Mohammed Abuhamad, Tamer Abuhmed, DaeHun Nyang*, and David Mohaisen*

Multi-$\mathcal{X}$: Identifying Multiple Authors from Source Code Files

Abstract: Most authorship identification schemes assume that code samples are written by a single author. However, real software projects are typically the result of a team effort, making it essential to consider a fine-grained multi-author identification in a single code sample, which we address with Multi-$\mathcal{X}$. Multi-$\mathcal{X}$ leverages a deep learning-based approach for multi-author identification in source code, is lightweight, uses a compact representation for efficiency, and does not require any code parsing, syntax tree extraction, nor feature selection. In Multi-$\mathcal{X}$, code samples are divided into small segments, which are then represented as a sequence of n-dimensional term representations. The sequence is fed into an RNN-based verification model to assist a segment integration process which integrates positively verified segments, i.e., integrates segments that have a high probability of being written by one author. Finally, the resulting segments from the integration process are represented using word2vec or TF-IDF and fed into the identification model. We evaluate Multi-$\mathcal{X}$ with several Github projects (Caffe, Facebook's Folly, TensorFlow, etc.) and show remarkable accuracy. For example, Multi-$\mathcal{X}$ achieves an authorship example-based accuracy (A-EBA) of 86.41% and per-segment authorship identification of 93.18% for identifying 562 programmers. We examine the performance against multiple dimensions and design choices, and demonstrate its effectiveness.

Keywords: Code Authorship Identification, program features, deep learning identification, software forensics

DOI Editor to enter DOI
Received ..; revised ..; accepted ...

Mohammed Abuhamad: University of Central Florida, E-mail: abuhamad@knights.ucf.edu
Tamer Abuhmed: Sungkyunkwan University, E-mail: tamer@skku.edu
***Corresponding Author: DaeHun Nyang:** Ewha Womans University, E-mail: nyang@ewha.ac.kr
***Corresponding Author: David Mohaisen:** University of Central Florida, E-mail: mohaisen@ucf.edu

1 Introduction

Authorship identification is well-known in the natural language understanding community with several approaches to capture authors' distinctive stylometric features [3, 34, 55]. However, most of those approaches are inapplicable to authorship identification on structured code due to the inflexibility of the written code expressions established by the syntax rules. Recently, several efforts have investigated authorship identification of structured code, such as computer programs [1, 16, 17, 39]. These efforts established that programmers have distinctive and identifying programming styles that can survive compilation [18, 39].

Code authorship identification has many useful applications and can be either binary- or source code-based identification. Binary-based techniques [5, 18, 39, 48] are applicable to applications such as malware, proprietary software, and legacy code [39]. Source code-based techniques are applicable when the source code is available, e.g., in software copyright infringement [28], code authorship disputes [59], plagiarism detection [15], and code integrity investigations [38]. Moreover, such techniques could help in identifying malware authors who could leave source code in a compromised system for compilation or where some source-code fragments could be recovered from the decompiled binaries. Real-world examples of such codes (or leaked source) include Mirai and derivatives, Dendroid, Betabot, GM-bot, Mazar, TinyNuke, etc. (all available on Github). On the other hand, code authorship identification also poses significant privacy risks, e.g., for open source projects whereby contributors wish to stay anonymous. Exploring such risks by understanding the power of advanced and fine-grained identification techniques can raise awareness of the problem and may lead to potential defenses, which we tackle in this work.

Most of the existing code authorship identification techniques assume a single author per code sample, an assumption that does not always hold. For example, modern software projects are often the result of collaborative efforts, even with malware development due to shared code [39]. As a result, a multi-author and fine-grained identification from small segments of code within a single file becomes essential, especially for forensic applications. Auto-executable code in Java Applets, ActiveX controls, pushed content, plug-ins, and scripts are widely used as an attack vector, and they can benefit from fine-grained identification operating on smaller snippets. Recent studies show that more than 87% of Alexa top 75k sites use JavaScript code, which is subject to several attacks, including Cross-Site Scripting (XSS) and Cross-site Request Forgery (CSRF). Such attacks are executed using scripts in their source forms [4, 44, 50]. Being able to identify malicious code at a fine-granularity would help in malware attribution. However, conducting a fine-grained identification would have more subtle privacy implications.

Unlike the single author identification, multi-author identification in a single code sample raises more challenges, such as defining the boundaries of code pieces to be analyzed for authorship attributions. Moreover, the number of contributing programmers to a code sample could be arbitrarily large, and a multi-author identification system should be capable of identifying code authors even with a single line of code. Identifying programmers given such limited information requires powerful tools and abstractions to capture authorship attributes for accurate identification.

This paper contributes to multi-author code authorship identification by introducing Multi-$\mathcal{X}$, a fine-grained approach for identifying multiple authors of a code sample based on deep learning. The proposed approach addresses the above challenges and incorporates techniques for code representation, deep learning, and ensemble classification. We accomplish the multi-author identification by following five main steps (Figure 1). We begin with the source code presentation, where any given code sample is divided into a sequence of small segments to avoid authorship collision of the same segment. These segments are then incrementally integrated into larger segments using the verification process, where RNN models are trained to decide whether two consecutive segments are written by the same author or not. Following that, the integrated segments are fed to an authorship identification process, where RNN models are trained to identify the authors of these segments. In evaluating Multi-$\mathcal{X}$, we investigate the effect of various techniques for code representation, deep learning architectures, and classifiers.

Contributions. We propose Multi-$\mathcal{X}$, a fine-grained method for identifying multiple authors contributing to a single source file. We evaluate Multi-$\mathcal{X}$ using a large dataset of multi-author source-code files collected from Github and show its accuracy. We evaluate Multi-$\mathcal{X}$ across multiple dimensions and design choices. Multi-$\mathcal{X}$ enables multi-author verification and identification on small fractions of code; i.e., it can identify multiple authors line-by-line. We examine the effect of code representation on modeling authorship attributions. We use a *word2vec* technique to generate distributed representations of code terms that enable authorship verification on small segments (e.g., segments with one line of code). Moreover, we also use TF-IDF technique to represent larger segments for the authorship identification task. Multi-$\mathcal{X}$ is lightweight since (a) it does not produce a large number of sparse features of code samples, but a small number of compact representations in proportion to the sequence size, (b) it does not require code parsing, syntax tree extraction, nor explicit feature selection. Multi-$\mathcal{X}$ takes advantage of RNN-based deep learning techniques to generate discriminative author features.

Summary of Results. Using a large dataset of real open-source projects, our approach achieves high accuracy on the three targeted tasks:

- **Code Authorship Verification:** using *word2vec* representations of code segments with one line of code, the RNN model of Multi-$\mathcal{X}$ can achieve an F1-score exceeding 88% in determining whether two subsequent segments are written by the same programmer. The F1-score reaches 93.85% when using deeper (multi-layer) bi-directional RNN.
- **Code Segment Authorship Identification:** using ground-truth data, we examine the sufficient size and number of samples per author needed to identify authors. Our approach achieves an accuracy of 92.12% for 479 authors when the number of samples is ten per author and the size of each sample is at least ten lines of code. This accuracy increases to 94.4% when the sample count increases to 30 samples per author. Moreover, the accuracy increases when using TF-IDF representations instead of *word2vec*, to reach 92.82% when the sample count is 10, and 96.14% when the sample count is 30 samples per author.
- **Code Authorship Identification:** for the overall system evaluation, we were able to identify multiple authors in 5,321 code files including 562 programmers with an Authorship Example-Based Accuracy (A-EBA) of 86.41% and an overall per-code-segment authorship identification accuracy of 93.18%.

Organization. The remainder of the paper is structured as follows. We review the related work in §2. In §3, we present the overall deep learning-based system for source-code multi-author identification. In §4, we evaluate our approach and show the experimental results. §5 discusses the limitations of our work and directions for future work. Finally, we provide our conclusion in §6.

Table 1. A summary of the related work. (*) Single sample attribution, where the authorship attributions are extracted from a single sample. (+) Multiple sample attribution, where the authorship attributions are extracted/merged from multiple samples.

Reference	#Author	Author per code	Programming Languages	Accuracy	Technique
Abuhamad et. al. [1]	8903	Single	C++	92.30%	RNN and Random Forest
Abuhamad et. al. [1]	3458	Single	Python	96.20%	RNN and Random Forest
Abuhamad et. al. [1]	1952	Single	Java	97.24%	RNN and Random Forest
Caliskan-Islam et. al. [17]	1600	Single	C++	92.83%	Random Forest
Caliskan-Islam et. al. [17]	229	Single	Python	53.91%	Random Forest
Steven Burrows et. al. [16]	100	Single	C, C++	79.90%	ANN
Steven Burrows et. al. [16]	100	Single	C, C++	80.37%	SVM
Alsulami et. al. [6]	70	Single	Python	88.86%	LSTM and BiLSTM with Random Forest
Ding and Samadzadeh [24]	46	Single	Java	62.70%	Canonical Discriminant Analysis
Frantzeskou et al. [29]	30	Single	C++	96.90%	Nearest Neighbor with rank similarity
Ivan Krsul [35]	29	Single	C	73.00%	discriminant analysis
Lange and Mancoridis [36]	20	Single	Java	55.00%	Nearest Neighbor with rank similarity
Elenbogen et. al. [25]	12	Single	C++	74.70%	Decision Tree
Alsulami et. al. [6]	10	Single	C++	85.00%	LSTM and BiLSTM with Random Forest
Steven Burrows et. al. [14]	10	Single	C	76.78%	Mean Reciprocal Ranking
Frantzeskou et al. [29]	8	Single	C++	100.00%	Nearest Neighbor with rank similarity
MacDonell et al. [37]	7	Single	C++	81.10%	ANN and multiple discriminant analysis
MacDonell et al. [37]	7	Single	C++	88.00%	Case-Based Reasoning
Pellin [45]	2	Single	Java	88.47%	SVM
Meng et. al. [39]	284	Multiple	C, C++ binaries	65%	SVM and random forest
Dauber et. al. [22]	106	Multiple (*)	C++	73%	Random Forest
Dauber et. al. [22]	106	Multiple (+)	C++	99%	Random Forest
This work	282	Multiple (*)	C, C++	97.31%	RNN with random forest
This work	843	Multiple (*)	C, C++	88.89%	RNN with random forest

2 Related Work

Authorship attribution at a document-level started in the 19th century, with the first attempts to quantify authors' writing style [47]. However, it has not been until the past twenty years that researchers started to explore authorship attribution for software programmers. Generally, software authorship identification can be categorized into two types of works. First, the code authorship identification with the assumption that each code sample was written by a single programmer. Herein, the code sample can be in a source or binary format. Second, the code authorship identification with the assumption of collaborative authorship.

2.1 Single-Author Code Identification

Two approaches are considered to identify a programmer of software based on the available code form, which are source code and executable binaries. Several techniques for identifying programmers when only executable machine code is available [18, 39, 48]. On the other hand, most of the works on code authorship identification are conducted on source code [14, 16, 17, 24, 25, 29, 35–37, 45]. The prior works on a single source-code authorship identification assume that each sample of code is written by a single programmer and exhibit features and characteristics of the programmer, which can be used as indicative of authorship. Examples of such features include layout features (spacing, indentation, boarding characters, etc.), style features (variable naming, choice of statements, comments, etc.) and environment features (computer platform, programming language, compiler, text editor, etc.).

Several approaches and tools have been adopted for source-code authorship identification, such as statistical analysis, machine learning, or similarity-based ranking [16]. In particular, Krsul *et al.* were among the first to explore the possibility of identifying the author of source code using handcrafted features based on the programming style [35]. In their work, they applied a statistical analysis technique called the multivariate discriminant analysis to identify 29 programmers. Frantzeskou *et al.* [29] exploited low-level n-grams features extracted from source code along with a similarity ranking technique to identify a small number of programmers. Caliskan-Islam *et al.* [17] exploited the abstract syntax trees of source codes to extract stylometry features for programmers identification. Recently, Abuhamad *et al.* [1] utilized RNN to generate deep representations of authorship that enabled large-scale code authorship identification. Their experiments are conducted to cover all programmers participating in Google Code Jam competition in four programming languages. Similar work, presented in [2], proposed a convolutional neural network (CNN) approach to identify authors of source code. The authors reported that their CNN-based approach can achieve an identification accuracy exceeds 99% for 150 programmers.

2.2 Multi-Author Code Identification

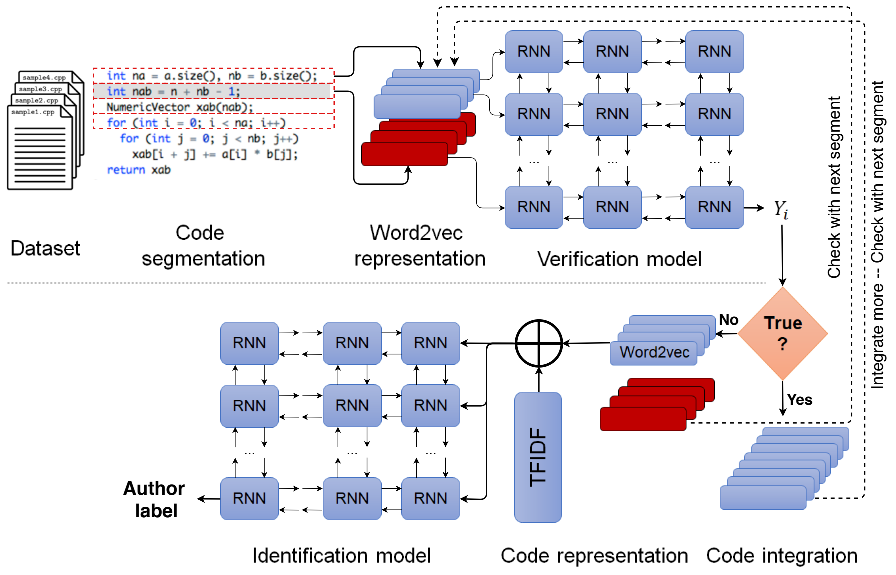
Identifying multiple authors of source code is related to multi-label learning, in which multiple labels are to be given to an unlabeled sample. Techniques for multi-label learning are well explored in several domains, including document classification and image recognition. The literature of multi-author identification is limited to the context of textual documents [23, 31, 43, 49], source code [22], and program executable binaries [39]. Payer *et al.* [43] introduced *deAnon*, a framework for de-anonymizing authorship of academic submissions. Using ensemble classifier, *deAnon* achieved an accuracy of 39.7% for identifying 1,405 possible authors from the first guess, and an accuracy of 65.6% from the first ten guesses. Dauber *et al.* [23] applied stylometry features to identify multi-authored documents from Wikia. The authors extended their analysis to include different possible application scenarios when using both relaxed classification and multi-label classification techniques. Sarwar *et al.* [49] proposed Co-Authorship Graph (CAG) technique to attribute different parts of documents to multiple authors. Using dataset of academic papers, CAG technique enabled accurate identification of 707 authors with an accuracy of 72.17%.

For source-code multi-author identification, the most closely related work is Dauber *et al.*'s seminal work in [22], which extends the work of Caliskan-Islam *et al.*'s [17] by attributing programmers of small code samples using a dataset obtained from Github considering multiple programmers for code files. The main difference between our work and Dauber *et al.*'s is as follows: First, their work used the stylometry features extraction process of [17], a process is shown to produce high-dimensional sparse representations, to extract the code features. Moreover, their feature extraction process requires code parsing, syntax tree extraction, and explicit feature evaluation and selection. Our work reduces the burden of feature extraction by taking advantage of deep learning to generate high-quality authorship attributions that enable large-scale identification. Our feature extraction relies on an RNN-based architecture that does not require additional steps nor feature selection process. Second, their work achieved an identification accuracy of 99% for 106 programmers given that each programmer has at least 150 code samples when considering multiple-sample attribution (by aggregating attributes of 50 samples). The authors addressed the multiple-sample attribution assuming that code segments can be collected from platforms with a version control system and accounts, e.g., Github, Gitlab, etc. The collection of segments for the same author can be attributed and aggregated to contribute to successful identification. Another suggested way to collect code samples for the same user is by clustering. Under this assumption, the presented results show a promising direction to identify multiple authors of source code. However, aggregating multiple samples (such as 50 samples) could limit the applicability of this method in practice. Our work considers identifying authors of source code based on a single-sample attribution and raising the challenge to scale even to more authors in open-source projects. Table 1 summarizes most related works for code authorship identification.

3 Multi-$\mathcal{X}$: An Overview

Multi-$\mathcal{X}$ contributes to solving the multi-author code authorship identification problem using an RNN-based system (§3.7) that incorporates five processes, which are: code processing and segmentation (§3.2), code sequence representation (§3.3), code authorship verification (§3.4), code segment integration (§3.5), and code authorship identification (§3.6). The overall process operates is shown in Figure 1. First, code samples are divided into small segments, then code segments are represented as a sequence of n-dimensional term representations. The *word2vec* representations are then fed into an RNN-based verification model to assist the segment integration process which integrates positively verified segments, i.e., integrate segments that have a high probability of being written by one author. Finally, the resulting segments from the integration process are represented using *word2vec* or TF-IDF embedding and fed into the authorship identification model.

However, before delving into the details of our RNN-based identification system, we first define some notations required for understanding the code multi-author identification task that we address in this work (§3.1).



```
int na = a.size(), nb = b.size();
int nab = n + nb - 1;
NumericVector xab(nab);
for (int i = 0; i < na; i++)
  for (int j = 0; j < nb; j++)
    xab[i + j] += a[i] * b[j];
return xab
```

Fig. 1. The general outline of the proposed approach. The code authorship verification includes processing code segments represented by *word2vec* to the verification model. The code authorship identification includes integrating code segments based on the verification process to be represented using *word2vec* or TF-IDF for the identification model.

3.1 Notations and Definitions

We treat the source-code sample C as a sequence of terms, $t_0, t_1, \ldots, t_{l-1}$ where $t_i \in \mathbb{Z}$ is the i-th term in the sequence. For example, a term can be a reserved keyword, a variable name, or an operator. We denote m segments in a sample code by $S_0, S_1, \ldots, S_m$, where a segment is a sequence of terms. Two segments can overlap if necessary. Terms of a segment S_i are labeled as $S_{i,t_0}, S_{i,t_1}, \ldots, S_{i,t_{l-1}}$. Segments are written by authors $a_0, a_1, \ldots, a_{n-1}$, where segment S_i is assigned to a single author a_i who contributed the most in writing it. Note that we defined the source code as segments of a set of terms, rather than functions. Therefore, Multi-$\mathcal{X}$ can handle incomplete codes without requiring a parser to extract functions or the abstract syntax tree (AST).
Task Definition. Given a source-code sample C without any information about the authors $(a_0, \ldots, a_{n-1})$ of this sample, the following tasks are defined:

– **Code Authorship Verification:** Given two subsequent segments of code S_i and S_{i+1}, determine whether the segments belong to the same author a_i.
– **Code Segment Authorship Identification:** Given S_i, identify a_i who wrote the segment.
– **Code authorship identification:** Given code C, identify the contributing authors $\{a_0, \ldots, a_{n-1}\}$ who wrote C. In other words, we identify all authors involved in writing all segments of C.
– **Open-World Identification:** Given code C, find $\{a_0, \ldots, a_{n-1}, a_{n+}\}$, where a_{n+} is one or more external authors who do not appear in the training data.

Code authorship identification is a superset of code authorship verification and segment authorship identification, while open-world identification is a superset that includes all of the other tasks. The foci of this work are the first three tasks; we leave the last as future work.

3.2 Code Processing and Segmentation

The first process for our fine-grained code authorship identification is segmentation. This process is performed using a sliding window, similar to the method adopted by Fifield and Follan [27], over the entire code sample. Applying a sliding window of size K and a stride R, the segmentation process generates a set of M code segments $\{S_0,\ S_1,\ \cdots,\ S_m\}$, where each segment S_i is assigned to an author a_i based on a ground-truth dataset. Consider a code C, presented as N pairs of lines and their corresponding ground-truth authors, i.e., $\{(l_0, a_0), (l_1, a_1), \ldots, (l_{n-1}, a_{n-1})\}$. The segmentation divides C into $M = \frac{N-K+p}{R} + 1$ segments, where p is the number of empty lines padded on the last segment ($p = K - (N \ mod \ K)$). For example, using $K = 6$ and $R = 4$ over a code file with 86 lines would result in $M = \frac{86-6}{4} + 1 = 21$ segments.

The purpose of this process is to divide the code into smaller segments for the verification task (i.e., checking

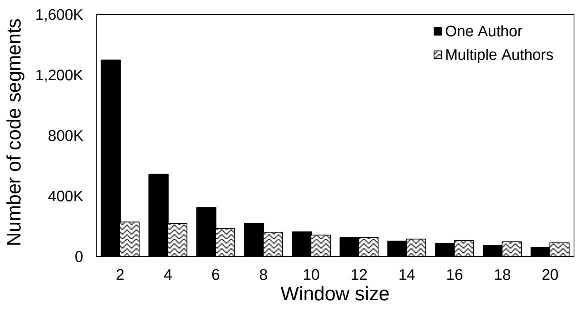


Fig. 2. Number of segments written by one author and multiple authors in nine open-source projects.

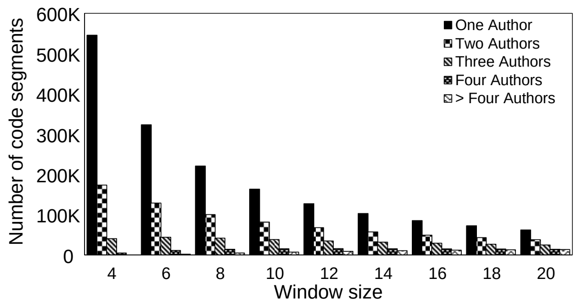


Fig. 3. Number of segments written by specific number of authors in nine open-source projects.

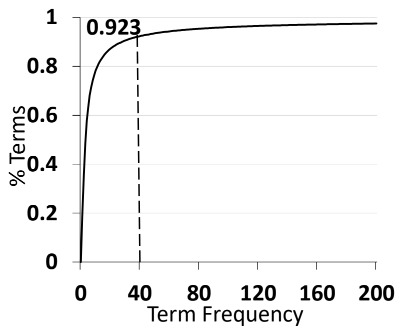


Fig. 4. Frequency of terms in our dataset. Note more than 92% of terms have less frequency than 40.

whether two consequent segments belong to the same author). With the assumption that this task is performed without any prior knowledge on the number of lines written by a single author in a code sample, the window size K can be a *hyperparameter*, tested and determined by experiments.

A segment is labeled based on the author who contributed most to it. Assigning authors in this way comes with some caveats since a segment can include codes of multiple authors, resulting in noise that may affect segments attribution. Thus, choosing the sliding window size is crucial. In particular, the sliding window should be small enough to recognize authors, and large enough to be correctly assigned to the right author. We based our selection of the window size on the experiments and statistics of real-world software projects.

Based on nine open-source libraries, Figure 2 shows that segments of code written by multiple programmers are very common. In fact, segments of length greater than 12 lines are more likely to be written by multiple programmers. This, in turn, motivates for introducing our fine-grained technique to identify programmers of a given code. By further analyzing the authorship of code segments, Figure 3 shows the number of segments written by a specific number of users. Even with small segments, e.g., six lines of code, there is a possibility that more than four programmers are involved. This possibility increases as the size of the segment increases. Therefore, defining the segment size for authorship identification is a challenging task that motivates our code authorship verification process prior to identification.

3.3 Code Sequence Representation

Code segments can be viewed as matrices, where each segment is a matrix with row representations, i.e., word embeddings, of tokens present in that segment. Given a large dataset, word embeddings can be learned using a prediction-based approach or a frequency-based approach. Recently, prediction-based methods, such as *word2vec* [58] and GloVe [46], have shown remarkable results. The frequency-based approach, such as TF-IDF, Co-Occurrence representations, and variations of both, are also studied. This work utilizes two methods of representing code samples: *word2vec* and TF-IDF.

Word2vec Representations. We use *word2vec* to represent code samples for deep learning models. Choosing the *word2vec* method is for several reasons. First, the *word2vec* approach provides distributed representations of tokens in a vector space allowing us to group similar tokens. Such a feature facilitates better language modeling [9]. Second, *word2vec*, as a learning method of generating distributed representations of tokens, has shown remarkable success in a wide range of applications (e.g., [20, 32, 41, 51, 53]).

We consider segments of source code as sequences of terms and expressions for training a *word2vec* model, which in turn is used to generate representations of code terms and expressions. Generating code representations using *word2vec* model requires some consideration due to the unconstrained and wide range of used terms. The source code includes variable names, language-specific keywords, and special characters that are part of the language rules. Unlike natural language texts, source codes include variable names that are not subject to syntactic or semantic rules. This results in a high number of terms with very low frequency as shown in Figure 4. To this end, we used around $153K$ unique terms out of a corpus of more than $26K$ C/C++ files to train a *word2vec* model. The *word2vec* model encodes similarities between terms as the distance between their representation vectors, where each term is represented as $\mathbb{R}^{128}$ vector of real values.

Representing segments of code as sequences of term representations, we aim to train RNN models that are capable of distinguishing authorship traits even with small sequences. This benefits the performance of the verification process, where often small segments (e.g., can be limited to one line of code) are targeted.

TF-IDF Representations. In TF-IDF, a term t in file d of a corpus $\mathcal{D}$ is assigned a weight using

$$\text{TF-IDF}(t, d, \mathcal{D}) = \text{TF}(t, d) \times \text{IDF}(t, \mathcal{D}),$$

where $\text{TF}(t, d)$ is the term frequency (TF) of t in d and

$$\text{IDF}(t, \mathcal{D}) = \log(|\mathcal{D}|/\text{DF}(t, \mathcal{D})) + 1,$$

where $|\mathcal{D}|$ is the number of documents in $\mathcal{D}$ and $\text{DF}(t, \mathcal{D})$ is the number of documents containing the term t. In evaluation, code samples are represented by TF-IDF representations of *uni-grams*, *bi-grams*, and *tri-grams*. Considering our dataset, the TF-IDF representations of code pieces are sparse and high-dimensional. Therefore, we represent code segments with the top 3,000 TF-IDF features based on the order of term frequencies across all code segments. Based on preliminary experiments, the top 3,000 features are sufficient to represent code segments. Even with this feature selection, small segments are represented in sparse vectors, thus we only use TF-IDF representations in the code identification.

Representation Learning. The *word2vec* models and TF-IDF vectorizers are constructed using the training dataset only. When applying the representation scheme, out-of-vocabulary (OOV) problem may occur during the validation and testing part of the experiment. There are several approaches for handling the OOV problem [10]. In this study, unseen terms are represented with zero-vectors when using *word2vec* and ignored in TF-IDF.

3.4 Code Authorship Verification

Adapting a fine-grained approach to identify multiple authors of a code sample requires distinguishing the boundaries of code pieces written by different authors. To accomplish that, we propose a code authorship verification process. This process aims to determine whether

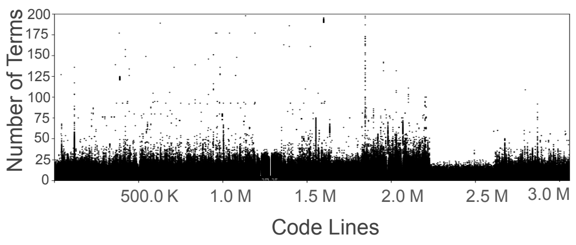


Fig. 5. Number of terms per code lines in our dataset. The maximum number of terms is 17,315, while the average is 7.6 $\approx$ 8.

two subsequent segments are written by the same author. This task requires training a model capable of establishing a decision of whether a given segment S_{i+1} belongs to the same author of S_i or not. We utilize RNN to perform this task and investigate the performance of various RNN model architectures under various experimental configurations. Given two subsequent code segments, S_i with length l and S_{i+1} with length k, the verification model takes vector representations of both segments' terms. $S_{i,t_0}, S_{i,t_1}, \ldots, S_{i,t_{l-1}}$ and $S_{i+1,t_0}, S_{i+1,t_1}, \ldots, S_{i+1,t_{k-1}}$, as an input of size $l + k$ (terms) and generates a decision based on an output probability of a softmax function that signifies whether the two segments are written by the same author. For this task, we preserve the order of terms in a code segment to enable the recognition of a pattern change when two segments are written by different authors.

3.5 Code Segments Integration

Subsequent segments that are written by the same author can be integrated into one larger segment. This step is important as larger segments exhibits more indicative authorship attribution than smaller segments. To automate the integration process using our fine-grained approach, we use the authorship verification model to essentially decide whether two subsequent segments are written by the same author. Subsequent segments that are positively verified for the same author are then integrated into one piece. It is designed to include as many lines as possible for the same author to help correctly identify the author since, intuitively, the more information available the better the identification accuracy. When two subsequent segments are not assigned to the same author, each segment is considered individually for the identification process.

Handling Small Segments. When choosing a small segmentation window (such as $K = 1$ line of code), the expected number of terms per segment is equal to eight terms, on average, as illustrated in Figure 5. However, there exists a number of segments that are very small (e.g., with length less than three terms). The intuitive reasoning behind such cases is that small segments are not written individually but are rather written by the same author of the previous or following segments. When looking at two subsequent segments with one line of code each, the chances that these two subsequent segments are written by the same programmer is approximately 85% as shown in Figure 2. Considering the distribution of the number of terms per line, these chances increase significantly (to more than 99%) when the number of terms is equal to or less than three terms. Therefore, when a small segment is presented, we integrate it with the previous one without verification. We understand that this assumption does not always hold, but excluding small segments from the code authorship verification positively enhances the overall performance, while not giving up significantly the accuracy.

3.6 Code Authorship Identification

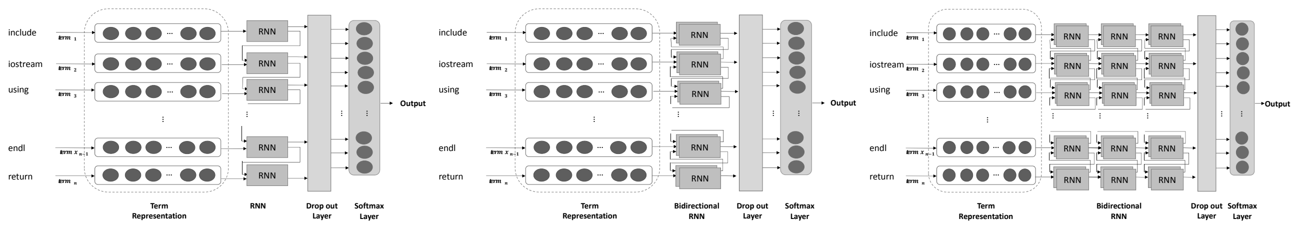
The essential step in our system is to identify multiple authors of a single code sample. Our approach to achieving accurate multi-author code authorship identification is adopting a fine-grained approach where the identification of code segments contributes to the overall identification accuracy. Assigning authors to code segments is performed using RNN models trained to capture distinctive authorship attributions to enable accurate identification. For this task, we investigate two methods to represent code samples for the RNN model (i.e., the sequence of *word2vec* and one vector of TF-IDF). For *word2vec* representation, segments with at least n lines and m terms per lines are represented as a sequence of $n \times m \times d$, where d is the dimension of term representation. Since the sequence length varies, we fix the length as the least number of lines n multiplied by $m = Line_{common} = 20$ number of terms. Therefore, code sequences are padded/truncated to fit the fixed size. The other representation method for code segments is the TF-IDF. Unlike the verification task, the identification task uses larger segments, and each exhibits a sufficient number of terms. Using TF-IDF representation, the input for the RNN model is one step sequence per sample.

Moreover, we investigate the performance of RNN models with both softmax classifier and a random forest classifier (RFC) [11]. For scalability and robustness, several works [1, 17, 22, 39] adopted RFC for code authorship identification. Therefore, we also use RFC over code sequence embeddings that are generated from trained RNN-based models. In all experiments, we construct RFC with 300 trees grown to the maximum extent. Based on the experiments, and using 300 trees is a sufficient trade-off between accuracy and efficiency.

3.7 RNN Models and Experiment Settings

Our method to capture code authorship attribution from a sequence of terms and expressions makes RNN as a prime candidate for this modeling task. RNN models are well-known to handle input sequences and capture temporal relations and distinctive patterns within the data. To this end, Multi-$\mathcal{X}$ explores the performance of different RNN structures namely, traditional simple RNN, Long Short-Term Memory (LSTM) [33], and Gated Recurrent Unit (GRU) [19]. The reason for investigating three units is that simple RNNs are efficient and capable of handling data sequences, but result in poor performance under long-range temporal dependencies in long sequences. Handling segments of code with a large number of terms could hinder the learning process of models when suffering from known conditions such as *vanishing* or *exploding* gradients [19, 33]. Thus, we extend our investigation to take advantage of the gating mechanism offered by LSTM and GRU to handle such problems. Moreover, LSTM and GRU have shown remarkable results in modeling long sequences [1]. We use RNN models for both authorship verification and identification tasks. Each model differs in purpose and structure since the output of each model is different (two softmax units in the verification models, while n-units for n-classes in the identification models). However, the general basic structure of the models includes one recurrent layer connected to a softmax layer as illustrated in Figure 6(a). In this section, we explain the model architectures adopted in this work as well as the procedure and considerations taken while training the models.

Bidirectional RNN. At each time step within the sequence, the simple RNN takes advantage of the information learned from past states in generating the current state that will also be propagated to future states. Learning sequential patterns in this way is important in many applications where the temporal component of the input data should not be ignored (e.g., real-time speech or handwriting recognition). However, for code authorship attribution, accessing the entire code sequence at once could enable not only learning from past states but also from future states. This can be achieved using bidirectional RNN, which incorporates two RNNs trained to make the output decision. The first RNN operates from



(a) One layer RNN model (b) Bidirectional RNN model (c) Multi-layer Bidirectional RNN model

Fig. 6. Different RNN model architectures used for code authorship verification and identification.

the beginning to the end of the sequence, while the other operates in the opposite direction as in Figure 6(b).

Multi-layer RNN. Deep RNNs with multiple hidden layers have shown a remarkable capability of capturing nonlinear patterns from long input sequences [52]. In this work, we also investigate the performance of Multi-χ using multi-layer RNN. Figure 6(c) shows an example of an RNN with multiple hidden layers.

Model Training and Settings. Since RNN models are parameterized, the training process aims to find appropriate parameters that enable the model to perform a given task. For authorship verification and identification, the model training is guided by minimizing the softmax cross-entropy loss between the ground-truth labels and the model output. The training process starts by initializing the model with weights drawn from a normal distribution near zero with zero-mean and small variance. Then, the optimization process is performed using the *Root Mean Square Propagation – RMSProp* [57] algorithm, which is commonly used with RNN [52]. The optimization process requires setting a learning rate that scales the entire gradient at each training step. Using a high learning rate can cause a divergence, while using a very low value can lead to a slow convergence or settling to a local optimum. In the literature, starting with a large learning rate and decreasing it over time during the training process has been an efficient way of setting the learning rate. In this work, we scale the learning rate to $\alpha_n = \alpha_c \times NI^{-\frac{1}{2}}$, where NI is the number of iterations, α_n is the new value, and α_c is the current value. We set the starting learning rate at 10^{-2} and the L2-regularization strength at 10^{-4}. To control the training process and prevent overfitting, we use the *dropout regularization* technique [54], which enables the neural network to reach better generalization capabilities. We set the dropout rate to 0.3 during the training of all RNN models. The termination criterion of the training is set to concluding 1,000 training iterations. The training hyperparameters are based on preliminary experiments on different tasks.

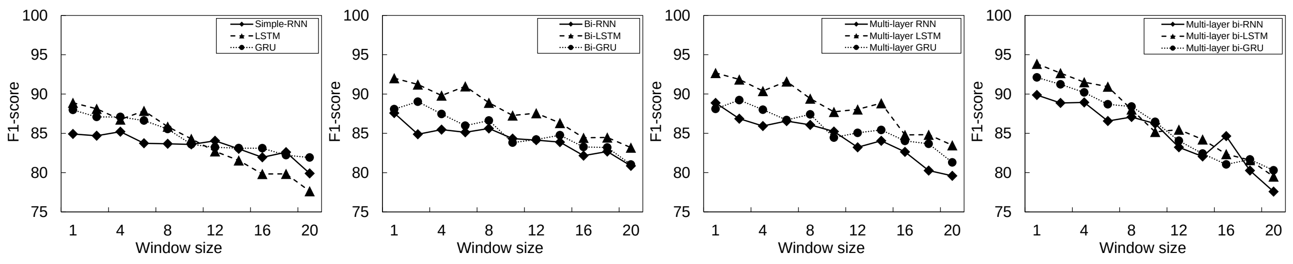
Dataset Handling and Splitting. Since we adopt a data-driven approach to obtain RNN-based models, the dataset is split into three sets, 70% for training, 15% for validation, and 15% for testing. The use of the three splits is straightforward, where the training set is used to train the model, the validation is used to validated the model during the model optimization, and the testing set is used to test the performance of the model on the targeted task. This mechanism is followed for training the RNN-models in all experiments of the authorship verification (§4.2) and segment authorship identification (§4.3) tasks. We note that the experiments in §4.2 and §4.3 aim to establish proper settings for fine-grained authorship identification approach (using end-to-end identification as in §4.4) by investigating the effects of code segment size, data representation, model structure, and experimental hyperparameters on the performed task. To this end, code segments are collected and processed based on the ground-truth dataset. The entire collection of code segments is then shuffled and split into training, validation, and testing sets.

Handling Class Imbalance. To address the class imbalance in our dataset, we use class weights (percentage) to penalize the wrong predictions and to scale the loss function during the training process.

Handling Code Segments of Different Length. Since segments consist of lines of code with a different number of terms, the resulting segments differ in length. The recurrent neural network can process sequences with different lengths, using dynamic RNN or sequence padding/truncating to a defined extent. Efficient handling of unequal input sequences may dictate using the mini-batch approach, where a number of segments are packed into a matrix of predefined dimension that becomes the dimension of the input sequences by padding short sequences or truncating long sequences. On the other hand, dynamic RNN computes gradients from one sample at a time raising the challenge of reducing the effects of the large variance of computed gradients. Thus, we adopt the mini-batch approach for efficiency since several segments are handled at once. In

Table 2. Code authorship verification: summary of results using different RNN architectures and different segment sizes

Window	LSTM			Bi-LSTM			Multi-layer LSTM			Multi-layer bi-LSTM		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1
1	82.76	96.00	88.89	86.73	98.00	92.02	90.48	95.00	92.68	93.11	94.60	93.85
2	81.66	95.70	88.12	85.55	97.70	91.22	89.17	94.70	91.85	91.27	94.10	92.66
4	79.83	95.00	86.76	83.62	97.00	89.81	87.04	94.00	90.39	89.88	93.20	91.51
6	81.87	94.80	87.86	85.82	96.80	90.98	89.50	93.80	91.60	88.62	93.40	90.94
8	78.73	94.40	85.86	82.46	96.40	88.89	85.77	93.40	89.42	84.08	92.40	88.04
10	76.42	94.00	84.30	80.00	96.00	87.27	83.04	93.00	87.74	79.31	92.00	85.19
12	73.97	93.80	82.72	80.64	95.80	87.57	83.76	92.80	88.05	79.97	91.80	85.48
14	72.44	93.30	81.56	78.89	95.30	86.32	85.62	92.30	88.84	78.17	91.30	84.23
16	69.92	93.00	79.83	76.00	95.00	84.44	78.63	92.00	84.79	75.21	91.00	82.35
18	70.12	92.70	79.84	76.25	94.70	84.48	78.92	91.70	84.83	72.88	92.70	81.60
20	67.15	92.00	77.64	74.60	94.00	83.19	77.12	91.00	83.49	70.96	90.40	79.51



(a) 1-layer RNN (b) 1-layer bi-directional RNN (c) Multi-layer RNN model (d) Multi-layer bi-direct. RNN

Fig. 7. Performance of authorship verification models with different architectures and RNN units. Notice that the performance enhances with bidirectional RNN and with more depth. All percentage are F1-score.

our experiments, code segments with K lines of code are padded/truncated to size $K \times Line_{common}$, where $Line_{common}$ is the line length threshold that most of the code lines satisfy. In our dataset, we use $Line_{common} = 20$ as illustrated in Figure 5.

4 Evaluation and Experiments

We evaluate Multi-$\mathcal{X}$ using real-world open-source code samples collected from Github. The evaluation includes the code authorship verification task and the code authorship identification task using various RNN-based models with different architectures and settings. All experiments are conducted on a workstation with 24 cores, one GeForce GTX Titan X GPU, and 128 GB of memory. The specific platform does not affect the results.

4.1 Dataset

Multi-$\mathcal{X}$ uses a real-world dataset of nine open-source projects available on Github, namely: Caffe Library, Cosmos Algorithms Collection repository [21], Dyninst API tools for binary instrumentation [13], Facebook Open-source Library (folly) [26], GNU Compiler Collection (GCC) [30], Apache HTTP Server [7], Open Source Computer Vision Library (OpenCV) [42], Swift Programming Language [8], and TensorFlow Library[56]. We use *git-author* [40] tool to collect the ground truth for authors of all projects. *Git-author* returns the author for each line of code. We process the code files to remove comments, empty lines, or files that do not have a code. After processing and cleaning all code files, the collected dataset contains 26,607 code files (84.7% C files while the rest are C++ files) with an average of 114 lines per file. The total number of authors is 2,220 programmers with an average of 1,377.9 code lines per programmer. We notice that the number of code lines per programmer is not balanced: for example, there is a programmer with 195,948 lines, while 170 other programmers have only one line of code in the entire collection of samples.

4.2 Code Authorship Verification

The purpose of this experiment is to obtain an authorship verification model that is able to distinguish segments from different authors. For this purpose, different architectures of RNN models are explored using different window sizes (i.e., lines of codes per segment). Since the verification task can be viewed as a binary classification, the results are reported using three evaluation metrics: $precision = \frac{TP}{TP+FP}$, $recall = \frac{TP}{TP+FN}$ and

F1-score$=2 \times \frac{P \times R}{P+R}$, where TP, FP, FN, P, and R are the true positive, false positive, false negative, precision, and recall respectively. Using these metrics provides a realistic evaluation of the verification model as the dataset contains unbalanced labels. For example, using a small segmentation window (e.g., one line of code) produces a dataset with a large number of subsequent segments written by the same author, and thus the positive labels are more prevalent than the negative labels. As F1-score provides a harmonic mean of precision and recall, we train the models with special emphasis on improving the recall to increase the sensitivity for negative verification. To this end, the class weights are used to weigh the loss function during the training process.

Word2vec Input Representation. For this experiment, we feed the RNN model with code segments represented as a sequence of *word2vec* representations. Segments with n lines are represented as a sequence of size $n \times Line_{common} \times d$, where d is the dimension of terms representation. For example, a segment with one line is represented as tensor of size $1 \times 20 \times 128$, since $Line_{common} = 20$ and the dimension of *word2vec* representations is 128. For the verification models, we use RNN with 64 hidden units and a maximum of two hidden layers when using multi-layers RNN architectures.

Results. Table 2 reports the verification performance of different LSTM architectures (i.e., Basic LSTM, Bi-LSTM, Multi-layer LSTM, and Multi-layer bi-LSTM) using datasets generated with different segmentation windows. The results reveal that segments with one line of code enable the best performance across different model architectures. We note that the best verification results are obtained using multi-layer bi-LSTM with F1-score of 93.85% verifying one-line segments. This can be because of the nature of the ground-truth, since labels are assigned to lines of code, making segments with multiple lines more prone to noise that hinder the verification process. This also explains the inferior results obtained with larger segments, e.g., 14.34% $(= 93.85 - 79.51)$ difference in F1-score between one-line segments and 20-lines segments.

The performance of different RNN units—simple RNN, LSTM, and GRU—is shown in Figure 7(a) using the F1-score. The results show that LSTM outperforms other units, especially when the window size is small. The bi-directional RNN shows an improvement over uni-directional RNN as in Figure 7(b). Moreover, deeper architectures with multiple layers achieve better results as illustrated in Figure 7(c) and Figure 7(d).

Table 3. Number of authors in the dataset based on the least number of samples and the minimum number of lines per sample.

		Least # of lines per sample			
		4	**6**	**8**	**10**
# of samples	6	843	730	660	608
	10	689	606	529	479
	20	525	459	393	346
	30	452	376	316	282

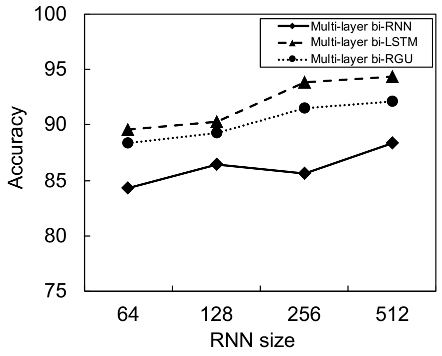


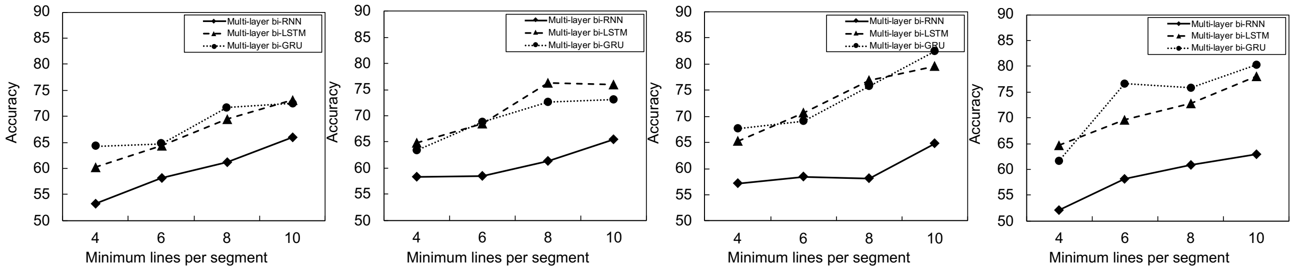
Fig. 8. Accuracy of authorship identification achieved by RFC using different *word2vec*-RNN-based embeddings sizes using a dataset of 282 programmers with at least 30 samples.

4.3 Code Authorship Identification

In this experiment of code segment authorship identification, we use the ground-truth data to collect code segments and the corresponding authors. Conducting experiments using the real ground-truth data allows us to define a baseline for the end-to-end system evaluation. Moreover, the ground-truth provides insights on the used methods at each phase and the sufficient amount of data required to achieve accurate authorship identification (e.g., the number of samples and the minimum number of code lines per sample).

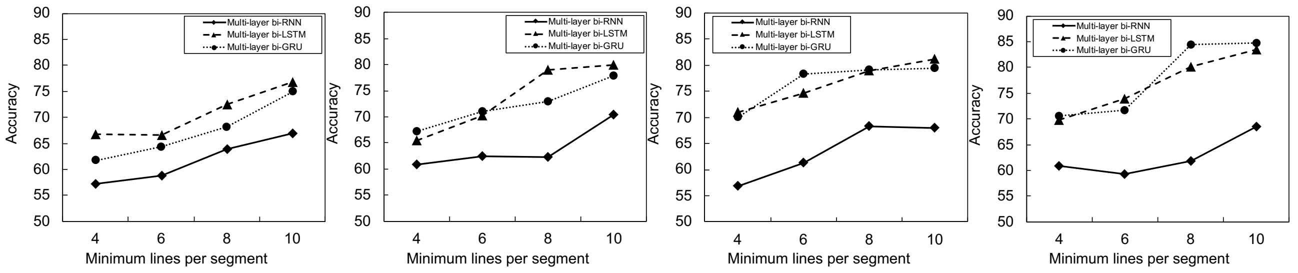
Collections of Code Samples. From the collected dataset, we created 16 subsets based on the number of samples per author of varying sample sizes. Table 3 shows the datasets used in the experiment. As we add more constraints to the dataset, the number of authors decreases. Using the datasets of Table 3, we investigate the minimum number of samples per author—6, 10, 20, and 30 samples—for successful authorship identification. Moreover, we also investigate the effect of code size, i.e., the number of code lines in a code segment, on the authorship identification. We consider segments with a minimum number of code lines, four to ten.

Model Architecture. Based on experiments and results obtained from the verification task in §4.2, we utilize two-layers of bidirectional RNN with 512 hidden units connected to a softmax layer. The experiments are conducted with traditional RNN, GRU, and LSTM



(a) At least 6 samples (b) At least 10 samples (c) At least 20 samples (d) At least 30 samples

Fig. 9. Accuracy of authorship identification for authors with at least specific number of samples with different sample sizes represented using *word2vec*. The RNN architecture is two-layers bi-LSTM with 512 units connected to a softmax classifier.



(a) At least 6 samples (b) At least 10 samples (c) At least 20 samples (d) At least 30 samples

Fig. 10. Accuracy of authorship identification for authors with at least specific number of samples with different sample sizes represented using TF-IDF. The RNN architecture is two-layers bi-LSTM with 512 units connected to a softmax classifier.

units. We use a large number of hidden units, i.e., 512 units, to allow the network to learn distinctive and high-quality sequence embeddings of authorship traits. These sequence embeddings enable the classifier, e.g., softmax classifier or RFC, to accurately identify programmers of presented code samples. For example, Figure 8 shows the results of the identification task performed by RFC using sequence embedding with different sizes. The reported results are produced using a dataset of authors with at least 30 samples with 10 lines of code as the minimum size for a sample. The results show an identification accuracy improvement when increasing the size of the embeddings, e.g., by $4.79\%(= 94.4 - 89.61)$ when increasing LSTM-based embeddings size from 64 to 512.

Experiment 1: Effects of Input Representation. We investigate the effect of using different representations of code samples on the accuracy of the proposed authorship identification task (i.e.,*word2vec* and TF-IDF). Figure 9 shows the accuracy of our identification approach using *word2vec* representation with varying samples per author and varying sample sizes. In particular, Figure 9(a) illustrates that our approach with LSTM unit is superior to other units and achieves 60.32% for 843 programmers with at least six code samples and four lines per sample. As the number of lines per sample reaches 10, we achieve an accuracy of 73.16% for 608 programmers. We also illustrate the impact of the number of samples per author on the performance of the identification process. Using LSTM, Figure 9(b) shows an accuracy of 75.94% for 479 programmers when the number of samples is at least ten, with at least ten lines per sample. This accuracy increases to 79.64% for 346 programmers when the number of samples is doubled as shown in Figure 9(c). However, the accuracy slightly decreases when increasing the number of samples to exceed 30 samples per author. Figure 9(d) shows an accuracy of 78.12% for 282 programmers using LSTM. This decrease can be explained by the fixed training process for all experiments, where the intuitive procedure for training a model with a large dataset required more time and a deeper architecture.

Using TF-IDF representations with the same experimental settings, Figure 10 shows the impact of using TF-IDF representation on the accuracy with a varying number of samples per author and varying sample sizes. For instance, Figure 10(a) shows that our approach with LSTM unit is superior to other units and achieves 66.84% with at least six code samples and four lines per sample. As the number of lines per sample reaches to 10, we achieve an accuracy of 76.87%. We also illustrate the impact of the number of samples per author on the performance of the identification process. Using LSTM, Figure 10(b) shows an accuracy of 79.88% when the number of samples is at least ten and when there are at least ten lines per sample. This accuracy increases to 81.11% when the number of samples is doubled as shown in Figure 10(c). The best accuracy reaches 83.45% for 282 programmers when using LSTM.

Key Insight. The number of samples per programmer influences the accuracy of identification, i.e., more samples means higher identification accuracy. However, the RNN model seems to learn authorship attributions even with a small number of samples, e.g., ten samples. Also, input representation affects the accuracy, i.e., TF-IDF shows better performance than *word2vec*.

Experiment 2: Identification with RFC. We conduct this experiment using the same setting as in experiment 1. However, instead of relying on the softmax classifier, we use the sequence embeddings generated by the RNN model to construct an RFC classifier with 300 trees grown to the maximum extent. We construct the RFC classifiers using the sequence embeddings of the same training dataset used for training the RNN-based models. The RFC models are then evaluated using the sequence embeddings of the test dataset. Similar to experiment 1, two different initial representations methods, namely, *word2vec* and TF-IDF, are used in this experiment. Using *word2vec* as the initial code representation, Figure 11 shows the identification accuracy of RFC over *word2vec*-based sequence embeddings generated with different RNN types. Figure 11(a) shows the accuracy of different RNN types when the least number of samples per author is six. Using sequence embeddings generated by LSTM enabled the best accuracy, regardless of the sample size, as it achieves 84.64% accuracy for 843 programmers. The improvement in accuracy becomes clearer as the minimum lines per sample exceed ten lines of code to reach 90.61% for the available 608 programmers. Figure 11(b) shows an accuracy of 92.12% for 479 programmers when the number of samples is at least ten with at least ten lines per sample. This accuracy increases to 92.87% for 346 programmers when the number of samples is doubled as shown in Figure 11(c). A similar improvement of accuracy is achieved when we use more than 30 samples per author to reach 94.4% as shown in Figure 11(d).

When using TF-IDF as our initial representation, the sequence embeddings seem to capture more distinctive features of the code samples. This can be shown by the obvious improvement of the obtained results illustrated in Figure 12. In Figure 12(a), LSTM-based sequence embeddings with RFC achieve 86.84% for 843 programmers. Compared to results achieved using *word2vec*-based sequence embeddings, the improvement in accuracy is 2.2% ($= 86.84 - 84.64$). When considering the minimum number of lines per sample, the accuracy reaches 91.24% with samples of more than ten lines. Figure 12(b) shows an accuracy of 92.82% when the number of samples is at least ten and with at least ten lines per sample. This accuracy increases to 95.12% when the number of samples is doubled as shown in Figure 12(c). The TF-IDF-based method seems to generate more robust sequence embedding than the ones generated by the *word2vec*-based method. This can be clearly seen in Figure 12(d) as it reaches to an accuracy of 96.14% when considering at least 30 samples per author since the accuracy improvement reaches 1.74% ($= 96.14 - 94.4$).

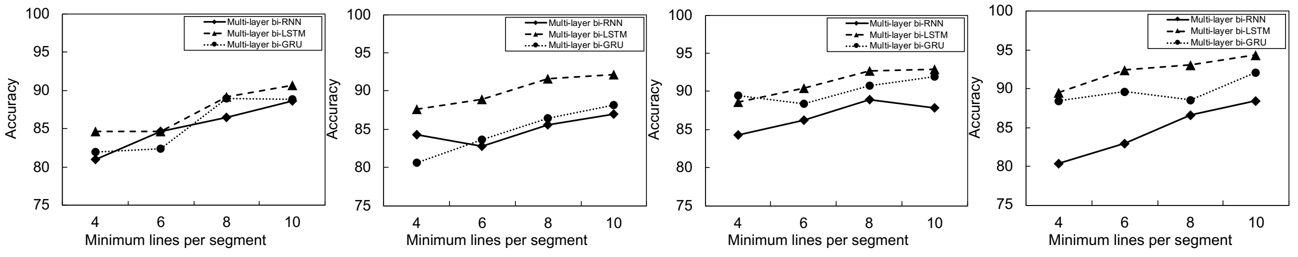
Key Insight. The reported results of this experiment show the impact of using robust classifier such as the RFC. For instance, the improvement of the achieved accuracy in identifying programmers for *word2vec*-based sequence embeddings using RFC compared to the softmax classifier is 16.28% ($= 94.4 - 78.12$) when considering programmers with at least 30 samples of minimum ten lines of code. Similarly, the improvement of achieved accuracy in identifying programmers for TF-IDF-based sequence embeddings using RFC compared to softmax classifier is 12.69% ($= 96.14 - 83.45$) when considering authors with at least 30 samples of minimum ten lines.

4.4 End-To-End Identification

In this experiment, we make use of the observations learned from previous experiments to design an overall system evaluation of Multi-$\mathcal{X}$. The main purpose of this evaluation is to demonstrate Multi-$\mathcal{X}$'s effectiveness in general rather than its accuracy on an individual task. For this evaluation, Multi-$\mathcal{X}$ should operate through the five stages, code segmentation, code representation, segment authorship verification, segment integration and finally authorship identification. The final end-to-end system evaluation depends on the performance of all incorporated stages, e.g., the segment verification plays an important role in segment integration that itself influences the identification.

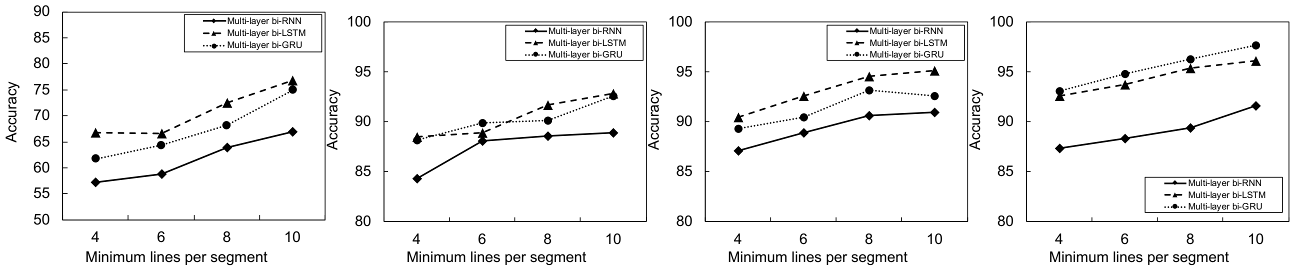
Experiment Settings. The setting of the experiment in a distinct stage is outlined as follows:

1. Code Segmentation: we use a one-line window.
2. Code Representation: firstly, code segments are represented as a sequence of 128-dimensional *word2vec* representations for the authorship verification task, where the sequence length equals to $Line_{common}$. Secondly, the integrated code segments, generated by accumulating positively verified code segments of a



(a) At least 6 samples (b) At least 10 samples (c) At least 20 samples (d) At least 30 samples

Fig. 11. Accuracy of authorship identification for authors with at least specific number of samples with different sample sizes represented using *word2vec*. The results are achieved by a RFC constructed using sequence embeddings generated from a trained two-layers bi-LSTM model with 512 units in each layer.



(a) At least 6 samples (b) At least 10 samples (c) At least 20 samples (d) At least 30 samples

Fig. 12. Accuracy of authorship identification for authors with at least a specific number of samples with different sample sizes represented using TF-IDF. The results are achieved by RFC constructed using sequence embeddings generated from a trained two-layers bi-LSTM model with 512 units in each layer.

programmer, are represented with the top-2,500 TF-IDF features for the identification task.

3. Code Authorship Verification: we use a two-layer bi-LSTM model with 64 hidden units for each layer. The verification models are trained from scratch.
4. Code Segment Integration: using the verification model, we go through the testing code files line by line integrating segments in an incremental manner.
5. Code Authorship Identification: we use a two-layer bi-LSTM with 512 units for each layer. The identification models are trained from scratch using the integrated segments produced by the verification. The identification models are fed with TF-IDF representations of the integrated segments and generate deeper representations of authorship attributions. Using deep representations of integrated segments, we construct RFC with 300 trees for identification.

Evaluation Metric. The traditional definition of accuracy, which corresponds to the exact prediction of tested samples (guess-all authors per file), can be inefficient in describing the level of correctness of our approach in identifying the authors of a source-code file [23, 43]. Therefore, the evaluation of the overall system performance can be calculated using a similar metric as the example-based accuracy (EBA) used in [23], which corresponds to the average correctness of the author assignments per code file. Since our system uses a fine-grained approach instead of a multi-labeled example, we use the average sum of per-segment identification accuracy and the author assignment accuracy for each code file. We call this evaluation metric as Authorship EBA (A-EBA), which we define as follows:

$$A\text{-}EBA = \frac{1}{2n}\sum_{i=1}^{n} s\_A_i + a\_A_i$$

where, (1) s_A_i is the per-segment authorship identification accuracy, defined as the proportion of correctly attributed segments in all tested segments for the sample file i. (2) a_A_i is the authors per example accuracy, defined as the proportion of correctly assigned authors in the total number of authors of example i. (3) n is the total number of tested code files. Using a_A_i or s_A_i separately does not provide high confidence in the overall system predictions. For example, consider a file with four segments and two authors; if three segments are correctly attributed then s_A_i is 75%, and the a_A_i can be either 50 or 100% depending on whether one or two authors are identified resulting in A-EBA of 62.5% or 87.5% for the two cases, respectively. Therefore, averaging the two measures can provide a better understanding of the system performance.

Results. For a real-world scenario, we run the evaluation on all code files in the testing set without altering or omitting little code contributions, i.e., removing authors with few code segments. Therefore, we split the dataset to 80% training set and 20% testing set, resulting in 21,286 sample files in the training set and 5,321 sample files in the testing set. Since the code files are randomly selected for the testing set, this might result in including files written (partially or entirely) by programmers who do not contribute to any sample in the training set. We exclude those programmers since attempting to identify them is an open-world problem, which is out of the scope of this work. The splitting of the dataset resulted in obtaining 617 programmers in the testing set, and only 562 have appeared in the training set.

We run the evaluation ten times and report the average result. For 562 programmers, Multi-$\mathcal{X}$ achieved an A-EBA of 86.41% and an overall per segment authorship identification of 93.18% and authors per example accuracy of 79.62%. Investigating the results further, we found that most misattributed segments are less than three lines of code. This misattribution of code segments factored on the authors per example accuracy, as authors with little contributions, e.g., one to three lines of code, on a given code file are very difficult to be identified. Moreover, it also becomes more challenging when the total number of contributions for an author is very small, e.g., less than six segments of code in the training data, which makes it hard for the classifier to learn distinct features for such an author. For example, in this experiment, the testing files include 109 programmers who have less than six samples in the training data. However, the proposed fine-grained approach achieved remarkable results with the utilization of term distributions and representations, deep learning, sequence embeddings, and ensemble classifiers.

5 Limitations

This work demonstrates that sufficient authorship attributions can be extracted from the smallest piece of code to enable accurate authorship identification. Nevertheless, Multi-$\mathcal{X}$ has several limitations concerning the ground-truth data used in the evaluation, dealing with binary code, and obfuscated code.

Ground-Truth Assumption. This work assumes authorship of code lines based on the *git-author* [40] tool. This means authorship is assigned to the Github committer of the project regardless of any consideration of other authors who worked offline in the submitted project. Thus, working with early commits of a project might not always allow authentic authorship. The continuous improvements and the dynamics of open-source projects enable the collaboration among authors and reduce the ramifications of ground-truth error. We chose nine open source projects with 2,220 programmers, with an average of 1,378 code lines per programmer.

Binary Code. Previous work [18] showed that a pseudo-code generated from the decompilation process of a binary can possess authorship traits of the binary program. The experiments were reported using a dataset with a single author per program, which simplifies the authorship assignment for the decompiled code. However, assigning multiple authors for a piece of decompiled code is very challenging, and for the best of our knowledge, there have been no previous attempts to map multiple authors to decompiled pseudo-code. We leave this investigation as future work.

Obfuscated Code. Previous work [1] showed that deep learning representation enabled accurate code authorship identification for obfuscated code. Another work by Brennan *et al.* [12] has studied adversarial stylometry to evaluate the performance of authorship identification when adversaries attempt to evade identification by hiding or impersonating another identity. We acknowledge such a limitation, and leave studying the effects of obfuscation or adversarial scenarios on identifying multiple authors of source code as future work.

6 Conclusion

We have proposed Multi-$\mathcal{X}$, a fine-grained approach for multi-author identification from source codes incorporating several techniques: code representation, recurrent neural networks, and ensemble classifiers. To the best of our knowledge, our work is the first to attempt at identifying multiple authors of a single source file from a real-world dataset collected in the wild (from Github), and in identifying authors line-by-line in source code. For the evaluation of Multi-$\mathcal{X}$, we have used a large scale dataset including nine real-world open-source projects. Multi-$\mathcal{X}$ achieves an authorship example-based accuracy of 86.41% and per-segment authorship identification of 93.18% for 562 programmers. Our results show that programmers' coding style is distinguishable even with small fractions of codes, and it is possible to identify multiple authors in single source code. We leave other representation techniques of code terms for higher identification accuracy for future investigation.

Acknowledgement. We would like to thank the anonymous reviewers, and our shepherd Rachel Greenstadt, for their feedback and helpful comments. This work was supported NRF grant 2016K1A1A2912757.

References

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

[2] Mohammed Abuhamad, Ji-su Rhim, Tamer AbuHmed, Sana Ullah, Sanggil Kang, and DaeHun Nyang. Code authorship identification using convolutional neural networks. *Future Generation Computer Systems*, 95:104–115, 2019.

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

[4] Abeer Alhuzali, Rigel Gjomemo, Birhanu Eshete, and VN Venkatakrishnan. Navex: precise and scalable exploit generation for dynamic web applications. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 377–392. USENIX Association, 2018.

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

[6] Bander Alsulami, Edwin Dauber, Richard Harang, Spiros Mancoridis, and Rachel Greenstadt. *Source Code Authorship Attribution Using Long Short-Term Memory Based Networks*, pages 65–82. Springer International Publishing, Cham, 2017.

[7] Apache. Apache http server mirror. https://github.com/apache/httpd, 2018. Accessed: 2018-05-04.

[8] Apple. Swift programming language, 2018. Accessed: 2018-05-04.

[9] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155, 2003.

[10] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146, 2017.

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

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

[13] Bryan Buck and Jeffrey K. Hollingsworth. An api for runtime code patching. *Int. J. High Perform. Comput. Appl.*, 14(4):317–329, November 2000.

[14] S. Burrows, A. L. Uitdenbogerd, and A. Turpin. Temporally robust software features for authorship attribution. In *2009 33rd Annual IEEE International Computer Software and Applications Conference*, volume 1, pages 599–606, Seattle, WA, USA, 2009. IEEE.

[15] Steven Burrows, S. M. M. Tahaghoghi, and Justin Zobel. Efficient plagiarism detection for large code repositories. *Softw. Pract. Exper.*, 37(2):151–175, February 2007.

[16] Steven Burrows, Alexandra L. Uitdenbogerd, and Andrew Turpin. Comparing techniques for authorship attribution of source code. *Software: Practice and Experience*, 44(1):1–32, 2014.

[17] Aylin Caliskan-Islam, Richard Harang, Andrew Liu, Arvind Narayanan, Clare Voss, Fabian Yamaguchi, and Rachel Greenstadt. De-anonymizing programmers via code stylometry. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, pages 255–270, Berkeley, CA, USA, 2015. USENIX Association.

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

[19] KyungHyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *CoRR*, abs/1409.1259, 2014.

[20] Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *the 25th international conference on Machine learning*, pages 160–167. ACM, 2008.

[21] Cosmos. Cosmos algorithms collection. https://github.com/OpenGenus/cosmos/, 2018. Accessed: 2018-05-04.

[22] Edwin Dauber, Aylin Caliskan, Richard Harang, Gregory Shearer, Michael Weisman, Frederica Nelson, and Rachel Greenstadt. Git blame who?: Stylistic authorship attribution of small, incomplete source code fragments. *Proceedings on Privacy Enhancing Technologies*, 2019(3):389 – 408, 2019.

[23] Edwin Dauber, Rebekah Overdorf, and Rachel Greenstadt. Stylometric authorship attribution of collaborative documents. In *International Conference on Cyber Security Cryptography and Machine Learning*, pages 115–135. Springer, 2017.

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

[25] Bruce S. Elenbogen and Naeem Seliya. Detecting outsourced student programming assignments. *J. Comput. Sci. Coll.*, 23(3):50–57, January 2008.

[26] Facebook. Facebook open-source library (folly). https://github.com/facebook/folly, 2018. Accessed: 2018-05-04.

[27] David Fifield, Torbjørn Follan, and Emil Lunde. Unsupervised authorship attribution. *arXiv preprint arXiv:1503.07613*, 2015.

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

[29] Georgia Frantzeskou, Efstathios Stamatatos, Stefanos Gritzalis, and Sokratis Katsikas. Effective identification of source code authors using byte-level information. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 893–896, 2006.

[30] GCC. Gnu compiler collection (gcc). https://github.com/gcc-mirror/gcc, 2018. Accessed: 2018-05-04.

[31] Chris Giannella. An improved algorithm for unsupervised decomposition of a multi-author document. *Journal of the Association for Information Science and Technology*, 67(2):400–411, 2016.

[32] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Domain adaptation for large-scale sentiment classification: A deep learning approach. In *the 28th international conference on machine learning (ICML-11)*, pages 513–520, 2011.

[33] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

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

[35] Ivan Krsul and Eugene H. Spafford. Refereed paper: Authorship analysis: Identifying the author of a program. *Comput. Secur.*, 16(3):233–257, January 1997.

[36] Robert Charles Lange and Spiros Mancoridis. Using code metric histograms and genetic algorithms to perform author identification for software forensics. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07, pages 2082–2089, 2007.

[37] S. G. Macdonell, A. R. Gray, G. MacLennan, and P. J. Sallis. Software forensics for discriminating between program authors using case-based reasoning, feedforward neural networks and multiple discriminant analysis. In *6th International Conference on Neural Information Processing, ICONIP '99.*, volume 1, pages 66–71, Australia, 1999. IEEE.

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

[39] Xiaozhu Meng, Barton P Miller, and Kwang-Sung Jun. Identifying multiple authors in a binary program. In *European Symposium on Research in Computer Security*, pages 286–304, Oslo, Norway, 2017. Springer.

[40] Xiaozhu Meng, Barton P Miller, William R Williams, and Andrew R Bernat. Mining software repositories for accurate authorship. In *29th IEEE International Conference on Software Maintenance (ICSM)*, pages 250–259. IEEE, 2013.

[41] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems 26*, pages 3111–3119. Curran Associates, Inc., 2013.

[42] OpenCV. Open source computer vision library (opencv). github.com/opencv/opencv, 2018. Accessed: 2018-05-04.

[43] Mathias Payer, Ling Huang, Neil Zhenqiang Gong, Kevin Borgolte, and Mario Frank. What you submit is who you are: a multimodal approach for deanonymizing scientific publications. *IEEE Transactions on Information Forensics and Security*, 10(1):200–212, 2015.

[44] Giancarlo Pellegrino, Martin Johns, Simon Koch, Michael Backes, and Christian Rossow. Deemon: Detecting csrf with dynamic analysis and property graphs. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1757–1771. ACM, 2017.

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

[46] Jeffrey Pennington, Richard Socher, and Christopher Manning. Glove: Global vectors for word representation. In *the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[47] Dylan Rhodes. Author attribution with cnns. *Avaiable online: https://www. semanticscholar. org/paper/Author-Attribution-with-Cnn-s-Rhodes/0a904f9d6b47dfc574f681f4d3b41bd840871b6f/pdf (accessed on 22 August 2016)*, 2015.

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

[49] Raheem Sarwar, Chenyun Yu, Sarana Nutanong, Norawit Urailertprasert, Nattapol Vannaboot, and Thanawin Rakthanmanon. A scalable framework for stylometric analysis of multi-author documents. In *International Conference on Database Systems for Advanced Applications*, pages 813–829. Springer, 2018.

[50] Michael Schwarz, Moritz Lipp, and Daniel Gruss. Javascript zero: Real javascript and zero side-channel attacks. *Network and Distributed System Security Symposium (NDSS)*, 2018.

[51] Holger Schwenk. Continuous space language models. *Computer Speech & Language*, 21(3):492–518, 2007.

[52] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. Recognizing functions in binaries with neural networks. In *24th USENIX Security Symposium*, pages 611–626, Washington, D.C., 2015. USENIX Association.

[53] Richard Socher, Cliff C Lin, Chris Manning, and Andrew Y Ng. Parsing natural scenes and natural language with recursive neural networks. In *the 28th international conference on machine learning (ICML-11)*, pages 129–136, 2011.

[54] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, January 2014.

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

[56] TensorFlow. Tensorflow library for numerical computation. github.com/tensorflow/tensorflow/. Accessed: 2018-05-04.

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

[58] Mikolov Tomas, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In *Proceedings of Workshop at International Conference on Learning Representations 2013*, Arizona, USA, May 2013.

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