Large-scale and Robust Code Authorship Identification with Deep Feature Learning

MOHAMMED ABUHAMAD, Loyola University Chicago
TAMER ABUHMED, Sungkyunkwan University
DAVID MOHAISEN, University of Central Florida
DAEHUN NYANG, Ewha Womans University

Successful software authorship de-anonymization has both software forensics applications and privacy implications. However, the process requires an efficient extraction of authorship attributes. The extraction of such attributes is very challenging, due to various software code formats from executable binaries with different toolchain provenance to source code with different programming languages. Moreover, the quality of attributes is bounded by the availability of software samples to a certain number of samples per author and a specific size for software samples. To this end, this work proposes a deep Learning-based approach for software authorship attribution, that facilitates large-scale, format-independent, language-oblivious, and obfuscation-resilient software authorship identification. This proposed approach incorporates the process of learning deep authorship attribution using a recurrent neural network, and ensemble random forest classifier for scalability to de-anonymize programmers. Comprehensive experiments are conducted to evaluate the proposed approach over the entire Google Code Jam (GCJ) dataset across all years (from 2008 to 2016) and over real-world code samples from 1,987 public repositories on GitHub. The results of our work show high accuracy despite requiring a smaller number of samples per author. Experimenting with source-code, our approach allows us to identify 8,903 GCJ authors, the largest-scale dataset used by far, with an accuracy of 92.3%. Using the real-world dataset, we achieved an identification accuracy of 94.38% for 745 C programmers on GitHub. Moreover, the proposed approach 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. Experimenting with executable binaries, our approach achieves 95.74% for identifying 1,500 programmers of software binaries. Similar results were obtained when software binaries are generated with different compilation options, optimization levels, and removing of symbol information. Moreover, our approach achieves 93.86% for identifying 1,500 programmers of obfuscated binaries using all features adopted in Obfuscator-LLVM tool.

CCS Concepts: • **Security and privacy** → *Software and application security;*

An earlier version of this work has appeared in ACM CCS 2018 [9].
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).
Authors' addresses: M. Abuhamad, Loyola University Chicago; email: mabuhamad@luc.edu; T. AbuHmed, Sungkyunkwan University, Seoul, South Korea; email: tamer@skku.edu; D. Mohaisen, University of Central Florida IL, USA; email: mohaisen@ucf.edu; D. Nyang (corresponding author), Ewha Womans University, Seoul, South Korea; email: nyang@ewha.ac.kr.

2471-2566/2021/07-ART23 $15.00
https://doi.org/10.1145/3461666

Additional Key Words and Phrases: Software authorship identification, program features, deep learning identification, software forensics

ACM Reference format:
Mohammed Abuhamad, Tamer AbuHmed, David Mohaisen, and DaeHun Nyang. 2021. Large-scale and Robust Code Authorship Identification with Deep Feature Learning. *ACM Trans. Priv. Secur.* 24, 4, Article 23 (July 2021), 35 pages.
https://doi.org/10.1145/3461666

1 INTRODUCTION

Authorship identification of natural language text is a well-known problem that has been studied extensively in the literature [46, 47, 50, 65]. However, far fewer works are dedicated to authorship identification in structured code, such as the source code of computer programs [27]. Software authorship identification is the process of software developer identification by associating a programmer to a given code based on the programmer's distinctive stylometric features. A code of software can be presented with the original source code or the executable binaries, which can be decompiled to generate pseudo-code as higher level construction of the binary instructions [29, 60]. The problem is, however, difficult and different from authorship identification of natural language text. This fundamental difficulty is due to the inherent inflexibility of the presented code expressions established either by the syntax rules of compilers or the reverse engineering of binaries.

Software authorship identification relies on extracting features from software code that a programmer produces based on the programmer's preferences in structuring and developing software pieces. Given these features, the main objective of software authorship identification is to correctly assign programmers to software codes based on the extracted features. Being able to identify software authors is both a risk and a desirable feature. On the one hand, software 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 software authors identification a de-anonymization problem. On the other hand, software 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 software is helpful with plagiarism detection [24], authorship disputes [69], copyright infringement [39], and software integrity investigations [55].

The problem of software author identification is challenging and faces several obstacles that prevent the development of practical identification mechanisms. In the case of source code authorship identification, 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 [26]. 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. In the case of binary code authorship identification, first, the toolchain provenance used to produce the binaries must be identified, since numerous resultant binaries for the same program can be generated by using different compilation processes. Second, most software binaries, especially malicious programs, are obfuscated making it difficult to be analyzed for authorship.

To address those challenges, recent attention to software authorship identification has revived more than two-decade old work [51, 63] by proposing several techniques [27, 28]. However, there are several limitations to 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 software 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 [35].

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 software 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 program samples per author? (iii) Can deep learning help identify authorship attributes that go beyond language specifics in an efficient way and without requiring prior knowledge of the language? (iv) Will deep authorship representation still be robust when the program is obfuscated? (v) Can deep authorship representation help identify authors of executable binaries? and (vi) Will deep authorship representation still be robust when different toolchain provenance is used to generate program binaries?

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 program 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 software 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 **Term Frequency-Inverse Document Frequency (TF-IDF)** that is already a well-known tool for textual data analysis [23, 40, 47]. Thus, our approach does not require prior knowledge of any specific programming language or high-level translations of program binaries, thus it is more resilient to language specifics when the source code is available and more robust to compilation settings when the target code is in binary format. When conducting experiments on large-scale source code dataset, 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 [28], 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 Reference [28]. 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 applicability of our approach to identify programmers from executables. Several previous works have shown that authorship attribution can be extracted from executable binaries, and identifying programmers of software binaries is possible [29, 56, 61]. In this work, we examine our approach on capturing authorship traits from high-level translations of binaries generated by simple straightforward reverse engineering process. The proposed approach achieves an accuracy of 98.4% and 95.74% for identifying 250 and 1,500 programmers of software binaries, respectively. We extend our experiments and analysis to examine the effects of different compilation settings such as levels of optimization and removal of symbol information in stripped binaries.

Fifth, 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 [1], and more sophisticated obfuscators, such as Tigress [6] 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 Reference [28].

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 article 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 software authorship identification. We proceed with a detailed overview of the experimental results of our approach using the software source code in Section 5 and using the binary code in Section 6. Section 7 shows the experimental results of authorship attribution of obfuscated software. In Section 8, we address authorship identification of software in real-world scenarios. Finally, the limitations of this work are outlined in Section 9, followed by concluding remarks in Section 10.

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 anonymously written documents with their authors were motivated by the interest of settling the authorship of disputed works, such as *The Federalist Papers*. Since the early 2000s, 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 micro-blogs entries. Abbasi and Chen [8] 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 [68] provided a comparative study of different stylometry methods used for authorship attribution and identification. Afroz et al. [12] investigated the possibility of identifying cybercriminals by analyzing their textual entries in underground forums, even when they

Table 1. Comparison between Our Work Using Deep Learning for Authorship Identification and Various Related Works from the Literature over the Used Classification Techniques, Used Languages, and Approaches

Authorship identification based on source code				
Reference	# Authors	Languages	Accuracy (%)	Classification Technique
Pellin [58]	2	Java	88.47%	Machine learning (SVM with tree kernel)
MacDonell et al. [54]	7	C++	81.10%	Machine learning (FFNN). Statistical analysis (MDA)
MacDonell et al. [54]	7	C++	88.00%	Machine learning (case-based reasoning).
Frantzeskou et al. [40]	8	C++	100.00%	Rank similarity measurements (KNN)
Burrows et al. [25]	10	C	76.78%	Information retrieval using mean reciprocal ranking
Elenbogen & Seliya [37]	12	C++	74.70%	Statistical analysis (decision tree model)
Lange & Mancoridis [52]	20	Java	55.00%	Rank similarity measurements (nearest neighbor)
Krsul & Spafford [51]	29	C	73.00%	Statistical analysis (discriminant analysis)
Frantzeskou et al. [40]	30	C++	96.90%	Rank similarity measurements (KNN)
Ding & Samadzadeh [35]	46	Java	62.70%	Statistical analysis (canonical discriminant analysis)
Burrows et al. [27]	100	C, C++	79.90%	Machine learning (neural network classifier)
Burrows et al. [27]	100	C, C++	80.37%	Machine learning (SVM)
Caliskan-Islam et al. [28]	229	Python	53.91%	Machine learning (random forest)
Caliskan-Islam et al. [28]	1,600	C++	92.83%	Machine learning (random forest)
Abuhamad et al. [11]	1,600	C++	96.2%	Machine learning (CNN)
Abuhamad et al. [11]	1,500	Python	94.6%	Machine learning (CNN)
Abuhamad et al. [11]	1,000	Java	95.8%	Machine learning (CNN)
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)
Authorship identification based on binary code				
Reference	# Authors	Languages	Accuracy (%)	Classification Technique
Rosenblum et al. [61]	10	Binary	81%	Machine learning (SVM)
Meng et al. [56]	284	Binary	65%	Machine learning (random forest)
Caliskan-Islam et al. [29]	600	Binary	83%	Machine learning (random forest)
This work	1,500	Binary	95.74%	Machine learning (RNN with random forest)

MDA, Multiple Discriminant Analysis; FFNN, Feed Forward Neural Network; RNN, Recurrent Neural Network; CNN, Convolutional Neural Network; KNN, K-Nearest Neighbor; SVM, Support Vector Machines. Results are excerpted from references.

use multiple identities. Stolerman et al. [66] 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 identities. Brennan et al. [22] studied three adversarial settings to circumvent authorship identification: obfuscation, imitation, and translation.

2.1 Authorship Attribution of Source Code

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 research works for more than two decades. Designing authorship attributions that reflect programmersâ.. stylistic characteristics has been investigated by multiple works, since the early work of Krsul et al. [51]. 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 [40], or as complex as control and data flow graphs [13, 56, 61] or even abstract syntax tree features [28, 58]. 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 [9, 10]. Krsul and Spafford [51] 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. [54] 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. [40] 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 [52] 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 [37] considered six features to establish programmersâ.. profiles based on personal experience and heuristic knowledge: the number of comments, lines of code, variablesâ.. count and name length, the use of for-loop, and program compression size. Burrows et al. [25] used a combination of *n*-gram and stylistic characteristics of programmers for authorship identification. Most recently, Caliskan-Islam et al. [28] 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 to classify code samples [27]: Statistical analysis methods are considered for limiting the feature space to discover highly-indicative features of authorship. Krsul and Spafford [51], MacDonell et al. [54], and Ding and Samadzadeh [35] used discriminant analysis for identifying authors. As for machine learning, various approaches are used for source code authorship identification: case-based reasoning [54], neural networks [27, 54], decision trees [37], support vector machine [27, 58], and random forest [28]. As a general approach of similarity measurement, a ranking approach based on similarity measurements can be used to compute the distance between a test instance and candidate instances in the feature space. Using a k-nearest neighbor approach is one way to assign instances to authors with similar instances. Lange and Mancoridis [52], Frantzeskou et al. [40], and Burrows et al. [25] implemented different ranking methods based on similarity measurements.

2.2 Authorship Attribution of Binary Code

Code authorship identification could also be done at the binary level, which is addressed separately in the literature. Binary-level techniques [13, 29, 56, 61] are advocated as a viable tool for software attribution where the source code is not available [56].

Rosenblum et al. [61] explored authorship attributions of program binary code in two tasks, authorship identification, and authorship clustering. The authors extracted a large number of authorship stylistic features from software binary code to enable attributing programmers efficiently

to identify them or categorize them based on extracted features. These features include n-grams, idioms, graphlets, supergraphlets, call graphlets, and library calls. Using these features, Rosenblum et al. [61] achieved 81% accuracy for identifying ten programmers and 51% for identifying almost 200 programmers. Alrabaee et al. [13] proposed a binary authorship identification method called *OBA2*, which extracts syntax-based and semantic-based features related to authorship.

Caliskan-Islam et al. [29] have introduced a different approach to extract authorship attribution from binary code by using simple straightforward reverse engineering process to obtain higher translations of program binary code. Using code stylometry features extracted from decompiled pseudo-code, their method achieved an accuracy of 96% for identifying 100 programmers and an accuracy of 83% for 600 programmers. Using the approach introduced by Caliskan-Islam et al. [29], the authors provided evidence that authorship stylometry features survive the compilation process and it is possible to identify programmers of executable binaries even if the binary codes were generated by compilation process included optimization and/or stripping of symbol information.

Meng et al. [56] explored the possibility of identifying multiple authors of binary code. The authors introduced a fine-grained approach to identify authors on the basic-block level. Meng et al. [56] evaluated their approach using real-world projects to achieve an accuracy of 65% for identifying 284 programmers as the first guess and accuracy of 82% when the correct author is from the top 10 suspects.

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, however, are more flexible and equally useful, especially in addressing incomplete pieces of code (which cannot be compiled). Even when operating on binaries, codelike artifacts are what is being actually analyzed. For example, Caliskan-Islam et al. [29] 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. In our experiments on identifying the programmers of executable binaries, we adopt the approach of Reference [29] by analyzing the decompiled code for authorship using deep learning.

3 BACKGROUND AND MOTIVATION

In this work, the analysis for software authorship attribution is done on source code or codelike artifacts extracted from executable binaries using a reverse engineering process. Authorship attributions are extracted from code files using a two-step process, i.e., TF-IDF as initial representation and then deep authorship representation using deep learning method. The extracted authorship attributions enable the identification of programmers using ensemble classifier. This section highlights the motivation and the underlying concepts of different used methods in our proposed system for software authorship attribution and identification.

3.1 Term Frequency-Inverse Document Frequency

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 that 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 **Inverse Document**

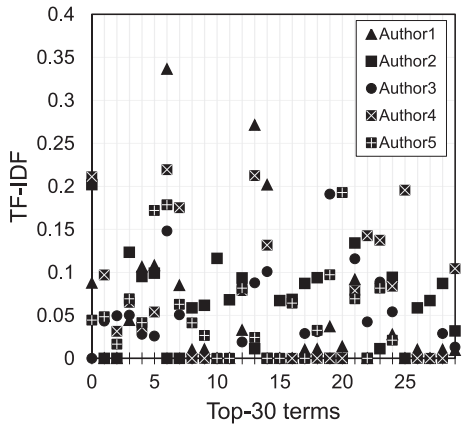


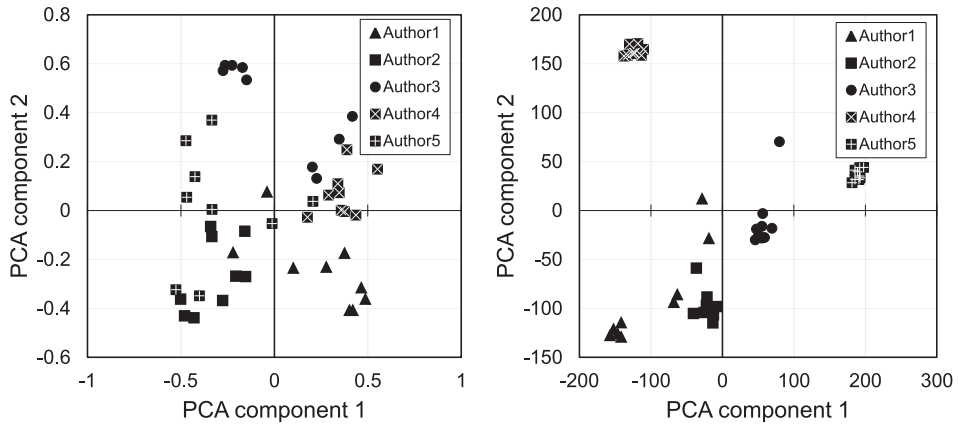
Fig. 1. The TF-IDF values of top-30 terms for five programmers. The value of a term is different among authors who use the same term. The terms are as follows: ("ans," "begin," "begin end," "bool," "break," "char," "cin," "cin int," "cmath," "cmath include," "const," "const int," "continue," "cout,"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

Frequency (IDF) $(t, \mathcal{D}) = \log(|\mathcal{D}|/\mathrm{DF}(t, \mathcal{D})) + 1$, where $|\mathcal{D}|$ is the number of documents in $\mathcal{D}$ and $\mathrm{DF}(t, \mathcal{D})$ is the number of documents containing the term t.

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 [23, 40, 47]. 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

Software 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 [17]. 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



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

Fig. 2. The PCA visualization of TF-IDF and deep representation of software attributions for five programmers.

community and has become a field in and of itself dedicated to enabling easier and more distinctive feature extraction processes [18]. 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 [20, 33, 43], object recognition [32, 59], natural language processing [15, 19], and multi-task and transfer learning [16, 41]. Since the breakthrough work of Hinton et al. [44], 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 References [16, 17]. 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 article are the **Long Short-Term Memory (LSTM)** and the **Gated Recurrent Units (GRU)**, which are two popular configurations of the RNN, one type of **Deep Neural Networks (DNN)**.

Deep LSTMs and GRUs [62] with multiple layers demonstrated remarkable capabilities in generating representations from long input sequences in other applications. This work investigates both LSTM's and GRU's capabilities of extracting software authorship attributions from TF-IDF code representations. Both of those approaches are a good fit for our defined problem, because they scale well, compared to alternatives; elaborate on this investigation in Section 4.2. We note that even though RNNs are particularly popular for their use to process and model sequential inputs, there are many applications of RNNs' variants on non-sequential data [30, 31, 53, 57]. In our work, the TF-IDF representations are fed into our deep neural network architecture (representation learning module) as one sequence per software sample to generate high-quality representations that will enable an 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 samples 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 [14, 38]. Figure 2 shows PCA visualizations of code samples for five programmers with nine samples each. In Figure 2(a), code samples are represented with the 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 deep learning techniques.

3.3 RFC over Deep Representations

To identify authors from feature representations of their code at scale, 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 [21] 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 DEEP LEARNING-BASED CODE AUTHORSHIP IDENTIFICATION SYSTEM

Our approach for large-scale software 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. Based on the available code format, the preprocessing phase aims to define the target code to be analyzed. For the source code of different programming languages, the preprocessing phase entails cleaning and preparing the code samples for the initial TF-IDF representations. On the other hand, for executable binary code, the preprocessing phase includes defining the toolchain provenance such as compiler family and version, compilation optimization level, and source code language, and so on. After defining the toolchain provenance of the presented binary code, a reverse engineering process takes place to obtain pseudo-codes as the higher translation of the program binary code. These pseudo-codes are then analyzed for authorship attribution and represented with the TF-IDF 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.

4.1 Data Preprocessing

The first phase of the proposed system is to handle available software samples to ensure an efficient initial representation process. This process varies considering the available code format, i.e., source code is a subject to a different preprocessing phase compared to executable binary code. However, both source and binary code files are represented initially with TF-IDF features by the end of the preprocessing phase. Note that previous works, e.g., References [25, 28, 29, 51, 52, 54] have used TF-IDF or a variation TF-IDF as part of the feature extraction of authorship attributes. In this

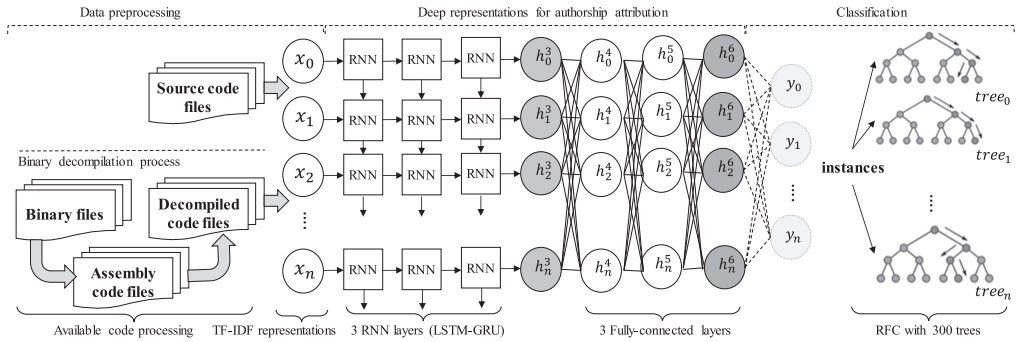


Fig. 3. A high-level illustration of the proposed deep learning-based software authorship identification system. This illustration shows the three phases of preprocessing (TF-IDF feature representation), better representation through learning (using the RNN and fully connected layers), and the classification (using 300 trees in a random forest classifier).

work, we only use TF-IDF representation as an initial representation for a deep learning model that is trained to extract more robust and distinctive authorship traits. The following describes the preprocessing phase with respect to the available code format.

Preprocessing for source code. The source code is processed to eliminate comments, copyright headers, program description, layout restrictions and features (such as tabs, spaces, and lines), and stop words. Since only n-grams are considered for the TF-IDF representation of code samples, the layout features and stop words are irrelevant, and are excluded from the initial representation. In this work, we conduct experiments on the source code of four programming languages, and by applying the same preprocessing steps. Moreover, obfuscated code using code-to-code obfuscation tools, e.g., Tigress or Stunnix, are treated as source code, and we follow the same procedure as with the source code from the different programming languages.

Preprocessing for binary code. When the available code is in a binary format, the first step is to identify the toolchain provenance such as compilation tools and settings. We assume the toolchain provenance of the presented binary codes is known, since the current state-of-the-art tools have this capability with a high degree of accuracy [60]. Being able to identify the source of the binary code, there are powerful tools to reverse engineer the binary code to higher level constructs via disassembly or/and decompilation process.

Both disassemblers and decompilers are capable of generating textual translations of binary code that can be an easier subject of analysis. On the one hand, disassemblers provide a straightforward one-to-one translation of binary instructions to instruction mnemonics. Among many powerful disassemblers available on the field, *radare2* [5] and *IDA-Pro* [4] are the most commonly utilized disassemblers with a wide-range of utilities. On the other hand, decompilers generate even higher-level translations of the binary code with concise C-like pseudocode. Compared to disassemblers, decompilers generate 5 to 10 times shorter outputs for the same binary program, e.g., typical binary program with size (400 KB–5 MB) can generate a decompiled code of size mostly less than 10 MB. Therefore, we use a decompilation process to generate high-level translations of software binaries. In our experiments on authorship attribution of executable binary code, we use Hex-Rays [3], a state-of-the-art commercial decompiler. The generated decompiled pseudo codes via a decompilation process are often larger in size than the original source code. However, we treat the generated pseudo code similarly as the source code in our analysis.

Both source and decompiled 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 single TF-IDF 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

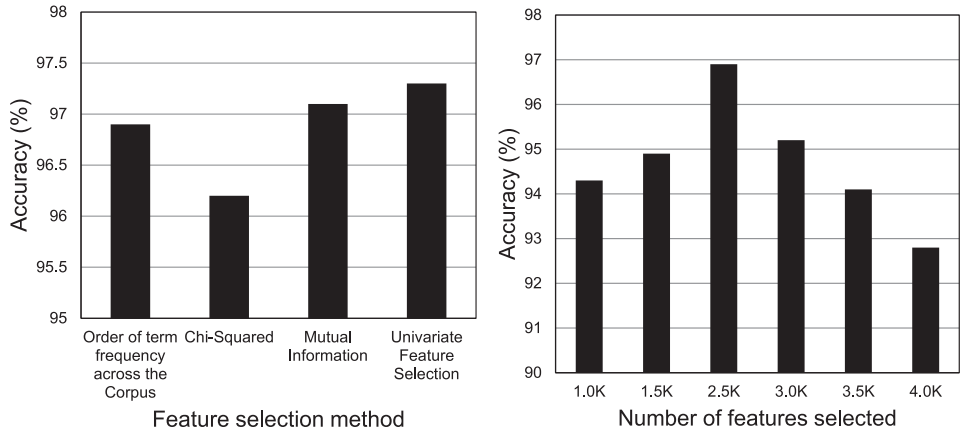
$$x_i = \bigcup_{j=1,\ldots,|\mathcal{D}|} \text{TF-IDF}(t_i, d_j, \mathcal{D}), \tag{1}$$

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 4.3). 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 [7], an open source symbolic math library for building and training neural networks using data flow graphs. We ran our experiments on a workstation with 24 cores, one GeForce GTX TITAN X GPU, and 256 GB of memory. We note



(a) Feature selection method.

(b) Number of features.

Fig. 4. Feature selection analysis.

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 [64]. 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 [64].

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 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 our deep learning model were chosen upon various iterations of experiments. We experimented with different architectures to explore the effects of the model design on its performance. Table 2 shows the results of using different models including simple feed-forward neural network, LSTM, and GRU. To explore the model design, we used a dataset of 250 C++ programmers with nine files each. The experiments were restricted to 500 training iterations and 1024 units for each adopted layer. The results show that LSTM and GRU improve the performance. We utilized RFC on top of the features learned through the deep learning architecture. The RFC is constructed using 100 trees grown to the maximum extent. Eventually, we chose an architecture with three RNN layers (i.e., LSTM or GRU) with a 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 the 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

Table 2. The Results of Different Model Architecture Designs

TF-IDF	FC	FC	FC	LSTM	LSTM	LSTM	GRU	GRU	GRU	FC	FC	FC	Sotmax	RFC
2500	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	78.62
2500	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	800	72.76	83.91
2500	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	1024	800	80.58	88.27
2500	✗	✗	✗	✗	✗	✗	✗	✗	✗	1024	1024	800	79.73	87.96
2500	1024	✗	✗	✗	✗	✗	✗	✗	✗	1024	1024	800	82.49	88.62
2500	1024	1024	✗	✗	✗	✗	✗	✗	✗	1024	1024	800	82.09	90.53
2500	1024	1024	1024	✗	✗	✗	✗	✗	✗	1024	1024	800	83.38	92.31
2500	✗	✗	✗	1024	✗	✗	✗	✗	✗	1024	1024	800	87.91	91.47
2500	✗	✗	✗	1024	1024	✗	✗	✗	✗	1024	1024	800	89.51	93.51
2500	✗	✗	✗	1024	1024	1024	✗	✗	✗	1024	1024	800	90.27	95.29
2500	✗	✗	✗	✗	✗	✗	1024	✗	✗	1024	1024	800	87.47	91.20
2500	✗	✗	✗	✗	✗	✗	1024	1024	✗	1024	1024	800	88.98	94.13
2500	✗	✗	✗	✗	✗	✗	1024	1024	1024	1024	1024	800	89.82	95.07

FC, fully connected layer; LSTM, Long-Short Term Memory; GRU, Gated Recurrent Unit; RFC, Random Forest Classifier. The results are obtained by a ninefold cross-validation process using a dataset of 250 C++ programmers. All deep learning models are trained for 500 iterations. The "✗" indicates excluding the layer, and the number indicates the layer size.

transformation that leads to the best probability of its correct class label. Targeting a large-scale code authorship identification problem with thousands of programmers, which translates to thousands of classes, the deep learning architecture alone does not accurately identify programmers. Thus, we use the output of the layer L_{k-1} (where the L_k is the softmax layer) to be the deep representations of code authorship features. Deep representations of the code authorship features are then subjected to a classification process using the RFC (Section 3.3), which is proven to be robust and scalable, accommodating large datasets. Table 2 shows that using the RFC on top of the deep representations improves the accuracy in all experimental settings. We note that, in our design we construct the RFC with 300 trees grown to the maximum extent to account for the scale of our dataset.

Training Procedure. The weights of the learning network were initialized using a normal distribution of small range near 0, a small variance, and mean of 0. To train our deep learning architecture, we used TensorFlow's **Adaptive Moment estimation (Adam)** [48] 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 [36], which is efficient for handling sparse gradients, and RMSProp [67], which is efficient for on-line and non-stationary settings [48].

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 [62]. 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 tradeoff between the model construction time and its accuracy [28].

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 in Reference [28], we report results of test accuracy using stratified k-fold cross-validation [49], where k depends on the number of observations per class in the dataset (i.e., ninefold used for nine files per author, sevenfold for seven 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 AUTHORSHIP IDENTIFICATION OF SOURCE CODE

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 [2]. 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 3. 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 3–5):

Table 3. 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	6,635	327	2,300	1,279
Across Years	7	8,903	566	3,458	1,952
Across Years	5	12,411	1,156	5,525	3,345
Average Lines of Code		71.53	65.20	44.44	86.70

Table 4. Two Datasets with the Corresponding Author Counts for Authors Who Had Seven Files at the GCJ 2015 and 2016 Competitions

Competition Year	Author Files	No. of Authors for Languages			
		C++	C	Python	Java
2015	7	2,241	41	398	132
2016	7	1,744	21	390	317
across 3 years*	7	292	NA	44	50
*Programmers participated in 2014, 2015 and 2016					

Table 5. A Dataset Used in Our Study to Demonstrate Identification across Multiple Languages

Competition Year	No. of Authors for Multiple Languages		
	C++-C	C++-Java	C++-Python
Across Years	1,897	855	626

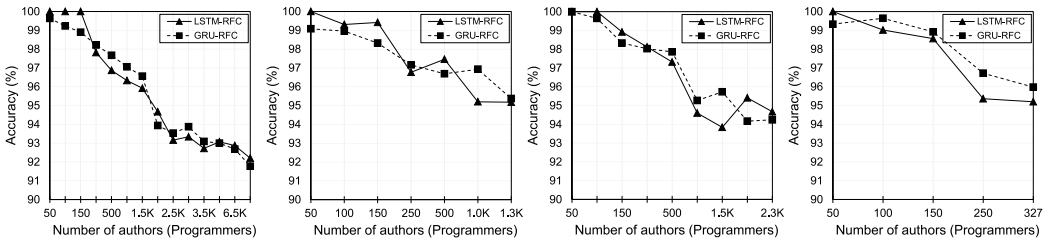
The dataset includes authors with nine files written in multiple languages.

Dataset 1: includes files across all years from 2008 to 2016 in a "cross-years" view, in Table 3.
Dataset 2: consists of code files for participants drawn from 2015 and 2016 competitions for four programming languages, as shown in Table 4.
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 5.

Number of Files. In Reference [28], 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 Section 5.3.

5.2 Large-scale Authorship Identification

Experiment. In this experiment, we used dataset 1 in Table 3. There are four large-scale datasets corresponding to four different programming languages with programmers who have exactly nine code files (first row in Table 3). The number of code files per author in this experiment was suggested in Reference [28] 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 ninefold 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.



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

Fig. 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.

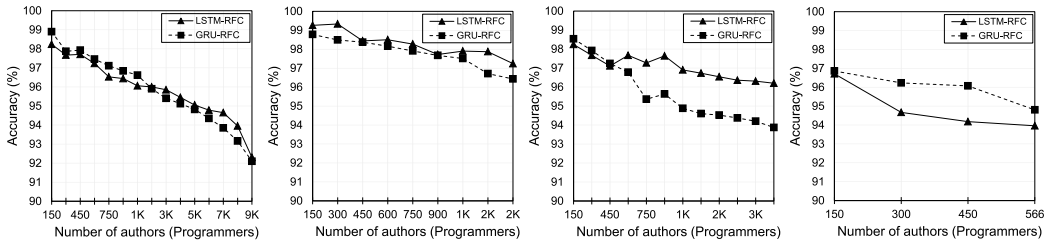
Evaluation Metric. For evaluation, we use the accuracy, defined as the percentage of code files correctly attributed over the total number of tested code files. Using the accuracy instead of other evaluation metrics (e.g., precision and recall) is enough, because the classes are balanced in terms of 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. 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 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 [26, 28].

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 3 (second row). We used RFC with stratified sevenfold 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 3). This experiment investigates the effects of having fewer files per author on the accuracy of our approach.



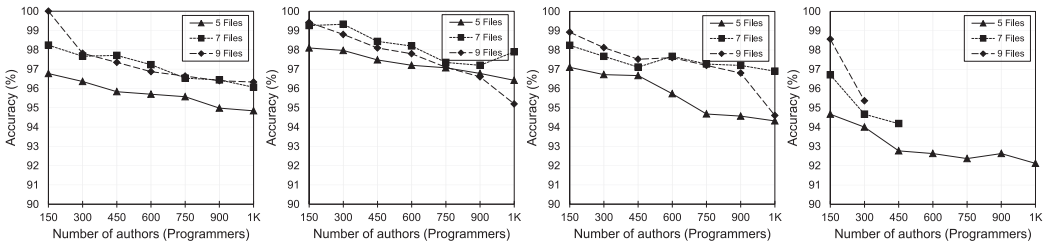
(a) C++ (7 files/programmer) (b) Java (7 files/programmer) (c) Python (7 files/programmer) (d) C (7 files/programmer)

Fig. 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.

Table 6. 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
	1,000	96.32	95.98
	1,500	95.88	95.22
	2,000	95.67	94.9
	2,241	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
	1,000	96.37	94.88
	1,500	95.66	94.14
	1,744	95.17	93.54

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 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.



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

Fig. 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 ~92% in the worst case.

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 3 (third row). We used RFC with stratified fivefold 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 3). This experiment further investigates the effects of having even fewer 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.

Comparison with Related Work. We conducted a comparison with the related work. Considering the work by Caliskan-Islam et al. [28] and Abuhamad et al. [11], Table 7 shows that our approach—using the LSTM architecture for deep feature representation followed by RFC for classification—outperforms other methods. For the work of Caliskan-Islam et al. [28], we selected the top-800 features by the information gain method to be comparable to our 800-dimensional

Table 7. A Comparison with Related Work on Source Code Authorship Identification

	C++			C		
	9 files (1000)	7 files (1000)	5 files (1000)	9 files (250)	7 files (500)	5 files (1000)
Caliskan-Islam et al. [28]	92.46	90.81	87.24	96.47	94.76	88.14
Abuhamad et al. [11] -TFIDF	86.24	84.19	82.82	91.64	89.54	86.42
Abuhamad et al. [11] -WE	81.67	78.43	72.91	89.61	84.92	71.90
This work	96.33	96.06	94.84	95.36	94.18	92.12

The results are reported using k-fold cross-validation method for a different number of files per author for a (total number of authors).

feature vector. The RFC model was constructed with 300 trees grown to the maximum extent. For the work of Abuhamad et al. [11], we used the stacked CNN architecture with three convolutional layers and with an input of TF-IDF representation (TF-IDF) and **word-embeddings (WE)**. We followed the implementation details in the original work, and trained the models to 5,000 iterations as no obvious improvement is shown after this number of iterations. Table 7 shows that our approach achieves remarkable results using different datasets.

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 [26, 28, 42]. 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 and (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 4. In this experiment, we used a random forest and stratified sevenfold cross validation to evaluate the accuracy of identifying programmers.

Results. Table 6 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 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 8, Table 9, and Table 10.

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 4.

Table 8. 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 9. The Accuracy of Authorship Identification for Programmers with Seven Programs Using the Python

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

Note that the accuracy is always above 96%.

Table 10. The Accuracy of Authorship Identification for Programmers Who Solved Seven Problems Using the C Programming Language

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

Notice the accuracy is always close to 100%.

Table 11. The Accuracy of Authorship Identification for Programmers Who Solved Seven Problems from Three Different Years (2014–2016)

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

The identification models were trained on data from 2014 and tested on data from 2015 and 2016.

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. Table 11 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.

Experiment 1. We use dataset 3 (Table 5), 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 (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.

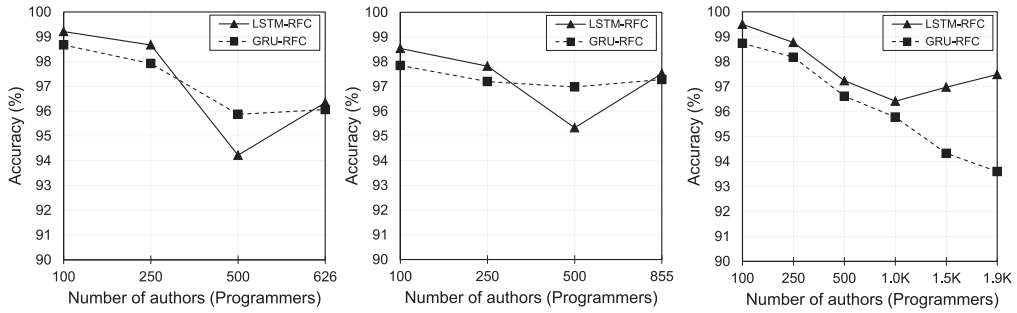
Key Insight. The reported test accuracy follows a stratified cross-validation, where every code file has been tested and contributed to the reported accuracy by being used in building the model. Therefore, the model is tested to identify programmers based on code samples written in a language that might 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 5), 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.

6 AUTHORSHIP IDENTIFICATION OF BINARY CODE

We examine the robustness of our approach in identifying authors of executable binaries. Previous research by Rosenblum et al. [61] extracted authorship features directly from the binary code to



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

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

enable the identification of 161 programmers with 51% accuracy, while the work of Caliskan-Islam et al. [29] improved this accuracy to 92% using features extracted from different levels of the decompilation process. Caliskan-Islam et al. [29] extracted features from assembly code and abstract syntax tree of decompiled code. These features enabled the binary code authorship identification of 600 programmers with an accuracy of 83%. The authors used an approach of four steps to identify programmers of binary code, namely: disassembly, decompilation, dimensionality reduction and classification. In this work, we use a similar approach without the requirement of extracting features from different levels. Instead, we use Hex-Rays, a commercial powerful decompiler, to decompile executable binaries to generate C-like pseudo code. The generated pseudo code can be described as a translation of the program instructions using higher level constructions that preserve the program's control structures such as loops and branches. These features have shown a high degree of significance in attributing programmers in previous works. Therefore, we conduct several experiments to evaluate our approach in identifying programmers of executable binaries.

Assumptions. We assume the availability of binary samples of authors to be identified. Additionally, we examine our approach to identify authors of tested executable binary programs under the assumption of knowing the toolchain provenance such as compiler family and version, compilation optimization level, and source code language, and so on. These assumptions consistent with the literature of identifying programmers of software binary code [29]. Moreover, state-of-the-art tools and methods can identify with high accuracy the toolchain provenance of a given exactable binary [60]. Therefore, we assume that such techniques are used to define the toolchain provenance of a given binary code and use our models that are trained using samples compiled with same settings.

6.1 Binary Code Dataset

Using a dataset of 6,962 C and C++ programmers participated in GCJ from 2008 to 2016 with at least nine files, we created a dataset of executable binaries for 1,500 programmers to evaluate our approach in identifying authors using binary format. We used GNU Compiler Collection's GCC or G++ to compile C and C++ source codes, respectively, into 32-bit Intel 80386 Unix binaries with Executable and Linkable Format. Moreover, we also use different compilation optimization levels to examine the effects of resultant binaries in the authorship identification process. GNU Compiler provides different optimization levels corresponds to different flags such as O1, O2, O3, and Os flags. When an optimization flag is turned on, the compiler generates different binaries that vary in some attributes such as code size, execution time, memory utilization, and so on. Using a higher

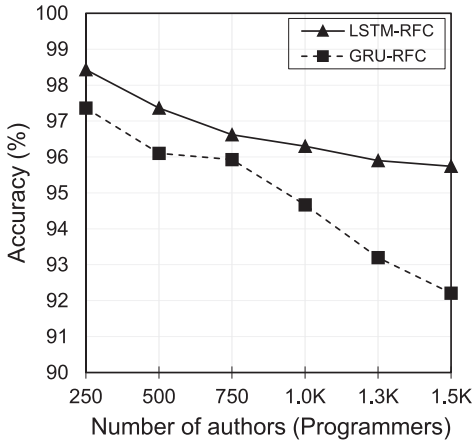


Fig. 9. The accuracy of the authorship identification of programmers using their binary samples compiled with no optimization.

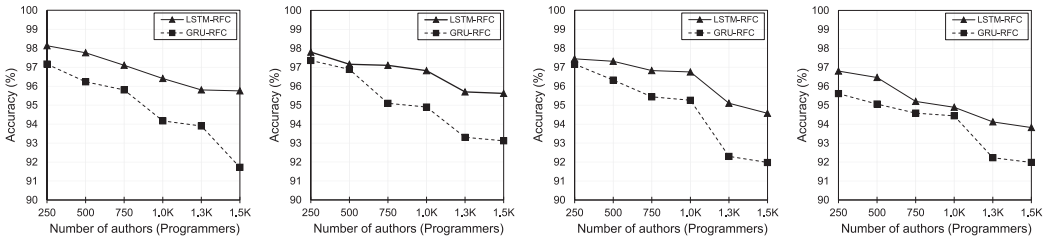
level of compilation optimization results in advanced and more optimized binary code. However, compilation with optimization requires more time and memory resources than compilation with no optimization.

6.2 Authorship Identification of Binary Code

In this experiment, we processed the dataset of 1,500 programmers with at least nine binary samples produced from a compilation without optimization process. Figure 9 shows that our approach accurately identified programmers on a large-scale binary dataset. Using ninefold-cross-validation LSTM-RFC, our approach achieved an accuracy of 98.4% for identifying 250 programmers. when increasing the scale of our experiment to include 1,500 programmers, our approach achieved an accuracy of 95.74%. These results show that our approach can identify programmers of executable binaries more accurately and on a larger scale than previous approaches.

6.3 Effect of Compilation Optimization

Since our approach achieved high accuracy in identifying programmers of binary code generated from a compilation process without optimization, this experiment explores the effects of different optimization levels on the code authorship identification using our approach. There are different optimization levels that can be incorporated with the compilation process to advance the optimization of certain attributes of an executable program. The optimization techniques transform a given program to a semantically equivalent program that is more efficient than the original program. For this experiment, we use four levels of optimization that can be turned on by O1, O2, and O3, Os flags for GCC compiler family. Using different optimization techniques that generates different binaries enables a better understanding of their effects on authorship attribution. Figure 10 shows the result of our approach in identifying 1,500 programmers with nine binary samples generated with different optimization level. Figure 10(a) shows the results of our approach using ninefold-cross-validation on a dataset generated by a compilation process with level-1 optimization. The LSTM-RFC approach achieved 98.1% accuracy for identifying 250 programmers, and 95.75% for identifying 1,500 programmers. Compared to the result obtained when no optimization is used, this result shows no accuracy degradation as the optimization technique is introduced. Using level-2 optimization, Figure 10(b) shows that our approach achieves an accuracy of 97.8% for identifying 250



(a) O1 optimization. (b) O2 optimization. (c) O3 optimization. (d) OS optimization.

Fig. 10. The accuracy of the authorship identification of programmers using their binary samples compiled with different optimization options, showing promising accuracy results even with decompiled binary samples.

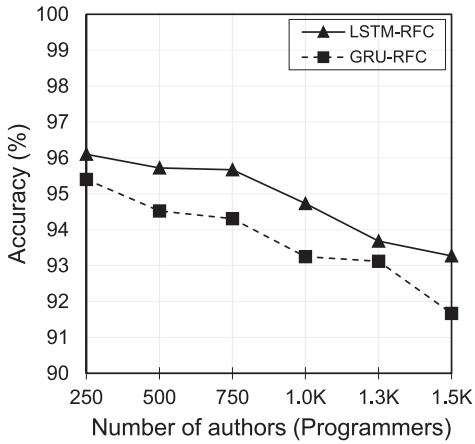


Fig. 11. The accuracy of the authorship identification of programmers using stripped binary dataset.

programmers and an accuracy of 95.6% for identifying 1,500 programmers. Similar results are obtained when using O3 optimization flag, Figure 10(c) shows that LSTM-RFC achieved an accuracy of 97.4% for identifying 250 programmers and an accuracy of 94.57% for identifying 1,500 programmers of binary code generated from a compilation with O3 optimization. Figure 10(d) shows that using Os optimization does not affect the identification accuracy as LSTM-RFC achieved 96.8% identification accuracy of 250 programmers and an identification accuracy of 93.82% for 1,500 programmers. The results achieved by different binary datasets generated from compilation with different optimization techniques show that our system is robust to different optimization and capable of extracting relevant authorship attributes that enabled accurate authorship identification.

6.4 Identification with Stripped Binary Code

In this experiment, we investigate the effects of stripping the symbol information from the binary code on the identification accuracy of our system. Using a fully stripped binary code, where all symbol table and relocation information are stripped using the *GNU strip* option, we show the effect of symbol information on authorship attribution. Figure 11 shows that the system was capable of generating high-quality deep representations that enabled accurate authorship identification. Comparing to previous works [29, 61], our approach shows robustness to different compilation settings including stripping symbol information. Even when symbol information is completely missing, the

Table 12. A Comparison with Related Work on Binary Code Authorship Identification

	No optimization	O1	O2	O3	OS	Stripped
Caliskan-Islam et al. [29]	91.36	89.57	86.21	86.76	81.61	68.97
This work	98.43	98.13	97.80	97.41	96.84	94.60

The results are reported using ninefold cross-validation method for a dataset of 250 programmers with 9 files each.

deep learning architecture is capable of transforming the input information presented in binary samples to robust deep representations of authorship attribution. Unlike other works that attempt to generalize features across different compilation settings, the deep learning architecture tune parameters that allow best representations based on a given input data or settings. For example, Caliskan-Islam et al. [29] showed that attempting to identify programmers of stripped binary code using the same feature set used for unstripped binaries can cause an accuracy degradation of 24%.

Comparison with Related Work. We conducted a comparison with the work of Caliskan-Islam et al. [29] using a dataset of binaries for 250 programmers with 9 files each. Table 12 shows the results achieved by the two works. Our approach maintained a high identification accuracy even with the different levels of optimization. When stripping the binaries, our approach achieved high identification accuracy of 94.60% for the 250 programmers, while the work of Caliskan-Islam et al. [29] has shown a degradation of 22.39% (i.e., 91.36–68.97).

7 AUTHORSHIP IDENTIFICATION OF OBFUSCATED SOFTWARE

The basic assumption for the operation of our approach is that TF-IDF can be extracted from the original software program, 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, a central question is, if the model is trained with obfuscated codes, will it be able to identify authors 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 features under obfuscation for testing obfuscated files. This requires the assumption that we know what obfuscation technique was used, where we transform the training set before building the model, which is a clear limitation of our approach. Deciding what obfuscation technique is used is out of scope of this article, but every obfuscation tool has a unique technique to amplify the obfuscation effect, which would be a hint to find the obfuscator.

The availability of several obfuscation tools and methods can allow programmers to attempt obfuscation as a method to ensure privacy and evade identification. Moreover, programmers might adopt obfuscation on the source code level or the binary level. In the following subsections, we show the effects of different obfuscation approaches on the software authorship identification.

7.1 Software Source Code Obfuscation

In this experiments, we investigate the effects of code-to-code obfuscation on authorship attribution. Different obfuscation tools are available, and two among them were chosen to evaluate our approach: Stunnix [1] and Tigress [6]. 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, however, 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 toward 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 12(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 12(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.

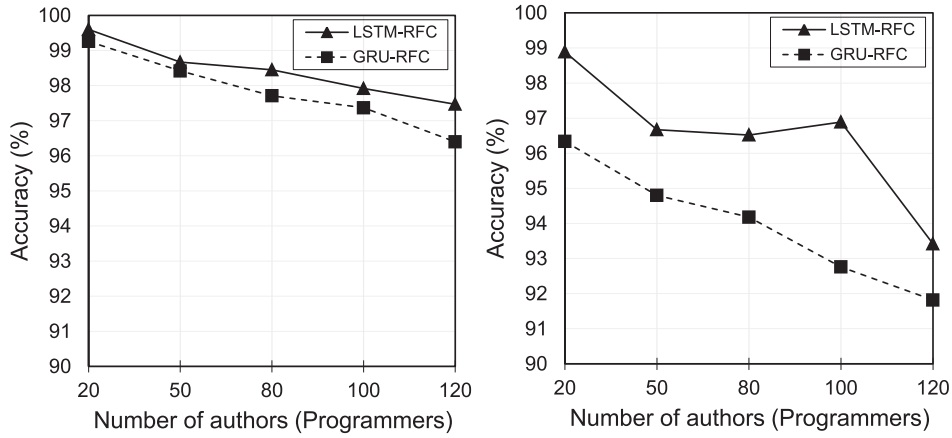
7.2 Software Binary Code Obfuscation

In this experiment, we investigate the effects of binary obfuscation on the identification accuracy. Among many tools for software binary obfuscation, we use Obfuscator-LLVM [45] to generate obfuscated binary code using different features. The Obfuscator-LLVM provides different levels of obfuscation including control flow flattening, instruction substitution, bogus code injection, and so on. The aim of this experiment is to examine the robustness of our approach in identifying programmers of obfuscated binary code even when different obfuscation levels are introduced.

For this experiment, we use the same dataset of 1,500 programmers used for the experiments on authorship identification of software binaries in Section 6. Programmers might adopt several techniques to circumvent authorship identification on the binary level of a program using control flow flattening, instruction substitution, bogus code injection or all options combined to make it difficult for analysis and authorship attribution. We address these different scenarios of obfuscation and show their effects on software authorship identification. Figure 13 shows results of different experiments conducted using different obfuscated binaries.

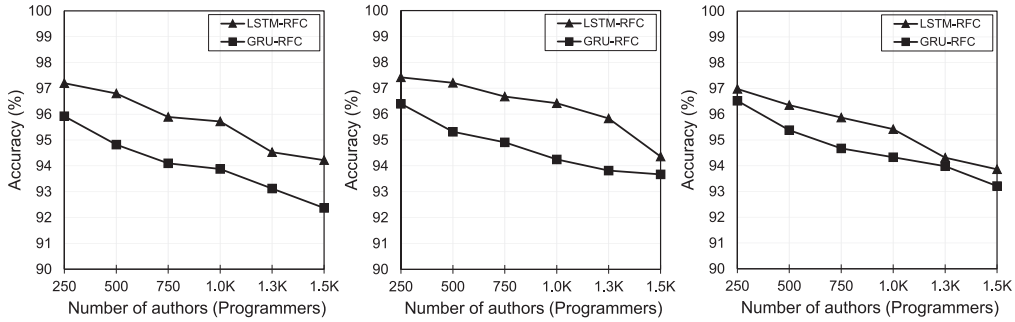
Experiment 1: Control flow flattening. Obfuscation through control flow flattening aims to hide the flow structure of a program using code transformations that target all basic blocks of a program. One way to achieve control flow flattening is to split all the program's basic blocks, e.g., functions, loops, branches, and so on, in a certain way that can be grouped inside one single infinite loop that operates on switch statement to control program's flow. This technique of obfuscation complicates the understanding of the program structure that can be indicative of authorship. We used the control flow flattening option in Obfuscator-LLVM tool to examine the effects of this technique on authorship attribution. Figure 13(a) shows the results of authorship identification of control flow flattened binaries using our approach. The results show that our approach still resilient to this kind of obfuscation by achieving an accuracy of 97.2% in identifying 250 programmers and accuracy of 94.22% for 1,500 programmers.

Experiment 2: Instruction substitution. Obfuscation through instruction substitution is a straightforward technique aims to replace standard instruction with a series of functionally equivalent instructions. This technique of obfuscation increases the code size and can be simply evaded by an optimization process. Therefore, such a technique alone might not be the optimal choice



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

Fig. 12. The accuracy of authorship identification with obfuscated source code, showing promising results even with the more sophisticated obfuscation approach (Tigress).



(a) Control flow flattening. (b) Instructions substitution. (c) Full obfuscation.

Fig. 13. The accuracy of authorship identification with obfuscated binary code using different Obfuscator-LLVM options.

for obfuscation but it can add complication when used with other obfuscation techniques. Using instruction substitution obfuscation, Figure 13(b) shows similar results for our approach with an accuracy of 97.42 % for identifying 250 programmers and an accuracy of 94.36% for identifying 1,500 programmers.

Experiment 3: Full Obfuscator-LLVM obfuscation. In this experiment, we allowed all obfuscation options offered by Obfuscator-LLVM, including, control flow flattening, instructions substitution, and bogus control flow. Using a dataset of fully-obfuscated binary code, our approach shows remarkable resilience by achieving high identification accuracy on different scales as presented in Figure 13(c). In Figure 13(c) shows that our approach achieved an accuracy of 96.98% for identifying 250 programmers and an accuracy of 93.86% when increasing the scale to 1,500 programmers.

Comparison with Related Work. We conducted a comparison with the related work on obfuscated code authorship identification, considering both code-to-code obfuscation and binary obfuscation. We followed the same obfuscation methods described in Section 7.1 to generate the C++

Table 13. A Comparison with the Related Work on Obfuscated Code Authorship Identification

	Stunnix-obfuscated	Tigress-obfuscated	O-LLVM-obfuscated
Caliskan-Islam et al. [28]	98.89	67.22	✗
Abuhamad et al. [11] -TFIDF	97.34	71.34	✗
Abuhamad et al. [11] -WE	95.87	74.38	✗
Caliskan-Islam et al. [29]	✗	✗	92.76
This work	99.60	98.89	98.94

The results are reported using ninefold cross-validation method for a dataset of 20 programmers with 9 files each.

Stunnix-obfuscated dataset and the C Tigress-obfuscated dataset of 20 programmers with nine files each. Table 13 shows the results achieved by different authorship identification methods on the obfuscated code. Our approach shows remarkable results, even with sophisticated code-to-code obfuscation such as Tigress. Similarly, for the binary obfuscated code, our approach maintains a high identification accuracy even with full Obfuscator-LLVM obfuscation (Section 7.2). Our approach achieves an accuracy of 98.94% compared to 92.26% achieved by the work of Caliskan-Islam et al. [29].

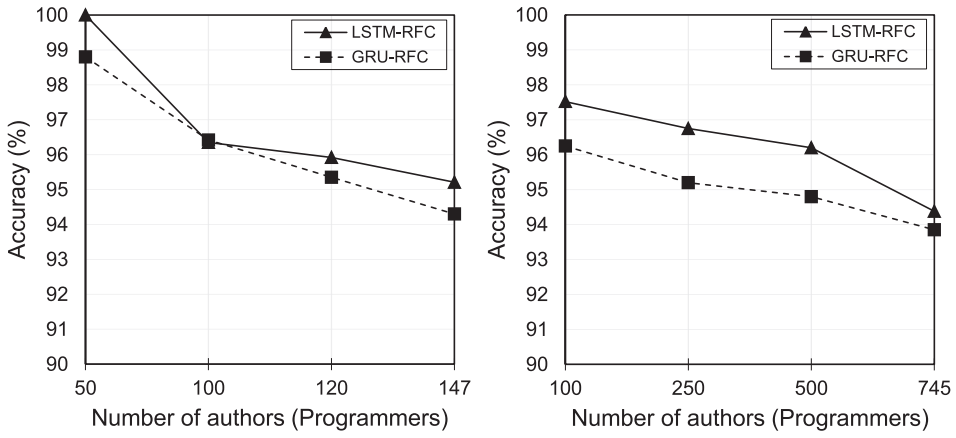
8 AUTHORSHIP IDENTIFICATION IN THE REAL WORLD

This section explores the robustness of our system using real-world scenarios. We examine our approach using a dataset collected in the wild from the code sharing platform (GitHub). Moreover, we show possible ways of handling the open-world assumption to identify new programmers who might not be seen by the model before. Handling such situations allows the model to have a certain validity when applied in the real world as the model might be tested on samples of programmers who have not included in the training process. Another possible application of our system is malware attribution. Although malware attribution is a challenging task due to the lack of ground-truth dataset, it is possible to apply deep authorship representation to assign malware to families and groups that enable sufficient analysis.

8.1 Software Authorship Identification in the Wild

This section investigates the applicability of our approach when the code samples are collected from public code sharing platforms such as GitHub. Handling software authorship attribution in the wild adds some challenges as there are no guarantees on the ground truth of authorship. The code reuse and multiple collaboration on software projects make attributing software much challenging. Since we had such success in identifying programmers participated in GCJ, we examine our system on a dataset collected in the wild.

We randomly sampled 2,000 repositories from GitHub that list C and C++ as the primary language and published by one contributor. We initially excluded 13 repositories that were not fit for the experiments for not having the targeted number of C++/C files. Upon processing the repositories and removing the incomplete files/samples, 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 [34]. 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*.



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

Fig. 14. The accuracy of the authorship identification of programmers using GitHub dataset, showing promising results even with real-world code samples.

Experiment 1: Source code authorship identification in the wild. For this experiment, we process the collected dataset using the source code files. Figure 14 shows the results of our approach using GitHub C++ and C datasets. Figure 14(a) shows an accuracy of 100% when using LSTM-RFC for 50 C++ programmers and 95.21% for 147 programmers. Figure 14(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.

Experiment 2: Binary code authorship identification in the wild. For this experiment, we compiled code files of the collected dataset in the wild using the same compilation options presented in Section 6. Using code files of 142 (C++) and 745 (C) programmers, we successfully generated a dataset of a total of 241 programmers who have at least nine files that we were able to compile. For the compilation process, we generated binaries with level O3 optimization and removed all debugging symbols. Using the dataset of binary code, our approach achieves an accuracy of 92.13% in identifying 241 programmers. This result demonstrates that using deep representations of authorship attribution enables accurate authorship identification in the wild.

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.

8.2 Software Authorship Identification in the Open World

Authorship identification using open-world assumption is applicable in a real-world scenario when attempting to identify the author of a given software, who might not be included in the suspect set. In contrast to the conventional machine learning approach, in which the model evaluation is based on unseen samples of labels that the model trained on during the training phase, addressing open world problem raise another challenge in indicating whether a given tested sample belongs

to a new unseen label. This setting is more reasonable for software forensics, since analysts aim to attribute pieces of software, e.g., malware, that can possibly be created by new programmers who are not part of the suspect set. Previous works by Caliskan et al. [29] and Dauber et al. [34] have addressed the problem of authorship identification in an open world scenario. The authors used classification confidence as an indicator of sample-label-membership, where high classification confidence demonstrates a high probability of classified labels, whereas low confidence signals model hesitation of the classification decision. In ensemble classifier, such as the adopted RFC, the percentage of voted trees for a certain label reflects the model classification confidence. For an author (A_i), the classification confidence of a RFC identification model is estimated by percentage of trees voted for (A_i) when testing a given sample, and it can be formulated as $Conf(A_i) = \frac{\sum_j Vote_j(A_i)}{\|T\|}$, where $Vote_j(A_i)$ is the vote of tree j for A_i and $\|T\|$ is the total number of trees in RFC model.

Addressing open world identification requires setting up a confidence threshold where classifications with higher confidence level are accepted, while classifications below the threshold are rejected and reported as possible membership of new unseen labels. One way of estimating the confidence threshold for a classifier is by classification margin defined by the difference between the highest and second highest $Conf(A_i)$ of a given sample [29].

Experiment 1: Setting confidence threshold. To establish a confidence threshold for our RFC identification models, we used a dataset of 1,000 C++ programmers with nine files each. Using the training set, we estimated the confidence threshold be averaging all confidence levels of classified samples as $\frac{1}{n}\sum_{j=0}^{n-1} Conf_j(A_i)$, where $Conf_j(A_i)$ is the confidence of classifying sample j for an author A_i, and n is the total number of samples. Using the RFC model of 300 trees trained to identify 1,000 C++ programmers with an accuracy of 96.2%, we adopted a stratified ninefolds cross-validation to calculate and evaluate the classification confidence threshold. Among the 9,000 code samples in the dataset, 8,658 code sample were correctly classified with average confidence of 32.12%. The other 342 code samples were misclassified with average confidence of 28.46%. Using this observation, we can set a confidence threshold to accept and reject classification based on the model confidence in assigning programmers to code samples.

Experiment 2: Identification in the open world. Setting a confidence threshold results in accepting and rejecting model decision on programmers identification. We can evaluate a certain threshold by calculating the recall and precision of accepting and rejecting model decisions. For example, accepting decisions for "out-of-world" samples is considered as a false positive (i.e., wrong decision to accept). On the other hand, rejecting decisions for "in-world" samples is considered as false negative. The precision and recall are then calculated as: $precision = \frac{truepositive}{truepositive+falsepositive}$ and $recall = \frac{truepositive}{truepositive+falsenegative}$. Based on the desired precision-recall tradeoff, a designer decide on a confidence threshold that satisfies the system requirement. In this work, we report the result of assessing different thresholds by the $F1 - score = 2 \times \frac{precision \times recall}{precision+recall}$ as the harmonic average of the precision and recall metrics.

To simulate the open world experiment, we used 9,000 "out-of-world" samples and test them with RFC model trained on 9,000 "in-world" samples. We passed all samples to classifier an observed the results achieved by adopting several confidence thresholds. We started by a low confidence threshold of 25% to achieve 71.4 precision, 62.3% recall, and 66.54 F1-score. When adopting a high confidence threshold of 40%, the results do not change significantly with a precision of 74.8%, recall of 68.1%, and F1-score of 71.29%. The obvious choice of selecting a threshold is by finding the best estimation between the average of confidence levels of "in-world" correct classification and the average of "out-of-world" misclassification. In our experiments, we found that 29% level of confidence to be the best threshold to achieve the best results with precision of 94.13%, recall of 88.2%, and F1-score of 91.1%.

9 LIMITATIONS

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

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.

Binary Code. Our experiments on binary code show that deep representations assist identifying programmers with higher accuracy and on a larger scale than state-of-the-art methods. However, the validity of our approach relies on the ability of successfully identifying the toolchain provenance of investigated executable binaries. Using specialized compilers that generate non-standard binary code may obstruct our approach, especially when failing to fingerprint the used compiler. Moreover, the ground-truth assumption when assigning one programmer to a binary code makes it easier to track programmers of decompiled codes. This process becomes more complicated when multiple programmers are involved, since it requires to trace authorship through the reverse engineering process. We leave this challenge to future work.

10 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.

REFERENCES

[1] 20120. Stunnix. Retrieved February 2, 2021 from http://stunnix.com/.

[2] 2020. Google Code Jam. Retrieved February 2, 2021 from https://codingcompetitions.withgoogle.com/codejam.

[3] 2020. Hex-Rays. Retrieved February 2, 2021 from https://www.hex-rays.com/products/decompiler/.

[4] 2020. IDA Pro. Retrieved February 2, 2021 from https://www.hex-rays.com/products/ida/.

[5] 2020. Radare. Retrieved February 2, 2021 from https://www.radare.org/.

[6] 2020. The tigress c obfuscator. Retrieved February 2, 2021 from https://tigress.wtf/.

[7] 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. arXiv:1603.04467. Retrieved from http://arxiv.org/abs/1603.04467.

[8] Ahmed Abbasi and Hsinchun Chen. 2008. Writeprints: A stylometric approach to identity-level identification and similarity detection in cyberspace. *ACM Trans. Inf. Syst.* 26, 2 (2008), 7.

[9] Mohammed Abuhamad, Tamer AbuHmed, Aziz Mohaisen, and DaeHun Nyang. 2018. Large-scale and language-oblivious code authorship identification. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 101–114.

[10] Mohammed Abuhamad, Tamer Abuhmed, DaeHun Nyang, and David Mohaisen. 2020. Multi-χ: Identifying multiple authors from source code files. *Proceedings on Privacy Enhancing Technologies (PoPETs)* 2020, 3 (2020), 25–41.

[11] Mohammed Abuhamad, Ji-su Rhim, Tamer AbuHmed, Sana Ullah, Sanggil Kang, and DaeHun Nyang. 2019. Code authorship identification using convolutional neural networks. *Fut. Gener. Comput. Syst.* 95 (2019), 104–115.

[12] Sadia Afroz, Aylin Caliskan Islam, Ariel Stolerman, Rachel Greenstadt, and Damon McCoy. 2014. Doppelgänger finder: Taking stylometry to the underground. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'14)*. IEEE, 212–226.

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

[14] Alexander T. Basilevsky. 2009. *Statistical Factor Analysis and Related Methods: Theory and Applications*. Vol. 418. John Wiley & Sons.

[15] Yoshua Bengio. 2008. Neural net language models. *Scholarpedia* 3, 1 (2008), 3881.

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

[17] Yoshua Bengio, Aaron Courville, and Pascal Vincent. 2013. Representation learning: A review and new perspectives. *IEEE Trans. Pattern Anal. Mach. Intell.* 35, 8 (2013), 1798–1828.

[18] Yoshua Bengio, Aaron C. Courville, and Pascal Vincent. 2012. Unsupervised feature learning and deep learning: A review and new perspectives. arXiv:1206.5538. Retrieved from http://arxiv.org/abs/1206.5538.

[19] Antoine Bordes, Xavier Glorot, Jason Weston, and Yoshua Bengio. 2012. Joint learning of words and meaning representations for open-text semantic parsing. In *Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'12)*, Vol. 22. 127–135.

[20] Nicolas Boulanger-Lewandowski, Yoshua Bengio, and Pascal Vincent. 2012. Modeling temporal dependencies in high-dimensional sequences: application to polyphonic music generation and transcription. In *Proceedings of the 29th International Conference on Machine Learning, ICML 2012, Edinburgh, Scotland, UK, June 26 - July 1, 2012*. icml.cc.

[21] Leo Breiman. 2001. Random forests. *Mach. Learn.* 45, 1 (2001), 5–32.

[22] Michael Brennan, Sadia Afroz, and Rachel Greenstadt. 2012. Adversarial stylometry: Circumventing authorship recognition to preserve privacy and anonymity. *ACM Trans. Inf. Syst. Secur.* 15, 3 (2012), 12.

[23] Steven Burrows and S. M. M. Tahaghoghi. 2007. Source code authorship attribution using n-grams. In *Proceedings of the 12th Australasian Document Computing Symposium (ADCS'07)*, A. Spink, A. Turpin, and M. Wu (Eds.), 32–39.

[24] 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

[25] 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, 699–713. https://doi.org/10.1007/978-3-642-00887-0_61

[26] S. Burrows, A. L. Uitdenbogerd, and A. Turpin. 2009. Temporally robust software features for authorship attribution. In *Proceedings of the 2009 33rd Annual IEEE International Computer Software and Applications Conference*, Vol. 1. 599–606.

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

[28] 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, 255–270. http://dl.acm.org/citation.cfm?id=2831143.2831160.

[29] Aylin Caliskan-Islam, Fabian Yamaguchi, Edwin Dauber, Richard Harang, Konrad Rieck, Rachel Greenstadt, and Arvind Narayanan. 2018. When coding style survives compilation: De-anonymizing programmers from executable binaries. In *Proceedings of the Network and Distributed System Security Symposium 2018 (NDSS'18)*.

[30] B. Chandra and Rajesh Kumar Sharma. 2017. On improving recurrent neural network for image classification. In *Proceedings of the 2017 International Joint Conference on Neural Networks (IJCNN'17)*. IEEE, 1904–1907.

[31] Chahes Chopra, Shivam Sinha, Shubham Jaroli, Anupam Shukla, and Saumil Maheshwari. 2017. Recurrent neural networks with non-sequential data to predict hospital readmission of diabetic patients. In *Proceedings of the 2017 International Conference on Computational Biology and Bioinformatics*. 18–23.

[32] Dan Ciregan, Ueli Meier, and Jürgen Schmidhuber. 2012. Multi-column deep neural networks for image classification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'12)*. IEEE, 3642–3649.

[33] George E. Dahl, Dong Yu, Li Deng, and Alex Acero. 2012. Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *IEEE Trans. Aud. Speech Lang. Process.* 20, 1 (2012), 30–42.

[34] Edwin Dauber, Aylin Caliskan, Richard Harang, and Rachel Greenstadt. 2018. Git blame who?: Stylistic authorship attribution of small, incomplete source code fragments. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, 356–357.

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

[36] John Duchi, Elad Hazan, and Yoram Singer. 2011. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.* 12 (Jul. 2011), 2121–2159.

[37] 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.

[38] Brian S. Everitt and Graham Dunn. 2001. *Applied Multivariate Data Analysis*. Vol. 2. Wiley Online Library.

[39] 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. *Int. J. Dig. Evid.* 6, 1 (2007), 1–18.

[40] 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, 893–896. https://doi.org/10.1145/1134285.1134445

[41] Ian J. Goodfellow, Aaron Courville, and Yoshua Bengio. 2012. Spike-and-slab sparse coding for unsupervised feature discovery. *CoRR abs/1201.3382 (2012)*. http://arxiv.org/abs/1201.3382.

[42] Niels Dalum Hansen, Christina Lioma, Birger Larsen, and Stephen Alstrup. 2014. Temporal context for authorship attribution. In *Proceedings of the Information Retrieval Facility Conference*. Springer, 22–40.

[43] Geoffrey Hinton, Li Deng, Dong Yu, George E. Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N. Sainath, et al. 2012. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Sign. Process. Mag.* 29, 6 (2012), 82–97.

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

[45] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. 2015. Obfuscator-LLVM—Software protection for the masses. In *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection (SPRO'15)*, Brecht Wyseur (Ed.). IEEE, 3–9. https://doi.org/10.1109/SPRO.2015.10

[46] Patrick Juola et al. 2008. Authorship attribution. *Found. Trends Inf. Retriev.* 1, 3 (2008), 233–334.

[47] 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'03)*, Vol. 3. 255–264.

[48] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015*. http://arxiv.org/abs/1412.6980

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

[50] Moshe Koppel, Jonathan Schler, and Shlomo Argamon. 2009. Computational methods in authorship attribution. *J. Assoc. Inf. Sci. Technol.* 60, 1 (2009), 9–26.
[51] 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.
[52] 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, 2082–2089. https://doi.org/10.1145/1276958.1277364
[53] Liang Li, Shuhui Wang, Shuqiang Jiang, and Qingming Huang. 2018. Attentive recurrent neural network for weak-supervised multi-label image classification. In *Proceedings of the 26th ACM International Conference on Multimedia*. 1092–1100.
[54] 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 *Proceedings of the 6th International Conference on Neural Information Processing (ICONIP'99)*, Vol. 1. 66–71. https://doi.org/10.1109/ICONIP.1999.843963
[55] Cameron H. Malin, Eoghan Casey, and James M. Aquilina. 2008. *Malware Forensics: Investigating and Analyzing Malicious Code*. Syngress.
[56] Xiaozhu Meng, Barton P. Miller, and Kwang-Sung Jun. 2017. Identifying multiple authors in a binary program. In *Proceedings of the European Symposium on Research in Computer Security*. Springer, Oslo, Norway, 286–304.
[57] Lichao Mou, Pedram Ghamisi, and Xiao Xiang Zhu. 2017. Deep recurrent neural networks for hyperspectral image classification. *IEEE Trans. Geosci. Remote Sens.* 55, 7 (2017), 3639–3655.
[58] Brian N. Pellin. 2000. Using classification techniques to determine source code authorship. White Paper. Department of Computer Science, University of Wisconsin.
[59] Salah Rifai, Yann Dauphin, Pascal Vincent, Yoshua Bengio, and Xavier Muller. 2011. The manifold tangent classifier. In *Proceedings of the Conference on Neural Information Processing Systems (NIPS'11)*, Vol. 271. 523.
[60] Nathan Rosenblum, Barton P. Miller, and Xiaojin Zhu. 2011. Recovering the toolchain provenance of binary code. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 100–110.
[61] Nathan Rosenblum, Xiaojin Zhu, and Barton Miller. 2011. Who wrote this code? Identifying the authors of program binaries. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS'11)*, 172–189.
[62] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. 2015. Recognizing functions in binaries with neural networks. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security'15)*. Washington, DC, 611–626.
[63] Eugene H. Spafford and Stephen A. Weeber. 1993. Software forensics: Can we track code to its authors? *Comput. Secur.* 12, 6 (1993), 585–595.
[64] 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.
[65] Efstathios Stamatatos. 2009. A survey of modern authorship attribution methods. *J. Assoc. Inf. Sci. Technol.* 60, 3 (2009), 538–556.
[66] 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.
[67] Tijmen Tieleman and Geoffrey Hinton. 2012. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *Neur. Netw. Mach. Learn.* 4, 2 (2012).
[68] Özlem Uzuner and Boris Katz. 2005. A comparative study of language models for book and author recognition. In *Proceedings of the International Conference on Natural Language Processing*. Springer, 969–980.
[69] Linda J. Wilcox. 1998. Authorship: The coin of the realm, the source of complaints. *J. Am. Med. Assoc.* 280, 3 (1998), 216–217.

Received May 2019; revised February 2021; accepted April 2021