent in empirical studies. In particular, threats to internal validity apply to our claim that our techniques isolate programmer style, rather than some other program property like program functionality. We addressed this issue by using a parallel corpus, where each author implemented the same programs; the fact that our authorship classifier is able to learn to recognize an author's programs despite differing functionality mitigate this threat. Our domain transfer results for authorship clustering provide further evidence that our techniques recover programmer style.

In this study, we assume that a program has a single author. This assumption may be violated in many scenarios, such as when programmers collaborate or when programs are assembled from commodity components. The binary code representation we use is not inherently restricted to representing the program as a single unity; our features could just as easily describe individual compilation units, functions, or arbitrary sequences of binary code, for example using the sequential model we have previously used to recover program provenance [15, 16]. The extension of authorship attribution to multiple authors and a sub-program model is an open question, and is the focus of our ongoing research.

7 Related Work

Previous work on program authorship attribution has focused almost exclusively on source code-level attribution. The use of code metrics like variable naming conventions, comment style, and program organization has been proposed several times [5, 18]; Krsul and Spafford [10] show the feasibility of this approach in a small pilot study. More recently, Hayes and Offutt [6] found further evidence that programmers can be distinguished through aggregate textual characteristics like average use of particular operators, placement of semicolons, and comment length.

Structural malware classification and behavioral clustering share many challenges with authorship attribution, as all three techniques involve extracting salient characteristics from binary code. The instruction-level features we use are similar to those used in malware classification [2, 8, 9], particularly n-grams; our idiom features differ from features based on instruction sequences through the use of wildcards and the abstraction of low-level details like the opcode and immediate values The instruction summary colors we use in the graphlet features are inspired by a technique to identify polymorphic malware variants [11]. Although some of the binary code representations we use are similar to existing work, our techniques are largely orthogonal: malware classification seeks to extract characteristics specific to a program or a family of programs with related behavior, while our authorship attribution techniques must discover more general properties of author style.

Authorship falls into the broad category of *program provenance*: those details that characterize the process through which the program was produced. Our previous investigation of *toolchain provenance* [15, 16] heavily informs this work, providing a general framework for extracting the characteristics of program binaries as well as providing the base representations on which we build more sophisticated authorship features. The current paper investigates a higher level of the *provenance hierarchy*, moving beyond those program properties that are attributable to the production toolchain.

8 Conclusion

We have presented techniques to extract stylistic characteristics from program binaries to perform authorship attribution and to cluster programs according to programmer style. Our authorship attribution techniques identify the correct author out of a set of 20 candidates with 77% accuracy, and rank the correct author among the top five 94% of the time. These techniques enable analysts to determine, for example, whether a new program sample is likely to have been written by a person of interest, or to test for the existence of multiple, stylisitically dissimilar authors in a collection of programs. Framing authorship attribution and clustering as machine learning problems, we designed instruction- and structure-based representations of binary code that automatically capture binary code details that reflect programmer style. We developed program clustering techniques that transfer stylistic knowledge across different domains, assigning new programs to clusters based on stylistic similarity with no training data. The results of our evaluation strongly support our claim that programmer style is preserved through the compilation process, and can be recovered from characteristics of the code in program binaries. Our approach to discovering stylistic features builds on our previous research into recovering toolchain provenance, and is part of a general framework for information retrieval in program binaries, with applications in security and software forensics.

References

[1] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, New York, 2006.

[2] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *IEEE Symposium on Security and Privacy (S&P '05)*, Oakland, CA, May 2005.

[3] C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20, 1995.

[4] 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*, 9:1871–1874, 2008.

[5] A. Gray, P. Sallis, and S. MacDonell. Software forensics: Extending authorship analysis techniques to computer programs. In *3rd Biennial Conference of the International Association of Forensic Linguists*, Durham, NC, September 1997.

[6] J. H. Hayes and J. Offutt. Recognizing authors: an examination of the consistent programmer hypothesis. *Software Testing, Verification and Reliability*, 2009.

[7] P. Juola. Authorship attribution. *Foundations and Trends in Information Retrieval*, December 2006.

[8] M. Karim, A. Walenstein, A. Lakhotia, and L. Parida. Malware phylogeny generation using permutations of code. *Journal in Computer Virology*, 1(1):13–23, November 2005.

[9] J. Z. Kolter and M. A. Maloof. Learning to detect and classify malicious executables in the wild. *Journal of Machine Learning Research*, 7:2721–2744, 2006.

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

[11] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic Worm Detection Using Structural Information of Executables. In *International Symposium on Recent Advances in Intrusion Detection (RAID '05)*, Seattle, WA, September 2005.

[12] P. C. Mahalanobis. On the generalised distance in statistics. In *Proceedings National Institute of Sciences of India*, volume 2, 1936.

[13] G. Palmer. A road map for digital forensic research. Technical Report DTR-T001-01 FINAL, Digital Forensics Research Workshop (DFRWS), 2001.

[14] Paradyn Project. ParseAPI: An application program interface for binary parsing. 2011. URL `http://paradyn.org/html/parse0.9-features.html`.

[15] N. E. Rosenblum, B. P. Miller, and X. Zhu. Extracting compiler provenance from program binaries. In *9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE '10)*, Toronto, Ontario, Canada, June 2010.

[16] N. E. Rosenblum, B. P. Miller, and X. Zhu. Recovering the toolchain provenance of binary code, 2011. URL `ftp://ftp.cs.wisc.edu/paradyn/papers/Rosenblum11Toolchain.pdf`. Under submission.

[17] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *ACM SIGMOD International Conference on Management of Data*, San Diego, CA, June 2003.

[18] E. H. Spafford and S. A. Weeber. Software forensics: Can we track code to its authors? Technical Report CSD-TR-92-010, Purdue University, February 1992.

[19] N. X. Vinh, J. Epps, and J. Bailey. Information theoretic measures for clusterings comparison: is a correction for chance necessary? In *26th Annual International Conference on Machine Learning*, ICML '09, Montreal, Quebec, Canada, June 2009.

[20] K. Q. Weinberger and L. K. Saul. Distance metric learning for large margin nearest neighbor classification. *Journal of Machine Learning Research*, February 2009.