Large-Scale and Language-Oblivious Code Authorship Identification

Mohammed Abuhamad
Inha University
abuhamad@seclab.inha.ac.kr

Tamer AbuHmed
Inha University
tamer@inha.ac.kr

Aziz Mohaisen
University of Central Florida
mohaisen@ucf.edu

DaeHun Nyang
Inha University
nyang@inha.ac.kr

ABSTRACT

Efficient extraction of code authorship attributes is key for successful identification. However, the extraction of such attributes is very challenging, due to various programming language specifics, the limited number of available code samples per author, and the average code lines per file, among others. To this end, this work proposes a Deep Learning-based Code Authorship Identification System (DL-CAIS) for code authorship attribution that facilitates large-scale, language-oblivious, and obfuscation-resilient code authorship identification. The deep learning architecture adopted in this work includes TF-IDF-based deep representation using multiple Recurrent Neural Network (RNN) layers and fully-connected layers dedicated to authorship attribution learning. The deep representation then feeds into a random forest classifier for scalability to de-anonymize the author. Comprehensive experiments are conducted to evaluate DL-CAIS over the entire Google Code Jam (GCJ) dataset across all years (from 2008 to 2016) and over real-world code samples from 1987 public repositories on GitHub. The results of our work show the high accuracy despite requiring a smaller number of files per author. Namely, we achieve an accuracy of 96% when experimenting with 1,600 authors for GCJ, and 94.38% for the real-world dataset for 745 C programmers. Our system also allows us to identify 8,903 authors, the largest-scale dataset used by far, with an accuracy of 92.3%. Moreover, our technique is resilient to language-specifics, and thus it can identify authors of four programming languages (e.g., C, C++, Java, and Python), and authors writing in mixed languages (e.g., Java/C++, Python/C++). Finally, our system is resistant to sophisticated obfuscation (e.g., using C Tigress) with an accuracy of 93.42% for a set of 120 authors.

KEYWORDS

Code Authorship Identification; program features; deep learning identification; software forensics

CCS '18, October 15–19, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5693-0/18/10…$15.00
https://doi.org/10.1145/3243734.3243738

ACM Reference Format:
Mohammed Abuhamad, Tamer AbuHmed, Aziz Mohaisen, and DaeHun Nyang. 2018. Large-Scale and Language-Oblivious, Code Authorship Identification. In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), October 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3243734.3243738

1 INTRODUCTION

Authorship identification of natural language text is a well-known problem that has been studied extensively in the literature [30, 31, 34, 45]. However, far fewer works are dedicated to authorship identification in structured code, such as the source code of computer programs [18]. Source code authorship identification is the process of code writer identification by associating a programmer to a given code based on the programmer's distinctive stylometric features. The problem is, however, difficult and different from authorship identification of natural language text. This fundamental difficulty is chiefly due to the inherent inflexibility of the written code expressions established by the syntax rules of compilers.

Code authorship identification relies on extracting features from source code that a programmer produces based on the programmer's preferences in structuring codes and naming variables.Given these features, the main objective of code authorship identification is to correctly assign programmers to source codes based on the extracted features. Being able to identify code authors is both a risk and a desirable feature. On the one hand, code authorship identification poses a privacy risk for programmers who wish to remain anonymous, including contributors to open-source projects, activists, and programmers who conduct programming activities on the side. Thus, in turn, this makes code authors identification a de-anonymization problem. On the other hand, code authorship identification is useful for software forensics and security analysts, especially for identifying malicious code (such as malware) programmers; e.g., where such programmers could leave source code in a compromised system for compilation, or where features of programmers could be extracted from decompiled binaries. Moreover, authorship identification of source code is helpful with plagiarism detection [15], authorship disputes [49], copyright infringement [26], and code integrity investigations [38].

The problem of code author identification is challenging, and faces several obstacles that prevent the development of practical identification mechanisms. First, programming "style" of programmers continuously evolves as a result of their education, their experience, their use of certain software engineering paradigms, and

their work environment [17]. Second, the programming style of programmers varies from language to another due to external constraints placed by managers, tools, or even languages. Third, while it is sometimes possible to obtain the source code of programs, sometimes it is not, and the source code is occasionally obfuscated by automatic tools, preventing their recognition.

To address those challenges, a recent attention to source code authorship identification has revived more than two-decade old work [35, 43] by proposing several techniques [18, 19]. However, there are several limitations with the prior work. Namely, (i) most software features used in the literature for author identification are not directly applicable to another language; features extracted in Java cannot be directly used as features in C or in Python for identifying the same author, (ii) techniques used for extracting code authorship features do not scale well for a large set of authors (see section 2), and (iii) the extracted features are usually large and not all of them are relevant to the identification task, necessitating an additional procedure for feature evaluation and selection [22].

To address the aforementioned issues, this work presents a technique that uses deep learning as a method for learning data representation. Our work attempts to answer the following questions. (i) How can deep learning techniques contribute to the identification of code authors? (ii) To what extent does an authorship identification approach based on deep learning scale in terms of the number of authors given a limited number of code samples per author? (iii) Can deep learning help identify authorship attributes that go beyond language specifics in an efficient way and without requiring a prior knowledge of the language? (iv) Will deep authorship representation still be robust when the source code is obfuscated?

Summary of Contributions. We summarize the main contributions of this work in multiple directions as follows: First, we design a feature learning and extraction method using a deep learning architecture with a recurrent neural network (RNN). The extraction process is fed by a complete or an incomplete source code to generate high quality and distinctive code authorship attributes. The prior work considers preprocessing data transformations which resulted in high quality features for effective code authorship identification. However, this feature engineering process is usually dependent on human prior knowledge of the programming language addressed in a given task. Our approach utilizes a learning process of large-scale code authorship attribution based on a deep learning architecture to efficiently generate high quality features. Also, as input to the deep learning network, we use the TF-IDF (Term Frequency-Inverse Document Frequency) that is already a well-known tool for textual data analysis [14, 27, 31]. Thus, our approach does not require a prior knowledge of any specific programming language, thus it is more resilient to language specifics. In the large-scale dataset experiment, we found that top features are mostly for keywords of the used programming language, which implies that a programmer cannot easily avoid being identified by simply changing the variable names but by dramatically changing his programming style. With this feature learning and extraction method, we were able to achieve comparable accuracy to (and sometimes better than) the state-of-the-art. For example, compared to 100% accuracy in detecting authorship over a small sample (35 C++ programmers) using features extracted from the abstract syntax tree of the source code [19], we provide a similar accuracy over a larger dataset (150 C++ programmers) and close to that accuracy (99%) for other programming languages using our scalable deep learning-based approach (a comparison is in Table 1).

Second, we experimentally conduct a large scale code authorship identification and demonstrate that our technique can handle a large number of programmers (8,903 programmers) while maintaining a high accuracy (92.3%). To make our authorship identifier work at a large scale, Random Forest Classifier (RFC) is utilized as a classifier of a TF-IDF-based deep representation extracted by RNN. This approach allows us to utilize both deep learning's good feature extraction capability and RFC's large scale classification capability. Compared to our work, the largest scale experiment in the literature used 1,600 programmers and achieved a comparable accuracy of 92.83% using nine files per author as shown in Table 9 of [19]. While our dataset includes more than 5.5 times the number of the programmers in the prior work, our technique required less data per author (only seven files) for the same level of accuracy at a lower computational overhead. Our experiments are complemented with various analyses. We explore the effect of limited code samples per author and conduct experiments with nine, seven, and five code samples per author. We investigate the temporal effect of programming style on our approach to show its robustness.

Third, we show that our approach is oblivious to language specifics. Applied to a dataset of authors writing in multiple languages, our deep learning architecture is able to extract high quality and distinctive features that enable code authorship identification even when the model is trained by mixed languages. We based our assessment on an analysis over four individual programming languages (namely, C++, C, Java, and Python) and three combinations of two languages (namely, C++/C, C++/Java, and C++/Python).

Fourth, we investigate the effect of obfuscation methods on the authorship identification and show that our approach is resilient to both simple off-the-shelf obfuscators, such as Stunnix [2], and more sophisticated obfuscators, such as Tigress [3] under the assumption that the obfuscators are available to the analyzer. We achieve an accuracy of 99% for 120 authors with nine obfuscated files, which is better than the previously achieved accuracy in [19].

Finally, we examine our approach on real-world datasets and achieve 95.21% and 94.38% of accuracy for datasets of 142 C++ programmers and 745 C programmers, respectively.

Organization. The remainder of the paper is structured as follows. We review the related work in section 2. We introduce the theoretical background required for understanding our work in section 3. In section 4 we present our deep learning based approach for source code authorship identification. We proceed with a detailed overview of the experimental results of our approach in section 5. Finally, the limitations of this work are outlined in section 6, followed by concluding remarks in section 7.

2 RELATED WORK

Broadly related to our work is the attribution of unstructured text. Authorship attribution for unstructured textual documents is a well-explored area, where earlier attempts to match anonymous written documents with their authors were motivated by the interest of settling the authorship of disputed works, such as *The Federalist*

Table 1: Comparison between our work using deep learning for authorship identification and various previous works in the literature, over the used classification techniques, languages, and approach. MDA=Multiple Discriminant Analysis, FFNN=Feed Forward Neural Network, RNN=Recurrent Neural Network, KNN=K-Nearest Neighbor. Results are excerpted from references.

Reference	# Authors	Languages	Accuracy (%)	Classification Technique
Pellin [40]	2	Java	88.47%	Machine learning (SVM with tree kernel)
MacDonell *et al.* [37]	7	C++	81.10%	Machine learning (FFNN). Statistical analysis (MDA)
MacDonell *et al.* [37]	7	C++	88.00%	Machine learning (case-based reasoning).
Frantzeskou *et al.* [27]	8	C++	100.00%	Rank similarity measurements (KNN)
Burrows *et al.* [16]	10	C	76.78%	Information retrieval using mean reciprocal ranking
Elenbogen *&* Seliya [24]	12	C++	74.70%	Statistical analysis (decision tree model)
Lange *&* Mancoridis [36]	20	Java	55.00%	Rank similarity measurements (nearest neighbor)
Krsul *&* Spafford [35]	29	C	73.00%	Statistical analysis (discriminant analysis)
Frantzeskou *et al.* [27]	30	C++	96.90%	Rank similarity measurements (KNN)
Ding *&* Samadzadeh [22]	46	Java	62.70%	Statistical analysis (canonical discriminant analysis)
Burrows *et al.* [18]	100	C, C++	79.90%	Machine learning (neural network classifier)
Burrows *et al.* [18]	100	C, C++	80.37%	Machine learning (support vector machines)
Caliskan-Islam *et al.* [19]	229	Python	53.91%	Machine learning (random forest)
Caliskan-Islam *et al.* [19]	1,600	C++	92.83%	Machine learning (random forest)
This work	566	C	94.80%	Machine learning (RNN with random forest)
This work	1,952	Java	97.24%	Machine learning (RNN with random forest)
This work	3,458	Python	96.20%	Machine learning (RNN with random forest)
This work	8,903	C++	92.30%	Machine learning (RNN with random forest)

Papers. Through the last two decades, studies of authorship attribution have focused on determining indicative features of authorship using the linguistic information (e.g., length and frequency of words or pairs of words, vocabulary usage, sentence structure, etc.). Recent works have shown high accuracy in identifying authors of various datasets such as chat messages, e-mails, blogs and microblogs entries. Abbasi and Chen [5] proposed *writeprints*, a technique that demonstrated a remarkable result in capturing authorship stylometry in diverse corpora including eBay comments and chat as well as e-mail messages of up to a hundred unique authors. Uzuner and Katz [48] provided a comparative study of different stylometry methods used for authorship attribution and identification. Afroz *et al.* [6] investigated the possibility of identifying cybercriminals by analyzing their textual entries in underground forums, even when they use multiple identities. Stolerman *et al.* [46] considered using classifiers' confidence to address the open-world authorship identification problem. Another body of work has investigated authorship attribution under adversarial settings either for the purpose of hiding the identity or impersonating (*i.e.,* mimicking) other identity. Brennan *et al.* [13] studied three adversarial settings to circumvent authorship identification: obfuscation, imitation and translation.

Addressing authorship attribution for structured data, such as source code, presents a challenge and another interesting body of work in the field of authorship attribution. A summary of the related work is in Table 1, with a comparison across four variables: the number of authors, the programming language, the accuracy, and the used technique. The method commonly followed in the literature for code authorship identification research has two main steps: feature extraction and classification. In the first step, software metrics or features representing an author's distinctive attributions are processed and extracted. In the second step, those features are fed into an algorithm to build models that are capable of discriminating among several authors. While the second step is a straightforward data-driven method, the first step leads to major challenges and has become the focus of several researches for more than two decades. Designing authorship attributions that reflect programmers' stylistic characteristics has been investigated by multiple works, since the early work of Krsul *et al.* [35]. Existing code authorship attribution methods include extracting features from different levels of programs, depending on the targeted code for analysis. These features can be as simple as byte-level or term-level features [27], or as complex as control and data flow graphs [7, 39, 41] or even abstract syntax tree features [19, 40]. The quality of extracted authorship attributes significantly affects the identification accuracy and the extent to which the proposed method can scale in terms of the number of authors. Krsul and Spafford [35] were the first to introduce 60 authorship stylistic characteristics categorized into three classes: programming layout characteristics (e.g., the use of white spaces and brackets), programming style characteristics (e.g., average variable length and variable names), and programming structure characteristics (e.g., the use of data structures and number of code lines per function). MacDonell *et al.* [37] adopted only 26 authorship stylistic characteristics extracted using custom-built software IDENTIFIED. Some of these characteristics were extracted by calculating the occurrence of features per line of code. Frantzeskou *et al.* [27] introduced Source Code Author Profiles using byte-level *n*-grams features for authorship attribution. Their work was inspired by the success of using *n*-gram in text authorship identification. Moreover, using *n*-gram have made the approach language-independent, an issue that limited preceding works. Lange and Mancoridis [36] were the first to consider a combination of text-based features and software-based features for code authorship identification. Their work used feature histogram distributions for finding the best combination of features that achieve the best identification accuracy. Elenbogen and Seliya [24] considered six features to establish programmers' profiles based on personal experience

and heuristic knowledge: the number of comments, lines of code, variables' count and name length, the use of for-loop, and program compression size. Burrows *et al.* [16] used a combination of *n*-gram and stylistic characteristics of programmers for authorship identification. Most recently, Caliskan-Islam *et al.* [19] showed the best results over a large scale dataset (1,600 programmers) by far, taking advantage of abstract syntax tree node bigrams. Their approach included an extensive feature extraction process for programmer code stylometry involving code parsing and abstract syntax tree extraction, resulting in large and sparse feature representations, and dictating a further feature evaluation and selection process. After authorship attributions have been introduced, most of the previous works on code authorship identification have adopted either a statistical analysis approach, a machine learning-based classification, or a ranking approach that is based on similarity measurements in order to classify code samples [18]: Statistical analysis methods are considered for limiting the feature space to discover highly-indicative features of authorship. Krsul and Spafford [35], MacDonell *et al.* [37], and Ding and Samadzadeh [22] used discriminant analysis for identifying authors. As for machine learning, various approaches are used for source code authorship identification : case-based reasoning [37], neural networks [18, 37], decision trees [24], support vector machine [18, 40], and random forest [19]. As a general approach of similarity measurement, ranking based on similarity measurements can be used to compute the distance between a test instance and candidate instances in the feature space. Using k-nearest neighbor is one method to assign instances to authors with similar instances. Lange and Mancoridis [36], Frantzeskou *et al.* [27], and Burrows *et al.* [16] implemented different ranking methods based on similarity measurements.

Code authorship identification could also be done on binaries, which is addressed in the literature. Binary-level techniques [7, 20, 39, 41] are advocated as a viable tool for malware, proprietary software, and legacy software attribution [39]. While very useful, binary-level techniques work under the assumption that a toolchain provenance is used to generate the binary code, including the operating system, compiler family, version, optimization level and source language are known to the analyzer. Source-level techniques, on the other hand, are more flexible and equally useful, especially in addressing incomplete pieces of code (which cannot be compiled). Even when operating on binaries, code-like artifacts are what is being actually analyzed. For example, Caliskan-Islam *et al.* [20] showed that a simple reverse engineering process of binary files can generate a pseudo-code that can be treated as a source code for code authorship identification.

3 BACKGROUND AND MOTIVATION

TF-IDF is a well-known tool for text data mining. The basic idea of TF-IDF is to evaluate the importance of terms in a document in a corpus, where the importance of a term is proportional to the frequency of the term in a document. However, it is highly likely to be emphasized by documents which have a very common term over a corpus. Therefore, how specific a given term is over a corpus should be considered. It can be quantified as an inverse function of the number of documents in which it appears. In building the data preprocessing component of our technique, a term t in a document d of a corpus $\mathcal{D}$ is assigned a weight using the formula $\text{TF-IDF}(t, d, \mathcal{D}) = \text{TF}(t, d) \times \text{IDF}(t, \mathcal{D})$, where $\text{TF}(t, d)$ is the term frequency (TF) of t in d and $\text{IDF}(t, \mathcal{D}) = \log(|\mathcal{D}|/\text{DF}(t, \mathcal{D})) + 1$, where $|\mathcal{D}|$ is the number of documents in $\mathcal{D}$ and $\text{DF}(t, \mathcal{D})$ is the number of documents containing the term t.

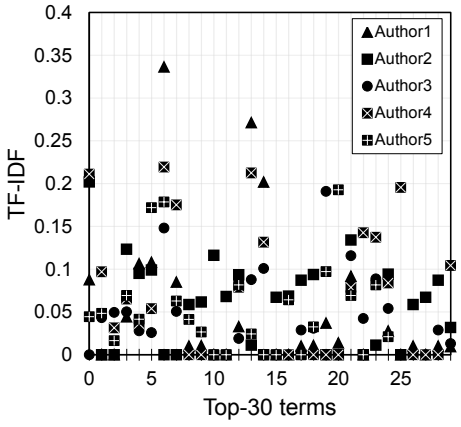


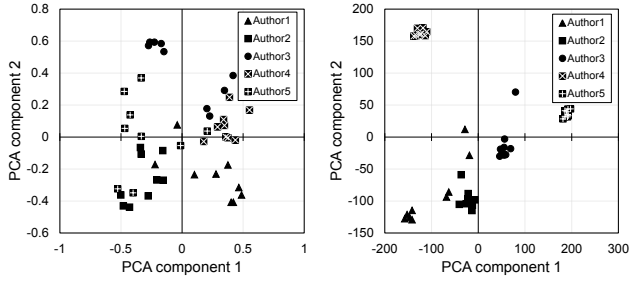
Figure 1: The TF-IDF values of top-30 terms for five programmers. The value of a term is different among authors who use the same term. The terms are: ('ans', 'begin', 'begin end', 'bool', 'break', 'char', 'cin', 'cin int', 'cmath', 'cmath include', 'const', 'const int', 'continue', 'cout', 'cout case', 'cstdlib', 'cstdlib include', 'cstring', 'cstring include', 'define', 'define pb', 'double', 'end', 'endl', 'false', 'freopen', 'include cmath', 'include cstdlib', 'include cstring', 'include map').

3.1 Term Frequency-Inverse Document Frequency (TF-IDF)

Using TF-IDF as initial representation for code files is motivated by its wide-range applications on processing textual data. Terms and n-grams features (frequency) are commonly used in information retrieval and have been adopted for code authorship identification [14, 27, 31]. TF-IDF features describe an author's preferences on using certain terms, or his/her preference for specific commands, data types, and libraries. Figure 1 illustrates the mean TF-IDF values of the top-30 terms used by five programmers in nine C++ files of code. Even with slight difference for some terms, the TF-IDF value differs from one programmer to another presenting its validity to be used as initial representation of code files. If the values are composed into one vector for each programmer, then we can distinguish more distinctively each author by observing the distribution of the values. Another observation is that the top features are for keywords of the used programming language. Such observation suggests that a programmer cannot easily avoid being identified by simply changing the variable names but rather by dramatically changing the programming style itself. For example, it seems that 'cout' should not have such a high TF-IDF score because it is a common command for printing out a message, but it has. This is because 'cout' has been used by only a small number of programmers solving problems in Google Code Jam, which in turn makes the keyword distinctive. Thus, frequent use of 'cout' can be regarded as some programmer's programming style.

3.2 Deep Representation of TF-IDF Features

Code authorship identification can be formulated as a classification problem, where authors are classified based on their distinctive authorship attributes. The performance of machine learning methods relies on the quality of data representation (features or attributes), which requires an expensive feature engineering process. This process is sometimes labor-intensive and heavily dependent on human prior-knowledge in the classification application field [10]. Identification of a large number of authors using TF-IDF directly cannot be easily achieved as can be seen in Figure 2(a). Recently, representation learning has gained increasing attention in the machine learning community and has become a field in and of itself dedicated to enabling easier and more distinctive feature extraction processes [11]. Among several representation learning methods, deep learning has achieved a remarkable success in capturing more useful representations through multiple non-linear data transformations. Deep learning representations have enabled several advancements in many machine learning applications such as speech recognition and signal processing [30–32], object recognition [33–35], natural language processing [36, 37], and multi-task and transfer learning [34,38]. Since the breakthrough work of Hinton *et al.* [29], multiple representation techniques using deep learning were presented in the literature. Those techniques have been employed in many fields, with various applications, as reported in [9, 10]. One potential application that was not previously explored in the literature is code authorship identification, which we explore in this work. The techniques used in this paper are LSTM (Long Short-Term Memory) and Gated Recurrent Units (GRU) that are sorts of Recurrent Neural Network (RNN) among various Deep Neural Networks (DNN).



(a) PCA visualization of TF-IDF. (b) Deep representation visual.

Figure 2: The PCA visualization of TF-IDF and deep representation of code attributions for five programmers.

Deep LSTMs and GRUs [42] with multiple layers demonstrated a remarkable capability to generate representations from long input sequences. In this work, we investigate both LSTM and GRU capabilities of extracting code authorship attributions from TF-IDF code representations, which are a good fit because of the scale of our problem.; we will elaborate on this investigation in section 4.2. The TF-IDF representations are then fed into our deep learning structure as one sequence per code sample to generate high quality representations that enable accurate authorship identification. To examine the characteristics of TF-IDF, we visualized TF-IDF values of top-30 terms of five authors. For visualizing code files of a programmer, we used the Principal Components Analysis (PCA). The PCA is a statistical tool that is widely used as a visualization technique that reflects the difference in observations of multidimensional data for the purpose of simplifying further analysis [8, 25]. Figure 2 shows PCA visualizations of C++ code samples for five programmers with nine sample each. In Figure 2(a), code files are presented with the initial TF-IDF features, which are insufficient to draw a decision boundary for all programmers. In Figure 2(b), however, the deep representations have increased the margin for decision boundary so distinguishing programmers has become easier. This visualization of the representations space (TF-IDF features and deep representations) illustrates the quality of representations obtained using the deep learning technique.

3.3 RFC over Deep Representations

To identify authors, we need a scalable classifier that can accommodate a large number of programmers. However, the deep learning architecture alone does not give us a good accuracy (e.g., 86.2% accuracy for 1,000 programmers). Instead of using the softmax classifier of the deep learning architecture, we use RFC [12] for the classification, and by providing the deep representation of TF-IDF as an input. RFC is known to be scalable, and our target dataset has more than 8,000 authors (or classes) to be identified. Such a large dataset can benefit from the capability of RFC.

Our authorship identifier is built by feeding a TF-IDF-based deep representation extracted by RNN and then classifying the representation by RFC. This hybrid approach allows us to take advantage of both deep representation's distinguishing attribute extraction capability and RFC's large scale classification capability.

4 DL-CAIS: DEEP LEARNING-BASED CODE AUTHORSHIP IDENTIFICATION SYSTEM

Our approach for large-scale code authorship identification has three phases: preprocessing, representation through learning, and classification. We briefly highlight those phases in the following and explain each phase of the proposed approach in more details in the subsequent subsections.

Preprocessing. The first phase starts with data preprocessing to handle code samples and generate initial representations. The initial representations of code samples are later fed into a deep learning architecture to learn more distinctive features. Finally, deep representations of code authorship attributions are used to construct a robust random forest model. Figure 3 illustrates the overall structure of our proposed system. In the first phase, a straightforward mechanism is used to represent source code files based on a weighting scheme commonly used in information retrieval.

Representation by Learning. This phase includes learning deep representations of authorship from less distinctive ones. Those representations are learned using an architecture with multiple RNN layers and fully-connected layers.

Classification. After training the deep architecture, the resulting representations are used to construct a random forest classifier with 300 trees grown to the maximum extent.

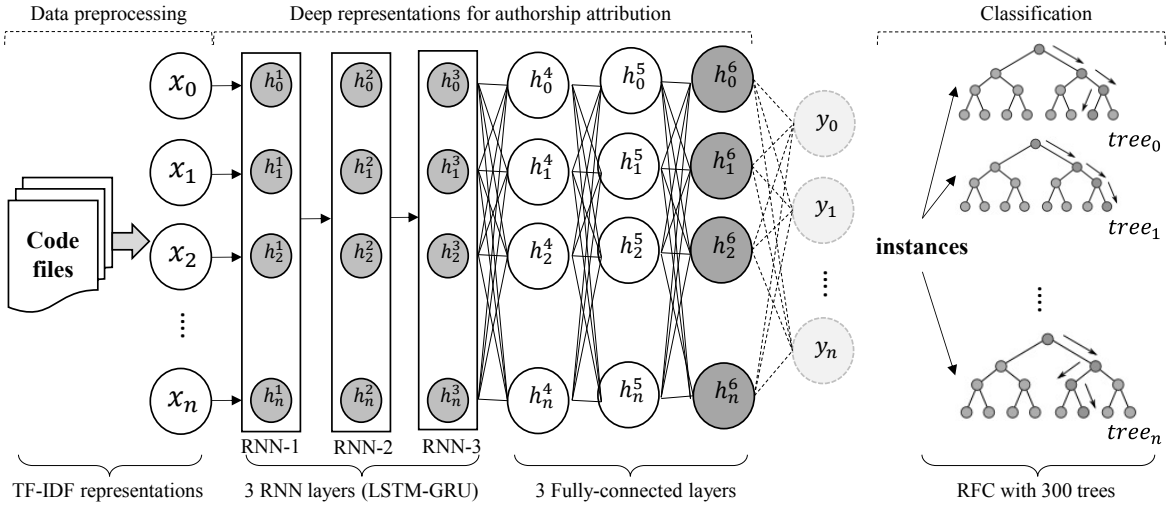


Figure 3: A high-level illustration of the Deep Learning-based Code Authorship Identification System (DL-CAIS). This illustration shows the three phases of preprocessing (TF-IDF feature representation), better representation through learning (using the RNN and fully-connected layers), and the classification (using 300 trees in a random forest classifier).

4.1 Data Preprocessing

Code files are represented by TF-IDF, as described in section 3.1. TF-IDF is a standard term weighting scheme commonly used in information retrieval and document classification communities. While we could have used TF instead, we use the TF-IDF to minimize the effect of frequent terms in a given corpus. This is due to the observation that more distinctive terms appear in certain documents (code files) rather than in most of the corpus. In our implementation, we use several methods for optimizing the representation of documents, such as eliminating stop words, normalizing representations, and removing indistinctive features. We note that TF-IDF representations cover word unigrams, bigrams, and trigrams in the presented code files, meaning a term can be a term of one, two, or even three words. As such, the input vector for a document d_i to the deep learning model is represented as follows:

$$[\text{TF-IDF}(t_1, d_i, \mathcal{D}), \text{TF-IDF}(t_2, d_i, \mathcal{D}), \ldots, \text{TF-IDF}(t_n, d_i, \mathcal{D}))],$$

where n is the total number of terms in the corpus $\mathcal{D}$. To train our model, a set of documents for each user is used to calculate the above vector. However, targeting a corpus of thousands of code files may lead to high-dimensional vector representations (*i.e.,* too many terms). Several feature selection methods that reduce the dimensionality using statistical characteristics of features exist. In this work, we investigated different feature selection methods for representing code files to be further fed into the deep learning model, and we found that all approaches lead to similar results. For every term t_i and every document d_i, we calculate

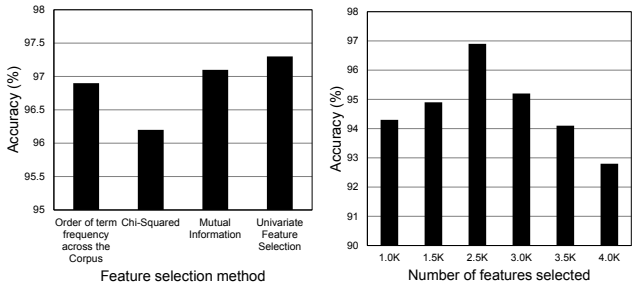
$$x_i = \bigcup_{j=1,\ldots,|\mathcal{D}|} \text{TF-IDF}(t_i, d_j, \mathcal{D}), \tag{1}$$

where $\cup$ is a feature selection operator such as the order of term frequency, chi-squared (χ^2) value, mutual information, or univariate feature selection. Using equation (1), x_i's for all terms in the corpus are calculated.

Feature Space. Among the n features, we choose the top-k terms for which x_i's are the largest to reduce the dimensionality and form an input vector to the learning model. For simplicity, we adopt the embedded function of selecting the top-k features by the TF-IDF vectorizer available by the scikit-learn package, which uses the order of term frequencies across all files. With TF-IDF as the method used to represent code files, the feature space needs to be sufficient to distinguish files' authors. For large dataset containing thousands of files (e.g., more than 1,000 programmer with nine files each), the top-k features (for a fixed k) may or may not be sufficient to enable the model to identify authors accurately. As such, we investigated the number of features considered to represent code files as an optimization problem of accuracy. This experiment suggested that 2,500 features are sufficient for the subsequent experiments. The high dimensionality is likely to introduce overfitting issues, but we addressed the overfitting issues by two regularization techniques (See section 4.2), and also conducted all the experiments by repeated k-fold cross validations (See section 5). Figure 4(a) shows the impact of feature selection, using four different approaches, on the accuracy of our approach using TF-IDF features in identifying code authors. In this experiment, we use 1,000 features to identify authors in a 250 C++ programmers experiment. The results demonstrate a substantial accuracy rate (of over 96%) for the given problem size. In Figure 4(b), we demonstrate the impact of the number of the selected TF-IDF features on the accuracy of the classifier. We note that the accuracy increases up to some value of the number of features after which it decays quickly. The accuracy, even with the smallest number of features, is relatively high.

4.2 Deep Representation of Code Attributes

For deep representations, we used multiple RNNs and fully-connected layers in a deep learning architecture. For our implementation, we used TensorFlow [4], an open source symbolic math library for building and training neural networks using data flow graphs. We



(a) Feature selection method. (b) Number of features.

Figure 4: Feature selection analysis

ran our experiments on a workstation with 24 cores, one GeForce GTX TITAN X GPU, and 256GB of memory. We note that our use of the GeForce GTX TITAN X GPU is purely performance driven, and the specific platform does not affect the end-results. Upon the release of our scripts and data, our findings can be reproduced on any other experimental settings.

Addressing Overfitting. To control the training process and prevent overfitting, two different regularization techniques were adopted. The RNN layers in our deep learning architecture included a *dropout regularization* technique [44]. In essence, this technique randomly and temporally excludes a number of units on the forward pass and weight updates on the backward pass during the training process. The dropout regularization technique has been shown to enable the neural network to reach better generalization capabilities [44].

The second technique concerns the fully-connected layers. For that we use the *L2 regularization*, which penalizes certain parameter configurations: given a loss function $\text{loss}(\theta,\ D) = \frac{1}{n}\sum_{i=1}^{n} d(y_i,\ \hat{y}_i)$, where θ is the set of all model parameters, D is the dataset of length n samples, and $d()$ indicates the difference between DNN's output $\hat{y}_i$ and a target y_i, the regularization loss becomes $\text{Reg}_{loss}(\theta, D) = \frac{1}{n}\sum_{i=1}^{n} d(y_i,\ \hat{y}I_i^{'}) + [\lambda \times \text{Reg}(\theta)]$, where λ is a constant that controls the importance of regularization and $\text{Reg}(\theta) = (\sum_{j=0}^{|\theta|} |\theta_j|^p)^{\frac{1}{p}}$, where $p = 1$ or 2 (hence, the $L1$ and $L2$ nomenclature).

Selecting Layers. The parameters of our final architecture of the deep learning model were chosen after various iterations of experiments, upon which we chose three RNN layers with dropout keep-rate of 0.6, followed by three fully-connected layers with ReLU activation. Each of the fully-connected layers has 1024 units except the last layer, which has 800 units representing the dimensionality of code authorship features for a given input file. During the representation learning process, this architecture is connected to the softmax output layer that represents the class of authors to direct the training process. The training process follows a supervised learning approach, where only the intended model is meant to provide a data transformation that leads to the best probability of its correct class label. Targeting a large-scale code authorship identification process with thousands of programmers (thousands of classes), the deep learning architecture alone does not accurately identify programmers (86.2% accuracy for 1000 programmers). Thus, we use the output of layer L_{k-1} (where the L_k is the softmax layer) to be the deep representations of code authorship features. Deep representations of code authorship features are then subjected to a classification process using RFC (section 3.3), which is proven to be robust and scalable for large datasets. The weights of the learning network were initialized using a normal distribution of small range near 0, a small variance, and mean of 0.

Training Procedure. To train our deep learning architecture, we used TensorFlow's Adaptive Moment estimation (Adam) [32] with a learning rate of 10^{-4}, and without reducing the learning rate over time. Adam is an efficient stochastic optimization method that only requires first-order gradients with little memory requirements. Using estimations of the first two moments of the gradients, Adam assigns different adaptive learning rates for different parameters. This method was inspired by combining the advantages of two popular stochastic optimization methods, AdaGrad [23], which is efficient for handling sparse gradients, and RMSProp [47], which is efficient for on-line and non-stationary settings [32].

Further Optimizations. In the training process of the deep learning architecture, we used a mini-batch size ranging from 64 to 256 observations. The idea of using mini-batches reduces the variance in gradients of individual observations since observations may be significantly different. Instead of computing the gradient of one observation, the mini-batch approach computes the average gradient of a number of observations at a time. This approach is widely accepted and commonly used in the literature [42]. The training termination mechanism was either to reach 100,000 iterations or to achieve an early termination threshold for the loss value.

4.3 Code Authorship Identification

Using deep authorship features learned in section 4.2, we construct an RFC for code authorship identification. In doing so, and based on various experiments, we select 300 decision trees for an RFC—this configuration has shown to provide the best trade-off between the model construction time and its accuracy [19].

Implementation. We used scikit-learn to implement the RFC using the default settings for building and evaluating features on each split, and all trees were grown to the largest extent without pruning. Following the approach adopted by [19], we report results of test accuracy using stratified k-fold cross-validation [33], where k depends on the number of observations per class in the dataset (i.e., 9-fold used for 9 files per author, 7-fold for 7 files per author, and so on). The k-fold cross-validation technique aims to evaluate how well our model will generalize to an independent dataset. In this model, the original dataset is randomly partitioned into k equal-sized subsets. Of the k subsets, a single subset is used for testing, and the remaining $k-1$ subsets are used for training. This cross-validation is repeated k times, where each subset is given a chance to be used for testing the model built from the $k-1$ subsets, and the evaluation metric (e.g., accuracy) is the computed as average of the k validations.

Parameters Tuning. Through various experiments we confirm that choosing less than 300 trees (and as few as 100 trees) may degrade the accuracy by only 2%.

5 EVALUATION AND DISCUSSION

In this section, we present the results of several experiments to address various possible scenarios of our identification approach. In our evaluation, we deliver the following: (1) We present results of code author identification over a large dataset. We demonstrate

our central results for programmer authorship identification and how our approach scales to 6,635 programmers with nine files each and to 8,903 programmers with seven files each. Our experiments cover the entire Google Code Jam dataset from 2008 to 2016, an unprecedented scale compared to the literature (see Table 1). (2) We investigate our system's performance with fewer code files per author and demonstrate its viability. (3) We evaluate the robustness of our identification system under programmers' style evolution and change in development environment, and demonstrate that changes minimally affect the performance of our approach. We complement this study by exploring the temporal effects of programming style on our approach of identification. (4) We push the state of identification evaluation by looking into mixed language identification. Particularly, we show results using two language files for programmers (C and C++, Java and C++, and Python and C++). (5) We examine how off-the-shelf obfuscators affect our system's performance. Our results are promising: we show that it is possible to identify authors with high accuracy, which may have several privacy implications for contributors who want to stay anonymous through obfuscation (see section 1). (6) We investigate the applicability of our approach using real-world dataset collected from Github, including two programming languages (e.g., C and C++).

5.1 Data Corpus

The Google Code Jam (GCJ) is an international programming competition run by Google since 2008 [1]. At GCJ, programmers from all over the world use several programming languages and development environments to solve programming problems over multiple rounds. Each round of the competition involves writing a program to solve a small number of problems—three to six, within a fixed amount of time. We evaluate our approach on the source code of solutions to programming problems from GCJ. The most commonly used programming languages at GCJ are C++, Java, Python, and C, in order. Each of those languages has a sufficient number of source code samples for each programmer, thus we use them for our evaluation. For a large-scale evaluation, we used the contest code from 2008 to 2016, with general statistics as shown in Table 2. The table shows the number of files per author across years, with the total number of authors per programming language and the average file size (lines of code, LoC). For evaluation, we create the following three dataset views (Tables 2–4):

(1) **Dataset 1:** includes files across all years from 2008 to 2016 in a "cross-years" view, as shown in Table 2.
(2) **Dataset 2:** consists of code files for participants drawn from 2015 and 2016 competitions for four programming languages, as shown in Table 3.
(3) **Dataset 3:** consists of programmers who wrote in more than one language (i.e., Java-C++, C-C++, and Python-C++) as shown in Table 4.

Number of Files. In [19], the use of nine files per programmer for accuracy is recommended. Our approach provides as good–or even better–accuracy with only seven files, as shown in §5.3.

5.2 Large-scale Authorship Identification

Experiment. In this experiment, we used dataset 1 in Table 2. There are four large scale datasets corresponding to four different programming languages with programmers who have exactly nine code files (first row in Table 2). The number of code files per author in this experiment was suggested by [19] to be sufficient for extracting distinctive code authorship attribution features. In our experiment, we started each dataset with a small number of programmers and increased this number until we included all programmers in the dataset. In particular, we used an RFC with stratified 9-fold cross validation to evaluate the accuracy of identifying programmers. We repeated the k-fold cross validation five times with different random seeds and reported the average.

Evaluation Metric. For evaluation, we use the accuracy, defined as the percentage of code files correctly attributed over the total number of tested code files. Using the accuracy instead of other evaluation metrics (e.g. precision and recall) is enough because the classes are balanced in terms of the number of presented files per class in the dataset.

Results. Figure 5 shows how well our approach scales for a large number of programmers, and for the different programming languages. The results report the accuracy when using different RNN units in learning code authorship attribution and RFC for authors identification (*i.e.,* using either LSTM-RFC or GRU-RFC unit). In Figure 5(a), the LSTM-RFC performance results show that our approach achieves 100% accuracy for 150 C++ programmers with randomly selected nine code files. We note here that FPR is trivially computed as (1 - accuracy), because the dataset is balanced. As we scale our experiments to more programmers, the accuracy remains high, with 92.2% accuracy for 6,635 programmers. Given the same experimental configuration, similar results are obtained for the Java programming language, as illustrated in Figure 5(b) with 100% accuracy when the number of programmers is 50 programmers.

Table 2: Datasets used in our study with the corresponding statistics, including the number of authors with at least a specific number of files across all years.

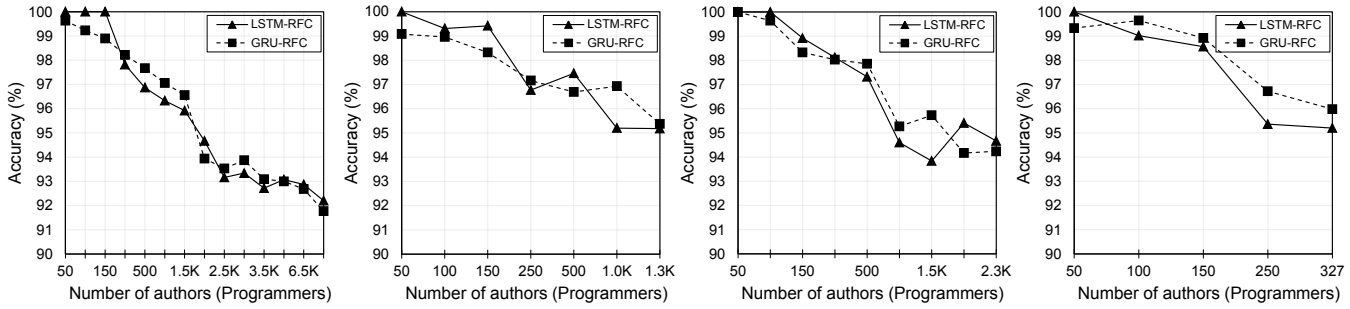
Competition Year	Author Files	No. of Authors for Languages			
		C++	C	Python	Java
Across Years	9	6635	327	2300	1279
Across Years	7	8903	566	3458	1952
Across Years	5	12411	1156	5525	3345
Average Lines of Code		71.53	65.20	44.44	86.70

Table 3: Two datasets with the corresponding author counts for authors who had seven files at the Google Code Jam (GCJ) 2015 and 2016 competitions.

Competition Year	Author Files	No. of Authors for Languages			
		C++	C	Python	Java
2015	7	2241	41	398	132
2016	7	1744	21	390	317
across 3 years*	7	292	NA	44	50
*Programmers participated in 2014, 2015 and 2016					

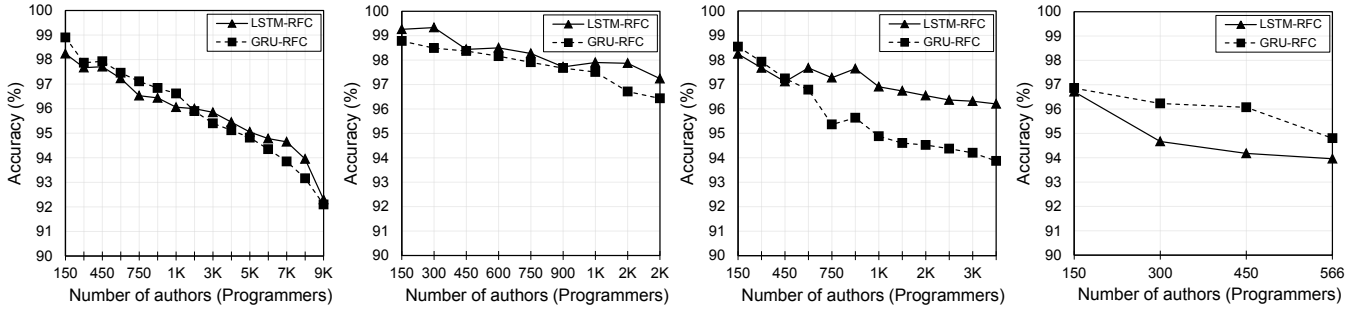
Table 4: A dataset used in our study to demonstrate identification across multiple languages. The dataset includes authors with nine files written in multiple languages.

Competition Year	No. of Authors for Multiple Languages		
	C++-C	C++-Java	C++-Python
Across Years	1897	855	626



(a) C++ (9 files/programmer) (b) Java (9 files/programmer) (c) Python (9 files/programmer) (d) C (9 files/programmer)

Figure 5: Accuracy of authorship identification of programmers with nine sample code files per programmer in four programming languages (C++. Java, Python, and C). Notice that the accuracy is always higher than 92% even with the worst of the two options of classifiers, and decay in the accuracy is insignificant despite a significant increase in the number of programmers.



(a) C++ (7 files/programmer) (b) Java (7 files/programmer) (c) Python (7 files/programmer) (d) C (7 files/programmer)

Figure 6: Accuracy of authorship identification of programmers with seven sample code files per programmer in four programming languages (C++. Java, Python, and C). Notice that the accuracy is always high even with large number of programmers.

Upon scaling the experiments to more programmers, we achieve 99.42% accuracy for 150 programmers, and 95.18% accuracy for 1,279 programmers. For the Python language dataset, our approach achieved an accuracy of 100% for 100 programmers, 98.92% for 150 programmers, and 94.67% for 2,300 programmers, as shown in Figure 5(c). Finally, for the C programmers, Figure 5(d) shows that the accuracy reaches 100% for 50 programmers, 98.56% for 150 programmers, and 95.2% for the total of 327 programmers. These results indicate that both deep LSTMs and GRUs are capable of learning deep representations of code authorship attribution that enable achieving large scale authorship identification regardless of the used programming language.

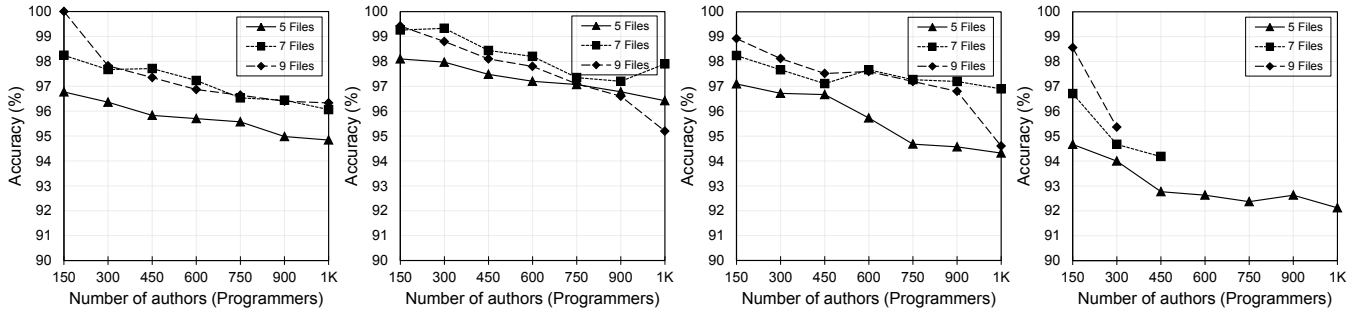
5.3 Effect of Code Samples Per Author

The availability of more code samples per author contributes to better code authorship identification, whereas less code samples restrain the extraction of distinctive features of authorship [17, 19].

Experiment 1: Seven Files per Author. For this experiment, we created two datasets with seven and five code samples per programmer for four different languages, as shown in Table 2 (second row). We used RFC with stratified 7-fold cross validation to evaluate the accuracy of identifying programmers at the dataset with seven files per programmer. As the number of available code samples per author decreased, we found that the number of authors increased (Table 2). The goal of this experiment is to investigate the effects of having less files per author on the accuracy of our approach.

Table 5: Results of the accuracy of our approach in authorship identification for programmers who solved seven problems using the C++ programming language.

Competition Year	# Authors	LSTM-RFC	GRU-RFC
2015	150	98.98	98.24
	300	98.64	97.94
	450	98.1	97.6
	600	97.56	97.21
	750	97.28	96.67
	900	96.34	96.4
	1000	96.32	95.98
	1500	95.88	95.22
	2000	95.67	94.9
	2241	95.23	94.67
2016	150	99.12	98.67
	300	98.34	98.31
	450	98	97.62
	600	97.54	96.84
	750	97.28	96.18
	900	96.7	95.64
	1000	96.37	94.88
	1500	95.66	94.14
	1744	95.17	93.54



(a) C++ programming language (b) Java programming language (c) Python programming language (d) C programming language

Figure 7: Accuracy comparison of authorship identification of programmers in case of five, seven, and nine sample code files per programmer in four programming languages (C++. Java, Python, and C). Notice that the accuracy is always higher than 92%, and regardless of the number of authors. While best results are achieved for the larger number of files, the lowest number of files (of 5) still provides $\sim$ 92% in the worst case.

Results. Figure 6 illustrates the results of our approach using the dataset of all programmers with seven code samples for four different programming languages. Figure 6(a) shows an accuracy of 98.24% when using LSTM-RFC for 150 C++ programmers, and an accuracy of 92.3% for 8,903 programmers. Figure 6(b) shows an accuracy of 99.26% for 150 Java programmers when using LSTM-RFC, and 97.24% accuracy when scaling the experiment to 1,952 programmers. Figure 6(c) shows an accuracy of 98.24% when using LSTM-RFC for 150 Python programmers, and an accuracy of 96.2% when scaling the experiment to 3,458. Finally, Figure 6(d) shows the result for C programmers, where LSTM-RFC is used: an accuracy of 96.71% for 150 C programmers, and 93.96% for 566 C programmers.

Comparison. Compared with the experimental result of identifying authors using nine code samples per author, as in section 5.2, the accuracy does not degrade even when using less code samples per author. Moreover, the results show that our approach is still capable of achieving high accuracy even with more authors compared to the previous experiments. This result presents the largest-scale code authorship identification by far, indicating that seven files per author are still sufficient for extracting distinctive features.

Experiment 2: Five Files per Author. We created a dataset with five source code samples per programmer in the four different programming languages, as shown in Table 2 (third row). We used RFC with stratified 5-fold cross validation to evaluate the accuracy of identifying programmers at the dataset with five files per programmer. As the number of available code samples per author decreased, we found that the number of authors increased (Table 2). The goal of this experiment is to further investigate the effects of having even lesser files per author on the accuracy of our approach.

Results. Figure 7 shows the results for 1,000 programmers, demonstrating the effect of decreasing the number of sample files for each author. Figure 7(a) shows that our approach provides an accuracy of 96.77% for attributing authors in 150 C++ programmers when using LSTM-RFC. Comparing the results of those datasets with the nine and seven source code samples for each programmer, the accuracy loss was only 3.23% and 1.47%, respectively. As we scale to 1,000 C++ programmers, our approach achieves an accuracy of 94.84%. This result proves that our approach still achieves high accuracy even with fewer sample files per programmer. The results of accuracy with smaller number of files per author generalize to other programming languages. Using the same approach and settings as above, Figure 7(b) shows an accuracy of 98.1% and 96.42% for 150 and 1,000 Java programmers, respectively. Figure 7(c) show an accuracy of 97.1% and 94.32% for 150 and 1,000 of Python programmers. Finally, Figure 7(d) shows an accuracy of 94.67% and 92.12% for 150 and 1,000 C programmers, respectively.

Using only five code samples per author, the accuracy of our approach does not significantly degrade. From those experiments we conclude that learning deep code authorship features using either deep LSTMs or GRUs enables large scale authorship identification even with limited availability of code samples per author.

5.4 Effect of Temporal Changes

The literature suggests that temporal effect is a challenge for code authorship identification, since the programming style of programmers evolves rapidly with time due to their education and experience [17, 19, 28]. We investigate the impact of temporal effect on source code authorship identification. The experiments include two parts: 1) exploring the existence of such impact on the identification process, 2) examining our approach against such effect.

Experiment 1: Temporal Effect on Accuracy. This experiment answers the following question: *Do temporal effects influence the accuracy of code authorship identification?*

To answer this question, we conducted an experiment where results from identifying authors from the same year is compared with results across different years throughout the competition. We examined our approach using a dataset of source codes written by programmers within one competition year, where all programmers solve the same set of problems. Two datasets of GCJ competition of the 2015 and 2016 code samples were created individually with seven code files per programmer, as shown in Table 3. In this experiment, we used a random forest and stratified 7-fold cross validation to evaluate the accuracy of identifying programmers.

Results. Table 5 summarizes the results of this experiment when applying LSTM-RCF and GRU-RCF for C++ programmers in two separate years. The accuracy of code authorship identification reaches

Table 6: The accuracy of authorship identification for programmers with seven samples problems (programs) using the Java programming language.

Competition Year	# Authors	LSTM-RFC	GRU-RFC
2015	132	99.64	99.12
2016	150	99.4	98.62
	300	98.34	97.56
	317	98.18	96.98

Table 7: The accuracy of authorship identification for programmers with seven programs using the Python. Note that the accuracy is always above 96%**.**

Competition Year	# Authors	LSTM-RFC	GRU-RFC
2015	150	98.96	97.6
	300	98.18	97.42
	398	98	97.1
2016	150	99.1	98.6
	300	98.67	97.34
	390	97.94	96.47

Table 8: The accuracy of authorship identification for programmers who solved seven problems using the C programming language. Notice the accuracy is always close to 100%

Competition Year	# Authors	LSTM-RFC	GRU-RFC
2015	41	100	99.44
2016	21	100	100

95.23% for 2,241 C++ programmers and 95.17% for 1,744 C++ programmers from 2015 and 2016 competitions, respectively. Our approach also shows high accuracy results for Java, Python, and C programming languages, as shown in Table 6, Table 7, and Table 8.

In comparison with the cross-year dataset, results of this experiment are shown to provide better accuracy, which indicates that temporal effects impact the accuracy of code authorship identification. However, these effects are insignificant—e.g., only 0.74% (=98.98%-98.24%) for C++ with seven files in the case of the year 2015. This is part due to the power of our approach in learning more distinctive and deep features of the studied domain.

Experiment 2: Testing Different Year's Dataset from Training Dataset. In this experiment we attempt to answer the following question: *If temporal effects do exist, can a model trained on data from one year identify authors given data from a different year?* To answer this question, we collected a dataset of sample codes for programmers who participated in three consecutive years from 2014 until 2016. The dataset include seven code files per programmer in each year. The total number of programmers included in the dataset of different languages is shown in Table 3.

Results. We trained our models (LSTM-RFC and GRU-RFC) on data from the year 2014 and used the data from 2015 and 2016 as a testing set. As a result, Table 9 shows that our approach of code authorship identification is resilient to temporal changes in the coding style of programmers as it achieves 100% accuracy for both Python and Java languages and 97.65% for the 292 C++ programmers.

5.5 Identification with Mixed Languages

Here, we investigate code authorship identification for programmers writing in multiple programming languages. In particular, in this section we attempt to answer the following question: *is it possible to identify programmers writing in multiple languages using one model trained with multiple languages*? Some programmers develop programming skills in multiple languages and use the preferable one based on the problem or the job at hand. To this end, we attempt to understand whether learning about a programmers' style in multiple languages without recognizing languages contributes in identifying the programmer given codes written in multiple languages. Despite the natural appeal to this problem and its associated research questions, there is no prior research work on this problem. Thus, we proceed to understand the potential of identification for multiple languages using our approach.

Table 9: The accuracy of authorship identification for programmers who solved seven problems from three different years (2014–2016). The identification models were trained on data from 2014 and tested on data from 2015 and 2016

	# Authors	LSTM-RFC	GRU-RFC
C++	292	97.65	96.43
Python	44	100	100
Java	50	100	100

Experiment 1. We use dataset 3 (Table 4), which corresponds to authors with nine files (selected randomly) written in multiple programming languages across all years. For training, we fed code files in two languages without letting it know the languages (thus, the training process is oblivious to the language itself). For testing, we also fed code files to the system without indicating what language they are written in (thus, the testing process is oblivious to the language too). Therefore, we aim to demonstrate that our system is language-oblivious even under this (stronger) mixed model.

Results. Figure 8 shows the accuracy of our approach with three datasets: C++/C, C++/Java, and C++/Python. Figure 8(a) shows an accuracy of 96.34% for a dataset of 626 C++/C programmers with LSTM-RFC, and its accuracy of 97.52% when used with LSTM-RFC on 855 C++/Java programmers, as illustrated in Figure 8(b). For the C++/Python dataset, Figure 8(c) shows that our approach provides an accuracy of 97.49% for 1,879 programmers.

Key Insight. The reported test accuracy follows a stratified cross-validation, where every code file has been tested and contributed to the reported accuracy by being used in building the model. Therefore, the model is tested to identify programmers based on code samples written in a language that might not be present in the training data. This experiment shows that our approach is oblivious to language specifics. Addressing a dataset of authors writing in multiple languages, our deep learning architecture is able to extract high quality and more distinctive features, preserving code authorship attributions through different programming languages.

Another observation is the non-monotonic results achieved using LSTM-RFC and GRU-RFC when extending the number of included authors in the dataset. As both models are parametric models, their performance depends on finding the best parameters within a fixed number of training iterations. Thus, the random initialization at the beginning might help the model converge to better results faster than the other (if at all). The non-monotonic results (1-2% difference) are explained by this optimality and convergence in independent runs with the fixed iterations.

Experiment 2. Another experiment was conducted to show the capability of our approach in identifying authors where the identification features are entirely extracted from a different programming language. The aim of this experiment is to answer the following question: *Given samples of code written by programmers in one language (e.g., C++), is it possible to identify those programmers when writing in a different language (e.g., C)*? From the 1,897 programmers who used C++ and C in dataset 3 (Table 4), we extracted a dataset of 224 programmers, where 70% of the samples per author are written in C++ while the remaining 30% are written in the C language. Using our approach, we trained an LSTM-RFC using the 70% of samples written by the 224 programmers in C++ and tested the LSTM-RFC model on the remaining 30% of C samples. As a result, our approach achieved 90.29% of accuracy for identifying programmers with features extracted from code written by them in a different programming language, highlighting its language-agnostic identification capabilities.

5.6 Identification in Obfuscated Domain

The basic assumption for the operation of our approach is that TF-IDF can be extracted from the original source code, presumably from an unobfuscated code. As such, one potential way to defeat our approach of authorship analysis (e.g., in a malware attribution application) is to obfuscate the code. In such a scenario, the underlying model would be built (in the training phase) using a certain dataset, and in the actual operation an obfuscated file would be presented to the model for identification. Our approach, if implemented in a straightforward manner, would possibly fail to address this circumvention technique. Thus, our question is if the model is trained with obfuscated codes, will it be able to identify authors correctly if obfuscated codes are presented for testing?

Assumption. We examine how obfuscation affects our approach, and whether it would be possible to still get attribution on obfuscated files for testing obfuscated files. This requires the assumption that we know what obfuscation technique was used, and we transform the training set before building the model, which is the clear limitaion of our approach. Deciding what obfuscation technique is used is out of scope of this paper, but every obfuscation tools have a unique technique to amplify obfuscation effect, which would be a hint to find the obfuscator.

Obfuscation Tools. Different obfuscation tools are available, and two among them were chosen to evaluate our approach: Stunnix [2] and Tigress [3]. The main reason for choosing these two obfuscation tools is because each represents a different approach for code-to-code obfuscation. Stunnix is a popular off-the-shelf C/C++ obfuscator that gives code a cryptic look while preserving its functionality and structure. Tigress, on the other hand, is a more sophisticated obfuscator for the C language; it implements function virtualization by converting the original code into an unreadable bytecode. For our experiment on code authorship identification of Tigress-obfuscated code, we turned on all of the features of Tigress.

Experiment 1: Stunnix. The first experiment is targeted towards a C++ dataset of 120 authors with nine source code files obfuscated using Stunnix. Our approach was able to reach 98.9% accuracy on the entire obfuscated dataset of 120 authors and 100% accuracy on an obfuscated dataset of 20 authors. Figure 9(a) shows the accuracy achieved using our approach on different Stunnix-obfuscated C++ datasets ranging from 20 to 120 authors using two different RNN units. The result of this experiment indicates that our approach is robust and resistant to off-the-shelf obfuscator.

Experiment 2: Tigress. We use a C dataset of 120 authors with nine source files each, obfuscated using Tigress. Even with this sophisticated obfuscator, our approach achieves 93.42% on the entire dataset while maintaining an accuracy of over 98% on a subset of 20 authors. Figure 9(b) shows the achieved accuracy on different Tigress-obfuscated C datasets ranging from 20 to 120 authors using two different RNN units. The results also indicate the resilience of our approach to sophisticated obfuscators such as Tigress. Despite the unreadability of the obfuscated code using Tigress, which makes such obfuscated code unreadable, the result of our experiment highlights that code files are no longer unidentifiable.

5.7 Identification with a Real-world Dataset

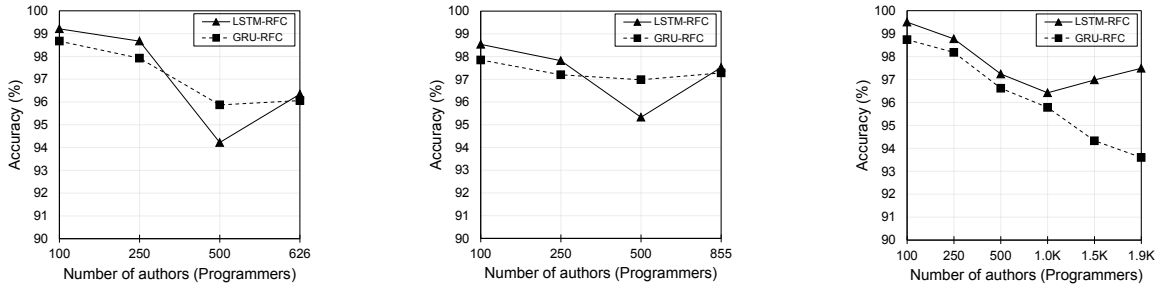
In this section, we examine our approach on real dataset of source code samples collected from GitHub. The collected dataset includes code samples from 1987 public repositories on GitHub, which list C and C++ as the primary language written by one contributor. After processing the repositories and removing incomplete data, the collected C++ and C datasets included 142 and 745 programmers, respectively, with at least five code samples each. Since some authors have more than 10 samples, we have randomly selected 10 samples per author. For the ground truth of our dataset, we collected repositories with a single contributor under the assumption that the collected samples are written by the same contributor of the repository. We acknowledge that this assumption is not always valid, because parts of the code samples might have been copied from other sources [21]. Even under those acknowledged limitations of the ground truth, our evaluation is still conservative with the respect to the end results: it attempts to distinguish between code samples that *may even include reused codes across samples*.

Results. Figure 10 shows the results of our approach using GitHub C++ and C datasets. Figure 10(a) shows an accuracy of 100% when using LSTM-RFC for 50 C++ programmers and 95.21% for 147 programmers. Figure 10(b) shows an accuracy of 94.38% for 745 C programmers using LSTM-RFC. This result shows that our approach is still effective when handling a real-world dataset.

Key Insight. The reported results using the GitHub dataset show some accuracy degradation in comparison with the results obtained using GCJ dataset given the same number of programmers. This degradation in the accuracy might be because of the authenticity of the dataset ground truth. The assumption behind establishing the ground truth for our dataset is only true to some extent since the contributor of a GitHub repository could copy code segments or even code files from other sources. Such ground truth problem influences the result of the authorship identification process. In real-world applications, this problem does not occur much often since most scenarios entails having authentic dataset.

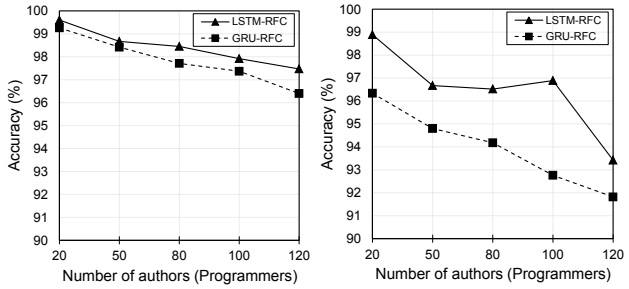
6 LIMITATIONS

While our work provides a high accuracy of code author identification across languages, it has several shortcomings which we outline in the following.



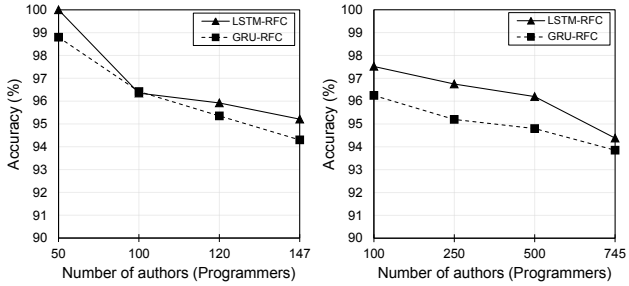
(a) C/C++ authors (b) Java/C++ authors (c) Python/C++ authors.

Figure 8: The accuracy of the authorship identification of programmers with sample codes of two programming languages.



(a) C++ obfuscated with Stunnix. (b) C code obfuscated with Tigress.

Figure 9: The accuracy of authorship identification with obfuscated source code, showing promising results even with the more sophisticated obfuscation approach (Tigress).



(a) GitHub C++ dataset. (b) GitHub C dataset.

Figure 10: The accuracy of the authorship identification of programmers using GitHub dataset, showing promising results even with real-world code samples.

Multiple authors. All experiments in this work are conducted under the assumption that a single programmer is involved in each source code sample. One shortcoming of our work is that this assumption does not always hold in reality, since large software projects are often the result of collaborative work and team efforts. The involvement of multiple authors in a single source code is almost inevitable with the increasing use of open development platforms. Using our approach to identify multiple authors can be an interesting direction for future work.

Authorship confusion. Since this work adopts a machine learning approach to identify programmers, it will only succeed if similar patterns from the training data are captured in the test dataset. As a pathological case, consider the *authorship confusion attack* or *mimicry attack* where the tested samples are contaminated to evade identification. Such contamination in the code could cause substantial changes of the programming style, thus making it difficult (if not impossible) to correctly identify the involved programmer.

Code size. The experiments in this work are conducted using datasets of source code samples that exhibit sufficient information (i.e., adequate average lines of code) to formulate distinctive authorship attribution for programmers. However, we have not investigated the minimal average lines of code to be considered as sufficient to distinguish programmers. For example, one could imagine that even though a small sample of code (e.g., with less than 10 lines of code) can present enough information to correctly identify the programmer, it is difficult to generalize this observation broadly. Investigating the sufficient code size to identify programmers is not examined in this work, and is an interesting future direction.

7 CONCLUSION AND FUTURE WORK

This work contributes to the extension of deep learning applications by utilizing deep representations in authorship attribution. In particular, we examined the learning process of large-scale code authorship attribution using RNN, a more efficient and resilient approach to language-specifics, number of code files available per author, and code obfuscation. Our approach extended authorship identification to cover the entire GCJ dataset across all years (2008 to 2016) in four programming languages (C, C++, Java and Python). Our experiments showed that the proposed approach is robust and scalable, and achieves high accuracy in various settings. We demonstrated that deep learning can identify more distinctive features from less distinctive ones. More distinctive features are more likely to be invariant to local changes of source code samples, which means that they potentially possess greater predictive power and enable large-scale code identification. One of the most challenging problems that authorship analysis confronts is the reuse of code, where programmers reuse others' codes, write programs as a team, and when a specific format is enforced by the work environment or by code formatters in the development environment. In the future, we will explore how code reuse affects the performance of our approach, and code authorship identification in general.

Acknowledgement. We would like to thank the anonymous reviewers and our shepherd, Apu Kapadia, for their feedback and helpful comments. This research was supported by the Global Research Lab. (GRL) Program of the National Research Foundation

(NRF) funded by the Ministry of Science, ICT and Future Planning (NRF-2016K1A1A2912757). D. Nyang is the corresponding author.

REFERENCES

[1] 2016. Google Code Jam. https://code.google.com/codejam/. (Accessed on 15/02/2017).

[2] 2017. Stunnix. http://stunnix.com/. (Accessed on 15/02/2017).

[3] 2017. The tigress diversifying c virtualizer. http://tigress.cs.arizona.edu. (Accessed on 15/02/2017).

[4] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *CoRR* abs/1603.04467 (2016). http://arxiv.org/abs/1603.04467

[5] Ahmed Abbasi and Hsinchun Chen. 2008. Writeprints: A stylometric approach to identity-level identification and similarity detection in cyberspace. *ACM Transactions on Information Systems (TOIS)* 26, 2 (2008), 7.

[6] Sadia Afroz, Aylin Caliskan Islam, Ariel Stolerman, Rachel Greenstadt, and Damon McCoy. 2014. Doppelgänger finder: Taking stylometry to the underground. In *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 212–226.

[7] Saed Alrabaee, Noman Saleem, Stere Preda, Lingyu Wang, and Mourad Debbabi. 2014. Oba2: An onion approach to binary code authorship attribution. *Digital Investigation* 11 (2014), S94–S103.

[8] Alexander T Basilevsky. 2009. *Statistical factor analysis and related methods: theory and applications*. Vol. 418. John Wiley & Sons.

[9] Yoshua Bengio. 2009. Learning Deep Architectures for AI. *Found. Trends Mach. Learn.* 2, 1 (Jan. 2009), 1–127. https://doi.org/10.1561/2200000006

[10] Yoshua Bengio, Aaron Courville, and Pascal Vincent. 2013. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence* 35, 8 (2013), 1798–1828.

[11] Yoshua Bengio, Aaron C. Courville, and Pascal Vincent. 2012. Unsupervised Feature Learning and Deep Learning: A Review and New Perspectives. *CoRR* (2012). arXiv:1206.5538 http://arxiv.org/abs/1206.5538

[12] Leo Breiman. 2001. Random Forests. *Machine Learning* 45, 1 (2001), 5–32.

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

[14] Steven Burrows and S. M. M. Tahaghoghi. 2007. Source code authorship attribution using n-grams. In *Proceedings of the Twelfth Australasian Document Computing Symposium (ADCS'07)*. Spink A, Turpin A, Wu M (eds), 32–39.

[15] Steven Burrows, S. M. M. Tahaghoghi, and Justin Zobel. 2007. Efficient Plagiarism Detection for Large Code Repositories. *Softw. Pract. Exper.* 37, 2 (Feb. 2007), 151–175. https://doi.org/10.1002/spe.v37:2

[16] Steven Burrows, Alexandra L. Uitdenbogerd, and Andrew Turpin. 2009. Application of Information Retrieval Techniques for Source Code Authorship Attribution. In *Proceedings of the 14th International Conference on Database Systems for Advanced Applications (DASFAA '09)*. Springer-Verlag, Berlin, Heidelberg, 699–713. https://doi.org/10.1007/978-3-642-00887-0_61

[17] S. Burrows, A. L. Uitdenbogerd, and A. Turpin. 2009. Temporally Robust Software Features for Authorship Attribution. In *2009 33rd Annual IEEE International Computer Software and Applications Conference*, Vol. 1. 599–606.

[18] Steven Burrows, Alexandra L. Uitdenbogerd, and Andrew Turpin. 2014. Comparing techniques for authorship attribution of source code. *Software: Practice and Experience* 44, 1 (2014), 1–32. https://doi.org/10.1002/spe.2146

[19] Aylin Caliskan-Islam, Richard Harang, Andrew Liu, Arvind Narayanan, Clare Voss, Fabian Yamaguchi, and Rachel Greenstadt. 2015. De-anonymizing Programmers via Code Stylometry. In *Proceedings of the 24th USENIX Conference on Security Symposium (SEC'15)*. USENIX Association, Berkeley, CA, USA, 255–270. http://dl.acm.org/citation.cfm?id=2831143.2831160

[20] Aylin Caliskan-Islam, Fabian Yamaguchi, Edwin Dauber, Richard Harang, Konrad Rieck, Rachel Greenstadt, and Arvind Narayanan. 2015. When coding style survives compilation: De-anonymizing programmers from executable binaries. *arXiv preprint arXiv:1512.08546* (2015).

[21] Edwin Dauber, Aylin Caliskan Islam, Richard E. Harang, and Rachel Greenstadt. 2017. Git Blame Who?: Stylistic Authorship Attribution of Small, Incomplete Source Code Fragments. *CoRR* abs/1701.05681 (2017). arXiv:1701.05681

[22] Haibiao Ding and Mansur H. Samadzadeh. 2004. Extraction of Java program fingerprints for software authorship identification. *Journal of Systems and Software* 72, 1 (2004), 49 – 57. https://doi.org/10.1016/S0164-1212(03)00049-9

[23] John Duchi, Elad Hazan, and Yoram Singer. 2011. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *J. Mach. Learn. Res.* 12 (July 2011), 2121–2159.

[24] Bruce S. Elenbogen and Naeem Seliya. 2008. Detecting Outsourced Student Programming Assignments. *J. Comput. Sci. Coll.* 23, 3 (Jan. 2008), 50–57. http://dl.acm.org/citation.cfm?id=1295109.1295123

[25] Brian S Everitt and Graham Dunn. 2001. *Applied multivariate data analysis*. Vol. 2. Wiley Online Library.

[26] Georgia Frantzeskou, Efstathios Stamatatos, Stefanos Gritzalis, Carole E Chaski, and Blake Stephen Howald. 2007. Identifying authorship by byte-level n-grams: The source code author profile (scap) method. *International Journal of Digital Evidence* 6, 1 (2007), 1–18.

[27] Georgia Frantzeskou, Efstathios Stamatatos, Stefanos Gritzalis, and Sokratis Katsikas. 2006. Effective Identification of Source Code Authors Using Byte-level Information. In *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*. ACM, New York, NY, USA, 893–896. https://doi.org/10.1145/1134285.1134445

[28] Niels Dalum Hansen, Christina Lioma, Birger Larsen, and Stephen Alstrup. 2014. Temporal Context for Authorship Attribution. In *Information Retrieval Facility Conference*. Springer, 22–40.

[29] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. 2006. A fast learning algorithm for deep belief nets. *Neural computation* 18, 7 (2006), 1527–1554.

[30] Patrick Juola et al. 2008. Authorship attribution. *Foundations and Trends® in Information Retrieval* 1, 3 (2008), 233–334.

[31] Vlado Kešelj, Fuchun Peng, Nick Cercone, and Calvin Thomas. 2003. N-gram-based author profiles for authorship attribution. In *Proceedings of the conference pacific association for computational linguistics, PACLING*, Vol. 3. 255–264.

[32] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. *CoRR* abs/1412.6980 (2014).

[33] Ron Kohavi et al. 1995. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *The International Joint Conference on Artificial Intelligence*, Vol. 14. Stanford, CA, 1137–1145.

[34] Moshe Koppel, Jonathan Schler, and Shlomo Argamon. 2009. Computational methods in authorship attribution. *Journal of the Association for Information Science and Technology* 60, 1 (2009), 9–26.

[35] Ivan Krsul and Eugene H. Spafford. 1997. Refereed Paper: Authorship Analysis: Identifying the Author of a Program. *Comput. Secur.* 16, 3 (Jan. 1997), 233–257.

[36] Robert Charles Lange and Spiros Mancoridis. 2007. Using Code Metric Histograms and Genetic Algorithms to Perform Author Identification for Software Forensics. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation (GECCO '07)*. ACM, New York, NY, USA, 2082–2089. https://doi.org/10.1145/1276958.1277364

[37] S. G. Macdonell, A. R. Gray, G. MacLennan, and P. J. Sallis. 1999. Software forensics for discriminating between program authors using case-based reasoning, feedforward neural networks and multiple discriminant analysis. In *Neural Information Processing, 1999. Proceedings. ICONIP '99. 6th International Conference on*, Vol. 1. 66–71 vol.1. https://doi.org/10.1109/ICONIP.1999.843963

[38] Cameron H Malin, Eoghan Casey, and James M Aquilina. 2008. *Malware forensics: investigating and analyzing malicious code*. Syngress.

[39] Xiaozhu Meng, Barton P Miller, and Kwang-Sung Jun. 2017. Identifying Multiple Authors in a Binary Program. In *European Symposium on Research in Computer Security*. Springer, Oslo, Norway, 286–304.

[40] Brian N. Pellin. 2000. Using Classification Techniques to Determine Source Code Authorship. *White Paper:* Department of Computer Science, University of Wisconsin (2000).

[41] Nathan Rosenblum, Xiaojin Zhu, and Barton Miller. 2011. Who wrote this code? identifying the authors of program binaries. *Computer Security–ESORICS 2011* (2011), 172–189.

[42] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. 2015. Recognizing Functions in Binaries with Neural Networks. In *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C., 611–626.

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

[44] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *J. Mach. Learn. Res.* 15, 1 (Jan. 2014), 1929–1958.

[45] Efstathios Stamatatos. 2009. A survey of modern authorship attribution methods. *Journal of the Association for Information Science and Technology* 60, 3 (2009), 538–556.

[46] Ariel Stolerman, Rebekah Overdorf, Sadia Afroz, and Rachel Greenstadt. 2013. Classify, but verify: Breaking the closed-world assumption in stylometric authorship attribution. In *IFIP Working Group*, Vol. 11. 64.

[47] Tijmen Tieleman and Geoffrey Hinton. 2012. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning* 4, 2 (2012).

[48] Özlem Uzuner and Boris Katz. 2005. A comparative study of language models for book and author recognition. In *International Conference on Natural Language Processing*. Springer, 969–980.

[49] Linda J Wilcox. 1998. Authorship: the coin of the realm, the source of complaints. *The Journal of the American Medical Association* 280, 3 (1998), 216–217.