Source Code Authorship Attribution Using Long Short-Term Memory Based Networks

Bander Alsulami[1(✉)], Edwin Dauber[1(✉)], Richard Harang[2], Spiros Mancoridis[1], and Rachel Greenstadt[1]

[1] Drexel University, Philadelphia, USA
{bma48,egd34,spiros,rachel.a.greenstadt}@drexel.edu
[2] Sophos, Abingdon, UK
richard.harang@sophos.com

Abstract. Machine learning approaches to source code authorship attribution attempt to find statistical regularities in human-generated source code that can identify the author or authors of that code. This has applications in plagiarism detection, intellectual property infringement, and post-incident forensics in computer security. The introduction of features derived from the Abstract Syntax Tree (AST) of source code has recently set new benchmarks in this area, significantly improving over previous work that relied on easily obfuscatable lexical and format features of program source code. However, these AST-based approaches rely on hand-constructed features derived from such trees, and often include ancillary information such as function and variable names that may be obfuscated or manipulated.

In this work, we provide novel contributions to AST-based source code authorship attribution using deep neural networks. We implement Long Short-Term Memory (LSTM) and Bidirectional Long Short-Term Memory (BiLSTM) models to automatically extract relevant features from the AST representation of programmers' source code. We show that our models can automatically learn efficient representations of AST-based features without needing hand-constructed ancillary information used by previous methods. Our empirical study on multiple datasets with different programming languages shows that our proposed approach achieves the state-of-the-art performance for source code authorship attribution on AST-based features, despite not leveraging information that was previously thought to be required for high-confidence classification.

Keywords: Source code authorship attribution · Code stylometry · Long short-term memory · Abstract syntax tree · Security · Privacy

1 Introduction

Source code authorship attribution has demonstrated to be a valuable instrument in multiple domains. In legal cases, lawyers often need to dispute source code partnership conflicts and intellectual property infringement [6,28,57].

© Springer International Publishing AG 2017
S.N. Foley et al. (Eds.): ESORICS 2017, Part I, LNCS 10492, pp. 65–82, 2017.
DOI: 10.1007/978-3-319-66402-6_6

In educational institutions, detecting plagiarisms among students' submitted assignments is a growing interest [14,49]. In software engineering, source code authorship attribution is used to study software evolution through dynamic updates [26,36]. Source code stylometry is also used for code clone detection, automatic re-factorization, complexity measurement, and code design patterns enforcement [1,4,11,24,27,55]. In computer security, source code authorship attribution can be used to identify malware authors in post-incident forensic analysis [31,32]. Research has shown that syntactical features from the original source code can be recovered from decompiling the binary executable files [8]. However, building a profile for malware authors is still a challenging problem due to the lack of ground truth code samples. In the privacy domain, the ability to identify the author of anonymous code presents a privacy threat to some developers. Programmers might prefer to maintain their anonymity for certain security projects for political and safety reasons [7,8]. Even small contributions to public source code repositories can be used to identify the anonymous programmers [12]. Recent advances in source code stylometry comes from hand-crafted AST-based features.

This paper presents our contributions to source code authorship attribution using AST-based features. We demonstrate that our LSTM-based neural network models, that require only the structural syntactic features of the AST as input, learns improved features that substantially improve upon the performance of manually constructed ones. We measure the generalization of our models on different datasets with different programming languages. We also show the classification accuracy and performance scalability of our models on a large number of authors. The remainder of this paper is organized as follows: Sect. 2 describes the related work that is relevant to source code authorship attribution. Section 3 describes common obfuscation techniques used in source code. Section 4 describe background information about the AST features and the neural network models used by our models. The model architecture and the algorithm used for learning the feature of AST are described in Sect. 5. The experimental setup, training and testing data, and the evaluation of the results are described in Sects. 6 and 7. Section 8 summarizes our conclusions and potential future work.

2 Related Work

Source code authorship attribution is inspired by the classic literature authorship attribution problem. While natural languages have more flexible grammatical rules than programming languages, programmers still have a large degree of flexibility to reveal their distinguishing styles in the code they write. For example, experienced programmers exhibit different coding styles than exhibited by novice programmers [7]. Early work uses plain textual features of the source code to identify the authors of the source code. A popular feature extraction technique is using N-grams to extract the frequency of sequences of n-characters from the source code. N-gram techniques approach source code authorship attribution as a traditional text classification problem with the source code files as

text documents [15]. Other works use layout and format features of the source code as metrics to improve the accuracy of the authors' classification. Layout features include the length of a line of code, or the number of spaces in a line of code, and the frequency of characters (underscores, semicolons, and commas) in a line of code. Researchers often measure the statistical distributions, frequencies, and average measurements of the layout features [14]. For instance, some researchers use the statistical distribution of the length of lines, number of leading spaces, underscores per line, semicolons, commas per line, and words per line as discriminative features. They use Shannon's entropy to highlight important features, and a probabilistic Bayes classifier to identify the authors [28,41].

Latter work expands on source code features to lexical and style features to avoid the limitation of format features. Lexical features are based on the tokens of the source code for a particular programming language grammar. A token can be an identifier, function, class, keyword, or a language-specific symbol such as a bracket. The naming convention for classes, functions, and identifiers can also be used as lexical features. The naming convention feature has shown success in authorship identification [7,14,29,52]. For instance, researchers use the average length of variable names, the number of variables, the number of *for* loop statements and the total number of all loop statements in a feature set, and use C4.5 decision trees to detect outsourced student programming assignments [14]. Other work combines 6-grams of source code tokens such as keywords and operators with multiple similarity measurement methods to create a profile for students based on their submitted C/C++/Java source code files [49].

Recently, syntactic features, have shown significant success in source code authorship attribution [7,29,52]. The main syntax feature derived from source code is the Abstract Syntax Tree (AST). Syntactic features avoid many defects related to format and style features. For instance, ASTs capture the structural features of the source code regardless of the source code format or the development environment used for the writing of the code. AST-based features have been used to detect partial clones in C source code programs [29]. In that paper, the authors extract an AST tree for each program and then create a hash code for each subtree. Subtrees with similar hash values are grouped together to reduce the storage requirement and improve the speed of the code clone detection.

Previous studies combine different types of features to improve the accuracy of source code authorship attribution. Some early works combine format and lexical features and implement a feature selection technique to remove the least significant features [14,49]. Recent works use a large variety of format, lexical, and syntactic features, and use an Information gain and Random Forest ensemble to select the most important features to identify the authors of a source code file [7,52]. Because of the large number of features, the feature selection process becomes critical in the model's performance for source code authorship attribution. Our work is different from these efforts primarily in that we focus on identifying the authors of source code using only the abstract structure of the AST. We ignore the format and lexical features of source code. We also discard the attributes in the AST nodes such as identifiers, numeric constants, string

literals, and keywords. We avoid the hand-tuned feature engineering process by building deep neural network models that automatically learn the efficient feature representations of the AST. By using only AST features, we aim to build source code authorship attribution models that are resilient against source code obfuscation techniques, and are language-independent so that they can be automatically extended to programming language that supports AST.

3 Source Code Obfuscation

Obfuscation is the process of obscuring source code to decrease a human's ability to understand it. Programmers may use obfuscation to conceal parts of its functionality from a human or computer analysis. For instance, malware authors use obfuscation techniques to hide the malicious behavior of their programs and avoid detection from static malware detection [3,35]. Obfuscation also decreases the usability of reverse-engineering binary executable files. Commercial software might use obfuscation to increase the difficulty of reverse engineering their software and protect their software licensing [43].

Trivial source code obfuscation techniques can easily obscure the format features of the source code. For instance, they may remove/add random text to comment sections. They may also randomly eliminate the indentations and spaces in the source code files. Modern IDEs format source code file content based on particular formatting conventions. This results in a consistent coding style across all source code written using the same development tools. This reduces the confidence of using format features to identify the authors of source code. Advanced obfuscation tools target more sophisticated features such as lexical and style features of the source code. For example, variable, function, and class names can be changed to arbitrary random names that are hard to be interpreted by a human. Stunnix[1], an obfuscation tool for programs written in C/C++ languages, uses a cryptographic hash function to obfuscate identifier names, a hexadecimal encoding function to obfuscate strings literals, a random generation function to obfuscate source code file names. ProGuard[2], an obfuscation tool for Java, uses random names for classes, methods, identifiers, configuration files, and libraries.

Despite efforts to harden program source code from static analysis using various obfuscation techniques, the semantics of the program remain the same. That is, the structure of the AST and the control flow of the program remain largely intact. Control flow obfuscation techniques work on low-level machine code and incur performance and storage overhead [2]. This leads developers to use trivial obfuscation techniques without affecting the performance of their programs. Therefore, inferring a programmer's coding styles using structural features of an AST is more robust and resilient to most automatic obfuscation techniques. Obfuscating the syntactic features of the source code of a high-level programming language while preserving the program's behavior requires code refactorization. Fully automated code refactorization suffers from reliability issues which makes

[1] http://stunnix.com/prod/cxxo/.
[2] https://www.guardsquare.com/en/proguard.

it inefficient and unfeasible in most cases [9,33]. Code refactorization requires human interference to guarantee the correctness of the refactorization process.

4 Abstract Syntax Tree

An Abstract Syntax Tree (AST) is a tree that represents the syntactic structure of a program's source code written in a programming language. An AST is an abstract representation of the source code that omits information such as comments, special characters, and string delimiters. Each AST node has a specific type and might hold auxiliary information about the source code such as an identifier's name and type, string literals, and numeric values. Nodes in an AST can have multiple children that represent the building blocks of a program.

An AST is constructed by the compiler in the early stages of the compilation process. It represents information about the source code that is needed for later stages such as semantic analysis and code generation. Therefore, an AST contains no information about the format of the source code. Integrated Development Environments (IDEs) and source code editors enforce conventional formatting and naming conventions to improve the readability of source code. In the context of authorship identification, code formatting tools might contaminate and negatively affect the source code formatting features. In contrast, ASTs are less prone to the influence of development tools and can capture the programmer's coding style directly. Therefore, it is more reliable for authorship identification techniques to analyze a program using its AST rather than its source code.

Figures 1 and 2 show a code example in Python and its corresponding AST. *Module* represents the root node of the AST and has two child nodes: *FunctionDef* and *Expr*. Each node in the AST has a label that specifies a code block in the source code. Some AST nodes such as *Name* and *Num* have extra attributes (*square*) and a numeric constant (*2*), respectively. AST nodes often have a variable number of children depending on their type and context in the source code. For instance, *Call* nodes, in this example, have two children because function *square* is declared with only one argument. However, in other contexts, *Call* can have more than two child nodes when the function is declared with more than one parameter.

AST is tree-structured data that requires models that naturally operates on trees to extract useful features of the AST representation. Feature extraction techniques such as n-grams are limited and lose information over long-distance dependencies [42,58]. While a tree-like variant of the Long Short-Term Memory (LSTM) such as Tree-Structured Long Short-Term Memory Networks (Tree-LSTM) and Long Short-Term Memory Over Recursive Structures (S-LSTM) seem intuitive, the nature of ASTs, which often have a large number of child nodes in each subtree, presents a challenge for Tree-LSTM and S-LSTM implementations [47,59]. Tree variant networks have shown to be successful in modeling tree structure data with fixed number of children [30,47,59]. Long Short-Term Memory (LSTM) networks are a unique architecture of Recurrent Neural

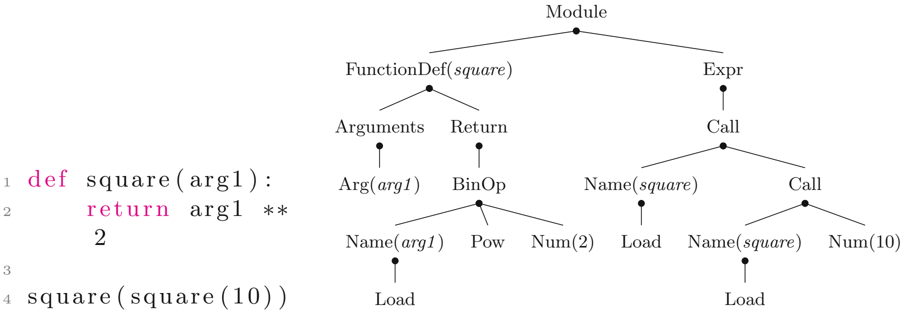

Fig. 1. Python code example **Fig. 2.** Abstract Syntax Tree for Python code example

Networks (RNN) [16,20,25]. An LSTM network has an internal state that allows it to learn the dynamic temporal behavior of long sequences over time. LSTM-based networks differ in architecture based on gate connections and information propagation. One successful architecture used for sequence classifications is the Bidirectional LSTM (BiLSTM). In contrast to the standard unidirectional LSTM, BiLSTM processes sequences in two different directions: forward and backward. Therefore, at each time step, the BiLSTM network has access to the past and future information.

5 Model Architecture

Our models traverse an AST using a *Depth First Search* algorithm. The model starts from the root node (the top node) of the AST and recursively examines all its inner nodes (nodes that have children) until it reaches a leaf node (a node with no child). An Inner node along with its children nodes is called a subtree. Therefore, an AST can be viewed as a root node with multiple subtrees. The model passes the leaf node to the *Embedding Layer* to generate a vector representation of that node. This process continues recursively for all the nodes in the AST. When all the vector representations of a subtree's nodes are retrieved, the model passes the subtree vectors to the *Subtree Layer*. The *Subtree Layer* encodes the subtree and returns a vector representation of that subtree. The model continues to encode each subtree as a vector, eventually, the AST is reduced into a final state vector representation that is passed into the final layer of the model (*Softmax Layer*). The *Softmax Layer* returns the predicted author for the AST. Algorithm 1 shows how to integrate the three layers in our models to learn the structural syntactic features of ASTs. The following subsections explain each layer's role in our model.

Algorithm 1. The Algorithm to learn the structural syntactic features of an AST.

1: **procedure** DFS(ast)
2: $count \leftarrow$ Number of children in ast
3: **if** $count = 0$ **then**
4: **return** $EmbeddingLayer(ast)$
5: **end if**
6: $tree_{vec} \leftarrow EmptyTree()$
7: **for** $i \leftarrow 1, count$ **do**
8: $tree_{vec}.child[i] \leftarrow DFS(ast.child[i])$
9: **end for**
10: $tree_{vec}.root \leftarrow EmbeddingLayer(ast.root)$
11: **return** $SubtreeLayer(tree_{vec})$
12: **end procedure**

5.1 Embedding Layer

The Embedding Layer maps individual AST nodes to their corresponding embedding vector representations. An embedding vector is a continuous fixed-length real-valued vector that can be trained with other parameters in the model. The number of embedding vectors defined in the model is equivalent to the number of unique nodes in the AST. The layer uses the node label to look up its corresponding embedding vector. Embedding representations have shown to improve the generalization of neural networks to multiple complex learning tasks [34,39,44,51].

5.2 Subtree Layer

The Subtree Layer encodes each subtree into a single vector representation. When the layer receives a subtree and its vector representation, the layer flattens the subtree into a sequence. That is, the layer processes the subtree sequentially in a pre-order fashion. Therefore, the root of the subtree is the first node in the sequence and the rest of the child nodes in the subtree are placed in the sequence from left to right. *Subtree Layer* can be implemented with any RNN architecture. In our work, we use LSTM and BiLSTM architectures and name them Subtree LSTM and Subtree BiLSTM, respectively.

Subtree LSTM processes the sequence of vector representations in a forward direction. The last hidden state in the sequence is used as a vector representation of the subtree. Subtree LSTM applies dropout on that hidden state, and propagates the results to the higher subtree. Subtree LSTM also resets its memory state before processing the next sequence. In the case of multi-layer Subtree LSTMs, the lower layer passes the hidden state vector of each time step, after applying dropout, as an input to the higher layer.

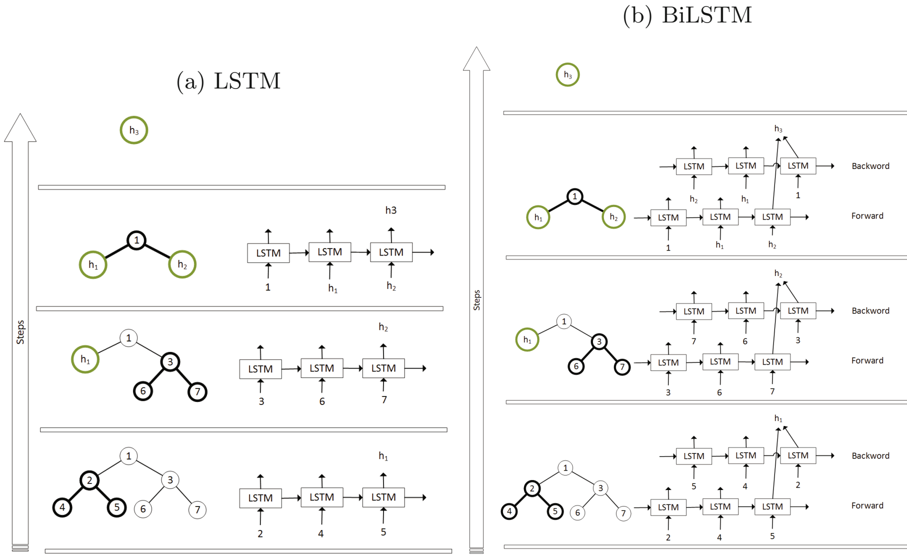

Fig. 3. An example of how the Subtree LSTM and the Subtree BiLSTM layers encode an AST.

Subtree BiLSTM processes subtrees as two sequences in two different directions. Similar to Subtree LSTM, the first sequence is processed forward from left to right. However, the second sequence is processed in backward, from right to left. The hidden states resulting from the forward and the backward passes are concatenated to generate a new vector representation that is used as an input for the next step. In the case of multi-layer BiLSTM Subtree, the lower layer passes the hidden states, after applying dropout, as an input to the higher layer at each step. The last hidden state of the highest layer is the final vector representation of the subtree.

Figure 3 gives an example on how the Subtree LSTM and the Subtree BiLSTM encode a subtree of an AST. The Subtree LSTM starts encoding the leftmost subtree as a sequence of 2, 4, and 5. A dropout is then applied on the last hidden state h_1, and the result is used as a vector representation of the subtree. h_1 replaces the subtree and becomes a new child node in the AST. Next, the Subtree LSTM resets its memory state and encodes the rightmost subtree as h_2 vector representation. Finally, Subtree LSTM encodes the AST as a sequence of 1, h_1, and h_2. The hidden state h_3 is used as the final vector representation of the AST. On the other hand, Subtree BiLSTM encodes the leftmost subtree as two sequences. The forward sequence is 2, 4, and 5, and the backward sequence is 5, 4, and 2. The last hidden state h_1 results from the merge of the last hidden states of the forward and backward sequences. A dropout is applied to h_1 and the result is used as a representation of the subtree and substitution in the AST. Next, the Subtree BiLSTM resets its memory states and encodes the rightmost subtree into h_2 vector representation. Finally, the Subtree BiLSTM encodes the

AST as forward and backward sequences of 1, h_1, and h_2 and h_2, h_1, and 1, respectively. The hidden state h_3 is used as the final vector representation of the AST.

5.3 Softmax Layer

The Softmax Layer is a linear layer with the Softmax activation function. The Softmax function is a generalized logistic regression function that is used for multi-class classification problems. The Softmax Layer generates a normalized probability distribution of the candidate source code authors. Given the last hidden state of the AST, the Softmax Layer applies a linear transformation on the input followed by the Softmax function to extract the probability distribution of authors. The author with the highest probability is selected as the final prediction of the model.

6 Experimental Setup

6.1 Data Collection

In this experiment, we collect two datasets for two different programming languages. The first and second datasets contain source code files from Python and C++, respectively. Our goal is to empirically evaluate the classification efficiency and the generalization of our models on different programming languages with different AST structures. The Python dataset is collected from Google Code Jam (GCJ)[3]. Google Code Jam is an annual international coding competition hosted by Google. The contestants are presented with programming problems and need to provide solutions to these problems in a timely manner. The Python dataset has 700 source code files from 70 programmers and 10 programming problems. Programmers work individually on each of the 10 problems. Therefore, each problem has 70 source code solutions with different programming styles. The C++ dataset is collected from Github[4]. Github is an online collaboration and sharing platform for programmers. We crawl Github starting from a set of prolific programmers and spidering out through other programmers they collaborate with, cloning any repositories for which over 90% of the lines of code are from the same programmer. We then group C++ files by author. To create sufficient training examples, we exclude any C++ file whose AST's depth is less than 10 levels or has 5 branches at most. The final dataset has 200 files from 10 programmers and 20 files per programmer.

Python AST files are extracted using a Python module called *ast*. The module is built into the Python 2.7 framework[5]. Each AST contains one root node called *Module* and represents a single Python source code file, as shown in Fig. 2.

[3] https://code.google.com/codejam.
[4] https://github.com.
[5] https://docs.python.org/2/library/ast.html.

The number of unique AST node types in Python 2.7 are 130 nodes. In addition, C++ AST files are extracted using the third party fuzzy parser *joern* [54]. Joern parses the C++ file, outputs the data into a graph database, and then python scripts can be used to explore the database to write machine-readable files containing AST information. A fuzzy parser performs the same basic function as a regular parser, but can operate on incomplete or uncompilable code [5]. Using such a parser allows us to attribute programs which are either incomplete or contain syntax errors, but more importantly, it means that we do not parse external libraries which are likely written by a different programmer. In contrast to Python ASTs, there are 53 unique node types for C++ ASTs. Each C++ source code file may contain multiple ASTs. The tool creates a separate AST for the global definition of a class, a struct, or a function. However, we merge each of these into a single AST per C++ file. That is, we create a root node called *Program* that includes the global blocks as children.

6.2 Training Models

Our models are trained using Stochastic Gradient Descent (SGD) with Momentum and compute the derivatives for the gradient using Backpropagation Through Structure [19,40,45]. SGD is an incremental optimization algorithm for minimizing the parameters of an objective function, also known as the loss function. The loss function in our models is the cross-entropy loss function. SGD computes the gradient of the parameters with respect to the instances in the training dataset. After computing the gradient, the parameters are updated in the direction of the negative gradient. Momentum is an acceleration technique that keeps track of the past updates with an exponential decay. Momentum has been successfully used to train large deep neural networks [22,45,46,48].

At the beginning of the training process, we set the learning rate to 1×10^{-2} and the momentum factor to 0.9. The models are trained up to 500 epochs with an early stopping technique to prevent overfitting [10]. We also use L_2 weight decay regularization with a factor of 0.001 to reduce overfitting [17]. We use a gradient clipping technique to prevent the exploding gradient during training [37]. The models' parameters are initialized with Glorot initialization to speed up the convergence during the training [18]. The biases for all gates in the LSTM and BiLSTM models are set to zero, while the bias for the forget gate is set to 1 [56]. We set the dropout rate to 0.2 and use inverted dropout to scale the input at training time and remove the overhead at test time. We use Chainer, a deep neural framework, to implement our LSTM and BiLSTM models [50].

7 Evaluation

In this section, we evaluate the complexity of our models and compare their classification accuracy and scaling capability to the state-of-the-art models in source code authorship attributions.

7.1 Model Complexity

We evaluate the complexity of LSTM and BiLSTM models by varying the recurrent architecture, the number of layers, and hidden units on 25 and 70 authors from the Python dataset. We examine the effectiveness of (1, 2) layers and (100, 250, 500) hidden units for LSTM and BiLSTM models. Figure 4 shows the effect of increasing the hidden unit size on the one and two layers of LSTM and BiLSTM models using 70 authors from the Python dataset. For the one layer models, the LSTM and BiLSTM models continue to improve their performance accuracy while increasing the hidden units until they reach 100 units. After that, the classification accuracy of the models decreases when more hidden units are added. However, the decline in the classification accuracy is minimal after exceeding 250 hidden units. Therefore, increasing the size of the hidden units to more than 100 does not improve the performance for one layer LSTM and BiLSTM models. On the contrary, two layers LSTM and BiLSTM models improve their classification accuracy until they reach 250 hidden units. However, the accuracy declines sharply when adding more hidden units. We think that larger layers might be over-fitting the training data. Therefore, 250 hidden units are the optimal size for two layered LSTM and BiLSTM models.

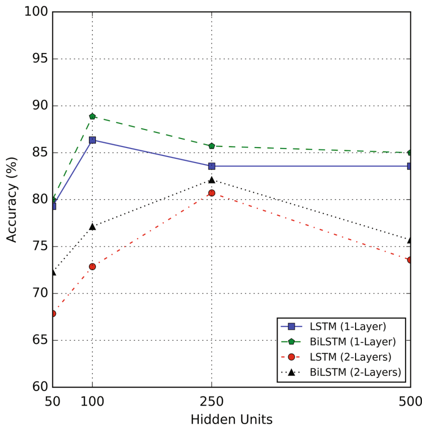

Fig. 4. The classification accuracy for (1,2) layers of LSTM and BiLSTM models with (50, 100, 250, 500) for 70 authors on the Python dataset.

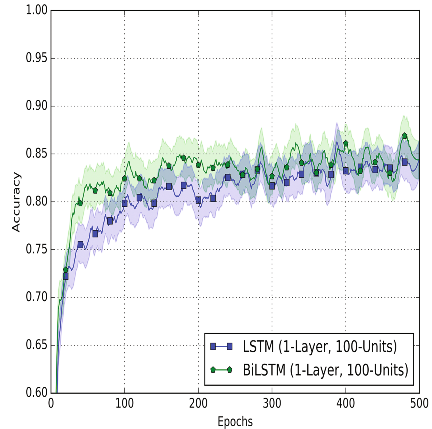

Fig. 5. The classification accuracy for one layer LSTM and BiLSTM with 100 hidden units on the Python test dataset.

Choosing the optimal recurrent architecture of RNN is crucial for improving the classification accuracy of our models. In our research, BiLSTM models show superior performance to LSTM models. These results are in agreement with recent experiments using LSTM-based networks [21,53]. Figure 5 shows the accuracy of the one layer LSTM and BiLSTM models with 100 hidden units

during the training process. We split the 70 authors from the Python dataset into 80% training and 20% testing sets with a balanced distribution of authors. We measure the accuracy of the models on the test dataset after each epoch for 500 epochs. As shown, the BiLSTM model achieves higher classification accuracy and converges quicker than the LSTM model.

7.2 Author Classification

We compare our LSTM and BiLSTM models to the state-of-the-art in source code authorship attribution [7,52]. The work in both research experiments uses a combination of layout, lexical, and syntactic features. We exclude the layout and lexical features from the evaluation and only include the syntactic features that are relevant to the structure of the AST. While excluding layout and lexical features degrades the accuracy of prior work, it enables a fair comparison between the structural/syntactic AST-based features of their work, and the structural/syntactic AST-based features we are developing. In [7], researchers use information gain as a feature selection to select the most important features and use Random Forest as the classifier. The work in [52] uses a greedy feature selection method and Linear SVM as the final classifier. We implement the classifiers using the Scikit-Learn machine learning framework [38]. We use a grid search technique to select the optimal hyperparameters for Random Forest and SVM. We evaluate the models on 25 and 70 authors from the Python dataset, and 10 authors from the C++ dataset. We split the datasets into 80% training and 20% testing sets with a balanced distribution of authors. We select one layer LSTM and BiLSTM with 100 hidden units for comparisons based on their superior performance.

Table 1. The classification accuracy for (1,2) layers of LSTM and BiLSTM with 100 hidden units, Linear SVM, and Random Forest models using 25 and 70 authors on the Python dataset, and 10 authors on the C++ dataset.

	Dataset		
	Python		C++
	25 (Authors)	70 (Authors)	10 (Authors)
Random forest*	86.00	72.90	75.90
Linear SVM*	77. 2	61.28	73.50
LSTM	92.00	86.36	80.00
BiLSTM	96.00	88.86	85.00

* The accuracy results differ from the results in the papers (Refer to Sect. 7.2)

Table 1 shows the results of the four authorship attribution models: Random Forest, Linear SVM, LSTM, and BiLSTM. The BiLSTM model achieves the best classification accuracy. The LSTM model achieves the second best accuracy.

As mentioned earlier, the accuracy results of Linear SVM and Random Forest models differ from the results in the original works because we focused only on the AST-based features and excluded extra features such as the layout and style features. The results show that LSTM and BiLSTM models can efficiently learn the abstract representation of ASTs for a large number of authors who have coded using different programming languages.

7.3 Scaling Author Classification

Large source code datasets often have a large number of authors. Deep neural networks have shown the capability to scale effectively to large datasets with a large number of labels [13,23,53]. A source code authorship classifier needs to handle a large number of different authors with a sufficient classification accuracy. In this experiment, we measure the effect of increasing the number of authors on the classification accuracy of our models. We vary the number of selected authors consecutively to 5, 25, 55, and 70 from the Python datasets. We use the one layer LSTM and BiLSTM models with 100 hidden units and compare the results to the Random Forest and Linear SVM models [7,52]. We obtain this results using 80% training and 20% testing sets with a balanced distribution of authors.

Figure 6 shows the performance of LSTM, BiLSTM, Linear SVM, and Random Forest models when increasing the number of authors in the Python dataset. In general, all the models suffer an inevitable loss in the classification accuracy when the number of authors is increased. However, LSTM and BiLSTM models suffer the least decrease and maintain a robust performance accuracy when the number of authors is large. The Random Forest model achieves an adequate performance, and the Linear SVM model suffers the most significant deterioration in classification accuracy.

7.4 Top Authors Predication

Random Forest, LSTM, and BiLSTM models predict the author with the highest probability as the potential author of an AST. In some cases, researchers increase the prediction to include the top n potential authors for further analysis, especially, when the difference between the authors' prediction probabilities is insignificant. Thus, researchers sometimes include the top n highest probabilities in the prediction process [46]. In this experiment, we measure the classification accuracy of our models when we pick the top n predictions for source code authors. We measure the ability of our models to narrow down the search for the potential authors. We compare the top 1, 5, 10, 15, and 20 predictions of the LSTM and BiLSTM models to the Random Forest [7]. We select one layer LSTM and BiLSTM with 100 hidden units and evaluate the models on 70 authors from the Python dataset. We obtain this results using 80% training and 20% testing sets with a balanced distribution of authors on the Python dataset.

Figure 7 shows the result of increasing the number of the predicted authors in the final prediction. The Random Forest model gains the largest improvement

in the classification accuracy when the top 5 candidate authors are included. The classification accuracy of the Random Forest model continues to improve as the number of top candidate authors increases. Surprisingly, the Random Forest model exceeds the BiLSTM model in the classification accuracy when including the top 20 predicted authors. For the LSTM model, the classification accuracy improves steadily while increasing the number of top candidate authors. The classification accuracy reaches its peak to a nearly perfect accuracy at 15 candidates. The LSTM model also exceeds the BiLSTM model after including the top 5 candidate authors. The BiLSTM model reaches its peak classification accuracy at 15 candidate authors. The BiLSTM model achieves lower classification accuracy than the LSTM model after including the top 5 predicted authors and less than the Random Forest model after including the top 15 predicted authors.

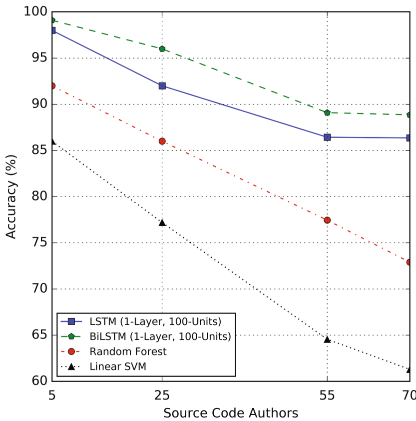

Fig. 6. The classification accuracy for one layer LSTM and BiLSTM with 100 hidden units, Random Forest, and Linear SVM models for 5, 25, 55, and 70 authors in the Python dataset.

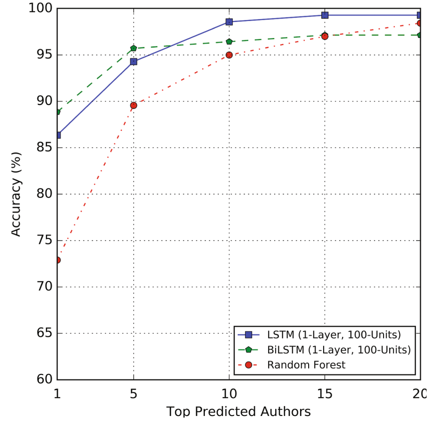

Fig. 7. The top predictions of one layer LSTM and BiLSTM models with 100 hidden units and Random Forest classifier.

8 Conclusions and Future Work

We present a novel approach to AST-based source code authorship attribution using LSTM and BiLSTM models. We show that our models are efficient at learning the structural syntactic features of ASTs. We evaluate our models on multiple datasets and programming languages. We improve the performance results from the previous state-of-the-art on source code authorship attribution using ASTs. We evaluate the scaling capability of our models on a large number of authors.

In the future, we would like to study source code with multiple authors, as large source code projects have multiple programmers collaborating on the

same code section. We would like to evaluate our models on ASTs with multiple authors. We would also like to harden our models against advanced obfuscation techniques that use code factorization for source code.

Acknowledgments. This work is supported by a Fellowship from the Isaac L. Auerbach Cybersecurity Institute at Drexel University, and by an appointment to the Student Research Participation Program at the U.S Army Research Laboratory administered by the Oak Ridge Institute for Science and Education through an interagency agreement between the U.S. Department of Energy and USARL.

References

1. Antoniol, G., Fiutem, R., Cristoforetti, L.: Using metrics to identify design patterns in object-oriented software. In: Proceedings of Fifth International Symposium on Software Metrics, 1998, pp. 23–34. IEEE (1998)
2. Balachandran, V., Tan, D.J., Thing, V.L., et al.: Control flow obfuscation for android applications. Comput. Secur. **61**, 72–93 (2016)
3. Barford, P., Yegneswaran, V.: An inside look at botnets. In: Christodorescu, M., Jha, S., Maughan, D., Song, D., Wang, C. (eds.) Malware Detection. Advances in Information Security, vol. 27, pp. 171–191. Springer, Boston, MA (2007). doi:10.1007/978-0-387-44599-1_8
4. Baxter, I.D., Yahin, A., Moura, L., Sant'Anna, M., Bier, L.: Clone detection using abstract syntax trees. In: 1998 Proceedings of International Conference on Software Maintenance, pp. 368–377. IEEE (1998)
5. Bischofberger, W.R.: Sniff (abstract): a pragmatic approach to a c++ programming environment. ACM SIGPLAN OOPS Messenger **4**(2), 229 (1993)
6. Burrows, S., Uitdenbogerd, A.L., Turpin, A.: Application of information retrieval techniques for source code authorship attribution. In: Zhou, X., Yokota, H., Deng, K., Liu, Q. (eds.) DASFAA 2009. LNCS, vol. 5463, pp. 699–713. Springer, Heidelberg (2009). doi:10.1007/978-3-642-00887-0_61
7. Caliskan-Islam, A., Harang, R., Liu, A., Narayanan, A., Voss, C., Yamaguchi, F., Greenstadt, R.: De-anonymizing programmers via code stylometry. In: 24th USENIX Security Symposium (USENIX Security), Washington, DC (2015)
8. Caliskan-Islam, A., Yamaguchi, F., Dauber, E., Harang, R., Rieck, K., Greenstadt, R., Narayanan, A.: When coding style survives compilation: De-anonymizing programmers from executable binaries. arXiv preprint (2015). arXiv:1512.08546
9. Calliss, F.W.: Problems with automatic restructurers. ACM SIGPLAN Notices **23**(3), 13–21 (1988)
10. Caruana, R., Lawrence, S., Giles, L.: Overfitting in neural nets: backpropagation, conjugate gradient, and early stopping. In: NIPS, pp. 402–408 (2000)
11. Chilowicz, M., Duris, E., Roussel, G.: Syntax tree fingerprinting for source code similarity detection. In: 2009 IEEE 17th International Conference on Program Comprehension, ICPC 2009, pp. 243–247. IEEE (2009)
12. Dauber, E., Caliskan-Islam, A., Harang, R., Greenstadt, R.: Git blame who?: stylistic authorship attribution of small, incomplete source code fragments. arXiv preprint (2017). arXiv:1701.05681
13. Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., Senior, A., Tucker, P., Yang, K., Le, Q.V., et al.: Large scale distributed deep networks. In: Advances in Neural Information Processing Systems, pp. 1223–1231 (2012)

14. Elenbogen, B.S., Seliya, N.: Detecting outsourced student programming assignments. J. Comput. Sci. Coll. **23**(3), 50–57 (2008)
15. Frantzeskou, G., Stamatatos, E., Gritzalis, S., Chaski, C.E., Howald, B.S.: Identifying authorship by byte-level n-grams: the source code author profile (SCAP) method. Int. J. Dig. Evid. **6**(1), 1–18 (2007)
16. Gers, F.A., Schmidhuber, J., Cummins, F.: Learning to forget: continual prediction with LSTM. Neural Comput. **12**(10), 2451–2471 (2000)
17. Girosi, F., Jones, M., Poggio, T.: Regularization theory and neural networks architectures. Neural Comput. **7**(2), 219–269 (1995)
18. Glorot, X., Bengio, Y.: Understanding the difficulty of training deep feedforward neural networks. In: AISTATS, vol. 9, pp. 249–256 (2010)
19. Goller, C., Kuchler, A.: Learning task-dependent distributed representations by backpropagation through structure. In: 1996 IEEE International Conference on Neural Networks, vol. 1, pp. 347–352. IEEE (1996)
20. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press, Cambridge (2016)
21. Graves, A., Schmidhuber, J.: Framewise phoneme classification with bidirectional LSTM and other neural network architectures. Neural Netw. **18**(5), 602–610 (2005)
22. Greff, K., Srivastava, R.K., Koutník, J., Steunebrink, B.R., Schmidhuber, J.: LSTM: a search space odyssey. IEEE Trans. Neural Netw. Learn. Syst. (2016)
23. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 770–778 (2016)
24. Heuzeroth, D., Holl, T., Hogstrom, G., Lowe, W.: Automatic design pattern detection. In: 2003 11th IEEE International Workshop on Program Comprehension, pp. 94–103. IEEE (2003)
25. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Comput. **9**(8), 1735–1780 (1997)
26. Kim, M., Notkin, D., Grossman, D.: Automatic inference of structural changes for matching across program versions. In: ICSE, vol. 7, pp. 333–343 (2007)
27. Koschke, R., Falke, R., Frenzel, P.: Clone detection using abstract syntax suffix trees. In: 2006 13th Working Conference on Reverse Engineering, WCRE 2006, pp. 253–262. IEEE (2006)
28. Kothari, J., Shevertalov, M., Stehle, E., Mancoridis, S.: A probabilistic approach to source code authorship identification. In: 2007 Fourth International Conference on Information Technology, ITNG 2007, pp. 243–248. IEEE (2007)
29. Lazar, F.M., Banias, O.: Clone detection algorithm based on the abstract syntax tree approach. In: 2014 IEEE 9th International Symposium on Applied Computational Intelligence and Informatics (SACI), pp. 73–78. IEEE (2014)
30. Li, J., Luong, M.T., Jurafsky, D., Hovy, E.: When are tree structures necessary for deep learning of representations? arXiv preprint (2015). arXiv:1503.00185
31. Marquis-Boire, M., Marschalek, M., Guarnieri, C.: Big Game Hunting: The Peculiarities in Nation-State Malware Research. Black Hat, Las Vegas (2015)
32. Meng, X.: Fine-grained binary code authorship identification. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 1097–1099. ACM (2016)
33. Mens, T., Tourwé, T.: A survey of software refactoring. IEEE Trans. Softw. Eng. **30**(2), 126–139 (2004)
34. Mikolov, T., Chen, K., Corrado, G., Dean, J.: Efficient estimation of word representations in vector space. arXiv preprint (2013). arXiv:1301.3781

35. Moser, A., Kruegel, C., Kirda, E.: Limits of static analysis for malware detection. In: 2007 Twenty-Third Annual Computer security Applications Conference, ACSAC 2007, pp. 421–430. IEEE (2007)
36. Neamtiu, I., Foster, J.S., Hicks, M.: Understanding source code evolution using abstract syntax tree matching. ACM SIGSOFT Softw. Eng. Notes **30**(4), 1–5 (2005)
37. Pascanu, R., Mikolov, T., Bengio, Y.: On the difficulty of training recurrent neural networks. ICML **3**(28), 1310–1318 (2013)
38. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al.: Scikit-learn: machine learning in python. J. Mach. Learn. Res. **12**, 2825–2830 (2011)
39. Pennington, J., Socher, R., Manning, C.D.: GloVe: global vectors for word representation. In: EMNLP, vol. 14, pp. 1532–1543 (2014)
40. Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Learning representations by backpropagating errors. Cogn. Model. **5**(3), 1 (1988)
41. Russell, S., Norvig, P., Intelligence, A.: A Modern Approach. Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs (1995). pp. 25, 27
42. Sak, H., Senior, A.W., Beaufays, F.: Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In: Interspeech, pp. 338–342 (2014)
43. Schrittwieser, S., Katzenbeisser, S., Kinder, J., Merzdovnik, G., Weippl, E.: Protecting software through obfuscation: can it keep pace with progress in code analysis? ACM Comput. Surv. (CSUR) **49**(1), 4 (2016)
44. Socher, R., Bauer, J., Manning, C.D., Ng, A.Y.: Parsing with compositional vector grammars. In: ACL, vol. 1, pp. 455–465 (2013)
45. Sutskever, I., Martens, J., Dahl, G.E., Hinton, G.E.: On the importance of initialization and momentum in deep learning. In: ICML (3), vol. 28, pp. 1139–1147 (2013)
46. Sutskever, I., Vinyals, O., Le, Q.V.: Sequence to sequence learning with neural networks. In: Advances in Neural Information Processing Systems, pp. 3104–3112 (2014)
47. Tai, K.S., Socher, R., Manning, C.D.: Improved semantic representations from tree-structured long short-term memory networks. arXiv preprint (2015). arXiv:1503.00075
48. Targ, S., Almeida, D., Lyman, K.: Resnet in resnet: generalizing residual architectures. arXiv preprint (2016). arXiv:1603.08029
49. Tennyson, M.F.: A replicated comparative study of source code authorship attribution. In: 2013 3rd International Workshop on Replication in Empirical Software Engineering Research (RESER), pp. 76–83. IEEE (2013)
50. Tokui, S., Oono, K., Hido, S., Clayton, J.: Chainer: a next-generation open source framework for deep learning. In: Proceedings of Workshop on Machine Learning Systems (LearningSys) in the Twenty-Ninth Annual Conference on Neural Information Processing Systems (NIPS) (2015)
51. Vinyals, O., Toshev, A., Bengio, S., Erhan, D.: Show and tell: a neural image caption generator. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 3156–3164 (2015)
52. Wisse, W., Veenman, C.: Scripting DNA: identifying the javascript programmer. Dig. Invest. **15**, 61–71 (2015)

53. Wu, Y., Schuster, M., Chen, Z., Le, Q.V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., et al.: Google's neural machine translation system: bridging the gap between human and machine translation. arXiv preprint (2016). arXiv:1609.08144
54. Yamaguchi, F., Golde, N., Arp, D., Rieck, K.: Modeling and discovering vulnerabilities with code property graphs. In: Proceedings of the IEEE Symposium on Security and Privacy (S&P) (2014)
55. Yu, D.Q., Peng, X., Zhao, W.Y.: Automatic refactoring method of cloned code using abstract syntax tree and static analysis. J. Chin. Comput. Syst. **30**(9), 1752–1760 (2009)
56. Zaremba, W.: An empirical exploration of recurrent network architectures (2015)
57. Zhang, F., Jhi, Y.C., Wu, D., Liu, P., Zhu, S.: A first step towards algorithm plagiarism detection. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis, pp. 111–121. ACM (2012)
58. Zhou, C., Sun, C., Liu, Z., Lau, F.: A C-LSTM neural network for text classification. arXiv preprint (2015). arXiv:1511.08630
59. Zhu, X.D., Sobhani, P., Guo, H.: Long short-term memory over recursive structures. In: ICML, pp. 1604–1612 (2015)