2020

Towards Large-Scale and Robust Code Authorship Identification with Deep Feature Learning

Mohammed Abuhamad
University of Central Florida

TOWARDS LARGE-SCALE AND ROBUST CODE AUTHORSHIP IDENTIFICATION WITH DEEP FEATURE LEARNING

by

MOHAMMED ABUHAMAD
M.S. The National University of Malaysia, 2013

A dissertation submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Summer Term
2020

Major Professor: David Mohaisen

ABSTRACT

Successful software authorship identification has both software forensics applications and privacy implications. However, the process requires an efficient extraction of quality authorship attributes. The extraction of such attributes is very challenging due to several factors such as the variety of software formats, number of available samples, and possible obfuscation or adversarial manipulation. We focus on software authorship identification from three central perspectives: large-scale single-authored software, real-world multi-authored software, and the robustness assessment of code authorship identification methods against adversarial attacks. First, we propose DL-CAIS, a deep Learning-based approach for software authorship attribution, that facilitates large-scale, format-independent, language-oblivious, and obfuscation-resilient software authorship identification. DL-CAIS incorporates learning deep authorship attribution using a recurrent neural network and identifying programmers using ensemble random forest. We demonstrate the effectiveness of DL-CAIS under different experimental settings and scenarios for identifying programmers of both source code and software binaries. Second, we propose Multi-χ, a fine-grained multi-author identification system of programmers in single code files. Multi-χ incorporates code segmentation, code representation, authorship verification, code integration, and authorship identification. We evaluate Multi-χ with several Github projects (Caffe, Facebook's Folly, TensorFlow, etc.) and show remarkable accuracy. We examine the performance of Multi-χ against multiple dimensions and design choices, and demonstrate its effectiveness. Finally, we propose Author-SHIELD to examine the robustness of six state-of-the-art code authorship attribution approaches against adversarial examples. We define three adversarial attacks on attribution techniques—confidence reduction, a programmer imitation, and evasion attacks—and realize them in targeted and non-targeted adversarial code perturbation. Our experiments demonstrate the vulnerability of current authorship attribution methods against adversarial attacks.

This dissertation is dedicated to my beloved wife Samara, my son Zayn, and my family for their support and encouragement.

ACKNOWLEDGMENTS

This work would not have been possible without the continued support, advice, and contribution of many people that I would like to thank for their help.

First, I would like to thank my advisor, Dr. David Mohaisen, for his contribution to my journey at UCF, including his advice, guidance, and training. His guidance, motivation, and immense knowledge helped me in conducting research and writing of this dissertation.

Besides my advisor, I would like to thank my dissertation committee members, Dr. Sung Choi, Dr. Paul Gazzillo, and Dr. Murat Yuksel, for their time and efforts in serving on my committee and for providing insightful feedback.

My sincere thanks also go to Dr. DaeHun Nyang for the continuous support and guidance through my Ph.D. study and related research.

I thank my fellow lab members at the Security Analytics Research Lab (UCF) and Information Security Research Lab (INHA University) for the stimulating discussions, for the sleepless nights we were working together before deadlines, and for all the nice moments we have had in the last five years. Also, I am grateful to Dr. Tamer Abuhmed for the insightful ideas and discussions that help me with my research.

Last but not the least, I would like to thank my family: my wife, my son, my parents, my brothers, and my sisters for supporting me throughout my Ph.D. journey and my life in general.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

Authorship identification of natural language text is a well-known problem that has been studied extensively in the literature [64, 97, 65, 68]. However, far fewer works are dedicated to authorship identification in structured code, such as the source code of computer programs [30]. Software authorship identification is the process of software developer identification by associating a programmer to a given code based on the programmer's distinctive stylometric features. A code of software can be presented with the original source code or the executable binaries, which can be decompiled to generate pseudo-code as higher level construction of the binary instructions [87, 32]. The problem is, however, difficult and different from authorship identification of natural language text. This difficulty is chiefly due to the inherent inflexibility of the presented code expressions established either by the syntax rules of compilers or the reverse engineering of binaries.

Software authorship identification relies on extracting features from software code that a programmer produces based on the programmer's preferences in structuring and developing software pieces. Given these features, the main objective of software authorship identification is to correctly assign programmers to software codes based on the extracted features. Being able to identify software authors is both a risk and a desirable feature. On the one hand, software authorship identification poses a privacy risk for programmers who wish to remain anonymous, including contributors to open-source projects, activists, and programmers who conduct programming activities on the side. Thus, in turn, this makes software authors identification a de-anonymization problem. On the other hand, software authorship identification is useful for software forensics and security analysts, especially for identifying malicious code (such as malware) programmers; *e.g.,* where such programmers could leave source code in a compromised system for compilation, or where features of programmers could be extracted from decompiled binaries. Moreover, authorship identification of software is helpful with plagiarism detection [27], authorship disputes [105], copyright

infringement [48], and software integrity investigations [72].

This work explores the problem of software authorship identification in three main directions. Firstly, this work presents, DL-CAIS, a technique that uses deep learning as a method for learning data representation. DL-CAIS attempts to answer the following questions. (i) How can deep learning techniques contribute to the identification of software authors? (ii) To what extent does an authorship identification approach based on deep learning scale in terms of the number of authors given a limited number of program samples per author? (iii) Can deep learning help identify authorship attributes that go beyond language specifics in an efficient way and without requiring prior knowledge of the language? (iv) Will deep authorship representation still be robust when the program is obfuscated? (v) Can deep authorship representation help identify authors of executable binaries?. (vi) Will these representation still be robust when different toolchain provenance is used to generate the binaries?

Secondly, addressing real-world open-source projects, we present, Multi-χ, fine-grained method for identifying multiple authors contributing to a single source file. Multi-χ is lightweight since (i) 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, (ii) it does not require code parsing, syntax tree extraction, nor explicit feature selection. Multi-χ takes advantage of RNN-based deep learning techniques to generate discriminative features.

Thirdly, since recent code authorship identification techniques are based on machine learning, adversaries are expected to develop incentives to manipulate the input data to force the identification models to generate specific desired output, *e.g.,* misclassification. We propose Author-SHIELD to investigate the impact of attacks on software authorship identification systems. Author-SHIELD involves generating adversarial software examples on the source code level to hinder the authorship identification while preserving the code's functionality. Such attacks will not only fool a classifier

to misidentify programmers but also establish a targeted attack, such as imitating or mimicking of a specific programmer. Investigating such capabilities by understanding the robustness of software authorship attribution against adversarial examples allows a finer understanding of the state-of-the-art methods in uniquely capturing authorship traits in code, and help us shed light on the shortcomings of those algorithms, especially with the increasing reliance on machine learning methods to recognize the coding style of programmers [5, 31].

1.1 Research Statement

The software authorship identification task is challenging and faces several obstacles that prevent the development of practical identification mechanisms. In the case of source code authorship identification, first, programming "style" of programmers continuously evolves as a result of their education, their experience, their use of certain software engineering paradigms, and their work environment [29]. Second, the programming style of programmers varies from language to another due to external constraints placed by managers, tools, or even languages. Third, while it is sometimes possible to obtain the source code of programs, sometimes it is not, and the source code is occasionally obfuscated by automatic tools, preventing their recognition. In the case of binary code authorship identification, first, the toolchain provenance used to produce the binaries must be identified since numerous resultant binaries for the same program can be generated by using different compilation processes. Second, most software binaries, especially malicious programs, are obfuscated making it difficult to be analyzed for authorship. To address those challenges, recent attention to software authorship identification has revived more than two-decade old work [95, 69] by proposing several techniques [30, 31]. However, there are several limitations to the prior work. Namely, (i) most software features used in the literature for author identification are not directly applicable to another language; features extracted in Java cannot be directly used as features in C

or in Python for identifying the same author, (ii) techniques used for extracting software authorship features do not scale well for a large set of authors, and (iii) the extracted features are usually large and not all of them are relevant to the identification task, necessitating an additional procedure for feature evaluation and selection [42].

Moreover, 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 [75]. 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. Further, 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.

Investigating the robustness of authorship identification systems against adversarial examples requires designing a framework for generating software adversarial examples. Generating source code-level adversarial examples is a challenging task since the generated adversarial source code should be syntactically correct, preserve the program's functionality, and not be easily detected. This can be done by a code transformation process [73, 84], similar to author obfuscation whereby authorship traits are hidden in the transformed code. However, code transformation requires analyzing and changing the code to target features from different categories including layout features, lexical and syntactic features, and control-flow and data-flow features. On the other hand, a code perturbation approach provides an effective alternative for targeted and non-targeted attacks, by generating adversarial code samples to meet a specific attack goal. Therefore, adversarial attacks on authorship identification systems should be designed specifically to meet the objectives of the adversaries using code perturbations applied directly to the software code.

1.2 Motivation

Code authorship identification has many useful applications and can be either binary- or source code-based identification. Binary-based techniques [32, 75, 88, 11] are applicable to applications such as malware, proprietary software, and legacy code [75]. Source code-based techniques are applicable when the source code is available, *e.g.,* in software copyright infringement [48], code authorship disputes [105], plagiarism detection [27], and code integrity investigations [72]. 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, GMbot, Mazar, TinyNuke, etc. (all available on Github).

Moreover, considering a multi-author and fine-grained identification from small segments of code within a single file is essential for several 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 [90, 10, 81]. Being able to identify malicious code at a fine-granularity would help in attributing programmers contributing to malicious software.

Since recent code authorship identification techniques are based on machine learning, adversaries are expected to develop incentives to manipulate the input data to force the identification models to generate specific desired output, *e.g.,* misclassification. One line of work for general machine learning algorithms has been utilizing small perturbations to the input domain, resulting in adversarial examples [79]. Those examples are very similar to the original ones, making it hard to distinguish them and posing serious threats to machine learning algorithms. Investigating such

capabilities by understanding the robustness of software authorship attribution against adversarial examples allows a finer understanding of the state-of-the-art methods in uniquely capturing authorship traits in code, and help us shed light on the shortcomings of those algorithms, especially with the increasing reliance on machine learning methods to recognize the coding style of programmers [5, 31]. Understanding the robustness of software authorship attribution can serve as a building block in maintaining the privacy of programmers in spite of identification techniques, although a rigorous definition and quantification of the privacy falls out of the scope of this research.

1.3 Contributions

In this dissertation, we make contributions to the field of software authorship identification form three central perspectives, 1) large-scale single-authored software using DL-CAIS, 2) real-world multi-authored software using Multi-χ, and 3) the robustness assessment of code authorship identification methods against adversarial attacks using Author-SHIELD. We summarize the main contributions of this dissertation in multiple directions as follows

Contributions of DL-CAIS. First, we design a feature learning and extraction method using a deep learning architecture with a recurrent neural network (RNN). The extraction process is fed by a complete or an incomplete program code to generate high quality and distinctive code authorship attributes. The prior work considers preprocessing data transformations which resulted in high-quality features for effective code authorship identification. However, this feature engineering process is usually dependent on human prior knowledge of the programming language addressed in a given task. Our approach utilizes a learning process of large-scale software authorship attribution based on a deep learning architecture to efficiently generate high-quality features. Also, as input to the deep learning network, we use the TF-IDF (Term Frequency-Inverse Document Frequency) that is already a well-known tool for textual data analysis [26, 65, 49]. Thus, our

approach does not require prior knowledge of any specific programming language or high-level translations of program binaries, thus it is more resilient to language specifics when the source code is available and more robust to compilation settings when the target code is in binary format. When conducting experiments on large-scale source code dataset, we found that top features are mostly for keywords of the used programming language, which implies that a programmer cannot easily avoid being identified by simply changing the variable names but by dramatically changing his programming style. With this feature learning and extraction method, we were able to achieve comparable accuracy to (and sometimes better than) the state-of-the-art. For example, compared to 100% accuracy in detecting authorship over a small sample (35 C++ programmers) using features extracted from the abstract syntax tree of the source code [31], we provide a similar accuracy over a larger dataset (150 C++ programmers) and close to that accuracy (99%) for other programming languages using our scalable deep learning-based approach.

Second, we experimentally conduct a large scale code authorship identification and demonstrate that our technique can handle a large number of programmers (8,903 programmers) while maintaining a high accuracy (92.3%). To make our authorship identifier work at a large scale, Random Forest Classifier (RFC) is utilized as a classifier of a TF-IDF-based deep representation extracted by RNN. This approach allows us to utilize both deep learning's good feature extraction capability and RFC's large scale classification capability. Compared to our work, the largest scale experiment in the literature used 1,600 programmers and achieved a comparable accuracy of 92.83% using nine files per author as shown in Table 9 of [31]. While our dataset includes more than 5.5 times the number of the programmers in the prior work, our technique required less data per author (only seven files) for the same level of accuracy at a lower computational overhead. Our experiments are complemented with various analyses. We explore the effect of limited code samples per author and conduct experiments with nine, seven, and five code samples per author. We investigate the temporal effect of programming style on our approach to show its robustness.

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

Fourth, we investigate the applicability of our approach to identify programmers from executables. Several previous works have shown that authorship attribution can be extracted from executable binaries, and identifying programmers of software binaries is possible [32, 88, 75]. We examine our approach on capturing authorship traits from high-level translations of binaries generated by simple straightforward reverse engineering process. The proposed approach achieves an accuracy of 98.4% and 95.74% for identifying 250 and 1,500 programmers of software binaries, respectively. We extend our experiments and analysis to examine the effects of different compilation settings such as levels of optimization and removal of symbol information in stripped binaries.

Fifth, we investigate the effect of obfuscation methods on the authorship identification and show that our approach is resilient to both simple off-the-shelf obfuscators, such as Stunnix [99], and more sophisticated obfuscators, such as Tigress [102] under the assumption that the obfuscators are available to the analyzer. We achieve an accuracy of 99% for 120 authors with nine obfuscated files, which is better than the previously achieved accuracy in [31].

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

Contributions of Multi-χ. We propose Multi-χ, a fine-grained method for identifying multiple authors contributing to a single source file. We evaluate Multi-χ using a large dataset of multi-

author source-code files collected from Github and show its accuracy. We evaluate Multi-χ across multiple dimensions and design choices. Multi-χ 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-χ 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-χ takes advantage of RNN-based deep learning techniques to generate discriminative author features. 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-χ 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%.

Contributions of Author-SHIELD. We propose adversarial attacks on code authorship identification using code perturbation. This approach does not change the original code, but add carefully-crafted code blocks that change the authorship attributes of the resulting code and lead to confidence reduction, identification evasion, and programmer imitation. We target six programmers' identification systems for our evaluation: RNN-based DL-CAIS [3], CNN-based WE-C-CNN, WE-S-CNN, TF-IDF-C-CNN, and TF-IDF-S-CNN [5], and Code Stylometry [31]. Using a dataset of 2,000 programmers, our results show that code perturbations allowed a misidentification rate exceeding 98.5% and decreasing the models' confidence to levels below 40%. Moreover, the results show successful programmers' imitation with a rate exceeds 48% for all targeted systems when using sufficient amount of code perturbation. The results show the feasibility of adversarial attacks using code perturbations, and without any complex code transformations.

1.4 Dissertation Organization

This dissertation encompasses material from four papers, two published papers by the author [3, 4], and two other paper under submission. Chapter 2 uses materials from the four papers, especially reference [3, 4], to highlight the related work in the field of software authorship identification. We organized the chapter of related work as follows. Section 2.1 provides the related work on authorship identification methods of single-authored source code files. Section 2.2 provides the related work on authorship identification methods of software binaries. We explore the methods for authorship attribution of multi-authored software in Section 2.3. For related work on adversarial attacks against authorship identification systems, we provide Section 2.4.

Chapter 3 uses material from reference [3], coauthored with Tamer AbuHmed, David Mohaisen, and DaeHun Nyang, which proposes the Deep Learning-based Code Authorship Identification System (DL-CAIS). Chapter 3 is organized as follows. We provide the motivation in Section 3.1 and the system design in Section 3.2. We provide the experiments addressing attributing authors from the source code in Section 3.3, from the binary code in Section 3.4, Obfuscated code in Section 3.5. Real-world scenarios are addressed in Section 3.6. We highlight the limitations in Section 3.7 and conclude in Section 3.8.

Chapter 4 uses material from reference [4], coauthored with Tamer AbuHmed, DaeHun Nyang, and David Mohaisen which proposes the deep learning-based approach for multi-author identification in source code (Multi-χ). Chapter 4 is organized as follows. We describe our approach in Section 4.1 and provide the experiments and evaluation in Section 4.2. We highlight the limitations in Section 4.3 and conclude in Section 4.4.

Chapter 5 uses material from under submission work, coauthored with DaeHun Nyang and David Mohaisen which proposes Author-SHIELD to examine the robustness of different code authorship attribution approaches against adversarial examples. Chapter 5 is organized as follows. We describe our approach in Section 5.1 and provide the experiments settings in Sections 5.2 and the evaluation in Section 5.3. We provide the discussion in Section 5.4 and conclude in Section 5.5.

CHAPTER 2: RELATED WORK

Broadly related to our work is the attribution of unstructured text. Authorship attribution for unstructured textual documents is a well-explored area, where earlier attempts to match anonymously written documents with their authors were motivated by the interest of settling the authorship of disputed works, such as *The Federalist Papers*. Through the last two decades, studies of authorship attribution have focused on determining indicative features of authorship using the linguistic information (*e.g.,* length and frequency of words or pairs of words, vocabulary usage, sentence structure, etc.). Recent works have shown high accuracy in identifying authors of various datasets such as chat messages, e-mails, blogs, and micro-blogs entries. Abbasi and Chen [2] proposed *writeprints*, a technique that demonstrated a remarkable result in capturing authorship stylometry in diverse corpora including eBay comments and chat as well as e-mail messages of up to a hundred unique authors. Uzuner and Katz [104] provided a comparative study of different stylometry methods used for authorship attribution and identification. Afroz *et al.* [8] investigated the possibility of identifying cybercriminals by analyzing their textual entries in underground forums, even when they use multiple identities. Stolerman *et al.* [98] considered using classifiers' confidence to address the open-world authorship identification problem. Another body of work has investigated authorship attribution under adversarial settings either for the purpose of hiding the identity or impersonating (*i.e.,* mimicking) other identities. Brennan *et al.* [24] studied three adversarial settings to circumvent authorship identification: obfuscation, imitation, and translation.

Addressing authorship attribution for structured data, such as source code, presents a challenge and another interesting body of work in the field of authorship attribution. A summary of the related work is in Table 2.1, with a comparison across four variables: the number of authors, the programming language, the accuracy, and the used technique.

2.1 Authorship Attribution of Source code

The method commonly followed in the literature for code authorship identification research has two main steps: feature extraction and classification. In the first step, software metrics or features representing an author's distinctive attributions are processed and extracted. In the second step, those features are fed into an algorithm to build models that are capable of discriminating among several authors. While the second step is a straightforward data-driven method, the first step leads to major challenges and has become the focus of several research works for more than two decades. Designing authorship attributions that reflect programmers' stylistic characteristics has been investigated by multiple works, since the early work of Krsul *et al.* [69].

Existing code authorship attribution methods include extracting features from different levels of programs, depending on the targeted code for analysis. These features can be as simple as byte-level or term-level features [49], or as complex as control and data flow graphs [75, 88, 11] or even abstract syntax tree features [82, 31]. The quality of extracted authorship attributes significantly affects the identification accuracy and the extent to which the proposed method can scale in terms of the number of authors. Krsul and Spafford [69] were the first to introduce 60 authorship stylistic characteristics categorized into three classes: programming layout characteristics (*e.g.,* the use of white spaces and brackets), programming style characteristics (*e.g.,* average variable length and variable names), and programming structure characteristics (*e.g.,* the use of data structures and number of code lines per function). MacDonell *et al.* [71] adopted only 26 authorship stylistic characteristics extracted using custom-built software IDENTIFIED. Some of these characteristics were extracted by calculating the occurrence of features per line of code.

Frantzeskou *et al.* [49] introduced Source Code Author Profiles using byte-level n-grams features for authorship attribution. Their work was inspired by the success of using n-gram in text authorship identification. Moreover, using n-gram have made the approach language-independent,

an issue that limited preceding works. Lange and Mancoridis [70] were the first to consider a combination of text-based features and software-based features for code authorship identification. Their work used feature histogram distributions for finding the best combination of features that achieve the best identification accuracy. Elenbogen and Seliya [44] considered six features to establish programmers' profiles based on personal experience and heuristic knowledge: the number of comments, lines of code, variables' count and name length, the use of for-loop, and program compression size. Burrows *et al.* [28] used a combination of n-gram and stylistic characteristics of programmers for authorship identification.

Caliskan-Islam *et al.* [31] showed the best results over a large dataset (1,600 programmers), taking advantage of abstract syntax tree node bigrams. Their approach included an extensive feature extraction process for programmer code stylometry resulting in large and sparse feature representations and dictating a further feature evaluation and selection process. After authorship attributions have been introduced, most of the previous works on code authorship identification have adopted either a statistical analysis approach, a machine learning-based classification, or a ranking approach that is based on similarity measurements in order to classify code samples [30]: Statistical analysis methods are considered for limiting the feature space to discover highly-indicative features of authorship. Krsul and Spafford [69], MacDonell *et al.* [71], and Ding and Samadzadeh [42] used discriminant analysis for identifying authors. As for machine learning, various approaches are used for source code authorship identification: case-based reasoning [71], neural networks [71, 30], decision trees [44], support vector machine [82, 30], and random forest [31]. As a general approach of similarity measurement, a ranking approach based on similarity measurements can be used to compute the distance between a test instance and candidate instances in the feature space. Using a k-nearest neighbor approach is one way to assign instances to authors with similar instances. Lange and Mancoridis [70], Frantzeskou *et al.* [49], and Burrows *et al.* [28] tested different ranking methods based on similarity measurements.

2.2 Authorship Attribution of Binary Code

Code authorship identification could also be done on binary code, which is addressed in the literature. Binary-level techniques [32, 75, 88, 11] are advocated as a viable tool for malware, proprietary software, and legacy software attribution [75].

Rosenblum *et al.* [88] explored authorship attributions of program binary code in two tasks, authorship identification, and authorship clustering. The authors extracted a large number of authorship stylistic features from software binary code to enable attributing programmers efficiently to identify them or categorize them based on extracted features. These features include n-grams, idioms, graphlets, supergraphlets, call graphlets, and library calls. Using these features, Rosenblum *et al.* [88] achieved 81% accuracy for identifying ten programmers and 51% for identifying almost 200 programmers. Alrabaee *et al.* [11] proposed a binary authorship identification method called *OBA2*, which extracts syntax-based and semantic-based features related to authorship.

Caliskan-Islam *et al.* [32] have introduced a different approach to extract authorship attribution from binary code by using simple straightforward reverse engineering process to obtain higher translations of program binary code. Using code stylometry features extracted from decompiled pseudo-code, their method achieved an accuracy of 96% for identifying 100 programmers and an accuracy of 83% for 600 programmers. Using the approach introduced by Caliskan-Islam *et al.* [32], the authors provided evidence that authorship stylometry features survive the compilation process and it is possible to identify programmers of binaries even if the binary codes were generated by compilation process included optimization and/or stripping of symbol information.

Meng *et al.* [75] explored the possibility of identifying multiple authors of binary code. The authors introduced a fine-grained approach to identify authors on the basic-block level. Meng *et al.* [75] evaluated their approach using real-world projects to achieve an accuracy of 65% for identifying

284 programmers as the first guess and accuracy of 82% when the author is from top ten suspects.

While very useful, binary-level techniques work under the assumption that a toolchain provenance is used to generate the binary code, including the operating system, compiler family, version, optimization level and source language are known to the analyzer. Source-level techniques, on the other hand, are more flexible and equally useful, especially in addressing incomplete pieces of code (which cannot be compiled). Even when operating on binaries, code-like artifacts are what is being actually analyzed. For example, Caliskan-Islam *et al.* [32] showed that a simple reverse engineering process of binary files can generate a pseudo-code that can be treated as a source code for code authorship identification. In our experiments on identifying the programmers of binaries, we adopt the approach of [32] by analyzing the decompiled code for authorship attribution.

2.3 Authorship Attribution of Multi-Authored Code Files

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 [80, 41, 89, 51], source code [40], and program executable binaries [75]. Payer *et al.* [80] 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.* [41] 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.* [89] proposed Co-Authorship Graph (CAG) technique to attribute different parts of documents to multiple authors. Using dataset of academic papers, CAG technique

Table 2.1: Comparison between our work using deep learning for authorship identification and various related works from the literature, over the used classification techniques, used languages, and approaches. MDA=Multiple Discriminant Analysis, FFNN=Feed Forward Neural Network, RNN=Recurrent Neural Network, CNN=Convolutional Neural Network, KNN=K-Nearest Neighbor. Results are excerpted from references.

Authorship identification based on source code				
Reference	# Authors	Languages	Accuracy (%)	Classification Technique
Pellin [82]	2	Java	88.47%	Machine learning (SVM with tree kernel)
MacDonell *et al.* [71]	7	C++	81.10%	Machine learning (FFNN). Statistical analysis (MDA)
MacDonell *et al.* [71]	7	C++	88.00%	Machine learning (case-based reasoning).
Frantzeskou *et al.* [49]	8	C++	100.00%	Rank similarity measurements (KNN)
Burrows *et al.* [28]	10	C	76.78%	Information retrieval using mean reciprocal ranking
Elenbogen & Seliya [44]	12	C++	74.70%	Statistical analysis (decision tree model)
Lange & Mancoridis [70]	20	Java	55.00%	Rank similarity measurements (nearest neighbor)
Krsul & Spafford [69]	29	C	73.00%	Statistical analysis (discriminant analysis)
Frantzeskou *et al.* [49]	30	C++	96.90%	Rank similarity measurements (KNN)
Ding & Samadzadeh [42]	46	Java	62.70%	Statistical analysis (canonical discriminant analysis)
Burrows *et al.* [30]	100	C, C++	79.90%	Machine learning (neural network classifier)
Burrows *et al.* [30]	100	C, C++	80.37%	Machine learning (support vector machines)
Caliskan-Islam *et al.* [31]	229	Python	53.91%	Machine learning (random forest)
Caliskan-Islam *et al.* [31]	1,600	C++	92.83%	Machine learning (random forest)
Abuhamad *et al.* [5]	1,600	C++	96.2%	Machine learning (CNN)
Abuhamad *et al.* [5]	1,500	Python	94.6%	Machine learning (CNN)
Abuhamad *et al.* [5]	1,000	Java	95.8%	Machine learning (CNN)
DL-CAIS	566	C	94.80%	Machine learning (RNN with random forest)
DL-CAIS	1,952	Java	97.24%	Machine learning (RNN with random forest)
DL-CAIS	3,458	Python	96.20%	Machine learning (RNN with random forest)
DL-CAIS	8,903	C++	92.30%	Machine learning (RNN with random forest)
Authorship identification based on binary code				
Reference	# Authors	Languages	Accuracy (%)	Classification Technique
Rosenblum *et al.* [88]	10	Binary	81%	Machine learning (SVM)
Meng *et al.* [75]	284	Binary	65%	Machine learning (random forest)
Caliskan-Islam *et al.* [32]	600	Binary	83%	Machine learning (random forest)
Authorship identification based on Multi-authored code				
Reference	# Authors	Languages	Accuracy (%)	Classification Technique
Meng et. al. [75]	284	C, C++	65%	SVM and random forest
Dauber et. al. [40]	106	C++	73%	Random Forest
Dauber et. al. [40]	106	C++	99%	Random Forest
Our work (Multi-χ)	282	C, C++	97.31%	RNN with random forest
Our work (Multi-χ)	843	C, C++	88.89%	RNN with random forest

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 [40], which extends the work of Caliskan-Islam *et al.*'s [31] 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 [31], 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.

2.4 Adversarial Attacks on Authorship Attribution

Brennan *et al.* [24] proposed adversarial stylometry to circumvent authorship identification of textual documents. The authors present a framework to create adversarial documents for two purposes: *obfuscation*, where an author of a document attempts to hide his identity, and *imitation*, where an author attempts to imitate the style of another author. Both approaches are conducted manually with human involvement. For code authorship attribution, Simko *et al.* [93] conducted quantitative and qualitative approaches to evaluate authorship identification under adversarial code forgeries. Their study included programmers to create code forgeries and human code analysts to detect and evaluate the forgeries. Meng *et al.* [74] have introduced adversarial attacks on binary-code authorship identification system using adversarial binaries that correspond to feature vector modifications calculated to meet goals of attacks. Matyukhina *et al.* [73] and Quiring *et al.* [84] proposed a transformation process to hide programmers' coding style or to imitate a specific programmer's coding style.

Studying adversarial examples, Yuan *et al.* [106] explored various approaches for generating adversarial examples along with possible applications and corresponding countermeasures. Since we address the code authorship identification, the closest applications can be found in the fields of natural language processing and malware analysis. Adversarial examples in the NLP domain are limited to perturbation at the word level, where charters of a word can be flipped or changed so that it will be unrecognizable. The number of changes should be as small as possible so that the attacks are undetectable by humans. While this approach has been shown to be effective in generating adversarial examples, it is inapplicable to source code for several reasons. For instance, the perturbation, in this way, will be only restricted to variable names since changing a language-specific keyword would result in an error. This is important since Abuhamad *et al.* [3] showed that the top TF-IDF features are the programming language keywords. Moreover, changing a variable

name in one position should be followed by changing that variable name in all other positions where the variable name appears. In the field of malware detection, deep learning has been used to provide static and dynamic patterns that enables detecting zero-day malware [107]. However, recent studies showed the effect of adversarial examples on evading such deep learning-based malware detection methods [6, 7, 9, 54, 60]. Studying adversarial examples for malware detection can help us understand effective methods of developing adversarial examples to evade authorship identification of codes.

CHAPTER 3: DL-CAIS: DEEP LEARNING-BASED CODE AUTHORSHIP IDENTIFICATION SYSTEM [1]

Successful software authorship de-anonymization has both software forensics applications and privacy implications. However, the process requires an efficient extraction of authorship attributes. The extraction of such attributes is very challenging, due to various software code formats from executable binaries with different toolchain provenance to source code with different programming languages. Moreover, the quality of attributes is bounded by the availability of software samples to a certain number of samples per author and a specific size for software samples. To this end, this chapter proposes DL-CAIS, a deep learning-based approach for software authorship attribution, that facilitates large-scale, format-independent, language-oblivious, and obfuscation-resilient software authorship identification. This proposed approach incorporates the process of learning deep authorship attribution using a recurrent neural network, and ensemble random forest classifier for scalability to de-anonymize programmers.

Comprehensive experiments are conducted to evaluate our approach over the entire Google Code Jam (GCJ) dataset across all years (from 2008 to 2016) and over real-world code samples from 1987 public repositories on GitHub. The results of our work show high accuracy despite requiring a smaller number of samples per author. Experimenting with source-code, our approach allows us to identify 8,903 GCJ authors, the largest-scale dataset used by far, with an accuracy of 92.3%. Using the real-world dataset, we achieved an identification accuracy of 94.38% for 745 C programmers on GitHub. Moreover, the proposed approach is resilient to language-specifics, and thus it can identify authors of four programming languages (*e.g.,* C, C++, Java, and Python), and authors writing in

[1] Part of this content was reproduced from the following article: Mohammed Abuhamad, Tamer AbuHmed, Aziz Mohaisen, DaeHun Nyang, "Large-scale and Language-oblivious Code Authorship Identification," In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, (CSS), 2018.

mixed languages (*e.g.,* Java/C++, Python/C++). Finally, DL-CAIS is resistant to sophisticated obfuscation (*e.g.,* using C Tigress) with an accuracy of 93.42% for a set of 120 authors.

Experimenting with executable binaries, our approach achieves 95.74% for identifying 1,500 programmers of software binaries. Similar results were obtained when software binaries are generated with different compilation options, optimization levels, and removing of symbol information. Moreover, our approach achieves 93.86% for identifying 1,500 programmers of obfuscated binaries using all features adopted in Obfuscator-LLVM tool.

3.1 Background and Motivation

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

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

TF-IDF is a well-known tool for text data mining. The basic idea of TF-IDF is to evaluate the importance of terms in a document in a corpus, where the importance of a term is proportional to the frequency of the term in a document. However, it is highly likely to be emphasized by documents which have a very common term over a corpus. Therefore, how specific a given term is over a corpus should be considered. It can be quantified as an inverse function of the

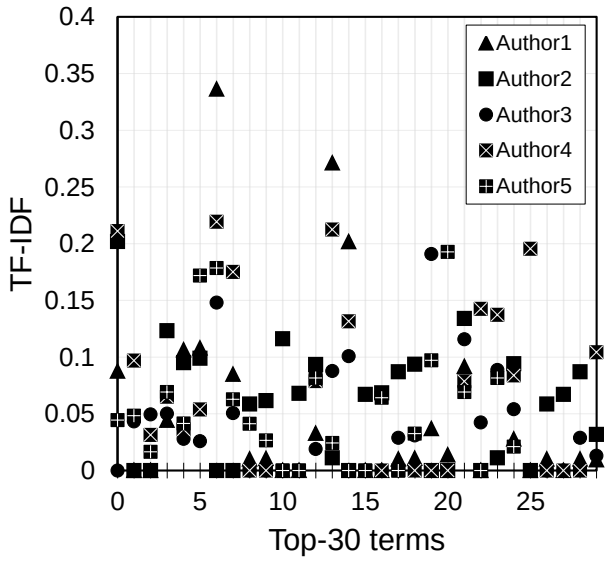


Figure 3.1: The TF-IDF of top-30 terms for five programmers. The value of a term is different among authors who use the same term. The terms are: (‘ans’, ‘begin’, ‘begin end’, ‘bool’, ‘break’, ‘char’, ‘cin’, ‘cin int’, ‘cmath’, ‘cmath include’, ‘const’, ‘const int’, ‘continue’, ‘cout’, ‘cout case’, ‘cstdlib’, ‘cstdlib include’, ‘cstring’, ‘cstring include’, ‘define’, ‘define pb’, ‘double’, ‘end’, ‘endl’, ‘false’, ‘freopen’, ‘include cmath’, ‘include cstdlib’, ‘include cstring’, ‘include map’).

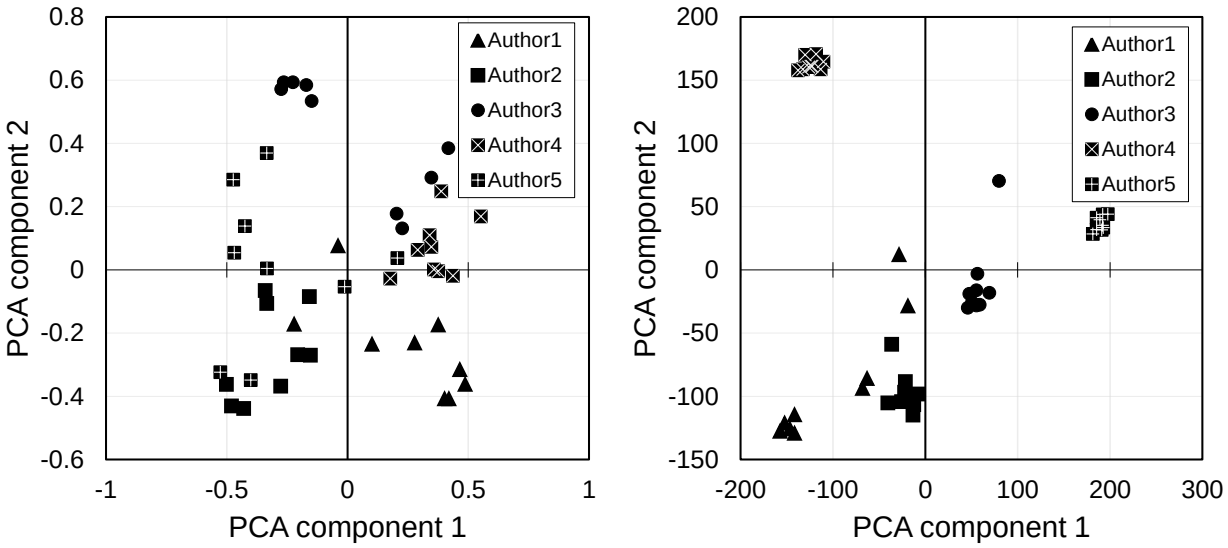
number of documents in which it appears. In building the data preprocessing component of our technique, a term t in a document d of a corpus $\mathcal{D}$ is assigned a weight using the formula TF-IDF$(t, d, \mathcal{D})$ = TF$(t, d) \times$ IDF$(t, \mathcal{D})$, where TF(t, d) is the term frequency (TF) of t in d and 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 DF$(t, \mathcal{D})$ is the number of documents containing the term t.

Using TF-IDF as initial representation for code files is motivated by its wide-range applications on processing textual data. Terms and n-grams features (frequency) are commonly used in information retrieval and have been adopted for code authorship identification [26, 65, 49]. TF-IDF features describe an author’s preferences on using certain terms, or his/her preference for specific commands, data types, and libraries. Figure 3.1 illustrates the mean TF-IDF values of the top-30

terms used by five programmers in nine C++ files of code. Even with slight difference for some terms, the TF-IDF value differs from one programmer to another presenting its validity to be used as initial representation of code files. If the values are composed into one vector for each programmer, then we can distinguish more distinctively each author by observing the distribution of the values. Another observation is that the top features are for keywords of the used programming language. Such observation suggests that a programmer cannot easily avoid being identified by simply changing the variable names but rather by dramatically changing the programming style itself. For example, it seems that 'cout' should not have such a high TF-IDF score because it is a common command for printing out a message, but it has. This is because 'cout' has been used by only a small number of programmers solving problems in GCJ, which makes the keyword distinctive. Thus, frequent use of 'cout' can be regarded as some programmer's programming style.

3.1.2 Deep Representation of TF-IDF Features

Software authorship identification can be formulated as a classification problem, where authors are classified based on their distinctive authorship attributes. The performance of machine learning methods relies on the quality of data representation (features or attributes), which requires an expensive feature engineering process. This process is sometimes labor-intensive and heavily dependent on human prior-knowledge in the classification application field [17]. Identification of a large number of authors using TF-IDF directly cannot be easily achieved as can be seen in Figure 3.2(a). Recently, representation learning has gained increasing attention in the machine learning community and has become a field in and of itself dedicated to enabling easier and more distinctive feature extraction processes [18]. Among several representation learning methods, deep learning has achieved a remarkable success in capturing more useful representations through multiple nonlinear data transformations. Deep learning representations have enabled several advancements in many machine learning applications such as speech recognition and signal processing [57, 38, 22],



(a) PCA visualization of TF-IDF.

(b) Deep representation visual.

Figure 3.2: The PCA visualization of TF-IDF and deep representation of software attributions for five programmers.

object recognition [35, 86], natural language processing [15, 21], and multi-task and transfer learning [16, 53]. Since the breakthrough work of Hinton *et al.* [58], multiple representation techniques using deep learning were presented in the literature. Those techniques have been employed in many fields, with various applications, as reported in [17, 16]. One potential application that was not previously explored is code authorship identification, which we explore in this work. The techniques used include LSTM (Long Short-Term Memory) and Gated Recurrent Units (GRU) that are sorts of Recurrent Neural Network (RNN) among various Deep Neural Networks (DNN).

Deep LSTMs and GRUs [92] with multiple layers demonstrated a remarkable capability to generate representations from long input sequences. In this work, we investigate both LSTM and GRU capabilities of extracting software authorship attributions from TF-IDF code representations, which are a good fit because of the scale of our problem. We will elaborate on this investigation in Subsection 3.2.2. The TF-IDF representations are fed into our deep learning structure as one

sequence per software sample to generate high quality representations that enable accurate authorship identification. To examine the characteristics of TF-IDF, we visualized TF-IDF values of top-30 terms of five authors. For visualizing code samples of a programmer, we used the Principal Components Analysis (PCA). The PCA is a statistical tool that is widely used as a visualization technique that reflects the difference in observations of multidimensional data for the purpose of simplifying further analysis [14, 45]. Figure 3.2 shows PCA visualizations of code samples for five programmers with nine samples each. In Figure 3.2(a), code samples are represented with the TF-IDF features, which are insufficient to draw a decision boundary for all programmers. In Figure 3.2(b), however, the deep representations have increased the margin for decision boundary so distinguishing programmers has become easier. This visualization of the representations space (TF-IDF features and deep representations) illustrates the quality of representations obtained using deep learning techniques.

3.1.3 RFC over Deep Representations

To identify authors, we need a scalable classifier that can accommodate a large number of programmers. However, the deep learning architecture alone does not give us a good accuracy (*e.g.,* 86.2% accuracy for 1,000 programmers). Instead of using the softmax classifier of the deep learning architecture, we use RFC [23] for the classification, and by providing the deep representation of TF-IDF as an input. RFC is known to be scalable, and our target dataset has more than 8,000 authors (or classes) to be identified. Such a large dataset can benefit from the capability of RFC.

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

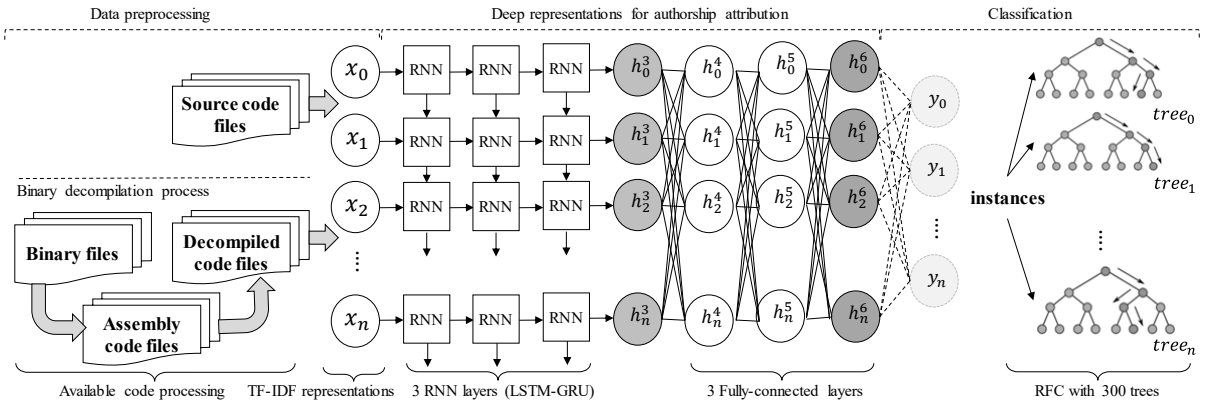


Figure 3.3: A high-level illustration of the proposed deep learning-based software authorship identification system. This illustration shows the three phases of preprocessing (TF-IDF feature representation), better representation through learning (using the RNN and fully-connected layers), and the classification (using 300 trees in a random forest classifier).

3.2 DL-CAIS: System Design

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

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

represented with the TF-IDF initial representations.

The initial representations of code samples are later fed into a deep learning architecture to learn more distinctive features. Finally, deep representations of code authorship attributions are used to construct a robust random forest model. Figure 3.3 illustrates the overall structure of our proposed system. In the first phase, a straightforward mechanism is used to represent source code files based on a weighting scheme commonly used in information retrieval.

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

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

3.2.1 Data Preprocessing

The first phase of the proposed system is to handle available software samples to ensure an efficient initial representation process. This process varies considering the available code format, *i.e.,* source code is a subject to a different preprocessing phase compared to executable binary code. However, both source and binary code files are represented initially with TF-IDF features by the end of the preprocessing phase. Note that previous works *e.g.,* [32, 31, 28, 71, 69, 70] have used TF-IDF or a variation TF-IDF as part of the feature extraction of authorship attributes. In this work, we only use TF-IDF representation as an initial representation for a deep learning model which is trained to extract more robust and distinctive authorship traits. The following describes the preprocessing phase with respect to the available code format.

Preprocessing for source code. The source code is processed to eliminate comments, copyright headers, program description, layout restrictions and features (such as tabs, spaces, and lines), stop words. Since only n-grams are considered for the TF-IDF representation of code samples, the layout features and stop words are irrelevant and therefore they are excluded from the initial representation. This work conducts experiments on the source code of four programming languages. Moreover, obfuscated code using code-to-code obfuscation tools, *e.g.,* Tigress or Stunnix, are treated as source code following the same procedure as source code.

Preprocessing for binary code. When the available code is in a binary format, the first step is to identify the toolchain provenance such as compilation tools and settings. We assume the toolchain provenance of the presented binary codes is known since the current state-of-the-art tools have this capability with a high degree of accuracy [87]. Being able to identify the source of the binary code, there are powerful tools to reverse engineer the binary code to higher level constructs via disassembly or/and decompilation process.

Both disassemblers and decompilers are capable of generating textual translations of binary code that can be an easier subject of analysis. On the one hand, disassemblers provide a straightforward one-to-one translation of binary instructions to instruction mnemonics. Among many powerful disassemblers available on the field, *radare2* [85] and *IDA-Pro* [61] are the most commonly utilized disassemblers with a wide-range of utilities. On the other hand, decompilers generate even higher-level translations of the binary code with concise C-like pseudocode. Compared to disassemblers, decompilers generate five to ten times shorter outputs for the same binary program, *e.g.,* typical binary program with size (400KB-5MB) can generate a decompiled code of size mostly less than 10MB. Therefore, we use a decompilation process to generate high-level translations of software binaries. In our experiments on authorship attribution of executable binary code, we use Hex-Rays [56], a state-of-the-art commercial decompiler. The generated decompiled pseudo codes via

a decompilation process are often larger in size than the original source code. However, we treat the generated pseudo code similarly as the source code in our analysis.

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

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

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

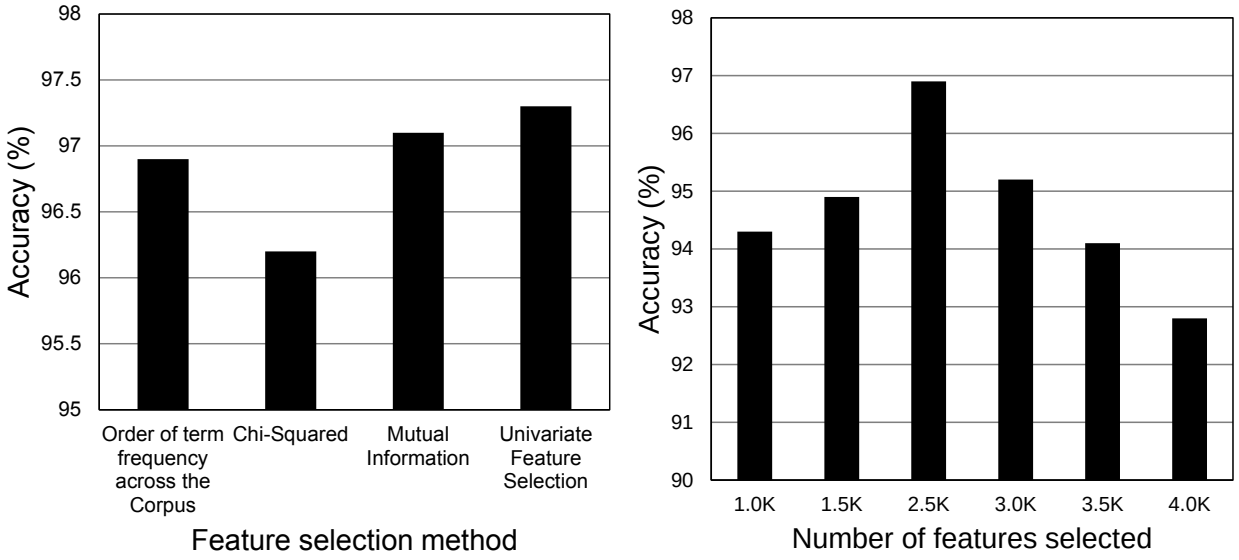
$$x_i = \bigcup_{j=1,\ldots,|\mathcal{D}|} \text{TF-IDF}(t_i, d_j, \mathcal{D}), \tag{3.1}$$

where $\cup$ is a feature selection operator such as the order of term frequency, chi-squared (χ^2) value,

mutual information, or univariate feature selection. Using equation (3.1), x_i's for all terms in the corpus are calculated.

Feature Space. Among the n features, we choose the top-k terms for which x_i's are the largest to reduce the dimensionality and form an input vector to the learning model. For simplicity, we adopt the embedded function of selecting the top-k features by the TF-IDF vectorizer available by the scikit-learn package, which uses the order of term frequencies across all files. With TF-IDF as the method used to represent code files, the feature space needs to be sufficient to distinguish files' authors. For large dataset containing thousands of files (*e.g.,* more than 1,000 programmer with nine files each), the top-k features (for a fixed k) may or may not be sufficient to enable the model to identify authors accurately. As such, we investigated the number of features considered to represent code files as an optimization problem of accuracy. This experiment suggested that 2,500 features are sufficient for the subsequent experiments.

The high dimensionality is likely to introduce overfitting issues, but we addressed the overfitting issues by two regularization techniques, and also conducted all the experiments by repeated k-fold cross validations. Figure 3.4(a) shows the impact of feature selection, using four different approaches, on the accuracy of our approach using TF-IDF features in identifying code authors. In this experiment, we use 1,000 features to identify authors in a 250 C++ programmers experiment. The results demonstrate a substantial accuracy rate (of over 96%) for the given problem size. Figure 3.4(b) demonstrate the impact of the number of the selected TF-IDF features on the accuracy of the classifier. We note that accuracy increases up to some value of the number of features after which it decays quickly. The accuracy, even with smallest number of features, is relatively high.



(a) Feature selection method.

(b) Number of features.

Figure 3.4: Feature selection analysis.

3.2.2 Deep Representation of Code Attributes

For deep representations, we used multiple RNNs and fully-connected layers in a deep learning architecture. For our implementation, we used TensorFlow [1], an open source symbolic math library for building and training neural networks using data flow graphs. We ran our experiments on a workstation with 24 cores, one GeForce GTX TITAN X GPU, and 256GB of memory. We note that our use of the GeForce GTX TITAN X GPU is purely performance driven, and the specific platform does not affect the end-results. Upon the release of our scripts and data, our findings can be reproduced on any other experimental settings.

Addressing Overfitting. To control the training process and prevent overfitting, two different regularization techniques were adopted. The RNN layers in our deep learning architecture included a *dropout regularization* technique [96]. In essence, this technique randomly and temporally ex-

cludes a number of units on the forward pass and weight updates on the backward pass during the training process. The dropout regularization technique has been shown to enable the neural network to reach better generalization capabilities [96].

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

Selecting Layers. The parameters of our final architecture of the deep learning model were chosen after various iterations of experiments, upon which we chose three RNN layers with dropout keep-rate of 0.6, followed by three fully-connected layers with ReLU activation. Each of the fully-connected layers has 1024 units except the last layer, which has 800 units representing the dimensionality of code authorship features for a given input file. During the representation learning process, this architecture is connected to the softmax output layer that represents the class of authors to direct the training process. The training process follows a supervised learning approach, where only the intended model is meant to provide a data transformation that leads to the best probability of its correct class label.

Targeting a large-scale code authorship identification process with thousands of programmers (thousands of classes), the deep learning architecture alone does not accurately identify programmers (86.2% accuracy for 1000 programmers). Thus, we use the output of layer L_{k-1} (where the L_k is the softmax layer) to be the deep representations of code authorship features. Deep representations of code authorship features are then subjected to a classification process using RFC, which is proven to be robust and scalable for large datasets. The weights of the learning network were

initialized using a normal distribution of small range near 0, a small variance, and mean of 0.

Training Procedure. To train our deep learning architecture, we used TensorFlow's Adaptive Moment estimation (Adam) [66] with a learning rate of 10^{-4}, and without reducing the learning rate over time. Adam is an efficient stochastic optimization method that only requires first-order gradients with little memory requirements. Using estimations of the first two moments of the gradients, Adam assigns different adaptive learning rates for different parameters. This method was inspired by combining the advantages of two popular stochastic optimization methods, AdaGrad [43], which is efficient for handling sparse gradients, and RMSProp [101], which is efficient for on-line and non-stationary settings [66].

Further Optimizations. In the training process of the deep learning architecture, we used a mini-batch size ranging from 64 to 256 observations. The idea of using mini-batches reduces the variance in gradients of individual observations since observations may be significantly different. Instead of computing the gradient of one observation, the mini-batch approach computes the average gradient of a number of observations at a time. This approach is widely accepted and commonly used in the literature [92]. The training termination mechanism was either to reach 100,000 iterations or to achieve an early termination threshold for the loss value.

3.2.3 Code Authorship Identification

Using deep authorship features, we construct an RFC for code authorship identification. In doing so, and based on various experiments, we select 300 decision trees for an RFC—this configuration has shown to provide the best trade-off between model construction time and its accuracy [31].

Implementation. We used scikit-learn to implement the RFC using the default settings for building and evaluating features on each split, and all trees were grown to the largest extent without

pruning. Following the approach adopted by [31], we report results of test accuracy using stratified k-fold cross-validation [67], where k depends on the number of observations per class in the dataset (*i.e.,* 9-fold used for 9 files per author, 7-fold for 7 files per author, and so on). The k-fold cross-validation technique aims to evaluate how well our model will generalize to an independent dataset. In this model, the original dataset is randomly partitioned into k equal-sized subsets. Of the k subsets, a single subset is used for testing, and the remaining $k - 1$ subsets are used for training. This cross-validation is repeated k times, where each subset is given a chance to be used for testing the model built from the $k - 1$ subsets, and the evaluation metric (*e.g.,* accuracy) is the computed as average of the k validations.

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

3.3 Authorship Identification of Source Code

In this section, we present the results of several experiments to address various possible scenarios of our identification approach. In our evaluation, we deliver the following: (1) We present results of code author identification over a large dataset. We demonstrate our central results for programmer authorship identification and how our approach scales to 6,635 programmers with nine files each and to 8,903 programmers with seven files each. Our experiments cover the entire Google Code Jam dataset from 2008 to 2016, an unprecedented scale compared to the literature (see Table 2.1). (2) We investigate our system's performance with fewer code files per author and demonstrate its viability. (3) We evaluate the robustness of our identification system under programmers' style evolution and change in development environment, and demonstrate that changes minimally affect the performance of our approach. We complement this study by exploring the temporal effects of programming style on our approach of identification. (4) We push the state of identification

evaluation by looking into mixed language identification. Particularly, we show results using two language files for programmers (C and C++, Java and C++, and Python and C++). (5) We examine how off-the-shelf obfuscators affect our system's performance. Our results are promising: we show that it is possible to identify authors with high accuracy, which may have several privacy implications for contributors who want to stay anonymous through obfuscation. (6) We investigate the applicability of our approach using real-world dataset collected from Github, including two programming languages (*e.g.,* C and C++).

3.3.1 Source Code Dataset

The Google Code Jam (GCJ) is an international programming competition run by Google since 2008 [62]. At GCJ, programmers from all over the world use several programming languages and development environments to solve programming problems over multiple rounds. Each round of the competition involves writing a program to solve a small number of problems—three to six, within a fixed amount of time. We evaluate our approach on the source code of solutions to programming problems from GCJ. The most commonly used programming languages at GCJ are C++, Java, Python, and C, in order. Each of those languages has a sufficient number of source code samples for each programmer, thus we use them for our evaluation. For a large-scale evaluation, we used the contest code from 2008 to 2016, with general statistics as shown in Table 3.1. The table shows the number of files per author across years, with the total number of authors per programming language and the average file size (lines of code, LoC). For evaluation, we create the following three dataset views (Tables 3.1–3.3):

1. **Dataset 1:** includes files across all years from 2008 to 2016 in a "cross-years" view, as shown in Table 3.1.

Table 3.1: Datasets used in our study with the corresponding statistics, including the number of authors with at least a specific number of files across all years.

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

Table 3.2: Two datasets with the corresponding author counts for authors who had seven files at the Google Code Jam (GCJ) 2015 and 2016 competitions.

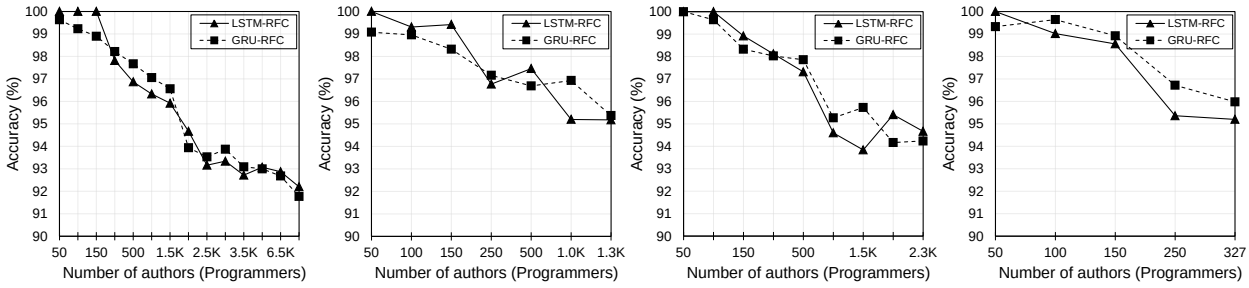
Competition Year	Author Files	No. of Authors for Languages			
		C++	C	Python	Java
2015	7	2241	41	398	132
2016	7	1744	21	390	317
across 3 years*	7	292	NA	44	50
**Programmers participated in 2014, 2015 and 2016*					

2. **Dataset 2:** consists of code files for participants drawn from 2015 and 2016 competitions for four programming languages, as shown in Table 3.2.
3. **Dataset 3:** consists of programmers who wrote in more than one language (*i.e.,* Java-C++, C-C++, and Python-C++) as shown in Table 3.3.

Number of Files. In [31], the use of nine files per programmer is recommended. Our approach provides as good–or even better–accuracy with only seven files, as shown in Subsection 3.3.3.

Table 3.3: A dataset used in our study to demonstrate identification across multiple languages. The dataset includes authors with nine files written in multiple languages.

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

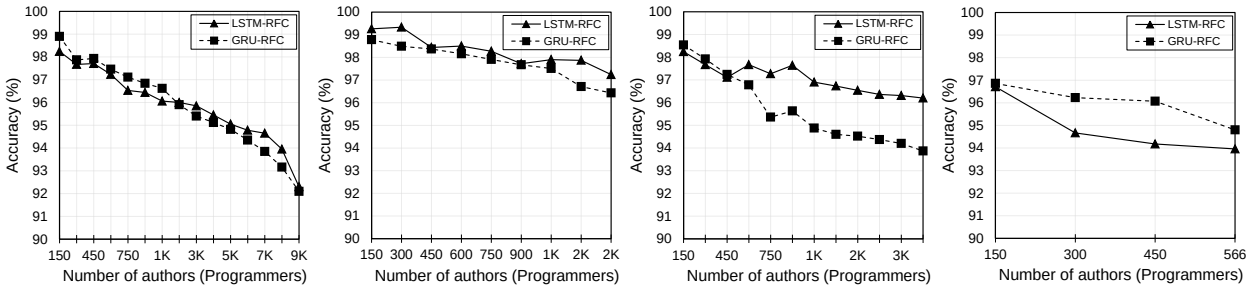


(a) C++ language (b) Java language (c) Python language (d) C language

Figure 3.5: Accuracy of authorship identification of programmers with nine sample code files per programmer in four programming languages (C++. Java, Python, and C). Notice that the accuracy is always higher than 92% even with the worst of the two options of classifiers, and decay in the accuracy is insignificant despite a significant increase in the number of programmers.

3.3.2 Large-scale Authorship Identification

In this experiment, we used dataset 1 in Table 3.1. There are four large scale datasets corresponding to four different programming languages with programmers who have exactly nine code files (first row in Table 3.1). The number of code files per author in this experiment was suggested by [31] to be sufficient for extracting distinctive code authorship attribution features. In our experiment, we started each dataset with a small number of programmers and increased this number until we included all programmers in the dataset. In particular, we used an RFC with stratified 9-fold cross validation to evaluate the accuracy of identifying programmers. We repeated the k-fold cross validation five times with different random seeds and reported the average.

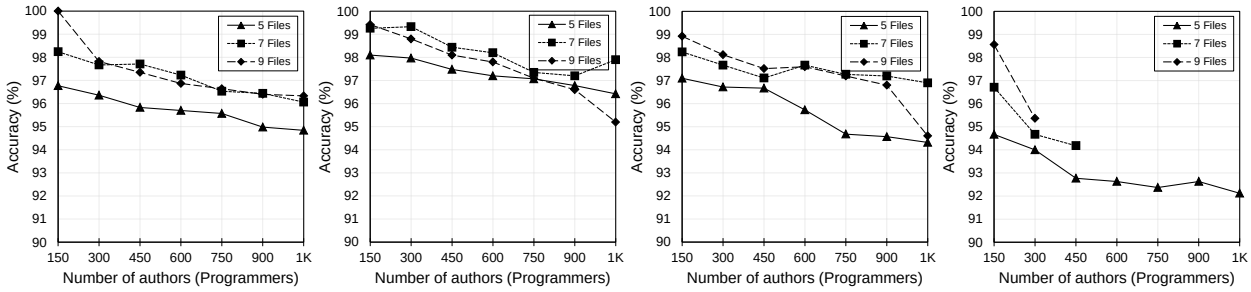


(a) C++ language (b) Java language (c) Python language (d) C language

Figure 3.6: Accuracy of authorship identification of programmers with seven sample code files per programmer in four programming languages (C++. Java, Python, and C). Notice that the accuracy is always high even with large number of programmers.

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

Results. Figure 3.5 shows how well our approach scales for a large number of programmers, and for the different programming languages. The results report the accuracy when using different RNN units in learning code authorship attribution and RFC for authors identification (*i.e.,* using either LSTM-RFC or GRU-RFC unit). In Figure 3.5(a), the LSTM-RFC performance results show that our approach achieves 100% accuracy for 150 C++ programmers with randomly selected nine code files. We note here that FPR is trivially computed as (1 - accuracy), because the dataset is balanced. As we scale our experiments to more programmers, the accuracy remains high, with 92.2% accuracy for 6,635 programmers. Given the same experimental configuration, similar results are obtained for the Java programming language, as illustrated in Figure 3.5(b) with 100% accuracy when the number of programmers is 50 programmers. Upon scaling the experiments to more programmers, we achieve 99.42% accuracy for 150 programmers, and 95.18% accuracy



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

Figure 3.7: Accuracy comparison of authorship identification of programmers in case of five, seven, and nine sample code files per programmer in four programming languages (C++. Java, Python, and C). Notice that the accuracy is always higher than 92%, and regardless of the number of authors. While best results are achieved for the larger number of files, the lowest number of files (of 5) still provides $\sim 92\%$ in the worst case.

for 1,279 programmers. For the Python language dataset, our approach achieved an accuracy of 100% for 100 programmers, 98.92% for 150 programmers, and 94.67% for 2,300 programmers, as shown in Figure 3.5(c). Finally, for the C programmers, Figure 3.5(d) shows that the accuracy reaches 100% for 50 programmers, 98.56% for 150 programmers, and 95.2% for the total of 327 programmers. These results indicate that both deep LSTMs and GRUs are capable of learning deep representations of code authorship attribution that enable achieving large scale authorship identification regardless of the used programming language.

3.3.3 Effect of Code Samples Per Author

The availability of more code samples per author contributes to better code authorship identification, whereas less code samples restrain the extraction of distinctive features of authorship [31, 29].

Experiment 1: Seven Files per Author. For this experiment, we created two datasets with seven and five code samples per programmer for four different languages, as shown in Table 3.1 (second

Table 3.4: Results of the accuracy of our approach in authorship identification for programmers who solved seven problems using the C++ programming language.

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

row). We used RFC with stratified 7-fold cross validation to evaluate the accuracy of identifying programmers at the dataset with seven files per programmer. As the number of available code samples per author decreased, we found that the number of authors increased (Table 3.1). The goal of this experiment is to investigate the effects of having less files per author on the accuracy.

Results. Figure 3.6 illustrates the results of our approach using the dataset of all programmers with seven code samples for four different programming languages. Figure 3.6(a) shows an accuracy of 98.24% when using LSTM-RFC for 150 C++ programmers, and an accuracy of 92.3% for 8,903 programmers. Figure 3.6(b) shows an accuracy of 99.26% for 150 Java programmers when using LSTM-RFC, and 97.24% accuracy when scaling the experiment to 1,952 programmers. Fig-

ure 3.6(c) shows an accuracy of 98.24% when using LSTM-RFC for 150 Python programmers, and an accuracy of 96.2% when scaling the experiment to 3,458. Finally, Figure 3.6(d) shows the result for C programmers, where LSTM-RFC is used: an accuracy of 96.71% for 150 C programmers, and 93.96% for 566 C programmers.

Comparison. Compared with the experimental result of identifying authors using nine code samples per author, as in Subsection 3.3.2, the accuracy does not degrade even when using less code samples per author. Moreover, the results show that our approach is still capable of achieving high accuracy even with more authors compared to the previous experiments. This result presents the largest-scale code authorship identification by far, indicating that seven files per author are still sufficient for extracting distinctive features.

Experiment 2: Five Files per Author. We created a dataset with five source code samples per programmer in the four different programming languages, as shown in Table 3.1 (third row). We used RFC with stratified 5-fold cross validation to evaluate the accuracy of identifying programmers at the dataset with five files per programmer. As the number of available code samples per author decreased, we found that the number of authors increased (Table 3.1). The goal is to further investigate the effects of having even lesser files per author on the accuracy of our approach.

Results. Figure 3.7 shows the results for 1,000 programmers, demonstrating the effect of decreasing the number of sample files for each author. Figure 3.7(a) shows that our approach provides an accuracy of 96.77% for attributing authors in 150 C++ programmers when using LSTM-RFC. Comparing the results of those datasets with the nine and seven source code samples for each programmer, the accuracy loss was only 3.23% and 1.47%, respectively. As we scale to 1,000 C++ programmers, our approach achieves an accuracy of 94.84%. This result proves that our approach still achieves high accuracy even with fewer sample files per programmer. The results of accuracy with smaller number of files per author generalize to other programming languages. Using

the same approach and settings as above, Figure 3.7(b) shows an accuracy of 98.1% and 96.42% for 150 and 1,000 Java programmers, respectively. Figure 3.7(c) show an accuracy of 97.1% and 94.32% for 150 and 1,000 of Python programmers. Finally, Figure 3.7(d) shows an accuracy of 94.67% and 92.12% for 150 and 1,000 C programmers, respectively.

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

3.3.4 Effect of Temporal Changes

The literature suggests that temporal effect is a challenge for code authorship identification, since the programming style of programmers evolves rapidly with time due to their education and experience [31, 29, 55]. We investigate the impact of temporal effect on source code authorship identification. The experiments include two parts: 1) exploring the existence of such impact on the identification process, 2) examining our approach against such effect.

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

To answer this question, we conducted an experiment where results from identifying authors from the same year is compared with results across different years throughout the competition. We examined our approach using a dataset of source codes written by programmers within one competition year, where all programmers solve the same set of problems. Two datasets of GCJ competition of the 2015 and 2016 code samples were created individually with seven code files per programmer, as shown in Table 3.2. In this experiment, we used a random forest and stratified 7-fold cross

validation to evaluate the accuracy of identifying programmers.

Results. Table 3.4 summarizes the results of this experiment when applying LSTM-RCF and GRU-RCF for C++ programmers in two separate years. The accuracy of code authorship identification reaches 95.23% for 2,241 C++ programmers and 95.17% for 1,744 C++ programmers from 2015 and 2016 competitions, respectively. Our approach also shows high accuracy results for Java, Python, and C programming languages, as shown in Table 3.5, Table 3.6, and Table 3.7.

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

Experiment 2: Testing Different Year's Dataset from Training Dataset. In this experiment we attempt to answer the following question: *If temporal effects do exist, can a model trained on data from one year identify authors given data from a different year?* To answer this question, we collected a dataset of sample codes for programmers who participated in three consecutive years from 2014 until 2016. The dataset include seven code files per programmer in each year. The total number of programmers included in the dataset of different languages is shown in Table 3.2.

Results. We trained our models (LSTM-RFC and GRU-RFC) on data from the year 2014 and used the data from 2015 and 2016 as a testing set. As a result, Table 3.8 shows that our approach of code authorship identification is resilient to temporal changes in the coding style as it achieves 100% accuracy for both Python and Java languages and 97.65% for the 292 C++ programmers.

Table 3.5: The accuracy of authorship identification for programmers with seven samples problems (programs) using the Java programming language.

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

Table 3.6: The accuracy of authorship identification for programmers with seven programs using the Python. Note that the accuracy is always above 96%.

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

3.3.5 Identification with Mixed Languages

Here, we investigate code authorship identification for programmers writing in multiple programming languages. In particular, in this section we attempt to answer the following question: *is it possible to identify programmers writing in multiple languages using one model trained with multiple languages*? Some programmers develop programming skills in multiple languages and use the preferable one based on the problem or the job at hand.

To this end, we attempt to understand whether learning about a programmers' style in multiple languages without recognizing languages contributes in identifying the programmer given codes written in multiple languages. Despite the natural appeal to this problem and its associated research

Table 3.7: The accuracy of authorship identification for programmers who solved seven problems using the C programming language. Notice the accuracy is always close to 100%.

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

Table 3.8: The accuracy of authorship identification for programmers who solved seven problems from three different years (2014–2016). The identification models were trained on data from 2014 and tested on data from 2015 and 2016.

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

questions, there is no prior research work on this problem. Thus, we proceed to understand the potential of identification for multiple languages using our approach.

Experiment 1. We use dataset 3 (Table 3.3), which corresponds to authors with nine files (selected randomly) written in multiple programming languages across all years. For training, we fed code files in two languages without letting it know the languages (thus, the training process is oblivious to the language itself). For testing, we also fed code files to the system without indicating what language they are written in (thus, the testing process is oblivious to the language too). Therefore, we aim to examine our system under this (stronger) mixed model.

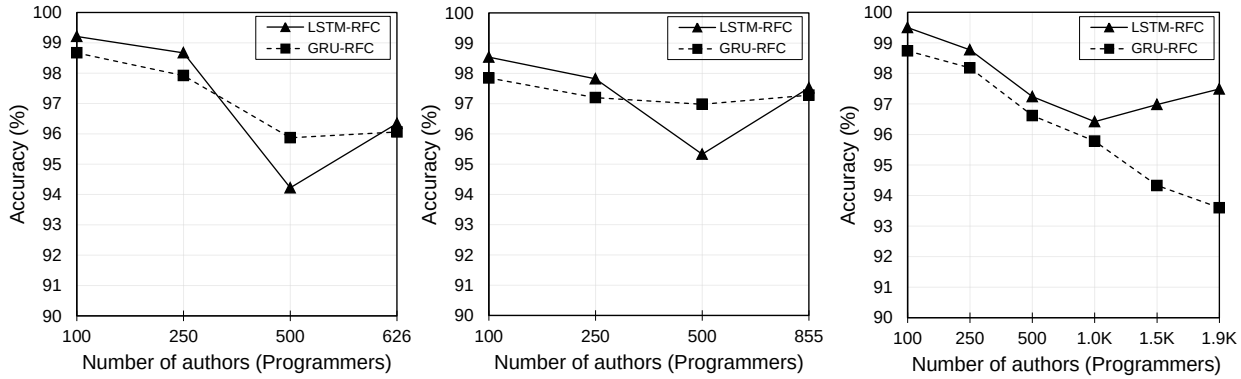
Results. Figure 3.8 shows the accuracy of our approach with three datasets: C++/C, C++/Java, and C++/Python. Figure 3.8(a) shows an accuracy of 96.34% for a dataset of 626 C++/C programmers with LSTM-RFC, and its accuracy of 97.52% when used with LSTM-RFC on 855 C++/Java programmers, as illustrated in Figure 3.8(b). For the C++/Python dataset, Figure 3.8(c) shows that

our approach provides an accuracy of 97.49% for 1,879 programmers.

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

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

Experiment 2. Another experiment was conducted to show the capability of our approach in identifying authors where the identification features are entirely extracted from a different programming language. The aim of this experiment is to answer the following question: *Given samples of code written by programmers in one language (*e.g., *C++), is it possible to identify those programmers when writing in a different language (*e.g., *C)*? From the 1,897 programmers who used C++ and C in dataset 3 (Table 3.3), we extracted a dataset of 224 programmers, where 70% of the samples per author are written in C++ while the remaining 30% are written in the C language. Using our approach, we trained an LSTM-RFC using the 70% of samples written by the 224 programmers in C++ and tested the LSTM-RFC model on the remaining 30% of C samples. As a result, our approach achieved 90.29% of accuracy for identifying programmers with



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

Figure 3.8: The accuracy of the authorship identification of programmers with sample codes of two programming languages.

features extracted from code written by them in a different programming language, highlighting its language-agnostic identification capabilities.

3.4 Authorship Identification of Binary Code

We examine the robustness of our approach in identifying authors of executable binaries. Previous research by Rosenblum *et al.* [88] extracted authorship features directly from the binary code to enable the identification of 161 programmers with 51% accuracy, while the work of Caliskan-Islam *et al.* [32] improved this accuracy to 92% using features extracted from different levels of the decompilation process. Caliskan-Islam *et al.* [32] extracted features from assembly code and abstract syntax tree of decompiled code. These features enabled the binary code authorship identification of 600 programmers with an accuracy of 83%. The authors used an approach of four steps to identify programmers of binary code, namely: disassembly, decompilation, dimensionality reduction and classification. In this work, we use a similar approach without the requirement of extracting features from different levels. Instead, we use Hex-Rays, a commercial powerful decompiler, to

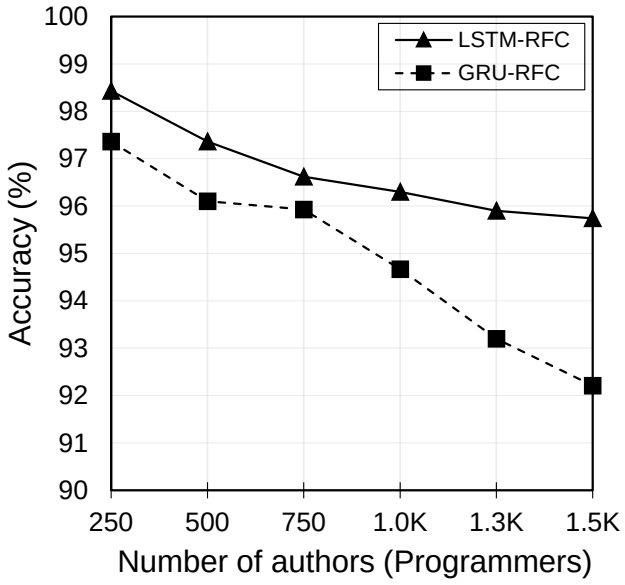


Figure 3.9: The accuracy of the authorship identification of programmers using their binary samples compiled with no optimization.

decompile executable binaries to generate C-like pseudo code. The generated pseudo code can be described as a translation of the program instructions using higher level constructions that preserve the program's control structures such as loops and branches. These features have shown a high degree of significance in attributing programmers in previous works. Therefore, we conduct several experiments to evaluate our approach in identifying programmers of executable binaries.

Assumptions. We assume the availability of binary samples of authors to be identified. Additionally, we examine our approach to identify authors of tested executable binary programs under the assumption of knowing the toolchain provenance such as compiler family and version, compilation optimization level, and source code language, etc. These assumptions consistent with the literature of identifying programmers of software binary code [32]. Moreover, state-of-the-art tools and methods can identify with high accuracy the toolchain provenance of a given exactable binary [87]. Therefore, we assume that such techniques are used to define the toolchain provenance of a given

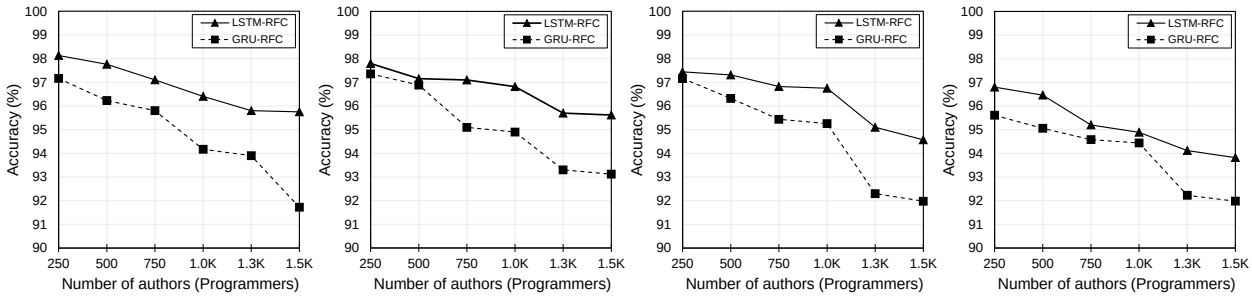
binary code and use our models that are trained using samples compiled with same settings.

3.4.1 Binary Code Dataset

Using a dataset of 6,962 C and C++ programmers participated in GCJ from 2008 to 2016 with at least nine files, we created a dataset of executable binaries for 1,500 programmers to evaluate our approach in identifying authors using binary format. We used GNU Compiler Collection's GCC or G++ to compile C and C++ source codes, respectively, into 32-bit Intel 80386 Unix binaries with Executable and Linkable Format (ELF). Moreover, we also use different compilation optimization levels to examine the effects of resultant binaries in the authorship identification process. GNU Compiler provides different optimization levels corresponds to different flags such as O1, O2, O3 and Os flags. When an optimization flag is turned on, the compiler generates different binaries that vary in some attributes such as code size, execution time, memory utilization, etc. Using a higher level of compilation optimization results in advanced binary code. However, compilation with optimization requires more time and memory resources than compilation with no optimization.

3.4.2 Authorship Identification of Binary Code

In this experiment, we processed the dataset of 1,500 programmers with at least nine binary samples produced from a compilation without optimization process. Figure 3.9 shows that our approach accurately identified programmers on a large-scale binary dataset. Using 9-fold-cross-validation LSTM-RFC, our approach achieved an accuracy of 98.4% for identifying 250 programmers. when increasing the scale of our experiment to include 1,500 programmers, our approach achieved an accuracy of 95.74%. These results show that our approach can identify programmers of executable binaries more accurately and on a larger scale than previous approaches.



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

Figure 3.10: The accuracy of the authorship identification of programmers using their binary samples compiled with different optimization options, showing promising accuracy results even with decompiled binary samples.

3.4.3 Effect of Compilation Optimization

Since our approach achieved high accuracy in identifying programmers of binary code generated from a compilation process without optimization, this experiment explores the effects of different optimization levels on the code authorship identification using our approach. There are different optimization levels that can be incorporated with the compilation process to advance the optimization of certain attributes of an executable program. The optimization techniques transform a given program to a semantically equivalent program that is more efficient than the original program. For this experiment, we use four levels of optimization that can be turned on by O1, O2, and O3, Os flags for GCC compiler family. Using different optimization techniques that generates different binaries enables a better understanding of their effects on authorship attribution. Figure 3.10 shows the result of our approach in identifying 1,500 programmers with nine binary samples generated with different optimization level. Figure 3.10(a) shows the results of our approach using 9-fold-cross-validation on a dataset generated by a compilation process with level-1 optimization. The LSTM-RFC approach achieved 98.1% accuracy for identifying 250 programmers, and 95.75% for identifying 1,500 programmers. Compared to the result obtained when no optimization is used, this

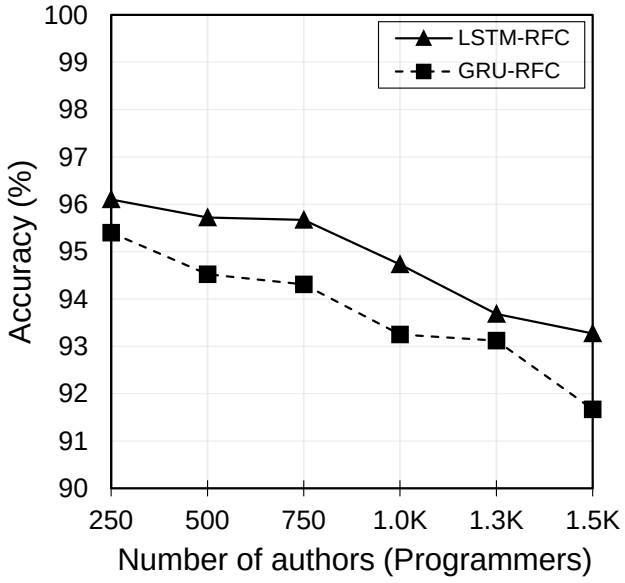


Figure 3.11: The accuracy of the authorship identification of programmers using stripped binaries.

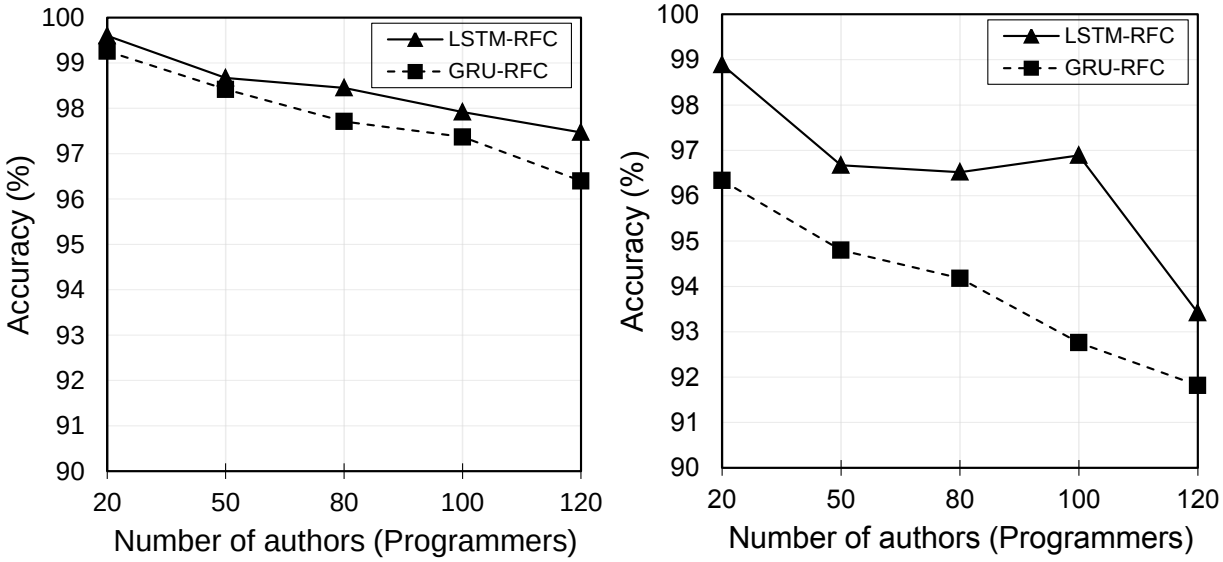
result shows no accuracy degradation as the optimization technique is introduced. Using level-2 optimization, Figure 3.10(b) shows that our approach achieves an accuracy of 97.8% for identifying 250 programmers and an accuracy of 95.6% for identifying 1,500 programmers. Similar results are obtained when using O3 optimization flag, Figure 3.10(c) shows that LSTM-RFC achieved an accuracy of 97.4% for identifying 250 programmers and an accuracy of 94.57% for identifying 1,500 programmers of binary code generated from a compilation with O3 optimization. Figure 3.10(d) shows that using Os optimization does not affect the identification accuracy as LSTM-RFC achieved 96.8% identification accuracy of 250 programmers and an accuracy of 93.82% for 1,500 programmers. The results achieved by different binary datasets generated from compilation with different optimization options show that our system is robust to different optimization and capable of capturing relevant authorship attributes that enabled accurate authorship identification.

3.4.4 Identification with Stripped Binary Code

In this experiment, we investigate the effects of stripping the symbol information from the binary code on the identification accuracy of our system. Using a fully stripped binary code, where all symbol table and relocation information are stripped using the *GNU strip* option, we show the effect of symbol information on authorship attribution. Figure 3.11 shows that the system was capable of generating high-quality deep representations that enabled accurate authorship identification. Comparing to the previous works by [32, 88], our approach shows robustness to different compilation settings including stripping symbol information. Even when symbol information is completely missing, the deep learning architecture is capable of transforming the input information presented in binary samples to robust deep representations of authorship attribution. Unlike other works that attempt to generalize features across different compilation settings, the deep learning architecture tune parameters that allow best representations based on a given input data or settings. For example, Caliskan-Islam *et al.* [32] showed that attempting to identify programmers of stripped binary code using the same feature set used for unstripped binaries can cause an accuracy degradation of 24%.

3.5 Authorship Identification of Obfuscated Software

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



(a) C++ obfuscated with Stunnix.

(b) C code obfuscated with Tigress.

Figure 3.12: The accuracy of authorship identification with obfuscated source code, showing promising results even with the more sophisticated obfuscation approach (Tigress).

trained with obfuscated codes, will it be able to identify authors correctly if obfuscated codes are presented for testing?

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

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

show the effects of different obfuscation approaches on the software authorship identification.

3.5.1 Software Source Code Obfuscation

In this experiments, we investigate the effects of code-to-code obfuscation on authorship attribution. Different obfuscation tools are available, and two among them were chosen to evaluate our approach: Stunnix [99] and Tigress [102]. The main reason for choosing these two obfuscation tools is because each represents a different approach for code-to-code obfuscation. Stunnix is a popular off-the-shelf C/C++ obfuscator that gives code a cryptic look while preserving its functionality and structure. Tigress, on the other hand, is a more sophisticated obfuscator for the C language; it implements function virtualization by converting the original code into an unreadable bytecode. For our experiment on code authorship identification of Tigress-obfuscated code, we turned on all of the features of Tigress.

Experiment 1: Stunnix. The first experiment is targeted towards a C++ dataset of 120 authors with nine source code files obfuscated using Stunnix. Our approach was able to reach 98.9% accuracy on the entire obfuscated dataset of 120 authors and 100% accuracy on an obfuscated dataset of 20 authors. Figure 3.12(a) shows the accuracy achieved using our approach on different Stunnix-obfuscated C++ datasets ranging from 20 to 120 authors using two different RNN units. The results indicate that our approach is robust and resistant to off-the-shelf obfuscator.

Experiment 2: Tigress. We use a C dataset of 120 authors with nine source files each, obfuscated using Tigress. Even with this sophisticated obfuscator, our approach achieves 93.42% on the entire dataset while maintaining an accuracy of over 98% on a subset of 20 authors. Figure 3.12(b) shows the achieved accuracy on different Tigress-obfuscated C datasets ranging from 20 to 120 authors using two different RNN units. The results also indicate the resilience of our approach to

sophisticated obfuscators such as Tigress. Despite the unreadability of the obfuscated code using Tigress, which makes such obfuscated code unreadable, the result of our experiment highlights that code files are no longer unidentifiable.

3.5.2 Software Binary Code Obfuscation

In this experiment, we investigate the effects of binary obfuscation on the identification accuracy. Among many tools for software binary obfuscation, we use Obfuscator-LLVM [63] to generate obfuscated binary code using different features. The Obfuscator-LLVM provides different levels of obfuscation including control flow flattening, instruction substitution, bogus code injection, etc. The aim of this experiment is to examine the robustness of our approach in identifying programmers of obfuscated binary code even when different obfuscation levels are introduced.

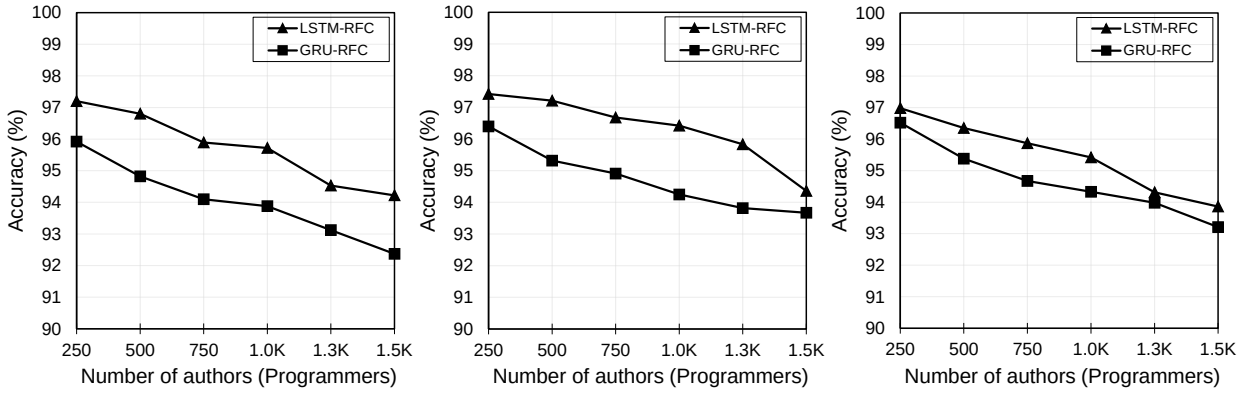
For this experiment, we use the same dataset of 1,500 programmers used for the experiments on authorship identification of software binaries in section Section 3.4. Programmers might adopt several techniques to circumvent authorship identification on the binary level of a program using control flow flattening, instruction substitution, bogus code injection or all options combined to make it difficult for analysis and authorship attribution. We address these different scenarios of obfuscation and show their effects on software authorship identification. Figure 3.13 shows results of different experiments conducted using different obfuscated binaries.

Experiment 1: control flow flattening. Obfuscation through control flow flattening aims to hide the flow structure of a program using code transformations that target all basic blocks of a program. One way to achieve control flow flattening is to split all the program's basic blocks, *e.g.,* functions, loops, branches etc., in a certain way that can be grouped inside one single infinite loop that operates on switch statement to control program's flow. This technique of obfuscation complicates

the understanding of the program structure that can be indicative of authorship. We used the control flow flattening option in Obfuscator-LLVM tool to examine the effects of this technique on authorship attribution. Figure 3.13(a) shows the results of authorship identification of control flow flattened binaries using our approach. The results show that our approach still resilient to this kind of obfuscation by achieving an accuracy of 97.2% in identifying 250 programmers and accuracy of 94.22% for 1,500 programmers.

Experiment 2: instruction substitution. Obfuscation through instruction substitution is a straightforward technique aims to replace standard instruction with a series of functionally equivalent instructions. This technique of obfuscation increases the code size and can be simply evaded by an optimization process. Therefore, such a technique alone might not be the optimal choice for obfuscation but it can add complication when used with other obfuscation techniques. Using instruction substitution obfuscation, Figure 3.13(b) shows similar results for our approach with an accuracy of 97.42 % for identifying 250 programmers and an accuracy of 94.36% for identifying 1,500 programmers.

Experiment 3: full Obfuscator-LLVM obfuscation. In this experiment, we allowed all obfuscation options offered by Obfuscator-LLVM, including, control flow flattening, instructions substitution, and bogus control flow. Using a dataset of fully-obfuscated binary code, our approach shows remarkable resilience by achieving high identification accuracy on different scales as presented in Figure 3.13(c). In Figure 3.13(c) shows that our approach achieved an accuracy of 96.98% for identifying 250 programmers and an accuracy of 93.86% when increasing the experiment scale to include 1,500 programmers.

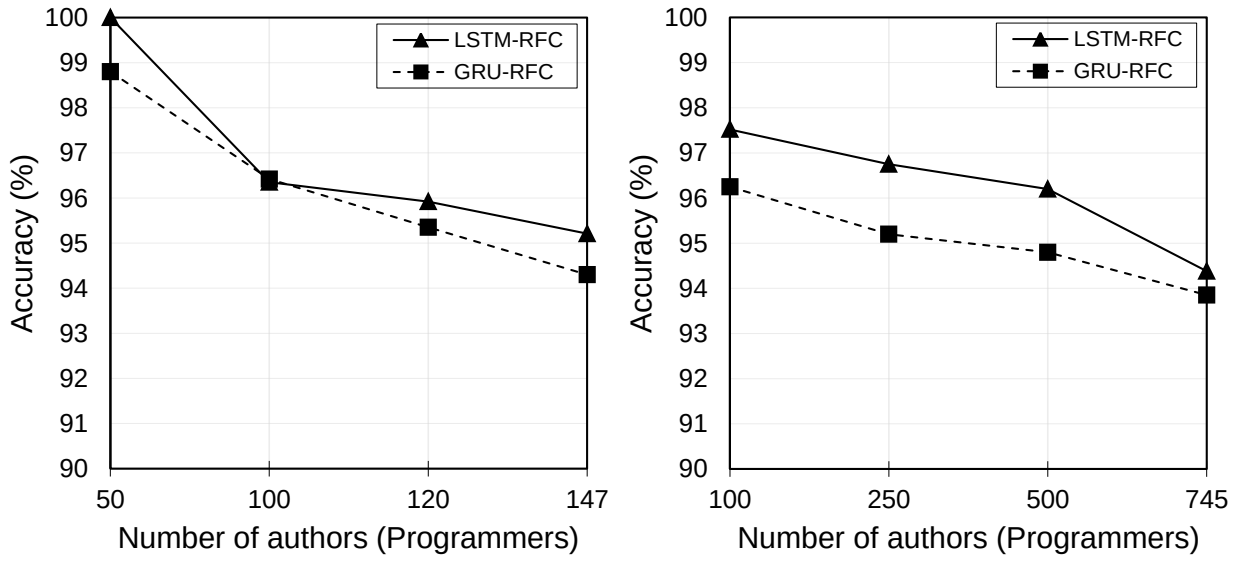


(a) Control flow flattening. (b) Instructions substitution. (c) Full obfuscation.

Figure 3.13: The accuracy of authorship identification with obfuscated binary code using different Obfuscator-LLVM options.

3.6 Authorship Identification in the Real-world

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



(a) GitHub C++ dataset.

(b) GitHub C dataset.

Figure 3.14: The accuracy of the authorship identification of programmers using GitHub dataset, showing promising results even with real-world code samples.

3.6.1 Software Authorship Identification In The Wild

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

The collected dataset includes code samples from 1987 public repositories on GitHub, which list C and C++ as the primary language written by one contributor. After processing the repositories and removing incomplete data, the collected C++ and C datasets included 142 and 745 programmers, respectively, with at least five code samples each. Since some authors have more than 10 samples,

we have randomly selected 10 samples per author. For the ground truth of our dataset, we collected repositories with a single contributor under the assumption that the collected samples are written by the same contributor of the repository. We acknowledge that this assumption is not always valid, because parts of the code samples might have been copied from other sources [39]. Even under those acknowledged limitations of the ground truth, our evaluation is still conservative with the respect to the end results: it attempts to distinguish between code samples that *may even include reused codes across samples*.

Experiment 1: Source code authorship identification in the wild. For this experiment, we process the collected dataset using the source code files. Figure 3.14 shows the results of our approach using GitHub C++ and C datasets. Figure 3.14(a) shows an accuracy of 100% when using LSTM-RFC for 50 C++ programmers and 95.21% for 147 programmers. Figure 3.14(b) shows an accuracy of 94.38% for 745 C programmers using LSTM-RFC. This result shows that our approach is still effective when handling a real-world dataset.

Experiment 2: Binary code authorship identification in the wild. For this experiment, we compiled code files of the collected dataset in the wild using the same compilation options presented in Section 3.4. Using code files of 142 (C++) and 745 (C) programmers, we successfully generated a dataset of a total of 241 programmers who have at least nine files that we were able to compile. For the compilation process, we generated binaries with level O3 optimization and removed all debugging symbols. Using the dataset of binary code, our approach achieves an accuracy of 92.13% in identifying 241 programmers. This result demonstrates that using deep representations of authorship attribution enables accurate authorship identification in the wild.

Key Insight. The reported results using the GitHub dataset show some accuracy degradation in comparison with the results obtained using GCJ dataset given the same number of programmers. This degradation in the accuracy might be because of the authenticity of the dataset ground truth.

The assumption behind establishing the ground truth for our dataset is only true to some extent since the contributor of a GitHub repository could copy code segments or even code files from other sources. Such ground truth problem influences the result of the authorship identification process. In real-world applications, this problem does not occur much often since most scenarios entails having authentic dataset.

3.6.2 Software Authorship Identification In The Open World

Authorship identification using open-world assumption is applicable in a real-world scenario when attempting to identify the author of a given software, who might not be included in the suspect set. In contrast to the conventional machine learning approach, in which the model evaluation is based on unseen samples of labels that the model trained on during the training phase, addressing open world problem raise another challenge in indicating whether a given tested sample belongs to a new unseen label. This setting is more reasonable for software forensics since analysts aim to attribute pieces of software, *e.g.,* malware, that can possibly be created by new programmers who are not part of the suspect set. Previous works by Caliskan *et al.* [32] and Dauber *et al.* [39] have addressed the problem of authorship identification in an open world scenario. The authors used classification confidence as an indicator of sample-label-membership, where high classification confidence demonstrates a high probability of classified labels, whereas low confidence signals model hesitation of the classification decision. In ensemble classifier, such as the adopted RFC, the percentage of voted trees for a certain label reflects the model classification confidence. For an author (A_i), the classification confidence of a RFC identification model is estimated by percentage of trees voted for (A_i) when testing a sample, and it can be formulated as $Conf(A_i) = \frac{\sum_j Vote_j(A_i)}{\|T\|}$, where $Vote_j(A_i)$ is the vote of tree j for A_i and $\|T\|$ is the total number of trees in RFC model.

Addressing open world identification requires setting up a confidence threshold where classifi-

cations with higher confidence level are accepted, while classifications below the threshold are rejected and reported as possible membership of new unseen labels. One way of estimating the confidence threshold for a classifier is by classification margin defined by the difference between the highest and second highest $Conf(A_i)$ of a given sample [32].

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

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

of the precision and recall metrics.

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

3.7 Limitations

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

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

Authorship confusion. Since this work adopts a machine learning approach to identify programmers, it will only succeed if similar patterns from the training data are captured in the test dataset.

As a pathological case, consider the *authorship confusion attack* or *mimicry attack* where the tested samples are contaminated to evade identification. Such contamination in the code could cause substantial changes of the programming style, thus making it difficult (if not impossible) to correctly identify the involved programmer.

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

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

3.8 Conclusion

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

CHAPTER 4: MULTI-X: IDENTIFYING MULTIPLE AUTHORS FROM SOURCE CODE FILES [1]

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-χ. Multi-χ 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-χ, 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-χ with several Github projects (Caffe, Facebook's Folly, TensorFlow, etc.) and show remarkable accuracy. For example, Multi-χ 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.

[1]This content was reproduced from the following article: Mohammed Abuhamad, Tamer Abuhmed, DaeHun Nyang, David Mohaisen, "Multi-χ: Identifying Multiple Authors from Source Code Files," In Proceedings of the 20th Privacy Enhancing Technologies Symposium, (PETS), 2020.

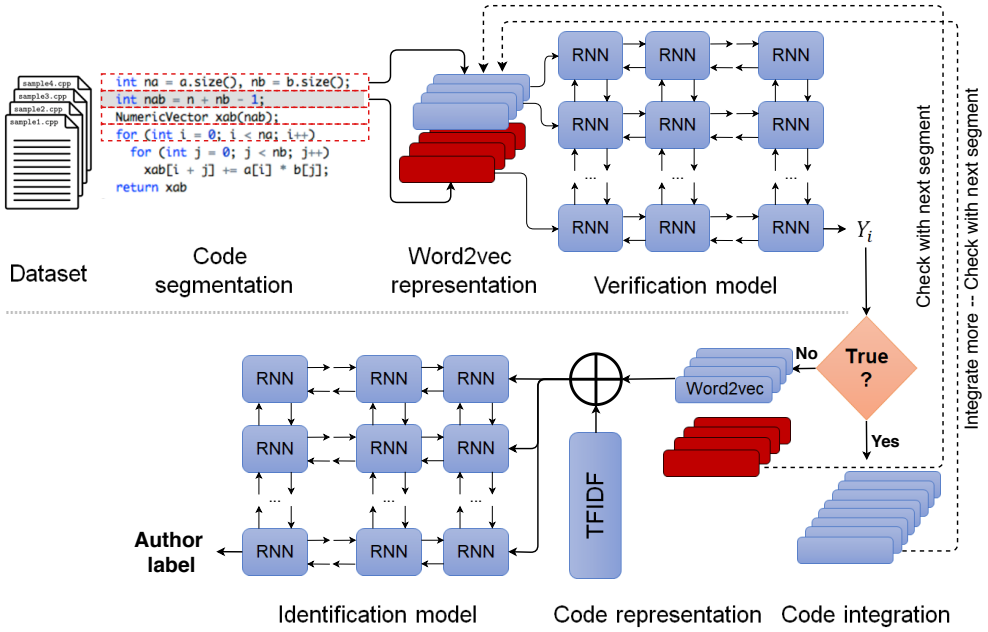


Figure 4.1: The general outline of the proposed approach. The code authorship verification includes processing code segments represented by *word2vec* to the verification model. The code authorship identification includes integrating code segments based on the verification process to be represented using *word2vec* or TF-IDF for the identification model.

4.1 Multi-X: System Design

Multi-χ contributes to solving the multi-author code authorship identification problem using an RNN-based system that incorporates five processes, which are: code processing and segmentation (Subsection 4.1.2), code sequence representation (Subsection 4.1.3), code authorship verification (Subsection 4.1.4), code segment integration (Subsection 4.1.5), and code authorship identification (Subsection 4.1.6). The overall process operates is shown in Figure 4.1. 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 (Subsection 4.1.1).

4.1.1 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-χ 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 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.

4.1.2 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 [47], 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 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 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

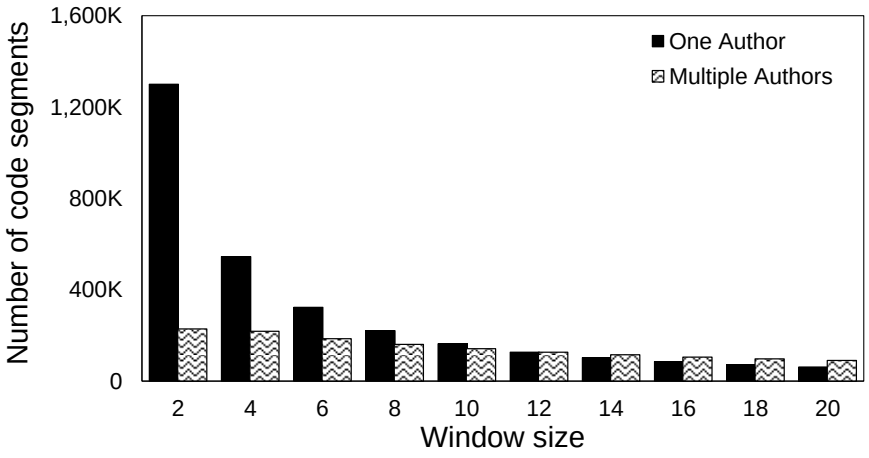


Figure 4.2: Number of segments written by one and multiple authors in nine open-source projects.

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 4.2 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 4.3 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.

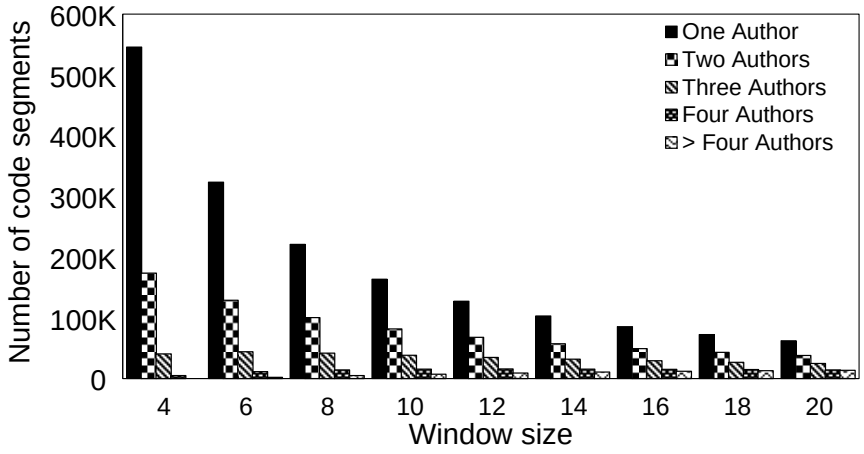


Figure 4.3: Number of segments written by a specific number of authors in nine open-source projects.

4.1.3 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* [103] and GloVe [83], 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 [19]. Second, *word2vec*, as a learning method of generating distributed representations of tokens, has shown remarkable success in a wide range of

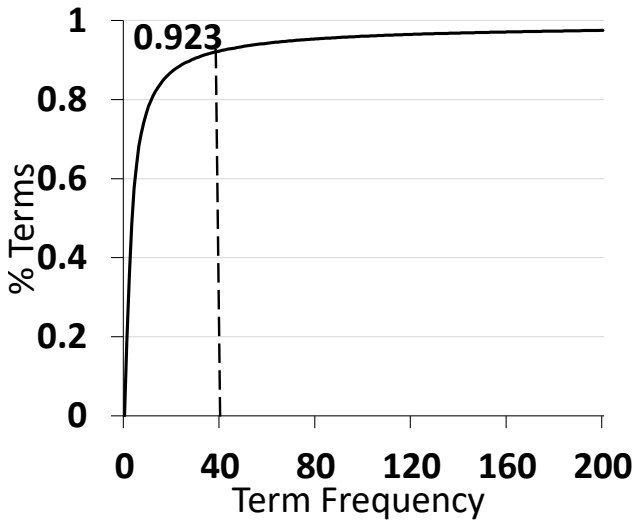


Figure 4.4: The frequency of terms in our dataset. Note more than 92% of terms have less frequency than 40.

applications (*e.g.,* [91, 77, 36, 94, 52]).

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 4.4. 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 [20]. In this study, unseen terms are represented with zero-vectors when using *word2vec* and ignored in TF-IDF.

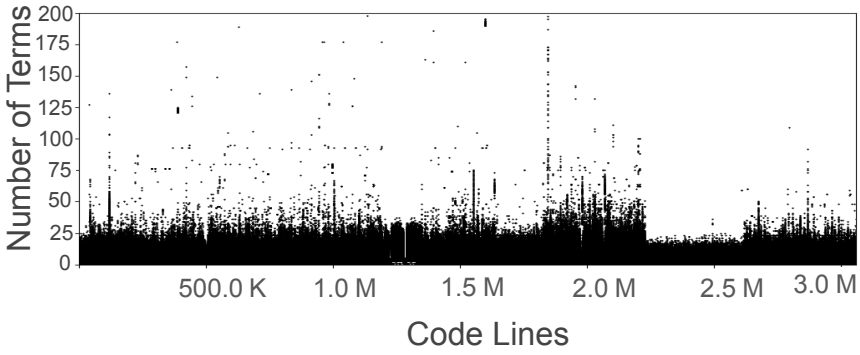


Figure 4.5: Number of terms per code lines in our dataset. The maximum number of terms is 17,315, while the average is 7.6 $\approx$ 8.

4.1.4 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 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.

4.1.5 Code Segments Integration

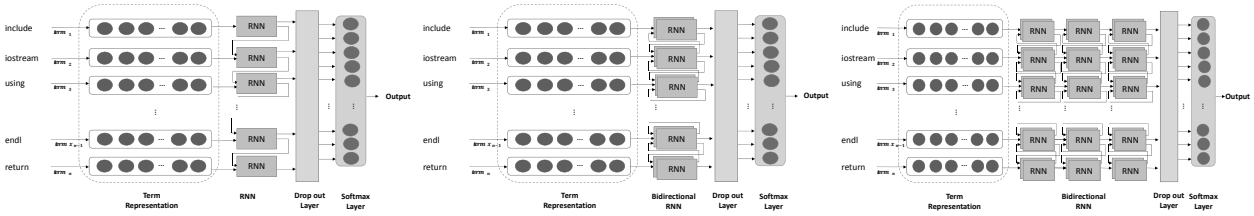
Subsequent segments that are written by the same author can be integrated into one larger segment. This step is important as larger segments exhibits more indicative authorship attribution than smaller segments. To automate the integration process using our fine-grained approach, we use the authorship 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 4.5. 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 4.2. 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.

4.1.6 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. Other representation method for code segments is the TF-IDF. Unlike the verification task, the identification task uses larger segments, 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) [23]. For scalability and robustness, several works [3, 31, 75, 40] 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.

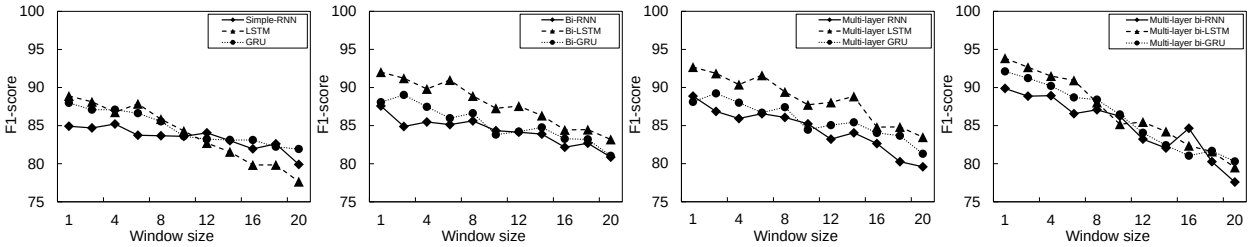


(a) One layer RNN model (b) Bidirectional RNN model (c) Multi-layer Bidirectional RNN model

Figure 4.6: Different RNN model architectures used for code authorship verification and identification.

4.1.7 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-χ explores the performance of different RNN structures namely, traditional simple RNN, Long Short-Term Memory (LSTM) [59], and Gated Recurrent Unit (GRU) [34]. 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 [59, 34]. 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 [3]. 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 4.6(a). In this section, we explain the model architectures



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

Figure 4.7: 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.

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 the beginning to the end of the sequence, while the other operates in the opposite direction as in Figure 4.6(b).

Multi-layer RNN. Deep RNNs with multiple hidden layers have shown a remarkable capability of capturing nonlinear patterns from long input sequences [92]. In this work, we also investigate the performance of Multi-χ using multi-layer RNN. Figure 4.6(c) shows an example of an RNN with multiple hidden layers.

Table 4.1: 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

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* [101] algorithm, which is commonly used with RNN [92]. 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 [96], 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 (Subsection 4.2.2) and segment authorship identification (Subsection 4.2.3) tasks. We note that the experiments in Subsection 4.2.2 and Subsection 4.2.3 aim to establish proper settings for fine-grained authorship identification approach (using end-to-end identification as in Subsection 4.2.4) by investigating the effects of code segment size, data representation, model structure, and experimental hyperparameters on performed task. The code segments are collected and processed based on ground-truth dataset. The 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 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 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 4.5.

4.2 Evaluation and Experiments

We evaluate Multi-χ 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.

4.2.1 Dataset

Multi-χ uses a real-world dataset of nine open-source projects available on Github, namely: Caffe Library, Cosmos Algorithms Collection repository [37], Dyninst API tools for binary instrumentation [25], Facebook Open-source Library (folly) [46], GNU Compiler Collection (GCC) [50], Apache HTTP Server [12], Open Source Computer Vision Library (OpenCV) [78], Swift Programming Language [13], and TensorFlow Library[100]. We use *git-author* [76] 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.

4.2.2 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 verifica-

tion models, we use RNN with 64 hidden units and a maximum of two hidden layers when using multi-layers RNN architectures.

Results. Table 4.1 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 4.7(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 4.7(b). Moreover, deeper architectures with multiple layers achieve better results as illustrated in Figure 4.7(c) and Figure 4.7(d).

4.2.3 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

Table 4.2: 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

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 4.2 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 4.2, 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 Subsection 4.2.2, 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 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 4.8 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

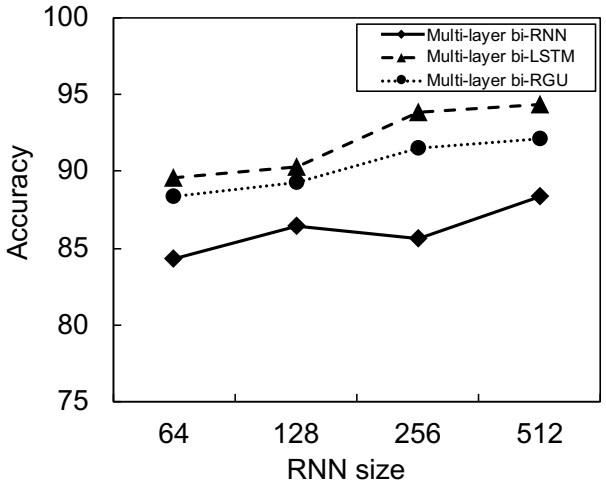
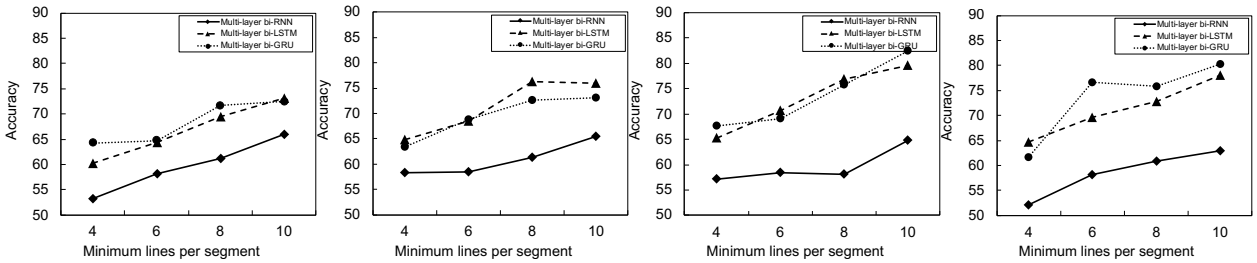


Figure 4.8: Accuracy of authorship identification achieved by RFC using different *word2vec*-RNN-based embeddings sizes using a dataset of 282 programmers with at least 30 samples.

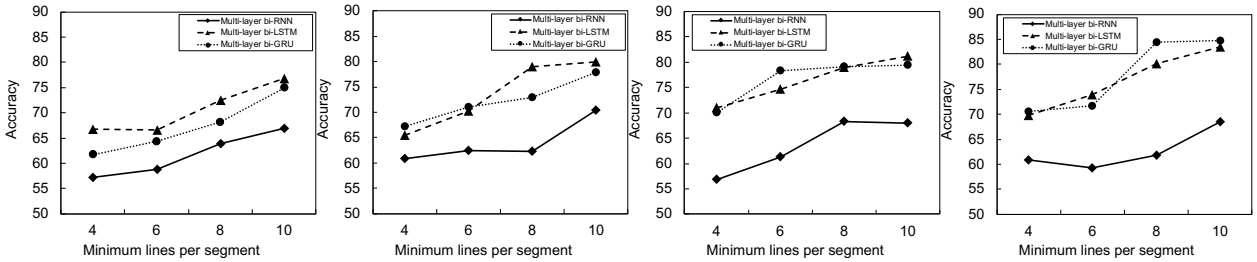


(a) At least 6 samples (b) At least 10 samples (c) At least 20 samples (d) At least 30 samples

Figure 4.9: 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.

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 4.9 shows the accuracy of our identification approach using

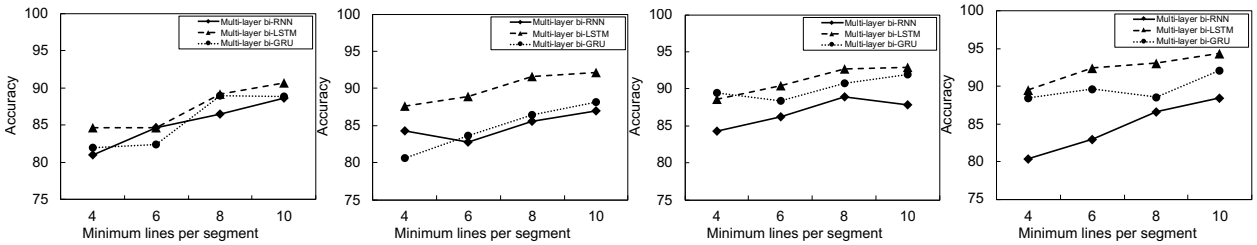


(a) At least 6 samples (b) At least 10 samples (c) At least 20 samples (d) At least 30 samples

Figure 4.10: 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.

word2vec representation with varying samples per author and varying sample sizes. In particular, Figure 4.9(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 4.9(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 4.9(c). However, the accuracy slightly decreases when increasing the number of samples to exceed 30 samples per author. Figure 4.9(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 4.10 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 4.10(a) shows that our approach with LSTM unit is



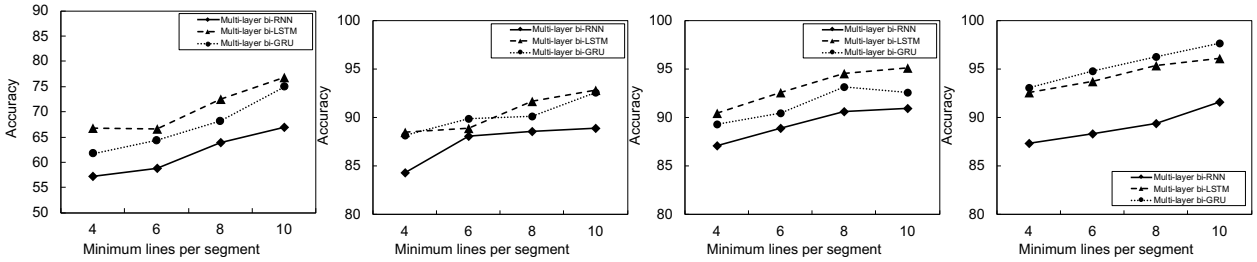
(a) At least 6 samples (b) At least 10 samples (c) At least 20 samples (d) At least 30 samples

Figure 4.11: 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.

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 4.10(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 4.10(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



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

Figure 4.12: 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.

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 4.11 shows the identification accuracy of RFC over *word2vec*-based sequence embeddings generated with different RNN types. Figure 4.11(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 4.11(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 4.11(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 4.11(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 4.12. In Figure 4.12(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 4.12(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 4.12(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 4.12(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.

4.2.4 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-χ. The main purpose of this evaluation is to demonstrate Multi-χ's effectiveness in general rather than its accuracy on an individual task. For this evaluation, Multi-χ 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 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 [80, 41]. Therefore, the evaluation of the overall system performance can be calculated using a similar metric as the

example-based accuracy (EBA) used in [41], 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-χ 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.

4.3 Limitations

This work demonstrates that sufficient authorship attributions can be extracted from the smallest piece of code to enable accurate authorship identification. Nevertheless, Multi-χ has several limitations concerning the ground-truth dataset, dealing with binary code, and obfuscated code.

Ground-Truth Assumption. This work assumes authorship of code lines based on the *git-author* tool [76]. 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 [32] 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 [3] showed that deep learning representation enabled accurate code authorship identification for obfuscated code. Another work by Brennan *et al.* [24] 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.

4.4 Conclusion

We have proposed Multi-χ, 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-χ, we have used a large scale dataset including nine real-world open-source projects. Multi-χ 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 accuracy for future investigation.

CHAPTER 5: AUTHOR-SHIELD: CIRCUMVENTING CODE AUTHORSHIP IDENTIFICATION USING ADVERSARIAL EXAMPLES

Authorship identification has become increasingly accurate, posing a serious privacy risk for programmers who wish to remain anonymous. In this chapter, we introduce Author-SHIELD to examine the robustness of different code authorship attribution approaches against adversarial examples. We define three adversarial attacks on attribution techniques—confidence reduction, a programmer imitation, and evasion attacks—and realize them in targeted and non-targeted adversarial code perturbation. We experiment with a dataset of 2,000 C++ programmers from the Google Code Jam competition to validate our methods targeting six state-of-the-art authorship identification methods that adopt a variety of techniques for extracting authorship traits from source-code, including RNN, CNN, and code stylometry. Our experiments demonstrate the vulnerability of current authorship attribution methods against adversarial attacks. For the confidence reduction attack, our experiments demonstrate the vulnerability of current authorship attribution methods against the attack, and show a degradation of the identification confidence ranging from 20% to 60% when introducing adversarial perturbations. This confidence reduction allowed a misidentification rate that exceeds 98% for all targeted systems, even with the smallest perturbation (one line of code). For the imitation attack, we show the possibility of impersonating a programmer using targeted-adversarial perturbations with an imitation success rate ranging from 48% to 55% for different authorship identification techniques. For authorship evasion, the results show a high evasion success rate reaching to 58%.

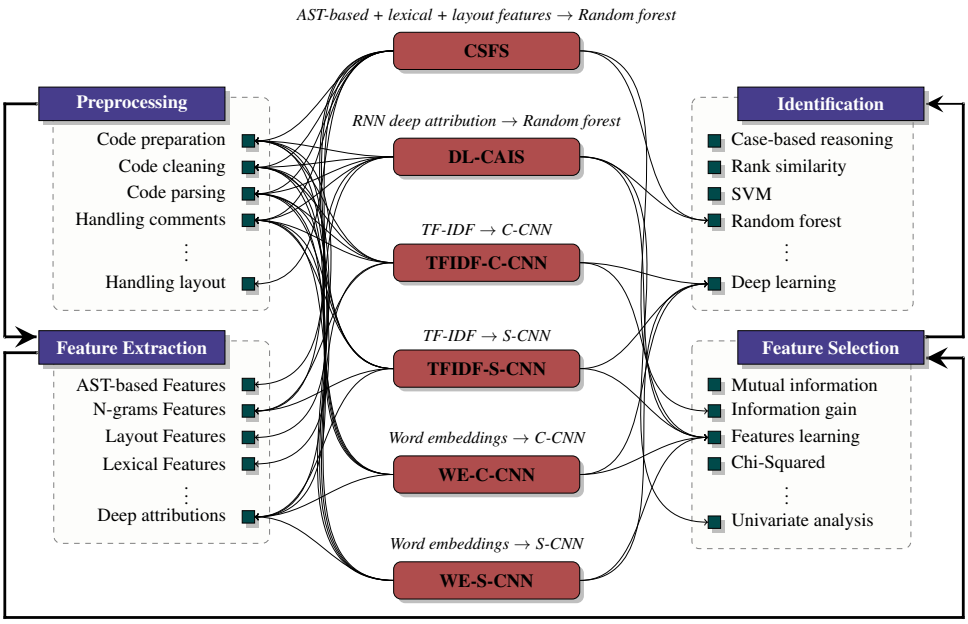


Figure 5.1: The workflow of source-code authorship identification, including four phases: preprocessing, feature extraction, feature selection, and identification.

5.1 Author-SHIELD: Methods

The workflow of typical source-code authorship identification systems includes three stages: data preparation and preprocessing, feature extraction, and authorship identification. The data preparation and preprocessing stage entails accessing software programs, removing undesired parts, *e.g.,* comments, links to external resources, etc., and normalizing layout features, *e.g.,* white spaces and lines. The feature extraction stage, often referred to as authorship attributes extraction, entails defining distinctive programmers' coding traits. Extracting such traits requires designing features that capture specific characteristics of different programming styles and methods. Such features may include: (1) syntactic features in the program structure and the implementation choices, (2) stylometric features in variables naming, documentation, language's reserved keywords usage,

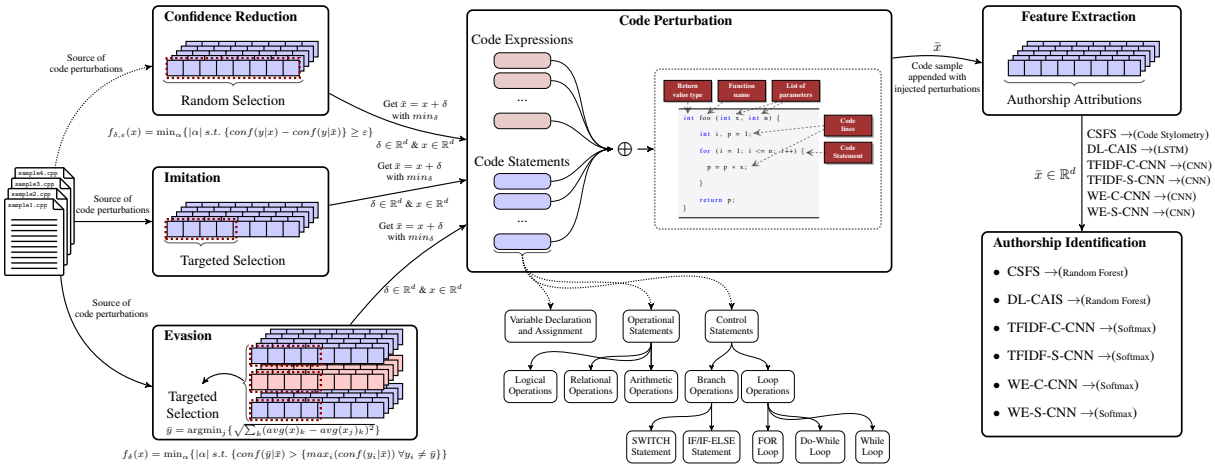


Figure 5.2: An overview of the adversarial attacks adopted by Author-SHIELD on different code authorship identification approaches.

etc., (3) layout features in white spaces and indentation usage, and (4) development environment features in the usage of platforms, editors, programming languages, etc.

The availability of a wide range of candidate features in large-scale cross-language identification tasks makes the feature selection process an important phase. The feature selection process typically includes evaluating the authorship attributes based on, for example, the information gain or mutual information between attributes and authors to determine a small subset of prominent features for identification. The final stage is the authorship identification, which is usually treated as a classification problem. Using the extracted (or selected) features, classification models, such as the Support Vector Machine (SVM) [82, 30], Random Forest Classifier (RFC) [3, 31], and neural networks [30], are often trained to identify programmers in a supervised manner.

The general workflow of the authorship identification is shown in Figure 5.1, highlighting methods used by the six approaches investigated in this study, namely: Code Stylometry (CSFS) [31] Deep Learning-based Code Authorship Identification System (DL-CAIS) [5], TF-IDF-based Concatenated CNN (TFIDF-C-CNN TF-IDF-based Stacked CNN (TFIDF-S-CNN) [5], Word Embedding-

based Concatenated CNN (WE-C-CNN) [5], and Word Embedding-based Stacked CNN (WE-S-CNN)[5].

Our robustness assessment for the targeted approaches includes adding code perturbations at the source code level rather than the feature space of authorship attributes, providing a more realistic scenario; *e.g.,* an adversary may have knowledge of the feature extraction techniques used by the targeted system, but without the ability to manipulate the features after submitting the code sample. This case only allows the adversary to manipulate the input samples on the source-code level. Moreover, perturbing feature representations might not reflect actual changes in the source code. To this end, we designed templates for code perturbation to inject into the source-code to serve the adversarial attack's objective. The configurations of different attacks are specified based on different scenarios and requirements. For example, one adversary could generate an attack for the purpose of misidentification, while another generates a different attack to imitate the coding style of a specific programmer.

This chapter investigates several aspects regarding code authorship adversarial attacks, including adversarial falsification (misclassification), adversarial specificity (imitation and evasion). Other aspects regarding the introduced perturbation should be considered, such as the size of perturbations, insights for defense, and adversarial perturbation detectability. Since the fundamental merit behind adversarial examples is the undetectability by humans, the perturbation should be as small as possible that make them imperceptible to a human. This can be a challenging task when working with source code samples since the adversarial examples should preserve the functionality and follow the syntax-rules of the targeted programming language. This work studies the code perturbation scope, limitation, and measurement. Figure 5.2 shows an overview of the approach adopted by Author-SHIELD to generate adversarial attacks on different authorship attribution approaches.

5.1.1 Code vs. Feature Perturbation

Adversarial perturbations are often introduced on the input feature space to allow enhanceing its effect while decreasing its size. However, introducing perturbations on the feature representations (*i.e.,* attributions) assumes a white-box attack where the adversary has full control over all stages of the system operation, limiting the approach's practicality.

Our Approach. Assuming that the adversary has limited knowledge, we investigate adding perturbations to the input code directly by code injection. The code perturbations can be added as variable declarations, loops and control statements, functions, etc. In reality, and for an author that tries to evade detection, this approach would be implemented by the author who would have full control over the source code. For preserving the functionality of the code, the added perturbations are not meant for execution, and we achieve that by adding code perturbations as methods (function) that never get called or executed in the main code. To enhance the code perturbations: ❶ we limit the size of perturbation to the minimum size that enables the attack, ❷ the function code should inherit the flow of the syntax rules of the programming language, ❸ the statements of the adversarial function should have the same size as other statements in the original code, ❹ variable names should be from the collection of vocabulary used by the programmers in the dataset.

Data used in our evaluation includes code samples written in C++, and perturbations are presented as C++ functions following the C++ function structure with a function body that includes multiple statements. Six major statements types are considered in the implementation: ❶ variable declaration and assignment, ❷ arithmetic, relational, and logical operations, ❸ IF, IF-ELSE statements, ❹ SWITCH statement, ❺ FOR loop statement, and ❻ DO-WHILE, WHILE loop statement. Each statement includes at least one line of code, *e.g.,* a declaration, an assignment, or an operation. The selection of statements' types and the naming criteria for variables and functions for the code perturbations are conducted based on the underlying assumptions of the attack. For ex-

ample in the imitation attack, we limit the considered names to the ones previously observed for the targeted programmer, since using random names might result in an out-of-vocabulary problem for approaches that use TF-IDF or word embeddings for representing the code sample. In such cases, the names that are not previously seen during the training are not considered in the code representation, and therefore they become useless in adding value to the perturbation. This work investigates code authorship attribution under three adversarial scenarios: confidence reduction, code style imitation, and evasion.

5.1.2 Identification Confidence Reduction

The confidence is defined as the probability distribution, $P(y_i|x_i) \quad \forall x_i \in \mathcal{X}$ and $y_i \in \mathcal{Y}$, of assigning programmers $\mathcal{Y} : \{y_0, y_1, \ldots, y_m\}$ to input data $\mathcal{X} : \{x_0, x_1, \ldots, x_n\}$. For the targeted approaches, confidence is defined as follows. DL-CAIS and CSFS adopt RFC where the confidence is the number of decision trees voting for a given class. For an author y_i, the identification confidence is the percentage of trees voted for y_i to be the programmer who wrote a given code sample x_i, and it is estimated as $\text{conf}(y_i) = \sum_j \text{vote}_j(y_i) \times \|T\|^{-1}$, where $\text{vote}_j(y_i)$ is the j-th tree voting for y_i and $\|T\|$ is the total number of trees in RFC. On the other hand, CNN-based models (WE-C-CNN, WE-S-CNN, TFIDF-C-CNN, and TFIDF-S-CNN) adopt softmax classifier where the confidence is the softmax score for a given class. For an author y_i, the identification confidence is $\text{conf}(y_i) = e^{y_i} \times (\sum_j e^{y_j})^{-1} \quad \forall\, j$.

Goals. This technique aims to delude the identification model by manipulating the input code files so that authorship attributes become ambiguous. More precisely, the goal is to decrease the confidence in the models' predictions to lead the model for misidentification or prediction rejection. This is conducted by adding code perturbation $\delta \in \mathbb{R}^d$ to the input code $x \in \mathbb{R}^d$ such as the generated adversarial code $\bar{x} = x + \delta$ serves the purpose of minimizing the confidence with at least

$\varepsilon > 0$. Formally, the adversary goal is:

$$f_{\delta,\varepsilon}(x) = \min_{\delta}\{|\delta| \quad s.t. \quad \{\text{conf}(y|x) - \text{conf}(y|\bar{x})\} \geq \varepsilon\}$$

The conf$(y|x)$ is correct assignment of x to the rightful programmer y. If conf$(y|\bar{x}) <$ conf$(\bar{y}|\bar{x})$, then the adversarial sample $\bar{x}$ is attributed to $\bar{y}$ (misidentification).

Capabilities. To achieve the above goal of confidence reduction, we assume that the adversary does not have knowledge at the level of the training data. Rather, the perturbation $\delta \in \mathbb{R}^d$ is generated randomly regardless of the code representation scheme adopted by the targeted system. Note that the adversary gains more advantage when having access to the training data since knowing the space of code expressions used in the dataset and their possible impact on the authorship attribution reduces significantly the size of δ.

5.1.3 Code Style Imitation

In this attack, we aim to fool the model to predict a specific class of interest (an author to imitated). This kind of attack occurs when the targeted models are used to predict multiple classes. Using the code authorship identification system, a targeted attack aims to maximize the probability of adversarial code sample to be classified as the targeted adversarial class. If successful, this attack will enable programmers to imitate the coding style of other programmers or at least mislead the identification system to predict the targeted programmer. Since the identification decision is the argmax_j conf$(y_j|x)$ $\forall j$, the model predictions can be changed based on the distribution of classes. Using the same definition of conf$(y_j|x)$, the adversary aims to minimize the confidence of predicting the right programmer and maximizing the probability of the target.

Goals. The goal of code style imitation is to maximize the confidence of the models' predictions towards a target class $\bar{y}$ to lead the model to predict $\bar{y}$. Similar to the previous model, this is

conducted by code perturbation $\delta \in \mathbb{R}^d$ on $x \in \mathbb{R}^d$ such as the generated adversarial code $\bar{x} = x + \delta$ serves the purpose of maximizing the confidence of predicting a targeted programmer $\bar{y}$. Thus, the goal is formally defined as:

$$f_\delta(x) = \min_\delta \{|\delta| \; s.t. \; \{\text{conf}(\bar{y}|\bar{x}) > \{\max_i (\text{conf}(y_i|\bar{x})) \; \forall y_i \neq \bar{y}\}\}$$

If the $\text{argmax}_j\{\text{conf}(y_j|\bar{x})\} = \text{conf}(\bar{y}|\bar{x})$, the model predicts the targeted programmer to be the author of $\bar{x}$.

Capabilities. The imitation attack assumes the adversary knows the training data and the code representation techniques without having any access to the system. The perturbation $\delta \in \mathbb{R}^d$ are generated based on the most representative features of the target programmer to enhance the adversarial code sample' probability to be assigned to the target programmer.

5.1.4 Code Style Disguise (Evasion)

In contrast to targeted attacks, the goal of this attack is to minimize the probability of assigning an adversarial example to a specific class. In this case, the model can output an arbitrary class except for the original one. For example, an adversary makes a given programmer's coding style as any other programmer so that it can be disguised to evade identification (evasion). Implementing this attack can be easier compared to targeted attacks (imitation) due to the flexibility of the feature space and options in assigning an adversarial class, especially when the number of classes is large.

Identification evasion can be conducted by generating adversarial examples in several ways. One way is by confidence reduction with random perturbation. However, this method of conducting the evasion attack does not guarantee the misidentification given a limited size of perturbation. On the other hand, conducting the disguise adversarial attack by implementing targeted imitations of other

classes that are most close to the adversarial code's programmer. In other words, the adversary can imitate or use the features of the closest coding style to generate limited perturbations.

Goals. This attack's goal is to maximize the second highest confidence of the models' predictions so that conf$(\bar{y}|\bar{x}) >$ conf$(y|\bar{x})$, where $\bar{y}$ is the closest programmer in coding style to the adversarial programmer. This does not necessarily mean that the final decision of the model is to predict $\bar{y}$ since argmax_j conf$(y_j|\bar{x})$ $\forall j$ can be any class but y. First, the adversary needs to find the closest programmer in coding style to be imitated. Let $\exists$ m programmers $\{y_1, y_2, \ldots, y_m\}$ with n code samples each. For y_i, the closest programmer $\bar{y}$ is the one with code samples that show the least Euclidean distance to the original samples by y_i. Let avg$(x) = \sum_n (\sum_k x_k^{(i)} \times k^{-1}) \times n^{-1}$ is the average of all n code samples such as $x^{(i)}$ that belong to a given programmer. The closest programmer $\bar{y}$ to a programmer y is: $\bar{y} = \text{argmin}_j \{\sqrt{\sum_k (\text{avg}(x)_k - \text{avg}(x_j)_k)^2}\}$. After defining the nearest programmer $\bar{y}$, the adversary retrieves the most influential features using the order of features to be the source for the perturbation. Similar to the imitation attack, this attack is conducted by adding code perturbation $\delta \in \mathbb{R}^d$ to the input code $x \in \mathbb{R}^d$ such as the generated adversarial code $\bar{x} = x + \delta$ maximize the confidence of predicting the programmer $\bar{y}$.

Capabilities. Similar to the imitation attack, the evasion attack assumes the adversary knows the training data and the feature extraction process without having any access to the system models. The perturbation $\delta \in \mathbb{R}^d$ is generated based on the most representative features of the target programmer selected based on the nearest proximity to the adversary.

5.2 Evaluation Settings

5.2.1 Dataset

The evaluation of adversarial code samples on the authorship identification is conducted using Google Code Jam (GCJ) competition [62]. GCJ is an international programming competition run by Google since 2008. At GCJ, programmers use several programming languages and development environments to solve programming problems over multiple rounds. The most common languages at GCJ are C++, Java, Python, and C, in order. For this work, we use a dataset of 2,000 C++ programmers with nine code samples. The dataset of C++ samples is collected from GCJ competition across all years from 2008 to 2016. The number of code samples, programmers, and years are consistent with prior work.

5.2.2 Baseline and Implementation Details

DL-CAIS. For DL-CAIS, the input code files are initially represented with TF-IDF scheme. Following the implementation described earlier, we used the top 2,500 TF-IDF features as a single step vector representation to the LSTM RNN for learning the deep representation. The LSTM architecture includes three layers of 128 LSTM units followed with three fully-connected layers with 1024 units connected to a softmax layer with the size of the considered number of programmers (classes). The LSTM model is trained in a supervised manner, where pairs of input code samples and their corresponding authors are used to train the model by minimizing the softmax-cross-entropy loss. After the training, the LSTM models are used to generate the deep representations of authorship attributions of code samples as the output of the second-last layer of the model. The deep representations of authorship attributes are used as 1024-dimensional vectors to construct RFC with 150 trees grown to the maximum extent.

CSFS. Using the implementation of Caliskan-Islam *et al.* [31], the input code samples are represented with the top 1024 IG-CSFS authorship attributes to construct RFC with 150 trees grown to the maximum extent.

WE-C-CNN. We set the word embedding matrix to generate 128-word representations. The input code samples are then represented as a matrix of $n \times 128$ representations, where n is the number of terms of the code sample. We set $n = 256$ to fix the length representations to the convolutional layers as short samples are padded with zeros and long samples are truncated. For the convolutional layers, three layers of 128 filters of different sizes (3, 4, and 5) are adopted to receive the input representation. The outputs of the three convolutional layers are concatenated and connected to a softmax layer with the same size as the number of the considered classes.

WE-S-CNN. Similar to the settings of WE-C-CNN, the code samples are represented as $x \in \mathbb{R}^{256\times128}$ since the word embeddings are set to be 128 vector representations. The CNN model architecture follows the typical stacked CNN layers with three consecutive layers each with 128 filters of different sizes (3, 4, and 5). For this architecture, we used max-pooling after each layer to reduce the size of feature maps produced by the convolutional layer. The last max-pooling layer is connected to the output softmax layer.

TFIDF-C-CNN. The input code samples are represented with the top 2500 TF-IDF features and fed to 1-dimensional convolutional layers. Similar to WE-C-CNN, we use three filter sizes and 128 filters per layer. The feature maps produced by the three convolutional layers are concatenated and connected to a softmax layer with the same size as the number of classes.

TFIDF-S-CNN. Similar to TFIDF-C-CNN, the input code samples are represented with the top-2500 TF-IDF features and fed to three stacked 1-dimensional convolutional layers. All convolutional layers consist of 128 filters and each layer has different filter size. We used max-pooling for

the stacked CNN architecture and the last max-pooling layer is connected to a softmax layer with the same size as the number of classes.

Deep-Learning Training. Both RNN-based and CNN-based models are trained using the Adam optimizer [66] with a static learning rate of 10^{-4} to minimize the softmax-cross-entropy loss since all models are trained in a supervised manner. The training process is terminated after 1,000 iterations. To prevent overfitting, we used *dropout regularization* with keep-rate of 70% and *L2 regularization* with λ regularization strength of 10^{-3}. Moreover, we used mini-batch approach with mini-batch size of 64 observations in training process of all deep learning-based architectures.

5.2.3 Evaluation Metrics

We evaluate the confidence reduction attack based on the degradation in the model confidence, while the misidentification rate is used as an indication of the success rate for fooling the identification model. When the predicted label $\text{argmax}_k\, P(y_k|\bar{x}_i)$ for the adversarial sample $\bar{x}_i$ is not the same as the correct class label y_i of the original sample x_i, this result is misidentification, and the rate is calculated as:

$$\frac{1}{n}\sum_{i} I(\underset{k}{\text{argmax}}\, P(y_k|\bar{x}_i) \neq y_i).$$

We evaluate the imitation attack by the imitation success rate as the proportion of correctly classified adversarial samples to the targeted programmer to the overall attempts. For targeted class $\bar{y}$, if the $\text{argmax}_k\, P(y_k|\bar{x}_i) = \bar{y}$, then the attack succeeds. The imitation success rate is calculated as:

$$\frac{1}{n \times m}\sum_{j}^{m}\sum_{i}^{n} I(\underset{k}{\text{argmax}}\, P(y_k|\bar{x}_i) = \bar{y}_j).$$

We evaluate the evasion attack by its success rate: correctly misidentifying a specific program-

mer to all presented code samples by that programmer, *i.e.,* similar to the misidentification rate. However, we adopted a targeted evasion attack where an adversarial code sample $\bar{x}_i$ is attributed to the closest programmer $\bar{y}$ with respect to the original programmer y_i, therefore we calculate the evasion success rate similar to the imitation success rate.

Magnitude of Perturbation. We use ℓ_p to measures the magnitude of perturbation by $p\text{–}norm$ distance as:

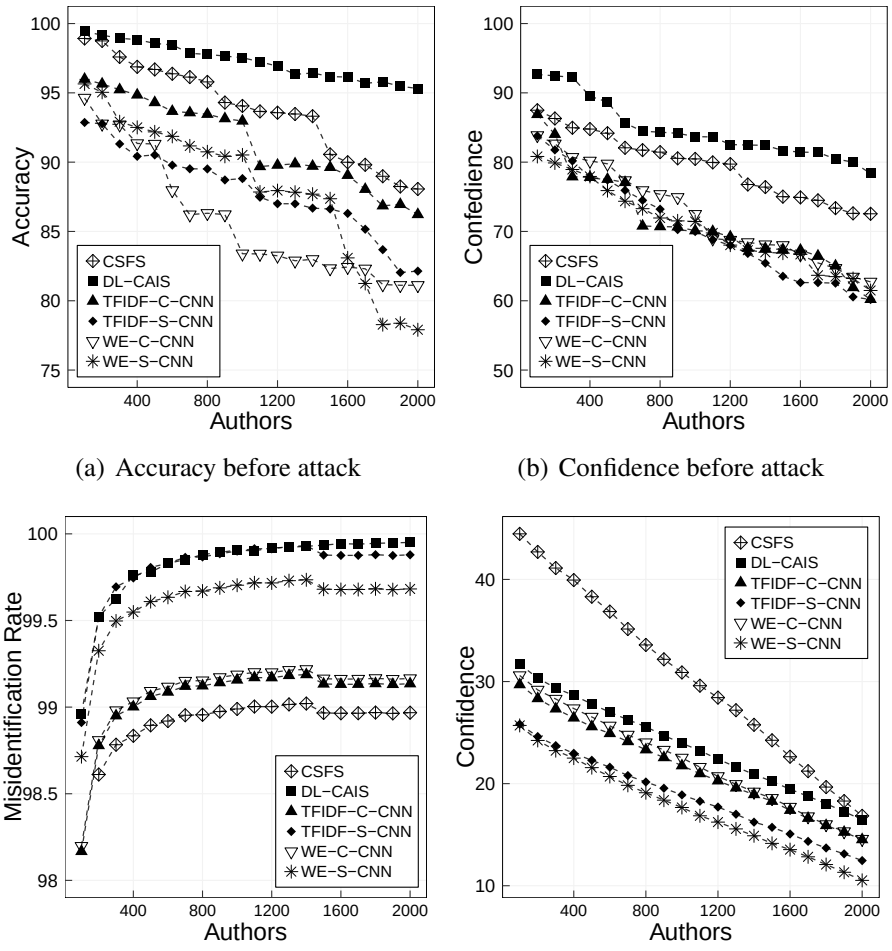
$$\|\delta\|_p = \left(\sum_{i=1}^{n} \|\bar{x}_i - x_i\|^p \right)^{\frac{1}{p}}$$

For $p\text{–}norm$, studying the ℓ_0, ℓ_2 and ℓ_∞ is very common [33]. The ℓ_0 is the count of changes in the adversarial example compared to the original sample, the ℓ_2 is the Euclidean distance between the adversarial sample and the original sample, and ℓ_∞ is the maximum changes with respect to all terms in the adversarial examples. In this work, we report the size of perturbation with respect to the number of code lines or code statements when showing the results. However, we discuss the magnitude of perturbation by $p\text{–}norm$ when considering different attacks implemented with a different number of code lines and statements in Section 5.4.

5.3 Experiments and Results

5.3.1 Confidence Reduction Attack

For this attack, we created a random function generator to inject perturbations to the original code. The optimization of the code perturbation size follows the minimization of the probabilities of assigning adversarial examples to the original programmers. That reflects on the number and types of statements as well as the number, types, and names of variable declarations. To evaluate the success of different adversarial attacks, we implemented different code authorship identification



(a) Accuracy before attack

(b) Confidence before attack

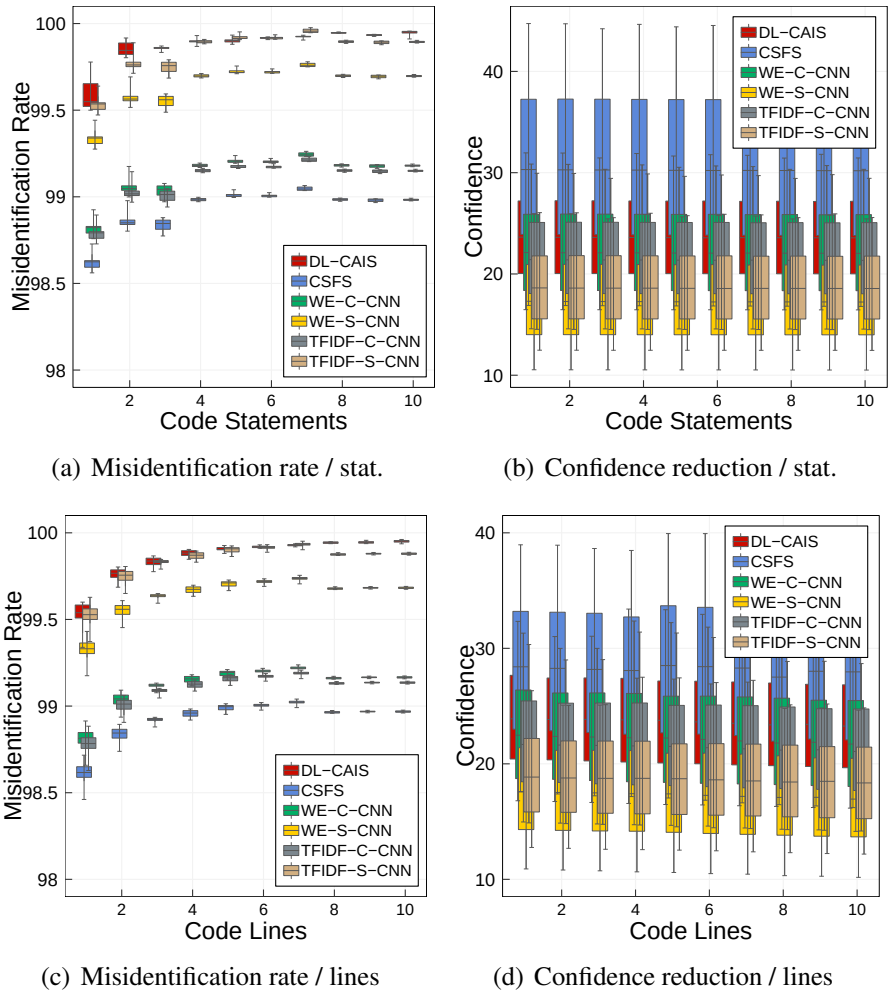
(c) Misidentification after attack

(d) Confidence after attack

Figure 5.3: The accuracy and confidence of identification models of the baseline approaches before and after launching the confidence reduction attack. The results before the attack are reported as the average results of 9-cross-validation evaluation.

systems to be the target of the attacks. Using a dataset of 2,000 programmers, the baseline systems have achieved remarkable results, outlined as follows.

Baseline Identification Results. Figure 5.3(a) shows that RNN-based deep representations adopted by DL-CAIS has enabled an identification accuracy ranging from 95.31% for identifying 2,000 programmers to 99.42% for identifying 100 programmers. Code stylometry features adopted in



(a) Misidentification rate / stat.

(b) Confidence reduction / stat.

(c) Misidentification rate / lines

(d) Confidence reduction / lines

Figure 5.4: The results of the confidence reduction attack: the misidentification rate and the confidence reduction of the targeted approaches categorized by the added perturbations of code lines (ranging from 1 to 10 per statement) and code statements (ranging from 1 to 10). Notice, the impact of added code statements is larger than the considered code lines within the statements.

CSFS has achieved identification accuracy ranging from 88.1% to 98.92% for different datasets. Using word embeddings to represent code files to CNN-based approaches, the WE-C-CNN and WE-S-CNN architectures have achieved an identification accuracy of 81.12% and 77.91% for identifying 2,000 programmers, respectively. Using TF-IDF representations with CNN, have enhanced the identification accuracy to reach to 86.22% and 82.13% for identifying 2,000 programmers

using TFIDF-C-CNN and TFIDF-S-CNN, respectively. Figure 5.3(a) shows the identification accuracy achieved for different sub-datasets with a different number of programmers using 9-fold cross-validation evaluation of all targeted approaches.

This identification accuracy is accompanied with high identification confidence. Figure 5.3(b) shows the average identification confidence of all targeted approaches when identifying different numbers of programmers. For all identification techniques, the confidence exceeds 60% in all settings. For systems that adopt RFC, such as DL-CAIS and CSFS, the model confidence is higher than the systems that utilize softmax layer for classification such as the CNN-based systems (WE-C-CNN, WE-S-CNN, TFIDF-C-CNN, and TFIDF-S-CNN). This is because the confidence of RFC models is defined by the number of trees in the random forest that vote for a given class, while the softmax layer captures the probability distribution of assigning the input data to all classes. In our experiments, the identification confidence increases as the number of programmers increases as shown in Figure 5.3(b).

Impact of Confidence Reduction Attacks. The results of this attack are shown in Figure 5.4 including the misidentification rate and the model confidence. For this attack, we use random code injections to reduce the confidence and to improve the misidentification success rate of programmers. Figure 5.4 shows the misidentification success rate of targeted approaches using different experimental settings. The attack allows an average misidentification success rate above 98.5% for all targeted approaches using different settings and across different numbers of programmers. This high misidentification accuracy is caused by the low confidence levels of identifying programmers as shown in Figure 5.4(b) and Figure 5.4(d). The confidence scores, with a high point of 43.51% achieved by CSFS and a low point of 10.14% achieved by WE-S-CNN, show the class ambiguity caused by the code perturbations.

Number of Programmers' Impact. Figure 5.4(a) shows the average targeted misidentification

rate by changing the number of code statements. Even when using only one statement the average achieved misidentification rate is higher than 98.5%, demonstrating the success of the confidence reduction attack. The average misidentification accuracy per a number of statements is reported for using a different number of lines per a statement ranging from one to ten lines. The smallest size of perturbation is to inject a function with one statement that includes only a declaration, while the largest size is to include ten statements with ten lines each. For 200 programmers, the average misidentification rate when adding one statement of perturbations is 99.61%, 98.59%, 98.79%, 99.30%, 98.76%, 99.50% for the targeted attacks on DL-CAIS, CSFS, WE-C-CNN, WE-S-CNN, TFIDF-C-CNN, and TFIDF-S-CNN, respectively. When increasing the size of perturbation to ten statements, the misidentification rate becomes 99.52%, 98.63%, 98.83%, 99.34%, 98.80%, and 99.54% for the targeted system DL-CAIS, CSFS, WE-C-CNN, WE-S-CNN, TFIDF-C-CNN, and TFIDF-S-CNN, respectively. Increasing the dataset size to include 2,000 programmers results in increasing the misidentification rate as the results show an average misidentification rate of 99.91%, 98.99%, 99.18%, 99.71%, 99.16%, and 99.90% for DL-CAIS, CSFS, WE-C-CNN, WE-S-CNN, TFIDF-C-CNN, and TFIDF-S-CNN, respectively, when using one statement of code perturbation. As the dataset size increases, the effects of increasing the perturbation size decreases; an average misidentification rate of 99.95%, 98.97%, 99.17%, 99.69%, 99.14%, and 99.89% for targeted system DL-CAIS, CSFS, WE-C-CNN, WE-S-CNN, TFIDF-C-CNN, and TFIDF-S-CNN.

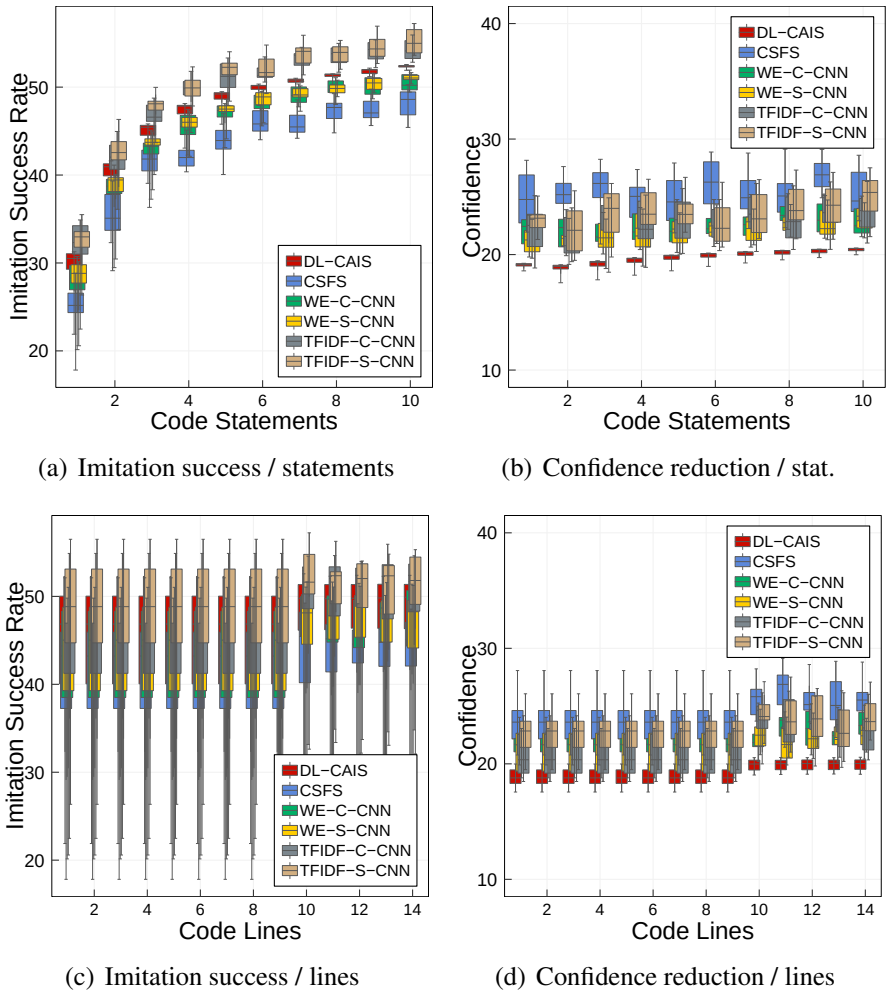
Lines of Code' Impact. Figure 5.4(c) shows the average misidentification rate with different lines of code in the injected statements. Typically, the number of lines as a variable does not seem to affect the overall performance of the attack, when using different numbers of statements, which is explained by the fact that the perturbations are introduced by the usage of statements themselves. The results of the experiments show that including a large number of lines of code within one programming statement does not always help with the misidentification rate. For example, and for the 2,000 programmers dataset, the difference in the achieved average misidentification rate

for using one line of code per statement and ten line of code is (99.95-99.94= 0.01%), (98.98-98.97=0.01%), (99.17-99.17=0%), (99.14-99.14=0%), (99.69-99.68=0.01%), and (99.89-99.89 = 0%) for the targeted systems DL-CAIS, CSFS, WE-C-CNN, WE-S-CNN, TFIDF-C-CNN, and TFIDF-S-CNN, respectively. These results show that using random code perturbation cripples the identification models' capabilities for identifying programmers. Moreover, even when using the smallest size of perturbation (one statement with one line of code), the achieved misidentification rate exceeds 98.5% for all approaches when using a dataset of 2,000 programmers.

5.3.2 Imitation Attack

To evaluate this attack, we randomly selected a subset of 100 programmers to simulate different experimental settings. In this experiment, we appointed each programmer in our subset dataset of 100 programmers to be the subject to the imitation attack while using all other programmers to imitate the coding style of the subject programmer. Since our dataset includes nine code samples per programmer, we generated ($9 \times 99 = 891$) imitation code samples for each subject programmer, *i.e.,* generating 89,100 adversarial code samples for each experimental setting considering the hundred included programmers. Figure 5.5 shows the achieved results by this attack against different code authorship identification systems using different settings. The results include the imitation success rate and identification confidence.

Statements' Impact. Figure 5.5(a) shows the effects of using different numbers of statements for the adversarial code perturbation when implementing the imitation attack. The figure shows clearly the increase in the imitation success rate as the number of statements increases. Using one statement, the average imitation success rate, using a different number of code lines, exceeds 25% for all targeted identification systems. Figure 5.5(a) shows that the average imitation success rates are 29.53%, 25.12%, 27.75%, 28.10%, 31.97%, and 32.35%, against DL-CAIS, CSFS, WE-C-CNN,



(a) Imitation success / statements

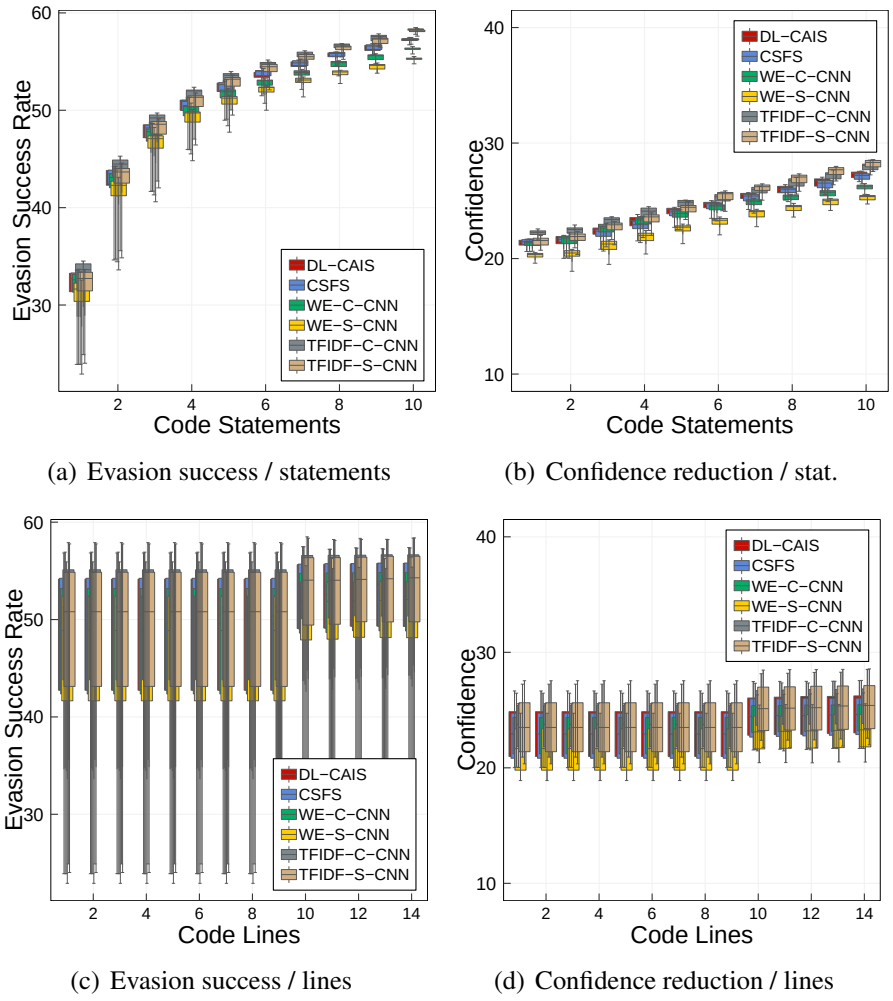
(b) Confidence reduction / stat.

(c) Imitation success / lines

(d) Confidence reduction / lines

Figure 5.5: The results of the imitation attack: the imitation success rate and the confidence reduction of the targeted approaches categorized by the added perturbations of code lines (ranging from 1 to 10 per statement) and code statements (ranging from 1 to 10). Notice, the impact of added code statements is larger than the considered code lines within the statements.

WE-S-CNN, TFIDF-C-CNN, and TFIDF-S-CNN, respectively. While the average success rates when using ten statements are 52.34%, 48.24%, 50.52%, 51.07%, 54.27%, and 55.28%, against DL-CAIS, CSFS, WE-C-CNN, WE-S-CNN, TFIDF-C-CNN, and TFIDF-S-CNN, respectively. This result shows that the vulnerability of the targeted identification systems against the authorship imitation attack, since the average imitation success rate exceeds 48% when the perturbations in-



(a) Evasion success / statements

(b) Confidence reduction / stat.

(c) Evasion success / lines

(d) Confidence reduction / lines

Figure 5.6: The results of the evasion: the evasion success rate and the confidence reduction of the targeted approaches categorized by the added perturbations of code lines (ranging from 1 to 10 per statement) and code statements (ranging from 1 to 10). Notice, the impact of added code statements is larger than the considered code lines within the statements.

clude ten statements of code. Introducing the smallest perturbation of only one statement with one code of line has enabled an imitation success rate of 21.89%, 17.82%, 20.14%, 20.59%, 22.48%, and 26.35% for the targeted approaches DL-CAIS, CSFS, WE-C-CNN, WE-S-CNN, TFIDF-C-CNN, and TFIDF-S-CNN, respectively. Even though the success rates vary when increasing the code perturbation size, the identification models' confidence levels are stable and significantly

lower than the baseline confidence levels shown in Figure 5.3(b). Figure 5.5(b) shows that the identification confidence is lower than 30% for all approaches and in all experimental settings.

Lines of Code's Impact. Figure 5.5(c) shows the effects of using different numbers of code lines per statement for the injected code perturbations for imitation attack. We use a range from one line of code per statement to 14 lines throughout our experiments. Using only one line of code per a statement, the average achieved imitation success rates, across different numbers of statements—ranging from one to ten statements, are 43.28%, 39.50%, 41.37%, 41.87%, 45.48%, and 46.56%, against DL-CAIS, CSFS, WE-C-CNN, WE-S-CNN, TFIDF-C-CNN, and TFIDF-S-CNN, respectively. Increasing the included lines of code to 14 lines does not significantly increase the imitation success rate, since using 14 lines of code results in imitation success rates of 47.32%, 43.23%, 46.10%, 45.62%, 49.65%, and 50.03%, against DL-CAIS, CSFS, WE-C-CNN, WE-S-CNN, TFIDF-C-CNN, and TFIDF-S-CNN, respectively. Similarly, the confidence levels of the identification models are not significantly influenced by the number of added lines of code for the code perturbations since Figure 5.5(d) shows that all confidence scores are tapered at a level below 30% with slight improvement after including more than ten lines.

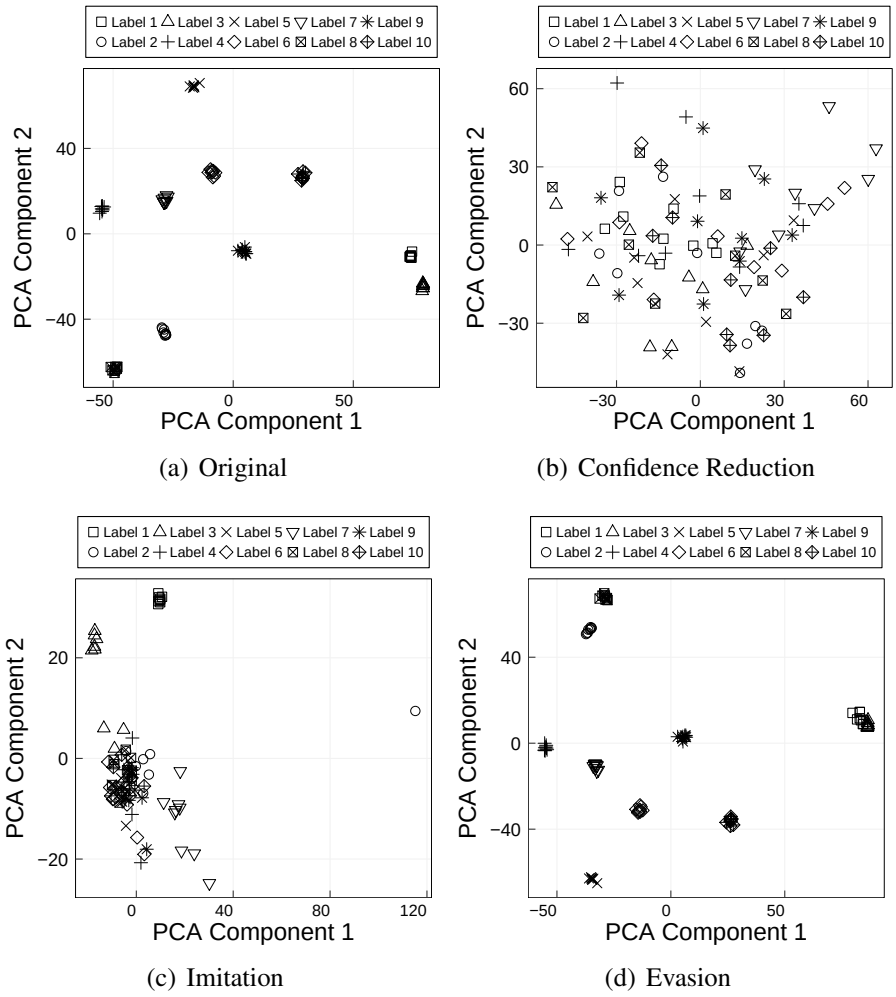
5.3.3 Evasion Attack

Similar to the imitation attack, the evaluation of this attack is conducted on a subset of 100 programmers. We appointed each programmer in our subset dataset of 100 programmers to be a subject to the evasion attack while using all other programmers as possible disguise subjects. For each programmer, we generate nine adversarial code samples based on the most influential features of the targeted subject, *i.e.,* generating ($9 \times 100 = 900$) samples for the selected 100 programmers in our dataset. Figure 5.6 shows the achieved results for this attack using various settings.

Statements' Impact. Figure 5.6(a) shows an improvement in the evasion success rates as the number of statements increases. Using only one statement, the average evasion success rate exceeded 30% for all targeted authorship identification approaches. Figure 5.6(a) shows that the average success rates of this attack using only one statement of code were 31.67%, 31.73%, 31.61%, 30.68%, 32.66%, and 31.77%, against DL-CAIS, CSFS, WE-C-CNN, WE-S-CNN, TFIDF-C-CNN, and TFIDF-S-CNN, respectively. While using ten statements, the evasion success rates increase to 57.19%, 57.22%, 56.25%, 55.23%, 58.24%, and 58.09%, for the targeted approaches, DL-CAIS, CSFS, WE-C-CNN, WE-S-CNN, TFIDF-C-CNN, and TFIDF-S-CNN. Even with the smallest size of code perturbation—with one statement that includes only one line of code, the evasion success rates are 23.90%, 23.91%, 23.90%, 22.91%, 24.88%, and 24.01%, for the targeted approaches, DL-CAIS, CSFS, WE-C-CNN, WE-S-CNN, TFIDF-C-CNN, and TFIDF-S-CNN, respectively.

This attack has also shown a great impact on the identification confidence of all targeted identification systems since all achieved levels of confidence are lower than 30%. Figure 5.6(b) shows that the number of included code statements contributes to the identification confidence. Using one code statement, the achieved average levels of confidence are 21.33%, 21.36%, 21.29%, 20.24%, 22.21%, and 21.45%, for DL-CAIS, CSFS, WE-C-CNN, WE-S-CNN, TFIDF-C-CNN, and TFIDF-S-CNN, respectively. While using ten statements, the achieved levels of confidence are 27.26%, 27.12%, 26.18%, 25.27%, 27.96%, and 28.21%, for DL-CAIS, CSFS, WE-C-CNN, WE-S-CNN, TFIDF-C-CNN, and TFIDF-S-CNN, respectively.

Lines of Code's Impact. Figure 5.6(c) shows the evasion rate is not significantly affected by the number of lines since the results are within the same range when increasing the number of lines per statement. The average evasion success rates when using one line of code per statement are 46.67%, 46.74%, 46.07%, 45.25%, 47.66%, and 47.21%, for the targeted approaches, DL-CAIS, CSFS, WE-C-CNN, WE-S-CNN, TFIDF-C-CNN, and TFIDF-S-CNN, respectively. Increasing



(a) Original (b) Confidence Reduction

(c) Imitation (d) Evasion

Figure 5.7: The PCA visualization of code authorship attributions of 10 programmers with nine code sample. The visualization shows the original authorship attributions along with the effects of different adversarial attacks on the generated attributions using DL-CAIS.

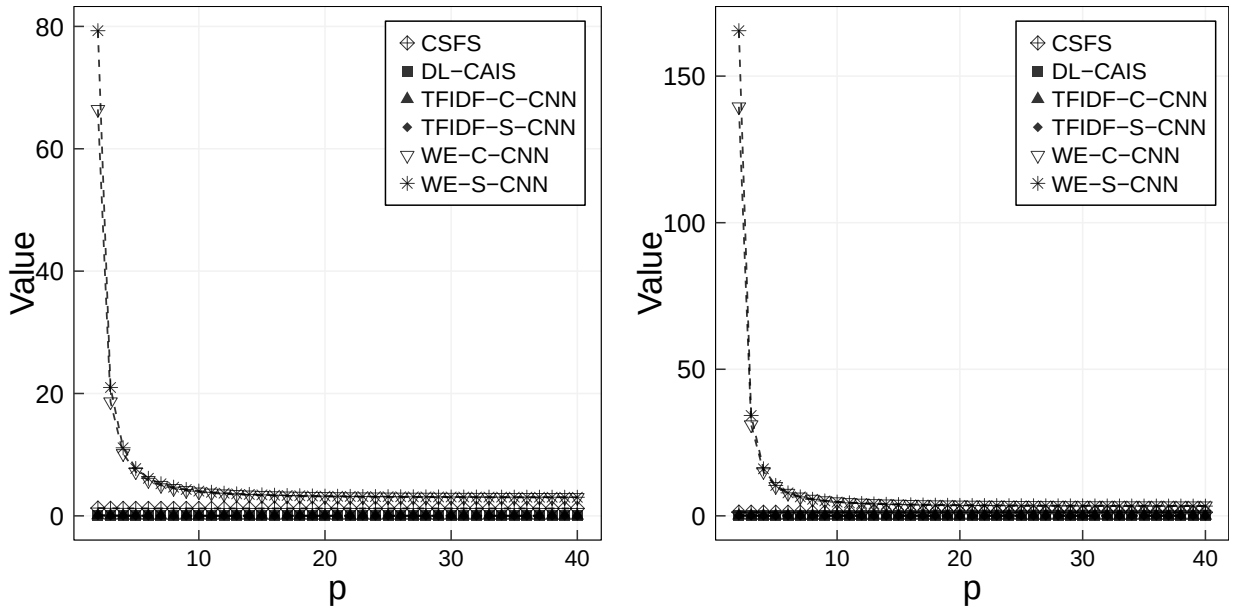
the number of code lines in code statements results in slight improvement of the evasion success rates since using 14 lines results in success rates of 50.98%, 51.05%, 50.30%, 49.48%, 51.98%, and 51.60%, against DL-CAIS, CSFS, WE-C-CNN, WE-S-CNN, TFIDF-C-CNN, and TFIDF-S-CNN, respectively, as confidence levels are below 30% Figure 5.6(d).

5.4 Discussion

In the following, we explore the effects of adversarial perturbations on the authorship attributes with different adversarial attacks. Second, we show the magnitude of perturbations when adding code blocks to the original code. Finally, we list the limitations of this study.

5.4.1 Adversarial Authorship Attributions

We show that code authorship identification systems can be vulnerable to adversarial attacks. For all attacks, the authorship attributes are greatly affected by the smallest size of perturbations. This effect can be shown in the PCA visualization of the authorship attributes of code samples generated by different adversarial settings, as shown visually in Figure 5.7 for code samples of ten programmers with nine code samples each in various adversarial settings. For this visualization, we used the DL-CAIS system to demonstrate the effect of different attacks. Figure 5.7(a) shows the original code representations of the ten programmers with nine code samples. It is clear that such authorship attributes are the reason for the accurate identification process since code samples of a certain programmer are located within very close proximity. However, when introducing adversarial examples as in Figure 5.7(b) during the confidence reduction attack, the authorship features become extremely scattered in the feature space so it is difficult to establish decision boundaries, and hence the high misidentification rate and low confidence for this attack. Figure 5.7(c) shows the attempt of all ten programmers to imitate one label (*e.g.,Label 10* in this figure), which leads to partial success as some programmers are still resilient with the same features as other programmers. The figure shows also that most code samples are within the same proximity as the code samples of the targeted programmer. However, some programmers have very distinct coding style such as *Label 1*, who has code samples that are not affected by the targeted perturbations. Other programmers are partially affected, such as (*Label 3* and label 7), who have some code samples that are close



(a) Small Code Perturbation

(b) Large Code Perturbation

Figure 5.8: The *p˙norms* of perturbations with different sizes with respect to the initial representations of the code using different approaches. Small perturbations are generated by one code statement, while large perturbations are generated by ten.

to the targeted programmer's samples, while some others are afar. In Figure 5.7(d), we show the effect of the evasion attack, in which the adversary aims to imitate the closest programming style to adversary's. In this figure, *Label 1* imitates the coding style of *Label 3* as their coding style is the closest among others as also can be seen in Figure 5.7(a). We explained this attack as if *Label 1* is using *Label 3* as a disguise to evade identification.

5.4.2 Perturbation Size

The perturbations size can be measured by ℓ_p to show their magnitude using the p–$norm$ distance. Figure 5.8 shows the values of p–$norm$, ranging from $p = 2$ to $p = 40$, for small code perturbations generated from one statement of code and for large perturbations from ten code statements. The

results shown in the figure are drawn from the average perturbation size using random perturbations generation (*i.e.,* as used in the confidence reduction attack) on a dataset of 100 programmers with nine code samples each. Generally, the effect of perturbations on the input representations varies based on the underlying method used for representing the code. The effects are highest when using word embeddings as initial representations, *e.g.,* WE-C-CNN and WE-S-CNN, due to the high-dimensional and compact representations of code sample using word embeddings, while the lowest effects are exhibited by approaches using TF-IDF as initial representations, *e.g.,* DL-CAIS, TFIDF-C-CNN, and TFIDF-S-CNN. Figure 5.8(a) shows the ℓ_p of perturbations with one statement that includes only one line of code. The ℓ_2 values are: 0.07, 1.27, 0.08, 0.08, 66.43, 79.30 for DL-CAIS, CSFS, WE-C-CNN, WE-S-CNN, TFIDF-C-CNN, and TFIDF-S-CNN, respectively. Increasing the value of p decreases slightly the value of the ℓ_p to reach $\ell_{40} \approx 3$ for WE-C-CNN, and WE-S-CNN. A similar observation is made using a large perturbation regarding the effect on the code representations. The ℓ_p values are shown to be slightly higher, which is due to the introduced size of code statements. However, these changes are not significant in magnitude, although they influence the outcome of the model as seen in Figure 5.4.

5.4.3 Limitations and Future Work

Programming Languages. In this work, we only used a C++ dataset and followed a restricted code injection scheme. We do not foresee any fundamental reason why the work at hand should not generalize, although exploring whether the same attacks are possible on different programming languages is yet a future direction. Moreover, it would be interesting to investigate the levels of difficulty to establish adversarial attacks based on the used programming languages.

Confidence Levels. All the results by the attacks showed very low levels of confidence, which can be an indication for out-of-world example typically to be excluded from the testing process.

In other words, this low level of confidence can be a sign for adversarial input and a possible way to detect the adversarial sample. An interesting future direction would be to investigate such an observation as well as other defense mechanisms against adversarial code samples.

Perturbation Detectability. It is important for the perturbations to be undetectable before entering the identification pipeline. This work uses a simple scheme to inject code perturbations, which can be detectable by experts, through manual inspection, as a separate part in the original code. Investigating methods to hide such perturbations, by compilation or obfuscation, is also to be explored as a future direction.

5.5 Conclusion

This work investigates the robustness of several code authorship identification systems under adversarial attacks utilizing code-level perturbation. We targeted six authorship identification systems with different underlying techniques and defined three attacks objectives—confidence reduction, imitation, and evasion. The adversarial attacks exploited code perturbations to hinder authorship recognition while preserving the functionality of the code. The process of generating adversarial code samples included producing code perturbations (targeted or non-targeted) to fulfill the adversaries' objectives. Our results showed the impact of code perturbations on authorship attributions, and how increasing the size of perturbation increases its effect, although the targeted techniques could be fooled with the smallest size of perturbations (one line). Among the approaches examined, the CSFS approach showed more robustness than the deep learning-based approaches against the three conducted adversarial attacks. However, all authorship attributions were compromised under adversarial scenarios such that the confidence levels of the identification models were very low in comparison to the baseline performance.

CHAPTER 6: CONCLUSION

At the beginning of this research, we studied authorship attribution of single-authored software using a deep learning-based approach (DL-CAIS). Through comprehensive experiments, we investigated the learning process of large-scale code authorship attribution using various RNN-based architectures. We show that DL-CAIS can produce deep representations of authorship features that a more efficient and resilient to language-specifics, number of code files available per author, and code obfuscation. For authorship attribution on the source code domain, we evaluated DL-CAIS on a large-scale dataset covering the entire GCJ dataset across all years (2008 to 2016) in four programming languages (C, C++, Java, and Python). Our experiments showed that our approach achieves remarkable results in terms of accuracy in various settings, more specifically, it achieved an accuracy of 92.3% for identifying 8,903 programmers with only seven files each.

We show that our approach is oblivious to language specifics, and therefore it can extract high-quality features that enable code authorship identification even when the model is trained by mixed languages. Moreover, we studied the effects of temporal changes and the availability of code samples on the identification accuracy.

For authorship attribution on the binary code domain, we investigated the performance of our approach on capturing authorship attributes from high-level translations of binaries produced by a simple straightforward reverse engineering process. The results showed that DL-CAIS achieves an accuracy of 95.74% for identifying 1,500 programmers of software binaries. Similar results were obtained when applying different compilation settings for producing the binaries. Further, we examined DL-CAIS on the attributing programmers on the obfuscation domain, using source code obfuscation tools (such as Stunnix and Tigress) and binary obfuscation tools (such as Obfuscator-LLVM). Our approach shows remarkable resilience to obfuscation by achieving high identification

accuracy on different scales and obfuscation settings.

The most interesting experiments in this research are shown in real-world scenarios for evaluating DL-CAIS. Those experiments evaluated the robustness of our approach in the wild using a dataset of code samples from the code-sharing platform (GitHub). We also considered the open-world assumption to identify new programmers who might not be seen by the model before. The results of those experiments give insights into how authorship identification task can be performed in the real-world. One of the most challenges that code authorship analysis confronts is 1) the reuse of code, where programmers reuse others' codes, 2) the collaborative efforts in writing a program as a team, and 3) when a specific format is enforced by the work environment or by code formatters in the development environment. Such challenges motivate for future work in this field.

The research extends the research on the field of software authorship attribution to multi-authored software. We propose Multi-χ, a fine-grained method for identifying multiple authors contributing to a single source file. This approach is more realistic to be applied for identifying programmers in collaborative projects.

To evaluate Multi-χ, we obtained a large dataset of multi-author source-code files collected from Github. The collected 26,607 code files belong to nine open-source projects and 2,220 involved programmers. Besides the scalability challenge (attributing a large number of programmers in multi-authored code samples), there are many challenges that we considered and explored such as selecting the proper segmentation window of code samples, the data imbalance, and the code representation and modeling techniques. For example, we observed that code segments of length greater than 12 lines of code are more likely to be written by multiple programmers. While this motivates for fine-grained approach, as Multi-χ, it makes it more challenging for the approach to detect authorship attributes form small segments. After experimenting with different design choices, Multi-χ showed remarkable results across multiple dimensions and experimental settings.

For example, Multi-χ achieves a per-segment authorship identification accuracy of 93.18% for identifying 562 programmers.

The dissertation also provides insights into the performance of authorship identification methods under adversarial settings. We introduced Author-SHIELD to investigate the robustness of various code authorship identification systems under different adversarial attacks (launched by adversarial examples generated by added code-level perturbation). For our analysis, we targeted six authorship identification systems with different underlying techniques and defined three attack objectives—confidence reduction, imitation, and evasion.

The code-level perturbations are added to meet different adversarial objectives while preserving the functionality of the code. Our experiments demonstrated the vulnerability of current authorship attribution methods against adversarial attacks. For example, the smallest perturbation (one line of code) can result in a misidentification rate that exceeds 98% for all targeted systems. This motivates for further investigation on robust solutions for the authorship identification task against adversarial attacks. Another direction for research is the detection of such adversarial examples that manipulate the system to generating false output.

APPENDIX A: COPYRIGHT INFORMATION

ACM Copyright and Audio/Video Release

Title of the Work: Large-Scale and Language-Oblivious Code Authorship Identification
Submission ID:ccsf036
Author/Presenter(s): Mohammed Abuhamad (Inha Univ.); Tamer Abuhmed (Inha Univ.); Aziz Mohaisen (Univ. of Central Florida); DaeHun Nyang (Inha Univ.);
Type of material: Full Paper

Publication and/or Conference Name: CCS '18: 2018 ACM SIGSAC Conference on Computer and Communications Security Proceedings

I. Copyright Transfer, Reserved Rights and Permitted Uses

* Your Copyright Transfer is conditional upon you agreeing to the terms set out below.

Copyright to the Work and to any supplemental files integral to the Work which are submitted with it for review and publication such as an extended proof, a PowerPoint outline, or appendices that may exceed a printed page limit, (including without limitation, the right to publish the Work in whole or in part in any and all forms of media, now or hereafter known) is hereby transferred to the ACM (for Government work, to the extent transferable) effective as of the date of this agreement, on the understanding that the Work has been accepted for publication by ACM.

Reserved Rights and Permitted Uses

(a) All rights and permissions the author has not granted to ACM are reserved to the Owner, including all other proprietary rights such as patent or trademark rights.

(b) Furthermore, notwithstanding the exclusive rights the Owner has granted to ACM, Owner shall have the right to do the following:

(i) Reuse any portion of the Work, without fee, in any future works written or edited by the Author, including books, lectures and presentations in any and all media.

(ii) Create a "Major Revision" which is wholly owned by the author

(iii) Post the Accepted Version of the Work on (1) the Author's home page, (2) the Owner's institutional repository, (3) any repository legally mandated by an agency funding the research on which the Work is based, and (4) any non-commercial repository or aggregation that does not duplicate ACM tables of contents, i.e., whose patterns of links do not substantially duplicate an ACM-copyrighted volume or issue. Non-commercial repositories are here understood as repositories owned by non-profit organizations that do not charge a fee for accessing deposited articles and that do not sell advertising or otherwise profit from serving articles.

(iv) Post an "Author-Izer" link enabling free downloads of the Version of Record in the ACM Digital Library on (1) the Author's home page or (2) the Owner's institutional repository;

(v) Prior to commencement of the ACM peer review process, post the version of the Work as submitted to ACM ("Submitted Version" or any earlier versions) to non-peer reviewed servers;

(vi) Make free distributions of the final published Version of Record internally to the Owner's employees, if applicable;

(vii) Make free distributions of the published Version of Record for Classroom and Personal Use;

(viii) Bundle the Work in any of Owner's software distributions; and

(ix) Use any Auxiliary Material independent from the Work.

When preparing your paper for submission using the ACM TeX templates, the rights and permissions information and the bibliographic strip must appear on the lower left hand portion of the first page.

The new ACM Consolidated TeX template Version 1.3 and above automatically creates and positions these text blocks for you based on the code snippet which is system-generated based on your rights management choice and this particular conference.

Please copy and paste the following code snippet into your TeX file between \begin{document} and \maketitle, either after or before CCS codes.

```
\copyrightyear{2018}
\acmYear{2018}
\setcopyright{acmcopyright}
\acmConference[CCS '18]{2018 ACM SIGSAC Conference on Computer and Communications Security}{October 15--19, 2018}{Toronto, ON, Canada}
\acmBooktitle{2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), October 15--19, 2018, Toronto, ON, Canada}
\acmPrice{15.00}
\acmDOI{10.1145/3243734.3243738}
\acmISBN{978-1-4503-5693-0/18/10}
```

ACM TeX template .cls version 2.8, automatically creates and positions these text blocks for you based on the code snippet which is system-generated based on your rights management choice and this particular conference.
Please copy and paste the following code snippet into your TeX file between \begin{document} and \maketitle, either after or before CCS codes.

```
\CopyrightYear{2018}
\setcopyright{acmcopyright}
\conferenceinfo{CCS '18,}{October 15--19, 2018, Toronto, ON, Canada}
\isbn{978-1-4503-5693-0/18/10}\acmPrice{$15.00}
\doi{https://doi.org/10.1145/3243734.3243738}
```

If you are using the ACM Microsoft Word template, or still using an older version of the ACM TeX template, or the current versions of the ACM SIGCHI, SIGGRAPH, or SIGPLAN TeX templates, you must copy and paste the following text block into your document as per the instructions provided with the templates you are using:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to

redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

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

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

NOTE: Make sure to include your article's DOI as part of the bibstrip data; DOIs will be registered and become active shortly after publication in the ACM Digital Library

☑ A. Assent to Assignment. I hereby represent and warrant that I am the sole owner (or authorized agent of the copyright owner(s)), with the exception of third party materials detailed in section III below. I have obtained permission for any third-party material included in the Work.

☐ B. Declaration for Government Work. I am an employee of the National Government of my country and my Government claims rights to this work, or it is not copyrightable (Government work is classified as Public Domain in U.S. only)

Are any of the co-authors, employees or contractors of a National Government? ○ Yes ◉ No

---

II. Permission For Conference Recording and Distribution

* Your Audio/Video Release is conditional upon you agreeing to the terms set out below.

I hereby grant permission for ACM to include my name, likeness, presentation and comments in any and all forms, for the Conference and/or Publication.

I further grant permission for ACM to record and/or transcribe and reproduce my presentation as part of the ACM Digital Library, and to distribute the same for sale in complete or partial form as part of an ACM product on CD-ROM, DVD, webcast, USB device, streaming video or any other media format now or hereafter known.

I understand that my presentation will not be sold separately as a stand-alone product without my direct consent. Accordingly, I give ACM the right to use my image, voice, pronouncements, likeness, and my name, and any biographical material submitted by me, in connection with the Conference and/or Publication, whether used in excerpts or in full, for distribution described above and for any associated advertising or exhibition.

Do you agree to the above Audio/Video Release? ◉ Yes ○ No

III. Auxiliary Material

Do you have any Auxiliary Materials? ○ Yes ◉ No

IV. Third Party Materials

In the event that any materials used in my presentation or Auxiliary Materials contain the work of third-party individuals or organizations (including copyrighted music or movie excerpts or anything not owned by me), I understand that it is my responsibility to secure any necessary permissions and/or licenses for print and/or digital publication, and cite or attach them below.

◉ We/I have not used third-party material.
○ We/I have used third-party materials and have necessary permissions.

V. Artistic Images
If your paper includes images that were created for any purpose other than this paper and to which you or your employer claim copyright, you must complete Part V and be sure to include a notice of copyright with each such image in the paper.
◉ We/I do not have any artistic images.
○ We/I have any artistic images.

VI. Representations, Warranties and Covenants

The undersigned hereby represents, warrants and covenants as follows:

(a) Owner is the sole owner or authorized agent of Owner(s) of the Work;

(b) The undersigned is authorized to enter into this Agreement and grant the rights included in this license to ACM;

(c) The Work is original and does not infringe the rights of any third party; all permissions for use of third-party materials consistent in scope and duration with the rights granted to ACM have been obtained, copies of such permissions have been provided to ACM, and the Work as submitted to ACM clearly and accurately indicates the credit to the proprietors of any such third-party materials (including any applicable copyright notice), or will be revised to indicate such credit;

(d) The Work has not been published except for informal postings on non-peer reviewed servers, and Owner covenants to use best efforts to place ACM DOI pointers on any such prior postings;

(e) The Auxiliary Materials, if any, contain no malicious code, virus, trojan horse or other software routines or hardware components designed to permit unauthorized access or to disable, erase or otherwise harm any computer systems or software; and

(f) The Artistic Images, if any, are clearly and accurately noted as such (including any applicable copyright notice) in the Submitted Version.

☑ I agree to the Representations, Warranties and Covenants

DATE: **08/02/2018** sent to abuhamad@seclab.inha.ac.kr at **21:08:34**

LICENSE TO PUBLISH

Please read the terms of this agreement, print, initial page 1, sign page 2, scan and send the document as one file attached to an e-mail to publication@petsymposium.org.

Article entitled ("Work" or "article"):

.....

Author/s: (also referred to as "Licensor/s")

.....

Corresponding author: (if more than one author)

.....

Journal Name

Proceedings on Privacy Enhancing Technologies (PoPETs)
.....

Journal Owner (also referred to as "You" in the Creative Commons license mentioned in section 1 below)

The Tor Project, Inc.
.....

1. License

The non-commercial use of the article will be governed by the Creative Commons Attribution-NonCommercial-NoDerivs license as currently displayed on http://creativecommons.org/licenses/by-nc-nd/3.0/, except that sections 2 through 8 below will apply in this respect and prevail over all conflicting provisions of such license model. Without prejudice to the foregoing, the author hereby grants the Journal Owner the exclusive license for commercial use of the article (for U.S. government employees: to the extent transferable) according to section 2 below, and sections 4 through 9 below, throughout the world, in any form, in any language, for the full term of copyright, effective upon acceptance for publication.

2. Author's Warranties

The author warrants that the article is original, written by stated author/s, has not been published before, contains no unlawful statements, does not infringe the rights of others, is subject to copyright that is vested exclusively in the author and free of any third party rights, and that any necessary written permissions to quote from other sources have been obtained by the author/s.

3. User Rights

Under the Creative Commons Attribution-NonCommercial-NoDerivs license, the author(s) and users are free to share (copy, distribute and transmit the contribution) under the following conditions: 1. they must attribute the contribution in the manner specified by the author or licensor, 2. they may not use this contribution for commercial purposes, 3. they may not alter, transform, or build upon this work.

4. Rights of Authors

Authors retain the following rights:
- copyright, and other proprietary rights relating to the article, such as patent rights,
- the right to use the substance of the article in future own works, including lectures and books,
- the right to reproduce the article for own purposes, provided the copies are not offered for sale,
- the right to self-archive the article.

5. Co-Authorship

If the article was prepared jointly with other authors, the signatory of this form warrants that he/she has been authorized by all co-authors to sign this agreement on their behalf, and agrees to inform his/her co-authors of the terms of this agreement.

6. Termination

This agreement can be terminated by the author or the Journal Owner upon two months' notice where the other party has materially breached this agreement and failed to remedy such breach within a month of being given the terminating party's notice requesting such breach to be remedied. No breach or violation of this agreement will cause this agreement or any license granted in it to terminate automatically or affect the definition of the Journal Owner. After the lapse of forty (40) years of the date of this agreement, this agreement can be terminated without cause by the author or the Journal Owner upon two years' notice. The author and the Journal Owner may agree to terminate this agreement at any time. This agreement or any license granted in it cannot be terminated otherwise than in accordance with this section 6.

7. Royalties

This agreement entitles the author to no royalties or other fees. To such extent as legally permissible, the author waives his or her right to collect royalties relative to the article in respect of any use of the article by the Journal Owner or its sublicensee.

8. Miscellaneous

The Journal Owner will publish the article (or have it published) in the Journal, if the article's editorial process is successfully completed and the Journal Owner or its sublicensee has become obligated to have the article published. Where such obligation depends on the payment of a fee, it shall not be deemed to exist until such time as that fee is paid. The Journal Owner may conform the article to a style of punctuation, spelling, capitalization and usage that it deems appropriate. The author acknowledges that the article may be published so that it will be publicly accessible and such access will be free of charge for the readers. The Journal Owner will be allowed to sublicense the rights that are licensed to it under this agreement. This agreement will be governed by the laws of the State of Massachusetts, in the USA.

9. Scope of the Commercial License

The exclusive right and license granted under this agreement to the Journal Owner for commercial use is as follows:

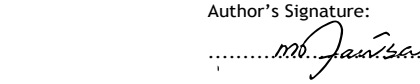
a. to prepare, reproduce, manufacture, publish, distribute, exhibit, advertise, promote, license and sub-license printed and electronic copies of the article, through the Internet and other means of data transmission now known or later to be developed; the foregoing will include abstracts, bibliographic information, illustrations, pictures, indexes and subject headings and other proprietary materials contained in the article,
b. to exercise, license, and sub-license others to exercise subsidiary and other rights in the article, including the right to photocopy, scan or reproduce copies thereof, to reproduce excerpts from the article in other works, and to reproduce copies of the article as part of compilations with other works, including collections of materials made for use in classes for instructional purposes, customized works, electronic databases, document delivery, and other information services, and publish, distribute, exhibit and license the same.

Where this agreement refers to a license granted to the Journal Owner in this agreement as exclusive, the author commits not only to refrain from granting such license to a third party but also to refrain from exercising the right that is the subject of such license otherwise than by performing this agreement.

The Journal Owner will be entitled to enforce in respect of third parties, to such extent as permitted by law, the rights licensed to it under this agreement.

If the article was written in the course of employment by the US or UK Government, and/or arises from NIH funding, please consult the Journal Owner for further instructions.

Author's Signature:



Name printed:

.....

Date:

.....

APPENDIX B: IRB APPROVAL/EXEMPTION LETTER

UNIVERSITY OF CENTRAL FLORIDA

Institutional Review Board
FWA00000351
IRB00001138, IRB00012110
Office of Research
12201 Research Parkway
Orlando, FL 32826-3246

Memorandum

To: Mohammed Abuhamad

From: UCF Institutional Review Board (IRB)

CC: Barbara Fritzsche

Date: May 11, 2020

Re: Request for IRB Determination

The IRB reviewed the information related to your dissertation *Towards Large-Scale and Robust Code Authorship Identification with Deep Feature Learning.*

As you know, the IRB cannot provide an official determination letter for your research because it was not submitted into our electronic submission system.

However, if you had completed a Huron submission, the IRB could make one of the following research determinations: “Not Human Subjects Research,” “Exempt,” “Expedited” or “Full Board.

Based on the data points you provided, this study would have been issued a Not Human Subjects Research determination outcome letter had a request for a formal determination been submitted to the UCF IRB through Huron IRB system.

If you have any questions, please contact the UCF IRB irb@ucf.edu.

Sincerely,

Renea Carver
IRB Manager

LIST OF REFERENCES

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. J. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Józefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. G. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. A. Tucker, V. Vanhoucke, V. Vasudevan, F. B. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467, 2016.

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

[3] M. Abuhamad, T. AbuHmed, A. Mohaisen, and D. 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.

[4] M. Abuhamad, T. Abuhmed, D. Nyang, and D. Mohaisen. Multi-χ: Identifying multiple authors from source code files. *Proceedings of the 20th Privacy Enhancing Technologies*, 1:17, 2020.

[5] M. Abuhamad, J.-s. Rhim, T. AbuHmed, S. Ullah, S. Kang, and D. Nyang. Code authorship identification using convolutional neural networks. *Future Generation Computer Systems*, 95:104–115, 2019.

[6] A. Abusnaina, M. Abuhamad, H. Alasmary, A. Anwar, R. Jang, S. Salem, D. Nyang, and D. Mohaisen. Deep learning-based fine-grained hierarchical learning approach for robust malware classification. *arXiv preprint arXiv:2005.07145*, 2020.

[7] A. Abusnaina, H. Alasmary, M. Abuhamad, S. Salem, D. Nyang, and A. Mohaisen. Subgraph-based adversarial examples against graph-based iot malware detection systems. In *International Conference on Computational Data and Social Networks*, pages 268–281. Springer, 2019.

[8] S. Afroz, A. C. Islam, A. Stolerman, R. Greenstadt, and D. McCoy. Doppelgänger finder: Taking stylometry to the underground. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 212–226. IEEE, 2014.

[9] H. Alasmary, A. Abusnaina, R. Jang, M. Abuhamad, A. Anwar, D. NYANG, and D. Mohaisen. Soteria: Detecting adversarial examples in control flow graph-based malware classifiers. In *40th IEEE International Conference on Distributed Computing Systems, ICDCS*, pages 1296–1305, 2020.

[10] A. Alhuzali, R. Gjomemo, B. Eshete, and V. 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.

[11] S. Alrabaee, N. Saleem, S. Preda, L. Wang, and M. Debbabi. Oba2: An onion approach to binary code authorship attribution. *Digital Investigation*, 11:S94–S103, 2014.

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

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

[14] A. T. Basilevsky. *Statistical factor analysis and related methods: theory and applications*, volume 418. John Wiley & Sons, 2009.

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

[16] Y. Bengio. Learning deep architectures for ai. *Found. Trends Mach. Learn.*, 2(1):1–127, Jan. 2009.

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

[18] Y. Bengio, A. C. Courville, and P. Vincent. Unsupervised feature learning and deep learning: A review and new perspectives. *CoRR*, 2012.

[19] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin. A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155, 2003.

[20] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146, 2017.

[21] A. Bordes, X. Glorot, J. Weston, and Y. Bengio. Joint learning of words and meaning representations for open-text semantic parsing. In *AISTATS*, volume 22, pages 127–135, 2012.

[22] N. Boulanger-Lewandowski, Y. Bengio, and P. Vincent. Modeling temporal dependencies in high-dimensional sequences: Application to polyphonic music generation and transcription. *arXiv preprint arXiv:1206.6392*, 2012.

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

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

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

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

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

[28] S. Burrows, A. L. Uitdenbogerd, and A. Turpin. Application of information retrieval techniques for source code authorship attribution. In *Proceedings of the 14th International Conference on Database Systems for Advanced Applications*, DASFAA '09, pages 699–713, Berlin, Heidelberg, 2009. Springer-Verlag.

[29] 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, 2009.

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

[31] A. Caliskan-Islam, R. Harang, A. Liu, A. Narayanan, C. Voss, F. Yamaguchi, and R. 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.

[32] A. Caliskan-Islam, F. Yamaguchi, E. Dauber, R. Harang, K. Rieck, R. Greenstadt, and A. Narayanan. When coding style survives compilation: De-anonymizing programmers

from executable binaries. *Network and Distributed System Security Symposium 2018 (NDSS)*, 2018.

[33] N. Carlini and D. A. Wagner. Towards evaluating the robustness of neural networks. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 39–57, 2017.

[34] K. Cho, B. van Merrienboer, D. Bahdanau, and Y. Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *CoRR*, abs/1409.1259, 2014.

[35] D. Ciregan, U. Meier, and J. Schmidhuber. Multi-column deep neural networks for image classification. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 3642–3649. IEEE, 2012.

[36] R. Collobert and J. 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.

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

[38] G. E. Dahl, D. Yu, L. Deng, and A. Acero. Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *IEEE Transactions on Audio, Speech, and Language Processing*, 20(1):30–42, 2012.

[39] E. Dauber, A. Caliskan, R. Harang, and R. Greenstadt. Git blame who?: stylistic authorship attribution of small, incomplete source code fragments. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 356–357. ACM, 2018.

[40] E. Dauber, A. Caliskan, R. Harang, G. Shearer, M. Weisman, F. Nelson, and R. Greenstadt. Git blame who?: Stylistic authorship attribution of small, incomplete source code fragments. *Proceedings on Privacy Enhancing Technologies*, 2019(3):389 – 408, 2019.

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

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

[43] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, 12:2121–2159, July 2011.

[44] B. S. Elenbogen and N. Seliya. Detecting outsourced student programming assignments. *J. Comput. Sci. Coll.*, 23(3):50–57, Jan. 2008.

[45] B. S. Everitt and G. Dunn. *Applied multivariate data analysis*, volume 2. Wiley Online Library, 2001.

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

[47] D. Fifield, T. Follan, and E. Lunde. Unsupervised authorship attribution. *arXiv preprint arXiv:1503.07613*, 2015.

[48] G. Frantzeskou, E. Stamatatos, S. Gritzalis, C. E. Chaski, and B. S. 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.

[49] G. Frantzeskou, E. Stamatatos, S. Gritzalis, and S. 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, New York, NY, USA, 2006. ACM.

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

[51] C. 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.

[52] X. Glorot, A. Bordes, and Y. 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.

[53] I. J. Goodfellow, A. Courville, and Y. Bengio. Spike-and-slab sparse coding for unsupervised feature discovery. *arXiv preprint arXiv:1201.3382*, 2012.

[54] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel. Adversarial examples for malware detection. In *European Symposium on Research in Computer Security*, pages 62–79. Springer, 2017.

[55] N. D. Hansen, C. Lioma, B. Larsen, and S. Alstrup. Temporal context for authorship attribution. In *Information Retrieval Facility Conference*, pages 22–40. Springer, 2014.

[56] Hex-Rays. Hex-rays. https://www.hex-rays.com/products/decompiler/, 2017. (Accessed on 15/02/2017).

[57] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

[58] G. E. Hinton, S. Osindero, and Y.-W. Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

[59] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, Nov. 1997.

[60] W. Hu and Y. Tan. Generating adversarial malware examples for black-box attacks based on gan. *arXiv preprint arXiv:1702.05983*, 2017.

[61] IDA-Pro. Ida pro. https://www.hex-rays.com/products/ida/, 2017. (Accessed on 15/02/2017).

[62] G. C. Jam. Google code jam. https://code.google.com/codejam/, 2016. (Accessed on 15/02/2017).

[63] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin. Obfuscator-LLVM – software protection for the masses. In B. Wyseur, editor, *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*, pages 3–9. IEEE, 2015.

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

[65] V. Kešelj, F. Peng, N. Cercone, and C. Thomas. N-gram-based author profiles for authorship attribution. In *Proceedings of the conference pacific association for computational linguistics, PACLING*, volume 3, pages 255–264, 2003.

[66] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

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

[68] M. Koppel, J. Schler, and S. Argamon. Computational methods in authorship attribution. *Journal of the Association for Information Science and Technology*, 60(1):9–26, 2009.

[69] I. Krsul and E. H. Spafford. Refereed paper: Authorship analysis: Identifying the author of a program. *Comput. Secur.*, 16(3):233–257, Jan. 1997.

[70] R. C. Lange and S. 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, New York, NY, USA, 2007. ACM.

[71] 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 *Neural Information Processing, 1999. Proceedings. ICONIP '99. 6th International Conference on*, volume 1, pages 66–71 vol.1, 1999.

[72] C. H. Malin, E. Casey, and J. M. Aquilina. *Malware forensics: investigating and analyzing malicious code*. Syngress, 2008.

[73] A. Matyukhina, N. Stakhanova, M. Dalla Preda, and C. Perley. Adversarial authorship attribution in open-source projects. In *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*, pages 291–302. ACM, 2019.

[74] X. Meng, B. P. Miller, and S. Jha. Adversarial binaries for authorship identification. *arXiv preprint arXiv:1809.08316*, 2018.

[75] X. Meng, B. P. Miller, and K.-S. Jun. Identifying multiple authors in a binary program. In *European Symposium on Research in Computer Security*, pages 286–304, Oslo, Norway, 2017. Springer.

[76] X. Meng, B. P. Miller, W. R. Williams, and A. R. Bernat. Mining software repositories for accurate authorship. In *29th IEEE International Conference on Software Maintenance (ICSM)*, pages 250–259. IEEE, 2013.

[77] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. 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.

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

[79] N. Papernot, P. D. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami. The limitations of deep learning in adversarial settings. In *Proceedings of the IEEE European Symposium on Security and Privacy*, pages 372–387, 2016.

[80] M. Payer, L. Huang, N. Z. Gong, K. Borgolte, and M. 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.

[81] G. Pellegrino, M. Johns, S. Koch, M. Backes, and C. 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.

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

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

[84] E. Quiring, A. Maier, and K. Rieck. Misleading authorship attribution of source code using adversarial learning. In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, page 479–496, USA, 2019. USENIX Association.

[85] Radare. Radare. https://www.radare.org/, 2017. (Accessed on 15/02/2017).

[86] S. Rifai, Y. Dauphin, P. Vincent, Y. Bengio, and X. Muller. The manifold tangent classifier. In *NIPS*, volume 271, page 523, 2011.

[87] N. Rosenblum, B. P. Miller, and X. Zhu. Recovering the toolchain provenance of binary code. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 100–110. ACM, 2011.

[88] N. Rosenblum, X. Zhu, and B. Miller. Who wrote this code? identifying the authors of program binaries. *Computer Security–ESORICS 2011*, pages 172–189, 2011.

[89] R. Sarwar, C. Yu, S. Nutanong, N. Urailertprasert, N. Vannaboot, and T. 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.

[90] M. Schwarz, M. Lipp, and D. Gruss. Javascript zero: Real javascript and zero side-channel attacks. *Network and Distributed System Security Symposium (NDSS)*, 2018.

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

[92] E. C. R. Shin, D. Song, and R. Moazzezi. Recognizing functions in binaries with neural networks. In *24th USENIX Security Symposium*, pages 611–626, Washington, D.C., 2015. USENIX Association.

[93] L. Simko, L. Zettlemoyer, and T. Kohno. Recognizing and imitating programmer style: Adversaries in program authorship attribution. *Proceedings on Privacy Enhancing Technologies*, 2018(1):127–144, 2018.

[94] R. Socher, C. C. Lin, C. Manning, and A. 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.

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

[96] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, Jan. 2014.

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

[98] A. Stolerman, R. Overdorf, S. Afroz, and R. Greenstadt. Classify, but verify: Breaking the closed-world assumption in stylometric authorship attribution. In *IFIP Working Group*, volume 11, page 64, 2013.

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

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

[101] T. Tieleman and G. 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.

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

[103] M. Tomas, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. In *Proceedings of Workshop at International Conference on Learning Representations 2013*, Arizona, USA, May 2013.

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

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

[106] X. Yuan, P. He, Q. Zhu, and X. Li. Adversarial examples: Attacks and defenses for deep learning. *IEEE transactions on neural networks and learning systems*, 2019.

[107] Z. Yuan, Y. Lu, Z. Wang, and Y. Xue. Droid-sec: deep learning in android malware detection. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 371–372. ACM, 2014.