information

Article

Authorship Identification of Binary and Disassembled Codes Using NLP Methods

Aleksandr Romanov, Anna Kurtukova *, Anastasia Fedotova and Alexander Shelupanov

Department of Security, Tomsk State University of Control Systems and Radioelectronics, 634050 Tomsk, Russia; alexx.romanov@gmail.com (A.R.); afedotowaa@icloud.com (A.F.)
* Correspondence: av.kurtukova@gmail.com

Abstract: This article is part of a series aimed at determining the authorship of source codes. Analyzing binary code is a crucial aspect of cybersecurity, software development, and computer forensics, particularly in identifying malware authors. Any program is machine code, which can be disassembled using specialized tools and analyzed for authorship identification, similar to natural language text using Natural Language Processing methods. We propose an ensemble of fastText, support vector machine (SVM), and the authors' hybrid neural network developed in previous works in this research. The improved methodology was evaluated using a dataset of source codes written in C and C++ languages collected from GitHub and Google Code Jam. The collected source codes were compiled into executable programs and then disassembled using reverse engineering tools. The average accuracy of author identification for disassembled codes using the improved methodology exceeds 0.90. Additionally, the methodology was tested on the source codes, achieving an average accuracy of 0.96 in simple cases and over 0.85 in complex cases. These results validate the effectiveness of the developed methodology and its applicability to solving cybersecurity challenges.

Keywords: authorship; source code; disassembly; neural network; machine learning

Citation: Romanov, A.; Kurtukova, A.; Fedotova, A.; Shelupanov, A. Authorship Identification of Binary and Disassembled Codes Using NLP Methods. *Information* **2023**, *14*, 361. https://doi.org/10.3390/info14070361

Academic Editor: Mark Stevenson

Received: 31 May 2023
Revised: 18 June 2023
Accepted: 24 June 2023
Published: 25 June 2023

1. Introduction

Identifying the author of a computer program is a critical task in digital forensics [1] and plagiarism detection [2]. Solutions to this task can be beneficial for litigation related to intellectual property and copyright issues and for various forensic investigations of malicious software.

The existing methods for identifying the author of a computer program can be categorized into three groups: those that analyze the source code [3–7], the assembly code of the disassembled program [8–14], and universal methods applicable to both cases [14,15]. Although these methodologies are based on different algorithms and approaches, all of them share a common principle: each author-programmer has a unique coding style. This style can be identified through the following elements (see Figure 1):

1. Design patterns.
2. Language construct.
3. Code block formatting.
4. Code comments style.
5. Identifiers, variables, functions naming.
6. "Code smells" [16]—poorly written source code that does not conform to the language's code writing conventions used by the author of the program.

These features belong to the original program source code but are not present in the binary code. However, some of these features remain identifiable even after the program has been compiled and disassembled. As a result, determining authorship based on binary and disassembled code is more methodologically complex and requires modern solutions tailored to this type of analysis.

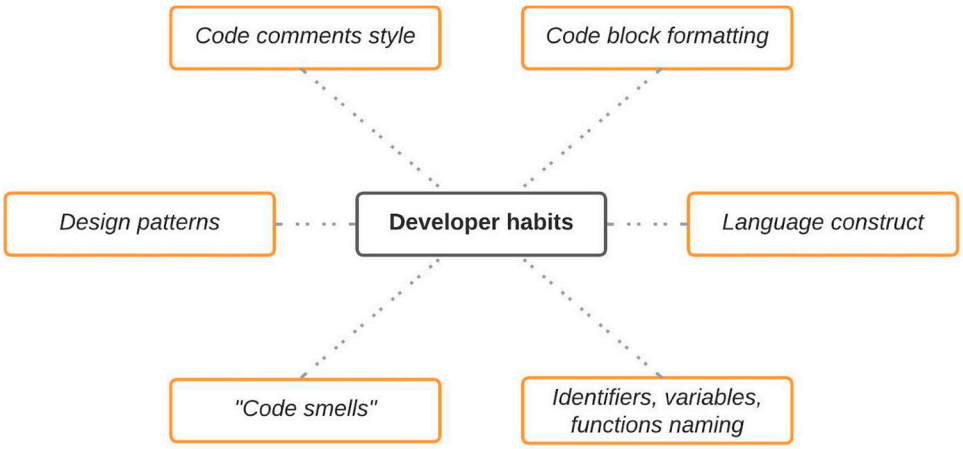

Figure 1. Developer habits.

This study aims to develop a comprehensive solution based on NLP algorithms that enables the precise identification of a program author with high accuracy, utilizing both source and compiled binary codes.

The development of such a methodology requires the utilization of state-of-the-art NLP methods. One actively growing and popular approach is multi-view learning [17–19]. Its principle lies in leveraging complementary data sources to provide a comprehensive representation of the research object. However, using this approach as a baseline carries certain risks for the following reasons:

1. Lack of multiple views. By definition, the multi-view learning process requires multiple distinct and complementary representations to ensure a comprehensive understanding of the research object. However, in the case of the textual representation of source code, additional views that reflect different aspects of the code may be lacking.
2. Complexity and interpretability. Program code is a highly structured text with numerous complex dependencies. Applying multi-view learning to source code can lead to increased computational complexity and make the interpretation of results more challenging. The machine learning methods used in this research allow for a more controlled process and a simpler interpretation of results.
3. Limited performance improvement. Multi-view learning can be highly beneficial in domains where different data representations provide additional insights into the research object. However, in the case of program code, such representations may not necessarily lead to improved performance compared to simpler and more interpretable machine learning methods. It is important for us to strike a balance between computational complexity and solution effectiveness.

Binary or disassembled code can be a distinct representation in addition to source code and be used in multi-view learning. This work focuses on the question of whether the use of binary code as a standalone component is possible for this purpose. As an alternative, we propose using an ensemble of classifiers.

The research presents a significant scientific novelty through the introduction of an ensemble of classifiers utilizing Natural Language Processing (NLP) methodologies. This ensemble comprises the author's hybrid neural network (HNN), SVM equipped with carefully selected feature space, and fastText utilizing experimentally optimized parameters. To the best of our knowledge, these specific approaches have not previously been employed for authorship identification where data are presented as disassembled code. This amalgamation of novel methodologies represents a pioneering contribution to the field.

The paper is organized as follows: Section 2 focuses on the literature review, including a detailed discussion of the studies aimed at identifying the author of binary programs;

Section 3 provides insights into our previous research concerning program source code authorship, addressing both simple and complex cases; in Section 4, we expound on the composition and formation of datasets of source and disassembled codes for programs; preliminary experiments conducted to address it are outlined in Section 5; Section 6 presents a universal method for identifying program authors based on the findings from the preliminary experiments; Section 7 focuses on the test cases of the methodology, examining its performance on both source and binary codes; in Section 8, we present a comprehensive summary of the results obtained and engage in a thorough discussion of the method's limitations and prospects for future development.

2. Literature Review

An effective system called Binary Authorship Verification with Flow-aware Mixture-of-Shared Language Model (BinMLM) has been developed for determining the authorship of binary codes [8]. This system utilizes a recurrent neural network (RNN) model that is trained on consecutive opcode traces extracted from the control flow graph (CFG). By combining these methods, unnecessary noise is eliminated, and the unique coding styles of developers can be accurately identified. Additionally, this system proves valuable in situations where limited computing resources are available. The authors conducted tests on disassembled datasets from Google Code Jam (GCJ), Codeforces, and real advanced persistent threat (APT) datasets. For the GCJ dataset, the average Area Under the ROC Curve (AUC-ROC) value was 0.865, with an average precision (AP) of 0.87. For the Codeforces dataset, the AUC-ROC values were 0.85 and 0.86, respectively. These results outperformed the n-gram approach and the method proposed by Caliskan-Islam et al. [14] by an average of 0.06 for the GCJ dataset and 0.19 for the Codeforces dataset. In addition to its high efficiency, the authors highlight that BinMLM is capable of providing organization-level validation, offering valuable information about the group responsible for an APT attack on software.

The authors of article [9] solve two related problems in binary code analysis: identifying the author of a program and finding stylistic similarities between programs written by unknown authors. The solution-finding process involves five steps. The first step involves collecting a corpus of programs with known authorship. In the second step, the CFG and instruction sequence are extracted for each binary file. Function template extraction is done using a recursive traversal parser. Among such templates, the authors identify idioms, graphlets, supergraphlets, libcalls, and call graphlets, as well as n-grams. Idioms are small instruction sequences that determine the stylistic features of the disassembled assembly code. Graphlets are subgraphs consisting of three nodes that are part of the CFG. Supergraphlets are adjacent nodes merged into one, and call graphlets are graphlets consisting exclusively of call instruction nodes. Libcalls are the names of imported libraries. In this study, n-grams are considered short byte sequences of length n. In the third step, a subset of functions corresponding to the programmer's style is selected by calculating mutual information between extracted functions and a specific developer, and then ranking them according to the calculated correlation. The fourth step involves training a Support Vector Machine (SVM) on the labeled corpus. The final step is clustering using the k-means algorithm. To avoid clustering based on the wrong feature, such as program functionality, the information obtained in the fourth step is used. A distance metric was used to transform unlabeled data before clustering, based on the labeled set. To test the approach, the authors used three datasets: GCJ 2009, GCJ 2010, and r CS537 2009. Accuracy was the metric used to evaluate classification performance based on cross-validation. For a combination of 5 function templates for 20 authors, it was 0.77 for GCJ 2009, 0.76 for GCJ 2010, and 0.38 for CS537. For five authors, the accuracy was higher, at 0.94, 0.93, and 0.84, respectively. Adjusted Mutual Information, Normalized Mutual Information, and the Adjusted Rand Index were used to evaluate clustering performance. For the GCJ 2010 dataset, they were 0.6, 0.72, and 0.48, respectively.

The study [10] focuses on BinGold, a system designed for binary code semantic analysis. The approach involves using dataflow to extract the semantic flow of the registers and semantic components of the control flow graph. These flows and components are then transformed into a new representation called the semantic flow graph (SFG). During binary analysis, many properties related to reflexivity, symmetry, and transitivity of relations are extracted from the SFG. The authors evaluated BinGold on 30 binary applications, including OpenSSL, Pageant, SQLite, and 7z, using precision, recall, and F1 as quality metrics. The metrics were calculated for each application and ranged from 0.66 to 0.9. In addition to similarity estimation, the authors conducted experiments on the GCJ dataset to address two other problems: binary code author identification and cloned component detection in executable files. The F0.5 metric [11] was 0.8 for the authorship identification task and 0.88 for the cloned component detection task, respectively. The authors argue that their proposed approach is more robust for binary code because the extracted semantic information is less susceptible to easy obfuscation, refactoring, and changes in compilation parameters.

A group of researchers from Princeton University has been working on solving the problem of software author identification for several years. In their previous studies, they focused on de-anonymizing source codes. However, in their recent work [12], they applied their prior experience to the analysis of binary codes. The authors argue that the most informative features used for source code classification are entirely absent in binary code. Therefore, they propose the utilization of supervised machine learning methods to de-anonymize the authors of binary files. The authors' experiment consisted of four steps. Initially, they disassembled the binary codes and then decompiled and translated the programs into C-like pseudocode. Subsequently, a fuzzy parser was employed to process the pseudocode, generating abstract syntax trees (ASTs) that contained both syntactic features and n-grams. To reduce dimensionality, the researchers performed a selection process that identified the most informative features from the disassembled and decompiled code using information gain and correlation-based feature selection methods. In the final step, they trained a random forest classifier on the resulting feature vectors. The accuracy of this approach, evaluated using the GCJ dataset, averaged 0.89. The authors emphasize that their method is robust against easy obfuscation techniques, changes in settings and compilation parameters, as well as binary files lacking their character tables.

The article [13] discusses the issue of performing forensic analysis on binary code files. To simplify reverse engineering in the context of forensic procedures, the authors present a system called the Onion Approach for Binary Authorship Attribution (OBA2). This system is based on three complementary levels: pre-processing, attribution using syntactic features, and attribution using semantic features. To extract meaningful functions, the authors utilize five predefined templates from [9] idioms, graphlets, supergraphlets, libcalls, and n-grams. These extracted features are then ranked according to their correlation with specific candidate authors and passed to the first level, known as the Stuttering Layer. At this level, a sequence of actions is proposed to identify user-defined functions and eliminate library code. These actions include binary disassembly, application of startup signatures, matching of entry points, compiler identification, accessibility of library signatures, creation of hashing-based patterns, marking of matched functions, and code filtration. The result of the first layer is the creation of function signatures for common libraries, which are essential for extracting user-defined functions. The next level is the Code Analysis Layer, where a combination of algorithms is employed to create an author's syntactic profile for each candidate author. These author profiles can later be utilized for clustering and classification tasks. The final level is the Register Flow Analysis Layer, which forms a binary code model called a register flow graph. This model captures programming style by analyzing the significant semantic aspects of the code and acts as an abstract intermediate representation between the source and assembly code of the program. The evaluation of the system was performed on the GCJ 2009 and GCJ 2010 datasets. The system achieved an accuracy of 0.93 for three program authors, 0.9 for five authors, and 0.82 for seven authors.

The study [15] presents an overview of current research in the field of program authorship attribution, focusing on both source code and binary code analysis. The authors aim to compile a comprehensive list of features and characteristics that are potentially relevant to attributing malware authorship. Based on their analysis, the authors identify several informative attributes for source code programs. These include linguistic features, such as the style and vocabulary used, as well as the presence of specific bugs and vulnerabilities. Other significant attributes encompass formatting, specific execution paths, and the AST, CFG, and Program Dependence Graph (PDG) representations of the code. When it comes to binary codes, the authors emphasize additional key features. Firstly, there is the compiler and system information, which can be inferred from unique instruction sequences or system-specific function calls. Furthermore, it is possible to gain insight into the programming language in which the program was written. Overall, the study sheds light on the diverse range of features that can be leveraged for program authorship attribution, both in the context of source code and binary code analysis. This is achieved through support subroutines and library calls preserved in the binary code. Secondly, it involves the opcode sequences, which represent specific actions dictated by assembly instructions such as imul, lea, jmp, and others. Thirdly, it includes strings, which are null-terminated American National Standards Institute (ANSI) strings, constant types used in calculations, and their values serving as offsets. Some informative features were examined in the aforementioned [9] study, including idioms, graphlets, and n-grams. In addition to these characteristics, system calls and errors are also significant. Within the experiment, three datasets were used: GitHub, GCJ, and Malware. The source codes were compiled with different settings using GNU Compiler Collection (GCC), Xcode, and Intel C++ Compiler (ICC). After disassembling the binary codes with Interactive DisAssembler (IDA), specific features and functions were extracted for subsequent ranking. Finally, the features with the highest ranks were used to evaluate the approaches presented in [12–14]. The approaches trained on informative features for 10 authors demonstrated the following accuracies: [12]—0.84, [9]—0.8, and [14]—0.89. Although the last approach proved to be the most effective, it exhibited the fastest decline in accuracy when the number of authors for classification increased. The approach in [12] was the most robust to changes in the number of authors.

It is worth noting that none of the previously discussed studies conducted additional experiments on method attacks. In software development, some programmers, particularly those involved in malware development, actively seek to protect their anonymity and employ de-anonymization tools. These tools can significantly undermine the accuracy of existing methodologies for determining program authorship. For instance, in [20], the authors focus on transforming a test binary to preserve the original functionality while inducing misprediction. The article highlights the inherent risks associated with attacks on binary code, where even a single-bit alteration can render the file invalid, lead to runtime failures, or lead to the loss of original functionality. The study explores two types of attacks on binary code: untargeted attacks, causing misprediction towards any incorrect author, and targeted attacks, causing misprediction towards a specific one among the incorrect authors. Researchers identify two primary avenues for attacks: modification of the feature vector and modification of the input binary itself. The authors' non-targeted attack achieved an average success rate of 0.96, indicating the effectiveness of obfuscating the authors' style. The targeted attack had a success rate of 0.46, highlighting the complexity and potential for manipulating the programming style of a specific author. In summary, the study's findings reveal that binary code authorship identification methods relying on code functions are susceptible to authorship attacks due to their vulnerability to modification.

Preventing attacks on methods is not the only aspect that researchers developing source code authorship identification solutions should consider. Table 1 shows the main advantages and disadvantages of each of the proposed approaches. It takes into account the most important aspects: First, the effectiveness of the approach for identifying the author-virus writer as the target application scenario. Second, robustness to noise in data,

that is, the ability to ignore uninformative and/or downgrade the effectiveness of the author's approach. Third, resistance to obfuscation, one of the most popular methods of attack. Fourth, applicability to both binary and source code, that is, the universality of the approach and the ability to determine authorship based on both the source and binary code of the program.

Table 1. Comparison of the advantages and disadvantages of approaches.

Study	Effective for Viruses	Resistant to Noise in Data	Resistant to Code Obfuscation	Applicable to Source and Binary Codes
Song Q. [8]	+	+	−	−
Rosenblum, N. [9]	−	+	−	−
Alrabaee S. [10]	−	−	+	−
Caliskan-Islam, A. [12]	−	+	−	−
Alrabaee S. [13]	−	+	−	−
Caliskan-Islam, A. [14]	−	+	−	+

There were several conclusions drawn from the analysis conducted:

1. The majority of the features utilized predominantly describe the functionality of the program rather than the author's style.
2. The feature space needs to be filtered based on the informativeness of each individual feature.
3. The effectiveness of the approaches depends on the specific domain. The features used may be characteristic of a particular domain, such as programs written for developer competitions or malicious software.
4. Feature source importance. The authors employ various decompilation tools that have different functionalities and settings. This variation can have a negative impact on the classification results.
5. Misleading features. The experimental results indicate that highly ranked informative features are often not informative and are not related to the author's style. The most common reason for this issue is that the highest scores in mutual information calculations are assigned to code fragments automatically generated by the compiler rather than genuine authorship-related features.
6. Attacks on authorship attribution methods have the potential to significantly decrease the effectiveness of modern methodologies, as they are not robust against obfuscation and de-anonymization tools. These attacks can undermine the reliability and robustness of the program authorship identification process.

3. Our Previous Research

We examined both simple [21] and complex cases [22,23] when addressing the task of determining the author of a source code. Simple cases encompassed scenarios where the authorship determination involved straightforward source codes containing explicit authorship features. On the other hand, complex cases involved analyzing obfuscated code, code that met coding standards, short commits, mixed training datasets, and artificially generated code.

As part of the study [21], a series of experiments were conducted to analyze source codes written in the eight most popular programming languages. These codes were written by developers with varying levels of qualification and experience, ranging from engineering students to professionals with extensive corporate development backgrounds.

In the experiments, both SVM and the author's HNN were considered classifiers. The feature set for SVM included lexical characteristics (such as loop nesting depth, average string length, average function parameter count, etc.), structural characteristics (such as the ratio of whitespace to non-whitespace characters, the ratio of empty lines to code length, the ratio of spaces to code length, etc.), and "code smells" (such as the average number of parameters in class methods, average lines of code in methods, comment length, etc.).

The fast correlation filter (FCF) was employed to filter out uninformative attributes. Its effectiveness compared to other feature selection methods has also been determined in previous studies [21–23]. This filter utilized a symmetric uncertainty measure to identify dependencies between features and form a subset of informative features. By using the fast correlation filter, only features that are informative for the specific case are included in the training set. This subset of informative features can account for 10% or more of the original feature set. As a result, we eliminate noise in the training data and enhance the discriminative power of the classifiers being used. The parameters of the SVM were determined based on previous work conducted on related topics [24].

The advantage of HNN (Figure 2) over SVM lies in its ability to independently extract informative features. The source code was transformed using the one-hot encoding method, creating a vector of 255 zeros and a single one positioned according to the American standard code for information interchange (ASCII) code of the character. This vector was then fed as input into the neural network. The neural network architecture consisted of convolutional, recurrent, and dense layers. The convolutional part employed Inception-v1 with filters of different dimensions (Conv1 $\times$ 1, 3 $\times$ 3, and 5 $\times$ 5) to capture both local and global distinctive characteristics. The recurrent component, represented by a bidirectional gated recurrent unit (BiGRU), captured short-term and long-term temporal dependencies. The dense layers were responsible for scaling the network. Finally, the output layer, which utilized the Softmax function, transformed the logits obtained from the dense layers into a probability distribution based on the number of classes (authors), N. The experimental results demonstrated that, on average across programming languages, HNN outperformed SVM by 5%. HNN achieved an accuracy of 97% in certain cases, while SVM achieved 96% accuracy.

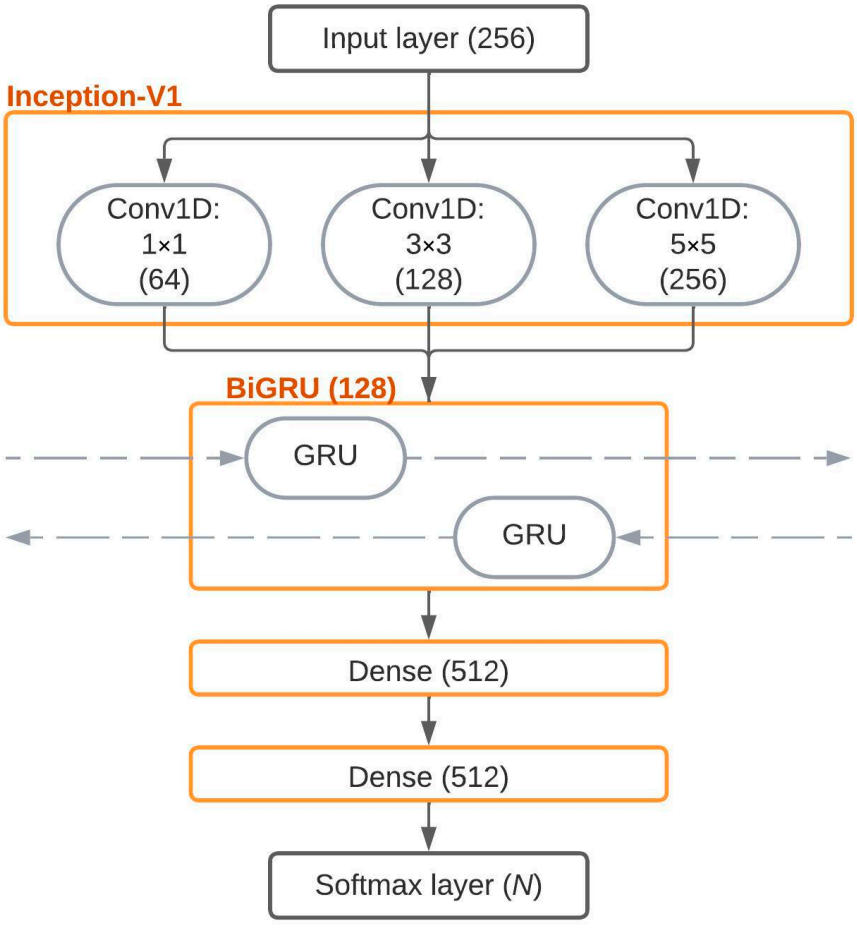

Figure 2. HNN architecture.

The focus of our previous study [22] revolved around two complex scenarios: obfuscated source codes and codes written according to coding standards. The author's HNN was employed to determine the authorship of the source code. Initial experiments revealed a significant decrease in accuracy; however, modifying certain parameters proved effective in making the classifier suitable even for complex cases. By employing updated parameters and expanding the training dataset, HNN exhibited high accuracy in both scenarios. The loss in identification accuracy for obfuscated source codes was found to be below 10%, with an average accuracy of 85%. Similarly, an average accuracy of 85% was achieved when the source code was written according to coding standards.

Other complex cases (mixed datasets, artificially generated source codes, and codes developed collaboratively) were examined in [23]. Classifiers such as the author's HNN, fastText, and Bidirectional Encoder Representations from Transformers (BERT) were evaluated for their effectiveness. Among these, the author's HNN demonstrated the highest performance with an equivalent amount of data. For mixed datasets consisting of two languages, the accuracy was 87%. In the case of datasets involving three or more languages, the accuracy remained at 76%. In scenarios where author identification was conducted based on commits in team-based development, the average accuracy ranged from 87% to 96%, depending on the volume of training data. Finally, for data generated by pre-trained Generative Pretrained Transformer (GPT) models on the source code, the average accuracy was 94%.

Based on the obtained experience, for the universal methodology of program author identification based on both its source and binary codes, three machine learning models are proposed:

- The author's HNN with experimentally determined parameters;
- fastText with default parameters;
- SVM trained on the feature set defined in [9].

Since disassembled code is significantly longer than the corresponding source code implementing the same program, BERT would require unreasonably high computational and time resources. Therefore, it is not considered within the scope of this research.

4. Dataset Description

The analysis of contemporary solutions for the problem of binary or disassembled code author identification has highlighted certain aspects that require special attention during the development of our own methodology. The primary consideration revolves around the approach to dataset formation for the study. Many researchers have utilized the publicly available GCJ source code dataset [25]. This choice carries both advantages and disadvantages.

On the one hand, using the GCJ dataset provides researchers with a substantial amount of experimental data, enhancing the objectivity of their evaluations. However, this dataset possesses specific characteristics. It primarily consists of solutions to the same problems and cases, limiting the relevance of the evaluation solely to this particular domain—programs created within the context of the competition.

Consequently, the approaches derived from such data have not been tested on other datasets, making it likely that they will exhibit significantly lower accuracy due to the domain-specific nature of the GCJ dataset.

Therefore, we decided to create our own dataset of source and binary/disassembled codes. This process involved several key steps (Figure 4):

1. The data downloading process from the hosting platform was performed using Python scripts that utilized the GitHub Application Programming Interface (API) [26].
2. Source code filtering. As a result of the collection, the raw dataset contains source codes up to five lines long, as well as those that are uncompilable or contain syntax errors. We consider such data to be uninformative, so they were filtered out. Additionally, we filtered out all the authors whose repositories contained less than 20 source code files. Thus, we use source codes that can be used with GCC, the compiler chosen

for the study [27]. The final statistics on the dataset are presented in Table 2, and the distribution of authors by the number of files in the dataset is shown in Figure 3.

3. Compiling source codes using the GCC compiler. The source codes, obtained after filtering, were compiled with the specified version of GCC. The compiler version, environment settings, and compilation parameters were either sourced from the README file or, if unavailable, set by default.
4. Disassembling executable files. It was a crucial step in the dataset creation process, as the primary objective was to develop a methodology for identifying the author of an unknown executable file. To accomplish this, the IDA Pro tool [28] was utilized for the disassembly process.

Table 2. Dataset statistics.

Statistical Measure	Source Codes	Disassembled Codes
General number of authors	167	167
General number of files	12,779	5661
Average number of files per author	63	25
Maximum number of files per author	132	51
Minimum number of files per author	20	20
Average number of lines of code	146	1677

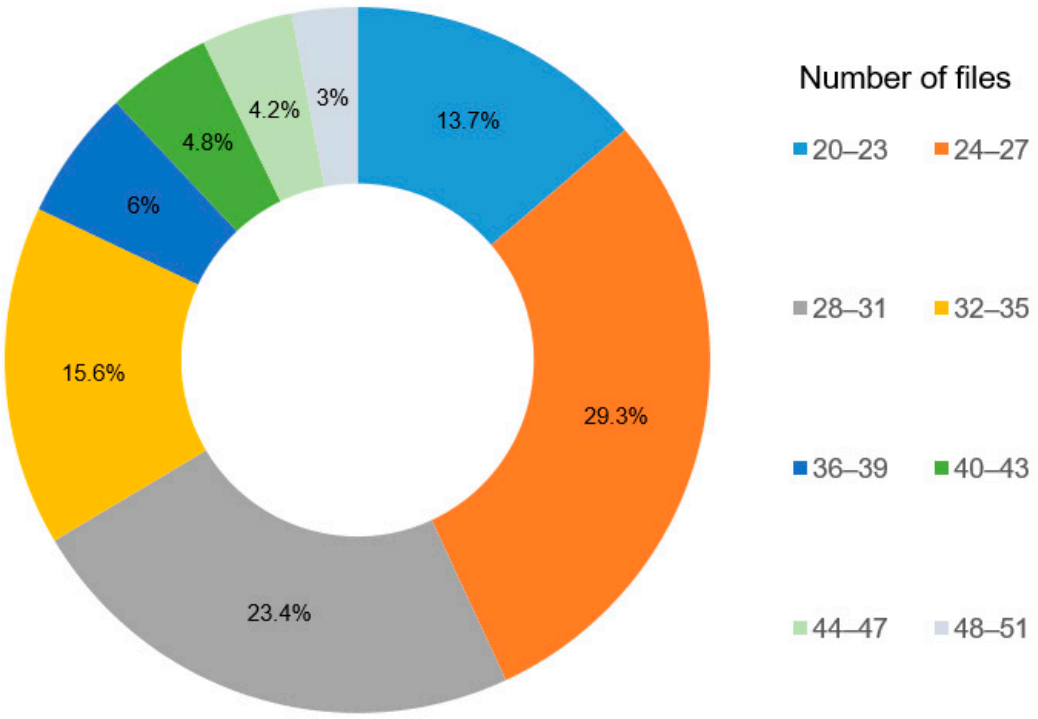

Figure 3. Distribution of authors by the number of files in the dataset.

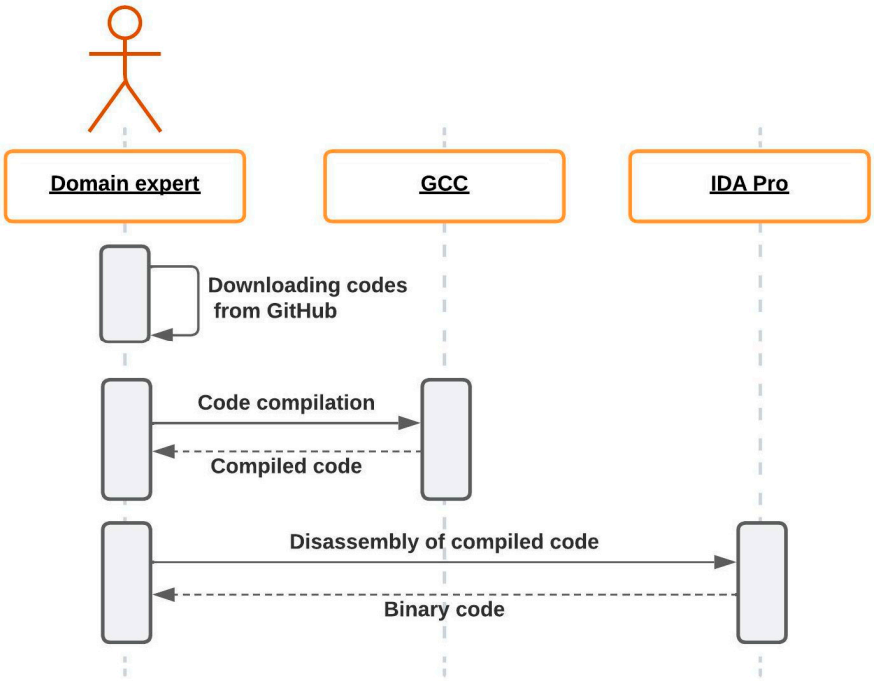

Figure 4. UML diagram of the dataset creating process.

5. Experimental Setup

In order to establish an efficient and universally applicable methodology, it is essential to conduct preliminary experiments using individual classifiers. These experiments are designed to assess the accuracy of the selected models, namely HNN, SVM, and fastText. By evaluating the performance of each classifier, we can determine their effectiveness in identifying the author of a program with high accuracy.

In our study, we utilized K-fold cross-validation and compared our results with other works and our own previous experiments [21,23,24], ensuring objectivity and comprehensive evaluation. While we acknowledge the importance of leave-one-user-out (LOUO) and other cross-validation techniques, we believe that K-fold cross-validation was a suitable choice for our research objectives, providing reliable and meaningful results. This approach ensures an objective evaluation and addresses the issue of excessive variability in estimates. Accuracy is used as a quality metric:

$$Acc = \frac{TP + TN}{TP + FP + TN + FN} \quad (1)$$

where *TP*—true positive, *TN*—true negative, *FP*—false positive, and *FN*—false negative.

These parameters are calculated based on the confusion matrix, which stores information about the right and wrong decisions made by the model for each class.

As part of the experiment, cases were considered with 2, 5, and 10 candidate authors, and the number of training files ranged from 10 to 30. The reason for this choice is that analyzing less than 10 files is insufficient for high-accuracy authorship identification and analyzing authors with more than 30 files requires significantly greater computational power due to the larger volume of data involved. Furthermore, it is worth noting that the majority of individuals typically do not possess more than 30 files. Considering these factors, we have chosen to present results only for authors within a range of 10 to 30 files (Figure 3). This range strikes a balance between obtaining a sufficiently representative sample size and managing computational requirements. The results are presented in Figures 5–7.

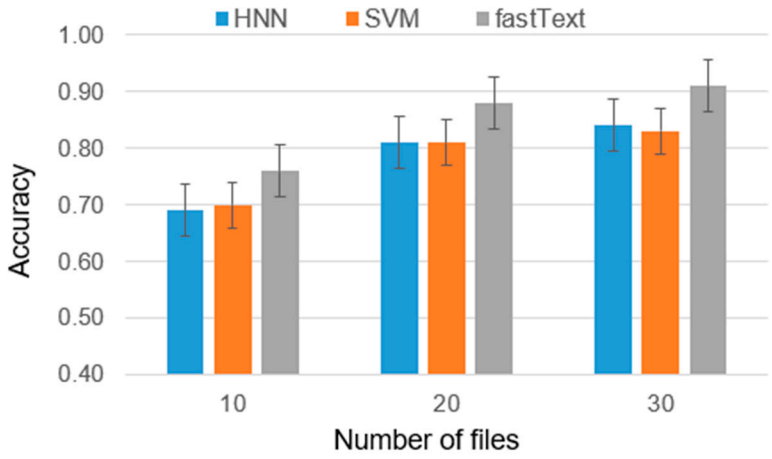

Figure 5. Accuracy metrics for two authors.

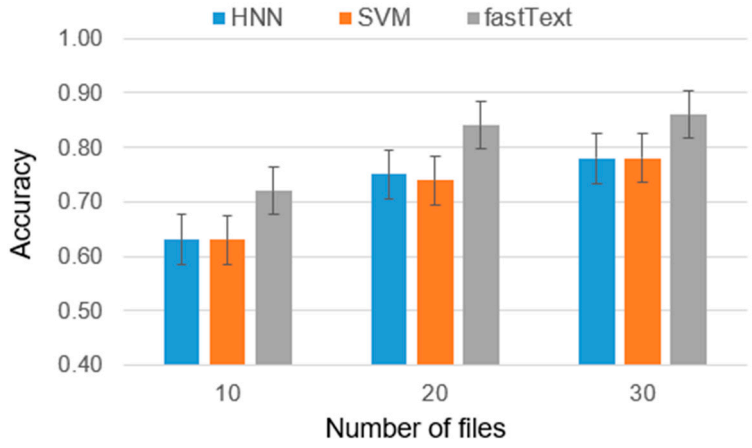

Figure 6. Accuracy metrics for five authors.

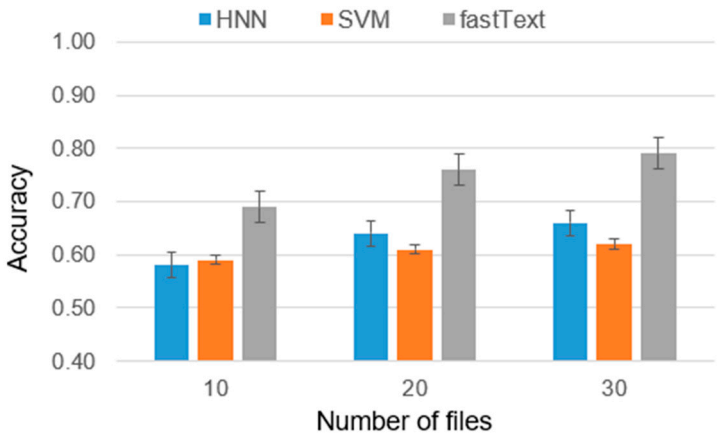

Figure 7. Accuracy metrics for 10 authors.

In order to determine the statistical significance of the obtained results, Non-parametric Friedman and Némenyi post hoc tests were employed. We formulated the null hypothesis, stating that the differences in accuracy between fastText, HNN, and SVM are random, and the alternative hypothesis, suggesting the statistical significance of the obtained results. To calculate the *p*-value for the Friedman statistical test [29], the results of the 10-fold cross-validation were used for 2, 5, and 10 authors with 30 files. The *p*-value for 2 authors was 0.0000549; for 5 authors, it was 0.0000914; and for 10 authors, it was 0.0000934. In none of the cases did the *p*-value exceed the threshold of 0.05, thus leading to the acceptance of the

alternative hypothesis, indicating that the differences in model accuracy are significant and not random. Némenyi's post hoc test [30] was utilized to assess the differences between the models. The graphical interpretation of the Demshar plot is presented in Figure 8, where subfigure (a) demonstrates the results for 2 authors, subfigure (b) for 5 authors, and subfigure (c) for 10 authors, respectively.

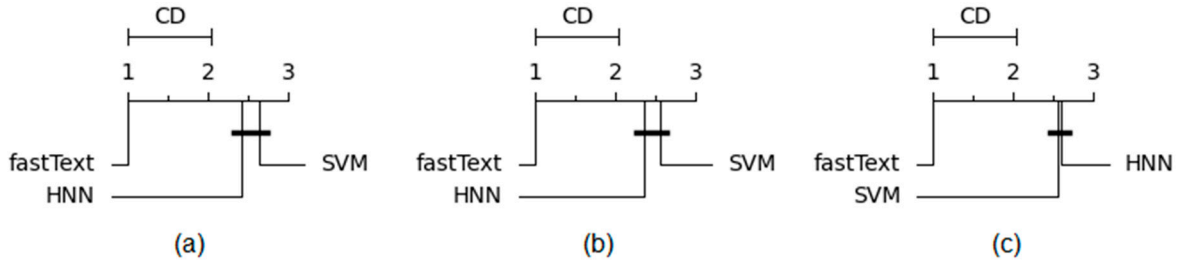

Figure 8. Results of Némenyi's post hoc test for (**a**)—2 authors, (**b**)—5 authors, (**c**)—10 authors.

The results obtained can be interpreted as follows: in the case of two authors, the difference between the average ranks of HNN and SVM is smaller than the critical difference (CD), indicating that the difference in the effectiveness of these models is insignificant. Furthermore, both models were significantly outperformed by fastText in terms of ranks. For five and ten authors, the results were similar, with the difference in effectiveness between HNN and SVM becoming nearly negligible.

Based on the final results of the experiments and tests, it can be concluded that fastText is particularly effective in identifying the author of the disassembled program code. With a sufficient number of codes per author, the accuracy of fastText reaches 0.91. It is important to note that in earlier studies on identifying the author of the source code of a program, fastText showed lower efficiency compared to HNN, with an average difference in accuracy of 5%. However, the combined use of these models as an ensemble of classifiers can provide a universal solution for determining program authorship.

6. Methodology Based on the Ensemble of Classifiers

A generalized methodology for determining the author of a source and/or binary/disassembled code based on an ensemble of classifiers is presented in Figure 9.

The first step, extensively described in Section 2, involves preparing the training data for vectorization. In the case of identifying the author of the program based on the source code, the file is directly passed to the vectorization module without any preprocessing. However, to identify the author of a program based on the binary code, the source code undergoes compilation with GCC and subsequent disassembly with IDA Pro. The resulting reverse-engineered file is forwarded to the vectorization module.

The second step is to convert the training data into a vector form. Each of the classifiers requires a corresponding format of input data:

- The input for SVM consists of a set of normalized and vectorized features. When analyzing assembly code, the feature space for SVM includes a set of features proposed in [9], such as idioms, graphlets, supergraphlets, libcalls, call graphlets, and *n*-grams. On the other hand, when analyzing source code, a set of features developed in the authors' previous studies [21] is utilized.
- HNN processes the data in raw text format without any preprocessing. The original text is directly fed into the one-hot vectorizer. One-hot encoding creates a vector containing 255 zeros and a single unit for each unique character in the text. The unit is positioned based on the character's ASCII code to represent its presence or absence in the text.
- fastText also requires the raw text format of data as its input. The model automatically generates word embeddings using skip-gram and Continuous Bag of Words (CBOW) methods.

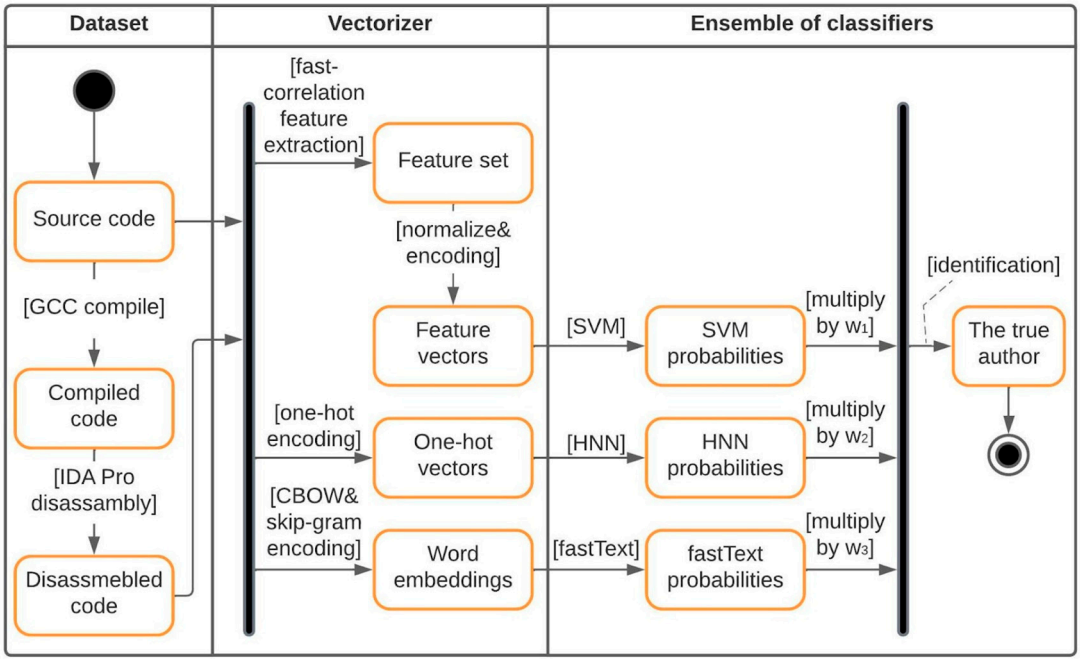

Figure 9. UML activity diagram for the process of determining the author of a program.

The final step is presented in Algorithm 1. This step involves training the ensemble of classifiers for author identification. The optimal hyperparameters for SVM and HNN classifiers were determined through experimental evaluation using the Grid search algorithm [31], while for fastText, the built-in autotune-validation function was utilized.

The following parameters were used for SVM [32]:

- SVM type: multi-class classification;
- Training algorithm: sequential optimization method;
- Kernel: sigmoid;
- Regularization parameter (C): 1;
- Tolerance error: 0.00001.

HNN hyperparameters [33] were chosen as follows:

- Optimization function: Adadelta;
- Activation function for the output layer: Softmax;
- Regularization function: 0.2 (using the Dropout function);
- Activation function for hidden layers: Rectified Linear Unit (ReLU);
- Learning rate: 10^{-4};
- Decay rate (rho): 0.95;
- Epsilon (eps): 10^{-7}.

The majority of fastText hyperparameters [34] were optimal for task solving:

- Learning rate: 0.9;
- Threads: 15;
- lrUpdateRate: 5.

The main principle of the ensemble is to assign weight coefficients (presented in the schema as w1, w2, and w3) to the outputs of individual classifiers. In the analysis of disassembled codes, the highest weight (w3 = 0.4) is assigned to the solutions provided by fastText, as it has proven to be the most accurate classifier for this case. The decisions made by HNN and SVM carry equal weights—w1 = 0.3 for SVM and w2 = 0.3 for HNN, respectively. Consequently, the author identified by fastText will be considered correct in all cases except when the solutions from both HNN and SVM coincide, resulting in a combined weight of 0.6 against fastText's weight coefficient of 0.4.

In the analysis of source codes, the highest weight is assigned to the solutions provided by HNN (w2 = 0.4), indicating its effectiveness in complex cases. The solutions from fastText and SVM classifiers, which are considered less effective in such scenarios, carry equal weights of 0.3. This is similar to the case with binary codes—w3 = 0.3 for fastText and w1 = 0.3 for SVM, respectively. The approach to identifying the author of the source code using this ensemble is similar to the analysis of assembly code, where the weights of the classifiers determine their contribution to the final identification.

Algorithm 1. Ensemble of classifiers.

```
Set x_src: >Source code feature vectors
Set x_asm: >Assembly code feature vectors
Set data_type: >'asm'
Set epochs: >10
Set Estimator 1: >SVM
Set Estimator 2: >HNN
Set Estimator 3: >fastText
Set model_list: >[Estimator 1, Estimator 2, Estimator 3]
Set anon_ex: >'path/filename.asm'
procedure predict(models_list, weights, ex, data_type):
        pred = {}
        for model_f, model_w in (models_list, weights):
                model = model.load(model_f)
                probs = model.predict(ex)
                pred[model] = [argmax(probs)×model_w, argmax(probs, axis = −1)]
        if (data_type = 'asm'):
                if pred['svm.bin'][0] = pred['hnn.bin'][0]:
                        if pred['svm.bin'][0] + pred['hnn.bin'][0] > pred['ft.bin'][0]:
                                return pred['svm.bin'][1]
                else:
                        return pred['ft.bin'][1]
        else:
                if pred['svm.bin'][0] = pred[ft.bin'][0]:
                        if pred['svm.bin'][0] + pred['ft.bin'][0] > pred['hnn.bin'][0]:
                                return pred['svm.bin'][1]
                else:
                        return pred['hnn.bin'][1]
procedure train_Estimator(Estimator, train_set, val_set, epochs):
        for i in epochs do:
                Estimator, logs = Estimator.fit(train_set, val_set, shuffle = True)
        return Estimator, logs
procedure set_data_weights():
        if (data_type = 'asm'):
                weights = [0.3, 0.3, 0.6]
                data = x_asm
        else:
                weights = [0.3, 0.6, 0.3]
                data = x_src
        return weights, data
begin
weights = set_weights(data_type)
for tr_set, ts_set in KFold(data, k_splits = 10):
        for model in model_list:
                model, logs = train_Estimator(model, tr_set, ts_set, epochs, repeats)
                model.save({model}.bin)
ex = vectorize(anon_ex)
true_author = predict(['svm.bin', 'hnn.bin', 'ft.bin'], weights, ex, data_type)
end
```

7. Testing of the Methodology for Identifying the Author of the Program

The developed methodology underwent testing for the source and binary codes of programs. Experiments were conducted to evaluate the methodology's performance in author identification, encompassing both simple and complex cases. In certain experiments, the number of disputed authors was increased to 20, thereby challenging the methodology's ability to accurately identify the correct author. The results of the ensemble of classifiers, specifically for the analysis of disassembled codes, are illustrated in Figure 10. Additionally, the results of the analysis of source codes in simple cases are presented in Figure 11. These figures provide visual representations of the performance metrics obtained from the experiments, enabling a comprehensive understanding of the classifier's effectiveness.

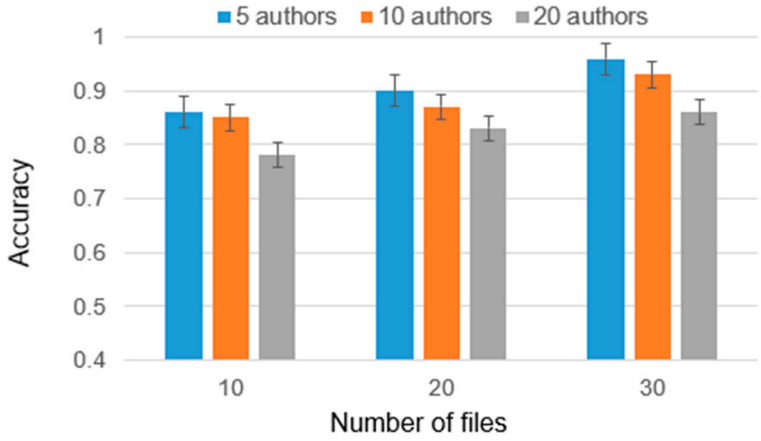

Figure 10. The results of binary/disassembled code author identification.

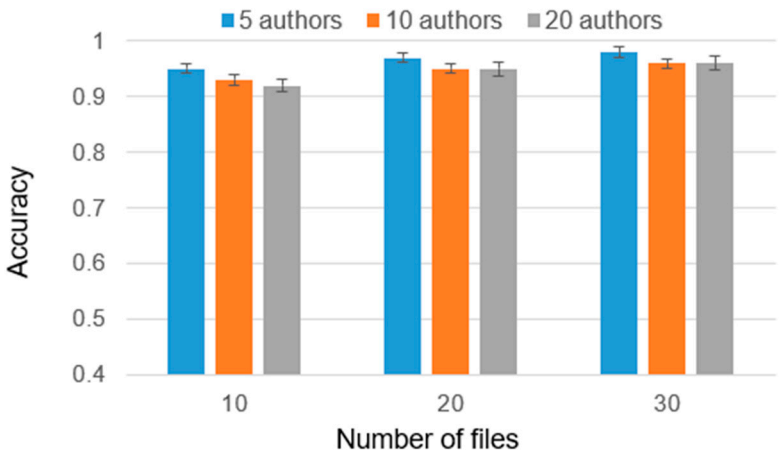

Figure 11. Source code identification results in simple cases.

Applying the ensemble to binary codes resulted in an accuracy improvement of over 0.1 when compared to using the classifiers individually. With an ample number of training files, the accuracy exceeds 0.9.

The results obtained by the ensemble for the original source codes are comparable to those obtained earlier [22], that is, the ensemble has no negative effect on this case.

In addition to simple cases of source code author identification, it is crucial to examine more complex cases to ensure the stability of the enhanced methodology. These scenarios include obfuscation, compliance with coding standards, team development, and the use of artificially generated code samples.

In Figure 12, the results of author identification for obfuscated source code are presented. The obfuscation tool used in this analysis was AnalyzeC [35]. The obfuscation process with this tool involves the following steps:

- Complete removal of comments, spaces, and line breaks;
- Adding pseudo complex code that does not change functionality;
- Using preprocessor directives: obfuscation may involve manipulating or transforming preprocessor directives, such as #define or #ifdef, to further obfuscate the code's logic or structure;
- Replacing strings with hexadecimal equivalents.

The results obtained by the ensemble are comparable to those obtained earlier [22], that is, the ensemble does not have a negative effect on this case.

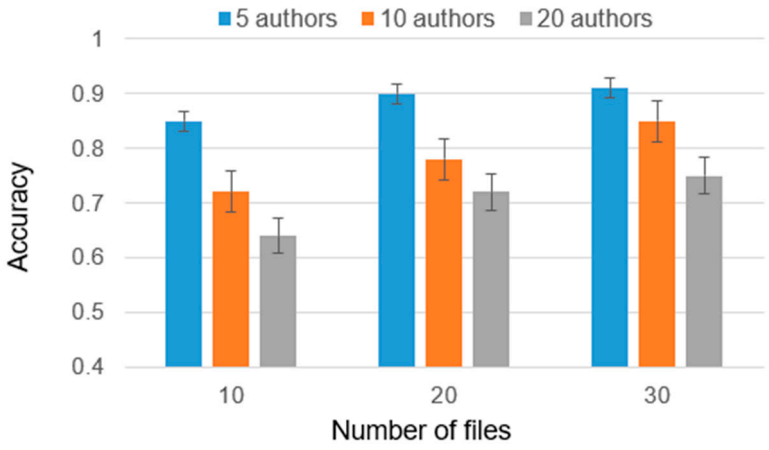

Figure 12. Results of authorship identification for obfuscated code.

The next case involves identifying the author of source code written by a development team. In this case, programmers utilize a version control system, such as GitLab or GitHub, and commit their changes to the project repository. It is common for source codes to contain indications of multiple authors. Hence, the ability to determine authorship based on commits becomes particularly significant.

During the data collection process, information regarding commits, their content, and authors was obtained utilizing the GitHub API. Figure 13 illustrates the results of author identification for the source code constructed from these commits. By employing an ensemble of classifiers instead of solely relying on a separate HNN [23], a noticeable increase in accuracy was achieved. On average, the accuracy gain obtained through the use of the ensemble amounted to 0.03.

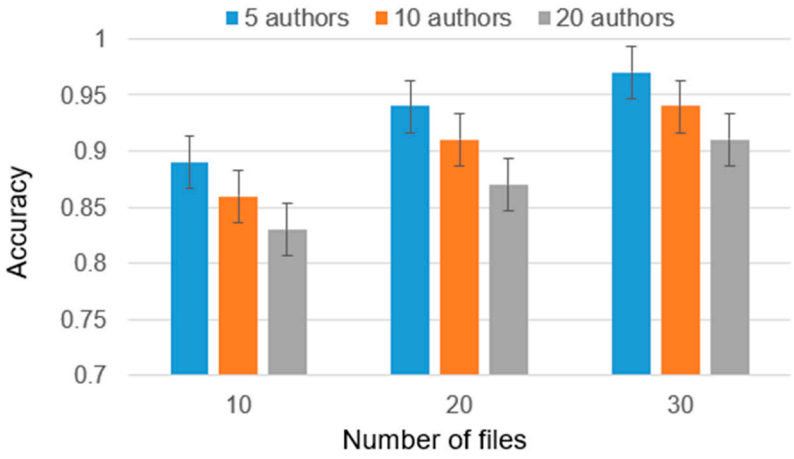

Figure 13. Results of authorship identification based on commits.

One of the most challenging cases in the early research was the authorship identification of source code written according to coding standards (see Figure 14). The primary

objective of coding standards is to facilitate code maintenance and enhance code readability. However, this can often result in minimizing the unique attributes that could help identify the author.

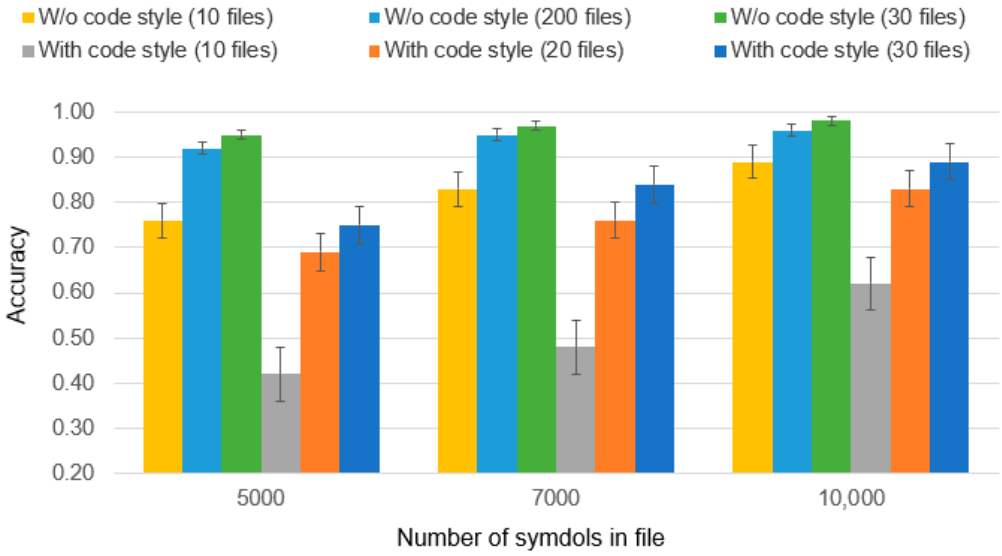

Figure 14. Results of authorship identification for code written according to coding standards.

In the evaluation of this particular case, the source codes of the Linux Kernel [36] were utilized. These codes, written in C/C++ following widely accepted standards [37], served as the basis for assessing author identification. Despite the inherent difficulties posed by code conforming to coding standards, an average increase of 0.03 in accuracy was achieved when compared to using the HNN separately [22].

The final challenging case arises from the increasing popularity of GPT models and their effectiveness in source code generation. Specifically, the experiments focused on the task of distinguishing authorship between different generative models (GPT-3, GPT-2, and RuGPT-3), as illustrated in Figure 15. To tackle this case, the ensemble approach was employed, which proved to be beneficial. The utilization of an ensemble of classifiers yielded a noteworthy increase of precisely over 0.07 in accuracy when compared to using the HNN alone [23].

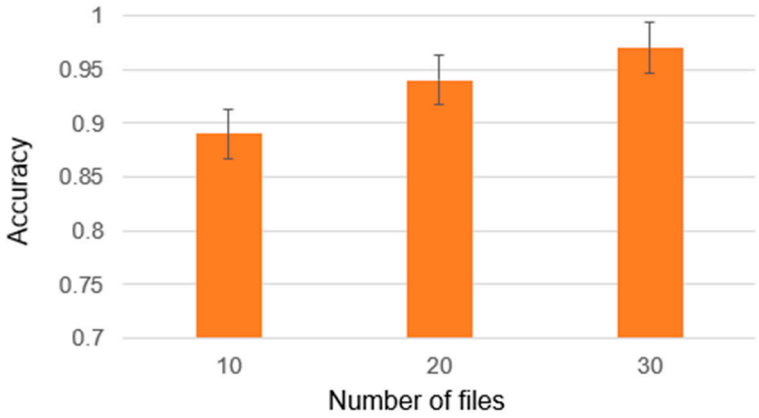

Figure 15. Results of authorship identification for artificially generated code.

To ensure that our method does not have a negative impact on complex cases compared to simple ones, we conducted a paired samples *t*-test. This test involves comparing the results of cross-validation between simple and complex cases and calculating a *p*-value for each pair. The null hypothesis, which states that the difference is not statistically significant,

is accepted when the *p*-value is greater than 0.05. The alternative hypothesis suggests a significant loss in accuracy. For the pair "simple source code—obfuscated source code", the *p*-value was 2.35. For the pair "simple source code—commit", the *p*-value was 0.06. For the pair "simple source code—artificially generated source code", the *p*-value was 0.88. Lastly, for the pair "simple source code—code written according to coding standards", the *p*-value was 1.83. None of these pairs yielded a result below the threshold of 0.05, indicating that the difference in accuracy between simple and complex cases is not statistically significant.

The summarized information on the results obtained in this study is presented in Table 3. To facilitate a clearer comparison, we considered the maximum number of authors (10) and files (30) for both simple and complex cases of source code authorship identification. The table includes the results of the ensemble method and individual classifiers. It also presents the results for SVM with and without the fast correlation filter for feature selection.

Table 3. The summarized information on the results obtained in this study.

Method	Binary Codes	Simple Source Codes	Obfuscated Source Codes	Commits	Source Codes Written According to Code Standards
SVM w/o FCF	0.39	0.43	0.32	0.4	0.31
SVM with FCF	0.62	0.89	0.74	0.7	0.65
HNN	0.66	0.95	0.85	0.93	0.88
fastText	0.79	0.88	0.75	0.87	0.74
SVM with FCF + HNN	0.69	0.92	0.8	0.89	0.87
SVM with FCF + fastText	0.8	0.88	0.77	0.89	0.74
HNN + fastText	0.82	0.93	0.85	0.9	0.87
fastText + HNN + SVM	0.93	0.96	0.85	0.94	0.89

The information about the distinction of authorship between generative models has been excluded from the summary table, as their comparison with other obtained results does not provide any valuable insights.

The results we obtained on the GCJ dataset are presented in Table 4. They are compared to the results reported by the authors in their published works [10,13,14].

Table 4. Results of comparative analysis.

Study	Number of Authors	Measure	Author's Result	Our Result	Efficiency Difference
Alrabaee, S. [10]	5	F0.5	0.8	0.96	+0.16
	10	F0.5	0.76	0.93	+0.17
	20	F0.5	0.71	0.88	+0.17
Rosenblum, N. [13]	5	Acc	0.935	0.97	+0.015
	20	Acc	0.765	0.87	+0.105
Alrabaee, S. [14]	3	Acc	0.93	0.99	+0.06
	5	Acc	0.9	0.97	+0.07
	7	Acc	0.82	0.96	+0.14

Several experiments were carried out to ensure that the developed methodology matches or even surpasses the accuracy of other research teams. Considering that the majority of papers relied on the GCJ dataset for evaluating their approaches, we chose to perform supplementary experiments using our methodology on the same dataset, specifically focusing on binary code analysis. To provide a fair and unbiased comparison, we incorporated the GCJ 2009 and 2010 data into our dataset. The metrics and author counts were consistent with those stated in the referenced articles. It should be noted that in one of

the papers [10], a performance metric different from accuracy (Acc) was used. The authors employed the F0.5 metric, which is calculated as follows:

$$F0.5 = \frac{1.25 \times Precision \times Recall}{0.25 \times Precision + Recall} \quad (2)$$

The table demonstrates that our ensemble-based methodology is on par with, if not superior to, the methods proposed by previous researchers. Notably, the ensemble demonstrates a significant improvement in accuracy in some cases. Furthermore, the ensemble's performance on the GCJ dataset surpasses that on GitHub. This can be explained by the fact that evaluating classifier accuracy on GCJ data lacks objectivity due to its specific characteristics. The GCJ dataset primarily comprises homogeneous data, enabling the classifier to focus solely on copyright features while disregarding functional differences and program specifics. In contrast, the GitHub dataset consists of heterogeneous data, including different programmer expertise and qualifications and a diverse range of tasks. Consequently, the experiments on GitHub simulate real-world applied problems, leading to a more objective assessment.

8. Conclusions

The article aims to develop a universal methodology for authorship attribution for both source code and assembly code. Several research studies were conducted to identify the most effective classifier among modern NLP algorithms. The results obtained demonstrated that the author's HNN, developed in previous studies, is the most accurate for analyzing source codes. In contrast, for analyzing assembly codes, fastText with optimal parameters was found to be the most accurate. Based on these findings, it was decided to combine these classifiers into an ensemble and supplement it with SVM, which operates on feature sets selected by experts. The solution based on the ensemble of classifiers was supplemented with weight coefficients that vary depending on the problem being solved. For assembly codes, the highest weight was assigned to fastText solutions, while for source codes, the HNN received the highest weight.

The developed methodology underwent testing for both simple and complex cases. In the simple case, where a sufficient number of authors' files were available, the accuracy for both source and binary codes exceeded 0.9. Additionally, the accuracy remained comparable to that achieved by the author's HNN for obfuscated source codes. Moreover, for source codes adhering to coding standards, formed from commits and artificially generated, an average increase in accuracy of 0.04 was achieved.

Thus, we have successfully improved the previously suggested methodology, adopted it to analyze assembly codes, and rendered it universally applicable. This methodology allows, firstly, the identification of the program author even in the absence of its source code, based solely on the disassembled code. This can be particularly useful in cases where authorship needs to be established for malicious software. Secondly, it enables resolving the authorship of the program in legal proceedings involving intellectual property and copyright. A third relevant application is the detection of plagiarism in the educational process, particularly in student programming assignments, aiming to enhance the objectivity of assessment.

Through our extensive research, we have identified the following key advantages of the methodology:

1. Universality: the ability to identify the author of both assembly and source code based on the program's text and extracted features.
2. Efficiency: the technique consistently achieves an accuracy exceeding 0.85 in all experiments, regardless of task complexity or data specificity. This level of accuracy is sufficient for practical applications.
3. Independence from complicating factors: the methodology remains robust against intentional factors such as obfuscation, coding standards, team collaboration, and

artificial generation, as well as unintentional factors such as stylistic changes due to increased experience and programmer skill.

The primary practical application of the proposed approach is authorship identification in malicious programs. This aspect is planned to be further developed in our future work.

The limitations of the methodology, as well as possibilities for further improvement, are described in the following aspects. Firstly, to achieve high authorship identification accuracy, the program needs to be pre-deobfuscated, as even minimal obfuscation significantly reduces the system's effectiveness. Secondly, the results are directly dependent on the number of authors and the amount of training data available for each author. Increasing the number of authors or lacking a sufficient number of training instances gradually decreases the system's effectiveness. Thirdly, the full capabilities of the system are not yet fully explored. There is a possibility that the specific programming language or compiler used to write the program may also have a negative impact on the system's effectiveness.

In future work, we plan to adapt multi-view learning techniques to textual data and incorporate complementary information from source and binary/disassembled code.

Author Contributions: Supervision, A.R. and A.S.; writing—original draft, A.K. and A.R.; writing—review and editing, A.R. and A.F.; conceptualization, A.K., A.R. and A.F.; methodology, A.R. and A.K.; software, A.K.; validation, A.F. and A.K.; formal analysis, A.R. and A.F.; resources, A.S.; data curation, A.S. and A.R.; project administration, A.R.; funding acquisition, A.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the Ministry of Science and Higher Education of Russia, Government Order for 2023–2025, project no. FEWM-2023-0015 (TUSUR).

Data Availability Statement: Data supporting reported results including links to publicly archived datasets and analysis code. Available online: https://github.com/kurtukova/src_asm_dataset (accessed on 18 June 2023).

Acknowledgments: The authors express their gratitude to the editor and reviewers for their work and valuable comments on the article.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Palmer, G. A Road Map for Digital Forensic Research. Technical Report DTR-T001-01 FINAL, Digital Forensics Research Workshop. Available online: https://dfrws.org/wp-content/uploads/2019/06/2001_USA_a_road_map_for_digital_forensic_research.pdf (accessed on 10 May 2023).
2. Schleimer, S.; Wilkerson, D.S.; Aiken, A. Winnowing: Local Algorithms for Document Fingerprinting. Available online: https://theory.stanford.edu/~aiken/publications/papers/sigmod03.pdf (accessed on 10 May 2023).
3. Abuhamad, M.; AbuHmed, T.; Mohaisen, A.; Nyang, D. Large-Scale and Language-Oblivious Code Authorship Identification. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, Toronto, ON, Canada, 15–19 October 2018; pp. 101–114.
4. Zhen, L.; Chen, G.; Chen, C.; Zou, Y.; Xu, S. RoPGen: Towards Robust Code Authorship Attribution via Automatic Coding Style Transformation. In Proceedings of the 2022 IEEE 44th International Conference on Software Engineering (ICSE), Pittsburgh, PA, USA, 25–27 May 2022; pp. 1906–1918.
5. Holland, C.; Khoshavi, N.; Jaimes, L.G. Code authorship identification via deep graph CNNs. In Proceedings of the 2022 ACM Southeast Conference (ACM SE '22), Virtual, 18–20 April 2022; pp. 144–150.
6. Bogomolov, E.; Kovalenko, V.; Rebryk, Y.; Bacchelli, A.; Bryksin, T. Authorship attribution of source code: A language-agnostic approach and applicability in software engineering. In Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, 23–28 August 2021; pp. 932–944.
7. Ullah, F.; Wang, J.; Jabbar, S.; Al-Turjman, F.; Alazab, M. Source code authorship attribution using hybrid approach of program dependence graph and deep learning model. *IEEE Access* **2019**, *7*, 141987–141999. [CrossRef]
8. Song, Q.; Zhang, Y.; Ouyang, L.; Chen, Y. BinMLM: Binary Authorship Verification with Flow-aware Mixture-of-Shared Language Model. In Proceedings of the 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), Honolulu, HI, USA, 15–18 March 2022; pp. 1023–1033.
9. Rosenblum, N.; Zhu, X.; Miller, B.P. Who Wrote This Code? Identifying the Authors of Program Binaries. In *Computer Security—ESORICS 2011*; Atluri, V., Diaz, C., Eds.; Lecture Notes in Computer Science, 6879; Springer: Berlin/Heidelberg, Germany, 2011.

10. Alrabaee, S.; Wang, L.; Debbabi, M. BinGold: Towards robust binary analysis by extracting the semantics of binary code as semantic flow graphs (SFGs). *Dig. Investig.* **2016**, *18*, 11–22. [CrossRef]
11. A Gentle Introduction to the Fbeta-Measure for Machine Learning. Available online: https://machinelearningmastery.com/fbeta-measure-for-machine-learning/ (accessed on 10 May 2023).
12. Caliskan-Islam, A. When Coding Style Survives Compilation: De-anonymizing Programmers from Executable Binaries. *arXiv* **2017**, arXiv:1512.08546.
13. Alrabaee, S.; Saleem, N.; Preda, S.; Wang, L.; Debbabi, M. OBA2: An Onion Approach to Binary code Authorship Attribution. *Dig. Investig.* **2014**, *11*, 94–103. [CrossRef]
14. Caliskan-Islam, A. Deanonymizing programmers via code stylometry. In Proceedings of the 24th USENIX Security Symposium, Washington, DC, USA, 12–14 August 2015; pp. 255–270.
15. Alrabaee, S.; Shirani, P.; Debbabi, M.; Wang, L. On the Feasibility of Malware Authorship Attribution. *Dig. Investig.* **2016**, *28*, 3–11. [CrossRef]
16. Zia, T.; Ilyas, M.I.J. Source Code Author Attribution Using Author's Programming Style and Code Smells. *Intell. Syst. Appl.* **2017**, *5*, 27–33.
17. Available online: https://doi.org/10.1016/j.eswa.2023.119614 (accessed on 20 June 2023).
18. Available online: https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0245230 (accessed on 20 June 2023).
19. Available online: https://www.sciencedirect.com/science/article/abs/pii/S1566253516302032?via%3Dihub (accessed on 20 June 2023).
20. Meng, X.; Miller, B.P.; Jha, S. Adversarial Binaries for Authorship Identification. *arXiv* **2018**, arXiv:1809.08316.
21. Kurtukova, A.V.; Romanov, A.S. Identification author of source code by machine learning methods. *Tr. SPIIRAN* **2019**, *18*, 741–765. [CrossRef]
22. Kurtukova, A.; Romanov, A.; Shelupanov, A. Source Code Authorship Identification Using Deep Neural Networks. *Symmetry* **2020**, *12*, 2044. [CrossRef]
23. Kurtukova, A.; Romanov, A.; Shelupanov, A.; Fedotova, A. Complex Cases of Source Code Authorship Identification Using a Hybrid Deep Neural Network. *Future Internet* **2022**, *14*, 287. [CrossRef]
24. Romanov, A.S.; Shelupanov, A.A.; Meshcheryakov, R.V. *Development and Research of Mathematical Models, Methods and Software Tools of Information Processes in the Identification of the Author of the Text*; V-Spektr: Tomsk, Russia, 2011; 188p.
25. Code Jam. Available online: https://codingcompetitions.withgoogle.com/codejam (accessed on 10 May 2023).
26. GitHub API. Available online: https://docs.github.com/en/rest?apiVersion=2022-11-28 (accessed on 10 May 2023).
27. GCC, the GNU Compiler Collection. Available online: https://gcc.gnu.org (accessed on 10 May 2023).
28. IDA Pro. Available online: https://hex-rays.com/ida-pro/ (accessed on 10 May 2023).
29. The Friedman Test. Available online: https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.friedmanchisquare.html (accessed on 10 May 2023).
30. Nemenyi Post hoc Test. Available online: https://scikit-posthocs.readthedocs.io/en/stable/generated/scikit_posthocs.posthoc_nemenyi_friedman/ (accessed on 10 May 2023).
31. Tuning the Hyper-Parameters of an Estimator. Available online: https://scikit-learn.org/stable/modules/grid_search.html (accessed on 10 May 2023).
32. Sklearn.svm.SVC. Available online: https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html (accessed on 20 June 2023).
33. Adadelta. Available online: https://pytorch.org/docs/stable/generated/torch.optim.Adadelta.html (accessed on 20 June 2023).
34. List of Options. Available online: https://fasttext.cc/docs/en/options.html (accessed on 10 May 2023).
35. AnalyseC. Available online: https://github.com/ryarnyah/AnalyseC (accessed on 10 May 2023).
36. Linux. Available online: https://github.com/torvalds/linux (accessed on 10 May 2023).
37. Linux Kernel Coding Style. Available online: https://www.kernel.org/doc/html/v4.10/process/coding-style.html (accessed on 10 May 2023).