symmetry

MDPI

Article

Source Code Authorship Identification Using Deep Neural Networks

Anna Kurtukova *, Aleksandr Romanov and Alexander Shelupanov

Faculty of Security, Tomsk State University of Control Systems and Radioelectronics, 634050 Tomsk, Russia; ras@fb.tusur.ru (A.R.); saa@tusur.ru (A.S.)
* Correspondence: kurtukova.a.745@e.tusur.ru

Received: 15 November 2020; Accepted: 8 December 2020; Published: 10 December 2020

Abstract: Many open-source projects are developed by the community and have a common basis. The more source code is open, the more the project is open to contributors. The possibility of accidental or deliberate use of someone else's source code as a closed functionality in another project (even a commercial) is not excluded. This situation could create copyright disputes. Adding a plagiarism check to the project lifecycle during software engineering solves this problem. However, not all code samples for comparing can be found in the public domain. In this case, the methods of identifying the source code author can be useful. Therefore, identifying the source code author is an important problem in software engineering, and it is also a research area in symmetry. This article discusses the problem of identifying the source code author and modern methods of solving this problem. Based on the experience of researchers in the field of natural language processing (NLP), the authors propose their technique based on a hybrid neural network and demonstrate its results both for simple cases of determining the authorship of the code and for those complicated by obfuscation and using of coding standards. The results show that the author's technique successfully solves the essential problems of analogs and can be effective even in cases where there are no obvious signs indicating authorship. The average accuracy obtained for all programming languages was 95% in the simple case and exceeded 80% in the complicated ones.

Keywords: source code; authorship; symmetry; software engineering; machine learning; deanonymization; neural networks

1. Introduction

Identifying the source code author is the task of determining the most likely author of a computer program based on available code samples from the candidate authors. The process of such identification consists of determining the author's style, identifying the programmer's individual habits, professional techniques, and approaches to writing program code. The solutions to this problem can be used as computer forensics tools [1–4].

Modern technologies, software systems, and other digitalization tools have great advantages that simplify a person's professional activity. However, there is a significant drawback behind the many advantages—the vulnerability of such developments to malfunctions and failures. Sources of similar problems are often defects of design or incorrect operation of the software. However, this does not exclude that an attacker deliberately breaks their work.

The most common cause of hardware and software failures is the injections of malicious code. Although the creation, using, and distribution of malicious computer software are prohibited by law, it entails criminal liability in almost all states, and there are still no technical solutions capable of speeding up and improving computer audits aimed at establishing source code authorship. Authorship investigation carried out on malware source codes are held manually by experts.

This approach may be appropriate only in simple cases. However even then, it is overly resource-intensive. Due to the complexity of the relevant expertise and the non-availability of software tools to identify the source code author, the number of crimes is increasing each year, and the level of disclosure, conversely, is decreasing [5]. The development of a forensic examination procedure that includes using such kinds of software can be a solution to this problem and make it possible to effectively identify the author-virus writer even in complex cases.

Global court practice shows numerous cases of copyright infringement and plagiarism of software source codes. An automated system designed to accurately identify the source code author will able commercial organizations to prevent significant losses by using it in the process of resolving legal disputes over intellectual property.

Solutions to the problem of identifying the source code author may also be useful for technical university lecturers. Finding and identifying plagiarism in the work of pupils and students is an important part of the education process. By using appropriate tools, plagiarism will be identified and the evaluation of students' work will be more objective.

The task of identifying the source code author is popular among researchers in the field of natural language processing (NLP). They are interested in both creating new methods and improving existing ones. Positive and negative experiences with existing approaches, as well as the results obtained through their testing, should therefore be taken into account.

Approach [6] is to use back propagation (BP) neural network (NN). This is based on the particle swarm optimization (PSO) method. NN training is carried out on a set of 19 features calculated using PSO. This set contains lexical (keywords, comments, and variable names), structural (code block formatting and loop nesting depth), and syntactic features. This approach is evaluated for the Java language, accuracy is 91%.

Method [7] improves the abstract syntax tree (AST) method. The authors have tried various combinations of it with conventional and bidirectional models with long short-term memory (LSTM/BiLSTM). The essence of the method is the automatic extraction of the functions from the AST representation of the programmers' source codes using the NN. The method obtained an accuracy of 92% for LSTM and 96% for BiLSTM for a set of 25 Python programmers, as well as 80% and 85% for 10 authors writing in C++.

The source code author profiles (SCAP) method [8] uses a statistical metrics that are used to create a "handwriting profile", calculating the deviation from this profile for a specific author. The method has 88% accuracy for the Java programming language. The advantage of this method is independence from programming languages.

Approach [9] is based on the union of structural features highlighted by AST, *n*-gram tokens, statistical metrics, and the SCAP method. Classification is done using the support vector machine (SVM). This approach showed 80% accuracy for 34 JavaScript programmers.

A special feature of the entropy classification method using compression [10] is the implementation of file archivers: Anonymous source codes are joined to samples of codes of known authors, resulting documents are compressed by the file archiver and then the compression rate is checked. The most likely author is the one whose code has the highest compression rate. This method has achieved 100% accuracy when applied for viruses written in Delphi. The method allowed to identify all anonymous samples and achieved an accuracy of 100%. However, this method is not applicable to real-life tasks, as it requires the same programming language, platform, and compiler for all samples.

Approach [11] applies methods of fuzzy AST, stylometric features, and so-called "code smells". The latter are specific features that reflect authors' code defects, that breaks programming paradigms, and are subject to refactoring. Features extracted from source code are used for classification using SVM, J48, naive Bayesian classifier, and *k*-nearest neighbor algorithm (KNN). The accuracy of the approach did not exceed 75%, the best result was obtained by SVM.

Method [12] has been developed by a team of researchers from Drexel University who has been actively working to resolve the problem of identifying the source code author since 2015. The researchers' basic approach is a combination of fuzzy AST and random forest (RF) algorithms, which achieves 90% accuracy for Python programmers. This approach has been extended to include calibration curves in the work [13]. This allows the analysis of incomplete copies of the source code. Modification of the approach allowed obtaining 73% accuracy with one sample in C++. This method was also tested on C++ obfuscated source codes. The results support the hypothesis that obfuscation [12] harms on source code author identification methods—the drop in accuracy was 28%.

Burrows approach [14] is one of the first solutions to the problem of deanonymizing the source code author. The author is determined by counting key substrings in *n*-grams of the source code. The frequencies obtained are lined up in a "ranking" and the author who took the first position is considered to be the true author of the anonymous sample. The accuracy of the method reaches 94%.

The [15] method is based on dynamic and static source code analysis. The authors means static features as keywords, function lengths, number of imports, interpreter version, encoding, etc. The following are considered dynamic: Function calls, code runtime, memory usage, length of disassembled code, etc. Advantages of this method are the possibility of extending the source code set, the use of a less functions, and the independence of the result from the number of authors. The accuracy of the method reaches 94%.

Work [16] focuses on the Deep Learning-based Code Authorship Identification System (DL-CAIS). This system is based on the RF classifier. It scales and makes a decision about the true author based on the deep representations obtained using term frequency-inverse document frequency (TF-IDF) and multi-layer recurrent NN (RNN). The authors claim that the system is independent of the programming language. This is confirmed by the example of four languages: C++, Java, Python, and C. The average identification accuracy was 95%.

Despite the high accuracy achieved by the considered approaches, they have several disadvantages. The main one is the inability to apply the same approach for different programming languages. Approaches were tested only for one or two programming languages. Their effectiveness for other languages has not been proven. Therefore, the approaches cannot be called universal. Besides, it should be taken into account that the cases of identifying the source code author can be divided into simple and complex. In simple cases, identification is not complicated by factors such as obfuscation, coding standards, and incomplete or uncompilable code samples. Any of these cases are considered complex, respectively. The analyzed approaches were used only in simple cases. Their effectiveness in complex cases has not been proven experimentally.

Thus, it was decided to develop a software tool based on a NN that allows solving all mentioned problems and identify source code author, regardless of the language, qualification, code obfuscation, or formatting according to coding standards.

2. The Task of Identifying of the Source Code Author

Let there be a set of authors $\mathbf{A} = \{a_1, \ldots, a_l\}$ and a set of program source codes $\mathbf{S} = \{s_1, \ldots, s_k\}$, where each source code is considered as a feature vector $\mathbf{X} = \{x_1, \ldots, x_n\}$. For the source code subset $\mathbf{S}' = \{s_1, \ldots, s_m\} \subseteq \mathbf{S}$, where $m < k$—the authors are known, there are many "author-source code" pairs $(s_i, a_j) \in \mathbf{D} \subseteq \mathbf{S}' \times \mathbf{A}$, where $s_i \in \mathbf{S}'$, $a_j \in \mathbf{A}$. The solution is to identify the author belonging to set $\mathbf{A}$, who is the true author of anonymous samples of source codes of the subset $\mathbf{S}'' = \mathbf{S}/\mathbf{S}'$.

This problem should be considered as a multi-class classification problem. So, set $\mathbf{A}$ consists of many predefined classes and their labels, set $\mathbf{D}$ includes the source codes for training the classifier, and codes to be classified are included in set $\mathbf{S}''$.

The goal is to develop a model that solves the problem—finding the objective function $\mathbf{F} : \mathbf{S} \times \mathbf{A} \rightarrow [0, 1]$, which assigns some source code from the set $\mathbf{S}$ to its true author. The function value is described as the degree to which the object belongs to a class, where 1 corresponds to the full to the target class, and 0, on the contrary, is a complete mismatch. So, the output is a set of probabilities with length equal to the number of authors.

The decision-making model being developed must classify and describe the process of determining whether the author of the anonymous source code sample belongs to a specific class, based on the training data submitted to model input.

3. Modeling Neural Network Architectures

When analyzing the source codes, the features of the artificial language text as an object of study. In particular semantic and syntactic rules, structural features, and programming language paradigms, must be taken into account.

There are two approaches to training models based on the described aspects. The first of them implies an earlier manual determination of the set of informative features by the researcher. However, this approach has two significant drawbacks: The need for deep expert knowledge of programming languages, as well as the possibility of deliberate distortion of obvious features by the author-programmer. The second approach consists of extracting both obvious and unobvious informative features by model in the process of training and involves the use of deep NN architectures. A comparison of these approaches, as well as traditional and deep machine learning methods, was implemented in the work [2]. The results obtained allow us to conclude that it is appropriate to use deep NN to identify unobvious features that are not consciously controlled by the programmer.

The most common models among researchers are based on recurrent and convolutional architectures. The popularity became due to the peculiarities of their processing of text sequences [17–19].

Convolutional NN (CNN) is known for its effectiveness in solving computer vision problems. They are created in the likeness of the visual cortex of the brain and can focus on certain areas of images and highlight important features in it. This advantage makes them a powerful tool in NLP, where it is necessary to focus on individual text fragments.

In general, the convolutional layer of the CNN architecture can be described as follows:

$$x^l = f(x^{l-1} * k^l + b^l), \tag{1}$$

where x^l—output of layer l, $f()$—activation function; b^l—shift coefficient for layer l; and *—convolution operation of input x with kernel k.

Moreover, due to edge effects, the size of the original matrices is reduced:

$$x_j^l = f(\sum_i x_i{}^{l-1} * k_j{}^l + b_j{}^l), \tag{2}$$

where x_j^l—feature map j (output of layer l), $f()$—activation function; b^l—shift coefficient for layer l for feature map j; and k_j^l—convolution kernel of input x with kernel k, and *—convolution operation of input x with kernel k.

Recurrent NN accepts data sequences as separate, unrelated tokens as well as linked and complete sequences—volumetric texts with a specific meaning. The order in which data are fed to the model input directly influences changes in weight matrices and hidden state vectors during training. By the end of RNN training, information obtained in the previous steps is accumulated in hidden state vectors. In this way, recurrent models allow to record context dependencies from the text.

Based on the experience of researchers in the field of NLP, it was decided to implement the following models: Deep LSTM (D-LSTM) [20], CNN with self-attention (CNN-A) [21], CNN-Gated recurrent unit (C-GRU), CNN-LSTM (C-LSTM), CNN-Bidirectional GRU (C-BiGRU), and CNN-BiLSTM (C-BiLSTM) [22].

Deep LSTM is a modification of RNN. Its special feature is the ability of LSTM to learn long-term dependencies and successfully extract them even from large texts. The architecture of LSTM is often superior to classical RNN in terms of efficiency when dealing with related tasks [23,24], so this study decided to evaluate its effectiveness.

Formally, the states of the LSTM cell at time t can be described as:

$$f_t = \sigma(\mathbf{X_t} * \mathbf{U_f} + H_{t-1} * \mathbf{W_f}), \tag{3}$$

$$\overline{C_t} = tanh(\mathbf{X_t} * \mathbf{U_c} + H_{t-1} * \mathbf{W_c}), \tag{4}$$

$$I_t = \sigma(\mathbf{X_t} * \mathbf{U_i} + H_{t-1} * \mathbf{W_i}), \tag{5}$$

$$O_t = \sigma(\mathbf{X_t} * \mathbf{U_o} + H_{t-1} * \mathbf{W_o}), \tag{6}$$

where f_t—the state of the forget gate layer, $\mathbf{X_t}$—the input vector, H_{t-1}—the hidden state of the previous cells, C_{t-1}—memory state of the previous cell, H_t—hidden state of the current cell, C_t—state memory of the current cell, $\mathbf{W_f}$, $\mathbf{U_f}$—vector of weights for the forget gate layer, $\mathbf{W_c}$, $\mathbf{U_c}$—vector of weights for the gate candidates, $\mathbf{W_i}$, $\mathbf{U_i}$—vector of weights for the input gate, $\mathbf{W_o}$, $\mathbf{U_o}$—vector of weights for the output gate, σ is the sigmoidal function, and *tanh*—tangensoid function.

The key feature that distinguishes LSTM from RNN is the memory state $\overline{C_t}$. This element is determined by the need to delete or, on the contrary, add information and has a significant effect on the value of the forget gate with range [0;1], where 0 is forgetting all information from the previous state, and 1 is remembering all information from the previous state. Based on the current memory state $\overline{C_t}$, the following can be calculated:

$$C_t = f_t * C_{t-1} + I_t * \overline{C}_t, \tag{7}$$

where C_t—the current state of memory at step t.

The output value of the LSTM cell can be formally described as:

$$H_t = O_t * tanh(C_t), \tag{8}$$

Obtained C_t and H_t are passed to the next step and the process is repeated.

Another non-standard modification of the classical model, which became an innovation in the field of NLP was CNN with self-attention. Experiments show that this model can replace RNN in some tasks due to its ability to identify key dependencies regardless of their lengths. This is the main difference from simple CNN models that can only process sequences of fixed lengths according to the dimensions of convolutions [25]. The choice of CNN with attention for implementation is due to its high results when solving problems of sentiment analysis, text generation, and autoreference tasks.

In general, the principle of the self-attention mechanism is as follows. Request vectors (Q) and key-value pairs (K-V) are fed into the self-attention layer input. Each vector is then converted by a linear transformation. The resulting Q values are scalar multiplied with all K in turn. The result goes to the activation function and all vectors V are concatenated with the obtained weights into a single vector.

It should be noted that the self-attention mechanism can be used either independently or in an ensemble of mechanisms, forming a multi-headed approach, where different weights are used to obtain projections. In the second case, they will train in parallel and then concatenate in a single result and, having gone through linear transformations, enter the output.

The value of the q head can be formally represented as:

$$H_q^{(h)} = \sum_k softmax(\mathbf{A}_q^{(h)})_k \mathbf{X}_k \mathbf{W}_{\mathbf{val}}^{(\mathbf{h})}, \tag{9}$$

where A_q—attention value for head q, $\mathbf{X}_k$—input values, $\mathbf{W}_{\mathbf{val}}^{(\mathbf{h})}$—head value matrix, and k—key position.

The attention value for head q is calculated as follows:

$$A_q = \mathbf{W_{qry}} \mathbf{W_{key}^T} \mathbf{X}_k^{\mathbf{T}}, \tag{10}$$

where $\mathbf{W_{qry}}$—query matrix, $\mathbf{W_{key}}$—key matrix, and **T**—transpose operation.

Then the output value of self-attention mechanism with one or more heads:

$$H_q = concat(H_q^{(1)}, \ldots, H_q^{(N_h)})\mathbf{W_{out}} + b_{out}, \tag{11}$$

where N—number of heads and b—offset vector of the output measurement.

As a part of this study, it was decided to consider a self-attention mechanism with a single head.

The idea of creation a hybrid NN (HNN) based on the hypothesis that not only obvious and unobvious informative features (such as the naming of variables, classes, and methods, comments, andwhitespaces) need to be highlighted, but also long-term dependencies. HNN combines the advantages of both recurrent and convolutional architectures. Combinations of classic CNN with the LSTM, GRU, BiLSTM, and BiGRU models were implemented to evaluate the effectiveness of determining the author of the source code.

The GRU architecture was added in the experiments as a simplified version of LSTM. Although this model has fewer filters and the same accuracy as the more complex LSTM. Moreover, due to its imperceptibility, GRU, which is less prone to overfitting, has better generalization capacity for new data, and trains faster.

The last two combinations involve the use of bidirectional architectures. The only difference from simple models is the ability of these architectures to take into account not only the previous context but also the one following a particular token, which can positively affect the identification of the source code author.

4. Estimating the Accuracy of Neural Network Models

The training of any NN architecture and its further evaluation involves the use of representative data. Data were collected from the largest web-service for hosting IT-projects—GitHub [26]. We selected languages included in the rating of the most popular and fast-growing programming languages for the experiment. Source codes were collected from random repositories of GitHub users using a Python script.

The extended source code base of programs' authors included more than 200,000 samples written by 898 authors. The corpus included simple source codes on 13 different programming languages (C, C++, Java, Python, C#, JavaScript, PHP, Perl, Swift, Groovy, Ruby, Go, and Kotlin). Detailed information on the data is presented in Table 1 (languages are presented in descending order of release date).

Table 1. Information about dataset.

Language	Number of Source Codes, Files	Number of Authors	Average Length, Symbols	Average Number of Projects per Author	Average Number of Authors per Project	Average Number of Authors per Source Code
C	17,274	62	1162	18	5.5	3.1
C++	16,368	72	988	25	4.9	3.2
Perl	11,189	61	251	12	5.9	2.2
Python	18,783	57	532	19	7.9	2.1
Java	39,708	73	2409	29	4.7	3.5
JS	18,735	69	397	21	3.6	3.1
PHP	17,158	80	374	19	6.9	2.3
Ruby	19,150	58	304	12	8.5	1.7
C#	19,378	71	638	19	7.0	2.1
Groovy	14,002	68	167	13	3.5	1.9
Go	14,067	81	816	16	3.4	2
Swift	12,672	74	775	10	1.8	1.5
Kotlin	15,274	72	301	13	2.1	1.8

This dataset will allow for extensive research and objective evaluation of the effectiveness of the developed model. It contains a variety of source codes that differ from each other not only in author's writing style but also implicitly take into account the qualifications, experience, age, and education of the developers.

The quality of the models was evaluated with accuracy and 10-fold cross-validation. This procedure avoids the problem of excessive scattering of scores during the verification and provides a reliable result. Accuracy in each of the blocks is defined as:

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN}, \tag{12}$$

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

The *TP*, *TN*, *FP*, and *FN* parameters are calculated on the basis of the confusion matrix, which contains information on how many times the system has made the right decision and how many times it has made the wrong decision on samples of a given class.

The validation results of the models described in the previous section on a corpus of 10 authors are presented in Table 2.

Table 2. Information about experiments results.

	Accuracy ± Std (%)					
Language	**D-LSTM**	**CNN-A**	**CGRU**	**CLSTM**	**CBiLSTM**	**CBiGRU**
C	72 ± 4.25	80 ± 5.06	76 ± 4.19	73 ± 4.5	91 ± 4.18	92 ± 5.07
C++	81 ± 3.37	80 ± 4.19	81 ± 4.53	84 ± 4.03	65 ± 4.43	86 ± 4.52
Perl	70 ± 3.29	80 ± 4.13	81 ± 4.45	73 ± 3.95	79 ± 4.35	84 ± 4.44
Python	75 ± 4.13	84 ± 5.18	82 ± 4.18	83 ± 4.49	92 ± 4.04	91 ± 4.85
Java	73 ± 4.12	82 ± 4.31	76 ± 3.8	74 ± 4.08	90 ± 3.68	91 ± 4.4
JS	66 ± 5.07	77 ± 4.12	73 ± 3.76	76 ± 4.04	83 ± 3.64	87 ± 4.36
PHP	72 ± 4.19	81 ± 5.13	80 ± 4.19	82 ± 4.5	82 ± 4.05	86 ± 4.85
Ruby	68 ± 3.1	79 ± 4.16	73 ± 3.26	83 ± 3.5	92 ± 3.11	90 ± 3.78
C#	66 ± 3.14	90 ± 5.11	85 ± 3.7	86 ± 3.98	95 ± 3.58	93 ± 4.29
Groovy	85 ± 5.11	93 ± 5.27	93 ± 4.67	78 ± 5.01	95 ± 4.51	93 ± 5.4
Go	82 ± 5.15	84 ± 3.25	84 ± 3.78	81 ± 4.06	85 ± 3.65	84 ± 4.37
Swift	73 ± 4.21	78 ± 4.11	80 ± 3.74	72 ± 4.02	62 ± 3.61	85 ± 4.33
Kotlin	76 ± 4.23	83 ± 3.22	82 ± 3.35	72 ± 3.6	78 ± 3.24	85 ± 3.88
Average accuracy	74 ± 4.04	82 ± 4.4	80 ± 3.97	78 ± 4.14	84 ± 3.85	88 ± 4.5

The least efficient architecture was D-LSTM, which showed an average accuracy for all programming languages—74%. Because this is the only architecture that sequentially processes text and is not able to select local and informative features built from *n*-grams of characters.

High accuracy was achieved by combinations of CNN with BiGRU—88%, CNN with BiLSTM—84%, and CNN with attention—82%. All these models are distinguished by the ability to identify the context of a symbol and to assess its importance relative to the contexts of other symbols. Thus, these architectures perform key functions for identifying the author of the source code: Identifying the area around local feature of the context and determining short-term and long-term dependencies.

The obtained results allow us to conclude that CNN in combination with BiGRU is particularly effective, and showed an accuracy of more than 80% for all languages, had a significant advantage over other implemented architectures. This combination is the basis of the author's hybrid NN, which is detailed described in the next section.

5. Author's Technique of Identifying the Author of the Source Code

The technique for identifying the author of the source code includes all steps necessary for the model to make a decision about whether an anonymous sample of the source code belongs to one of the training classes. Since the basic decision-making algorithm is a deep NN, capable of highlighting informative features from the non-preprocessed text in vector form by itself, the technique includes only five steps:

1. Conversion of training and test datasets into vector form.
2. Training a deep model on a vectorized training dataset.
3. Validation of a deep model on a vectorized test dataset.
4. Converting an anonymous source code sample into a vector form.
5. Authorship identification of a vectorized anonymous source code sample by the model that showed the best accuracy on the validation result.

The author's technique involves using deep NN. Therefore, it was necessary to take into account that such a model can accepts input data only in vector form. Thus, it became necessary to find the method for data vectorization. It was decided to use one-hot encoding method. The choice is since of used data, categorical features are nominal, not ordinal, and, therefore, require the encoder to "binarize" values. Encoding occurs by creating for each character a vector of 255 zeros and single 1 at the position of the element with an index equal to the character code (in accordance with the ASCII encoding). Thus, the training and test datasets, as well as the anonymous source code, are transformed into a vector form, character by character.

The vectorized training set is fed to the input of the deep NN. Results of experiment were presented in the previous section. Obtained results allow us to conclude that the hybrid CNN and BiGRU (HNN) are most effective for solving the problem of identifying the source code author. However, before this combination was included in the technique, it was improved with additional layers. The final model is shown in Figure 1.

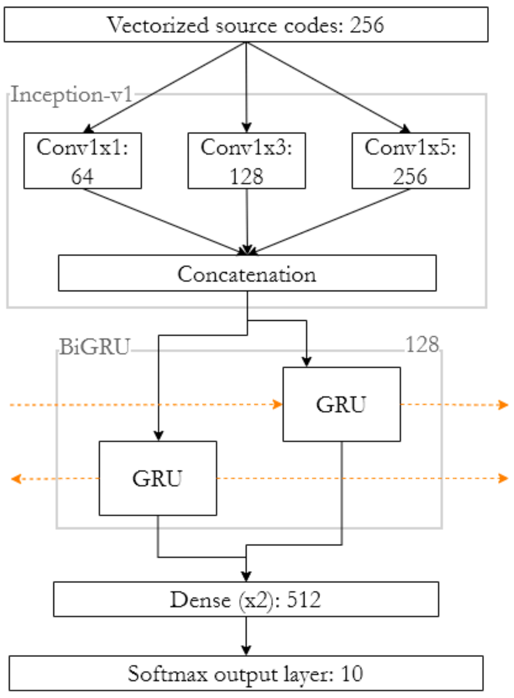

Figure 1. Generalized model of the hybrid neural network. To the right of the layer name, its dimension is indicated (for the case with 10 classes).

These are convolutional layers modeled as components of the Inception-1 architecture [27]. The decision is due to the need to analyze not only each individual symbol of the source code but also *n*-grams of symbols. Convolutions of dimensions 1×1, 1×3, and 1×5 allow parallel analysis of unigrams, trigrams, and 5-g, respectively. The model, obtained by combining Inception-v1 and BiGRU, will be able to perform deep analysis of the source code, take into account local and global features, the context, and various time dependencies.

This is not enough to find the best architecture to obtain high accuracy of decision making with HNN. Moreover, necessary to find effective regularization, activation, loss functions, and function for error minimization.

Optimization of the bias of synaptic weights and local parameters makes it possible to minimize the error obtained in the process of learning the NN. Many popular optimizers have a problem with paralysis or the greedy behavior of the algorithm. To avoid it, it was decided to use the adaptive step optimization—Adadelta [28].

First experiments with default settings used in [1] for the Adadelta optimizer (in the PyTorch framework) do not allow achieving high accuracy in complex cases. The following experimentally founded parameters allow identifying the source code author equally well in all cases (up to 15% increase for complex cases):

- lr = 10^{-4} (coefficient that scale delta before it is applied to the parameters);
- rho = 0.95 (coefficient used for computing a running average of squared gradients);
- eps = 10^{-7} (term added to the denominator to improve numerical stability).

To prevent possible overfitting of the model, it is necessary to carry out regularization. In this case, out 20% of the vector elements are zeroed and the model does not face the problem of overfitting. As in the previous work [1], Softmax [29], a generalization of the logistic function for the multidimensional case, was used as a function of the HNN output layer. Cross-entropy was used as a loss function, respectively.

Mini-batches of size 8–16 examples were used in the process of HNN training. This approach allows reducing the variance in gradients of individual examples, because they can be very different from each other [16]. The mini-batch calculates the average gradient over a number of examples at one time instead of calculating the gradient for one example.

The effectiveness of the presented author's technique was evaluated using examples of simple (see Section 6) and complex (see Sections 7 and 8) cases of identifying the author of the source code.

6. Identifying the Author of the Source Code in Simple Cases

Simple cases of author identification do not involve such factors as obfuscation and code formatting by coding standards. For such cases, it was decided to conduct some experiments aimed at determining the capabilities of the author's technique for identifying the true author based on HNN for further recommendations on its application.

The first step was the evaluation of the technique on corpus of various authors by the number of authors. For the experiment, were used subsets of 5, 10, and 20 authors. Selection of the subset of authors was made at random. Authors whose number and length of source codes were approximately equal were selected in one training subset. The results are presented in the Table 3.

Table 3. Experimental results for corpus of 5–20 authors.

	Accuracy ± Std (%)		
Language	**5 Authors**	**10 Authors**	**20 Authors**
C	96 ± 2.92	95 ± 2.76	94 ± 3.91
C++	92 ± 4.16	92 ± 3.84	90 ± 3.95
Perl	96 ± 3.12	91 ± 4.14	87 ± 3.98
Python	95 ± 4.16	92 ± 4.75	92 ± 4.75
Java	97 ± 2.76	93 ± 3.78	89 ± 3.61
JS	92 ± 2.78	84 ± 2.35	76 ± 3.88
PHP	92 ± 2.99	89 ± 2.76	86 ± 2.89
Ruby	95 ± 4.16	85 ± 3.84	77 ± 3.8
C#	96 ± 3.9	88 ± 4.44	83 ± 4.12
Groovy	99 ± 0.9	96 ± 3.44	93 ± 3.91
Go	93 ± 2.53	86 ± 2.88	83 ± 3.54
Swift	98 ± 1.08	94 ± 4.08	89 ± 3.74
Kotlin	91 ± 3.35	85 ± 3.47	81 ± 4.55
Average accuracy	95 ± 3.54	90 ± 3.57	86 ± 3.89

Based on the analysis of the approaches to identifying the source code author presented in Section 2, we can conclude that the proposed technique demonstrates the best results for all programming languages. Moreover, it turned out to be highly effective for languages not considered in the works of other researchers—C, C #, Ruby, PHP, Swift, Kotlin, Go, Groovy, and Perl. The peculiarities of the language in which the programmer writes affect performance, but in general the technique can be considered universal.

For the most popular Java programming language, the experiment has been extended to include scores of 2, 30, 40, and 50 authors. Results are shown in the Figure 2.

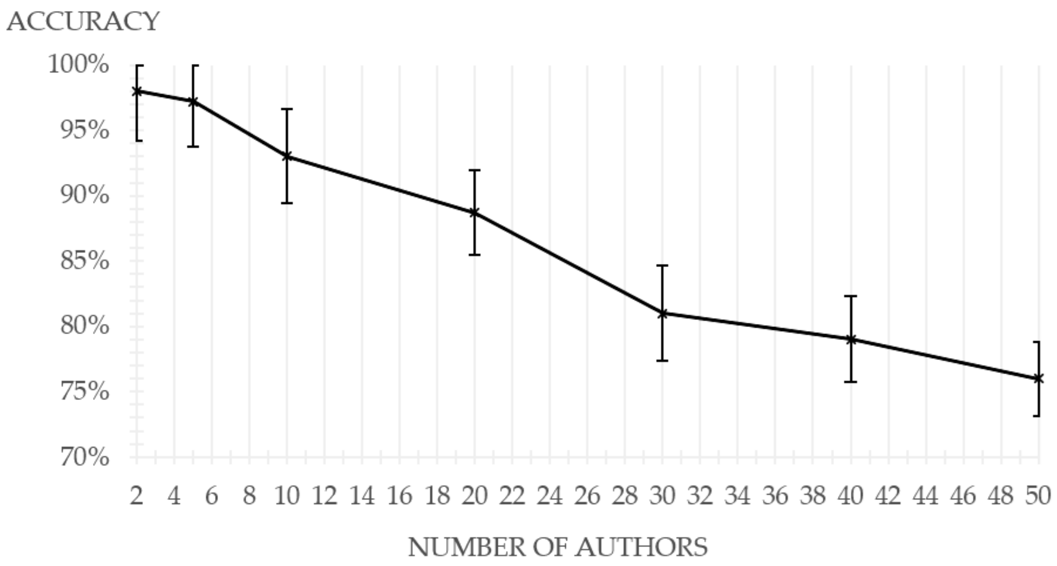

Figure 2. Graph of the dependence of accuracy on the number of authors participating in the experiment.

The results of the experiments aimed at evaluating the dependence of accuracy on the size of the corpus show a gradual decrease in accuracy as the number of target classes increases. Increasing the number of authors leads to the problem of multi-class classification in high-dimensional and, as a result, to the problem of data imbalance.

To evaluate the influence of the developer qualifications on the effectiveness of the author's technique, a small experiment was carried out on the corpus of the source codes written by technical students.

The corpus included works of 75 students of two to three courses. The programs are written by them to implement solutions of the same type of problems in the C++. This fact excludes an increase in the generalization ability due to the specificity of the functional purpose of the programs. The subsets were limited to 15 source codes of no more than 500 lines. The obtained results are presented on a histogram—Figure 3.

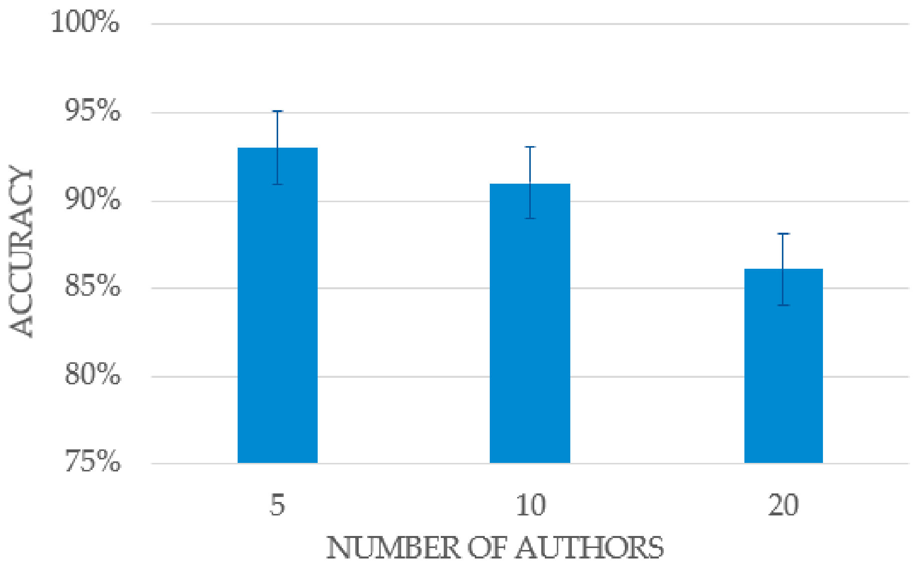

Figure 3. Accuracy histogram for student authorship identification task.

The presented results indicate that the proposed technique is effective for identifying the authorship of the code not only for professional programmers but also for beginners—the qualification does not affect the classification accuracy. This suggests that the application of the technique can be implemented the educational process for evaluating programming works.

7. Evaluation of the Technique Based on HNN on Obfuscated Data

Obfuscation of the source code [30–33] is a process aimed at modifying source code for hide its algorithm and semantic meaning and make it harder to understand by human.

The most common area of obfuscation application is software code protection from plagiarism, which is increasingly common in the field of commercial software development. Programmers obfuscate code with special tools. However, possibility for using such tools for to hiding unique techniques, habits, and style of coding, and, as a result, to anonymize the author of the program cannot be excluded. Virus writers often use obfuscation for this very purpose. In such situations, manual expert analysis of the source code becomes impossible and it is necessary to use technical analysis tools of identifying the source code author that is stable to the obfuscation.

Despite the large number of studies devoted to the problem of determining the author of the source code, the topic of obfuscation is hardly covered in them. The only work that takes into account the effect of obfuscation on the identification of the source code author is [12]. As mentioned in Section 2, the authors examined the influence of the Tiggers obfuscator [34] on the identification of the source code author. The loss in accuracy has reached almost 30%. This is because the features, on the basis of which the model makes a decision, lose their informative value due to renaming, encryption, and other code transformations used to hide the author's identity.

So, it was decided to evaluate the impact of obfuscation on the process of identifying the author of the program code using the HNN-based technique. Table 4 provides information about the obfuscators that were used to modify datasets.

Table 4. Information about obfuscators.

Language		Obfuscation Tool [35–42]
Interpreted	JavaScript	JS Obfuscator Tool [35] JS-obfuscator [36]
	Python	PyArmor [37] Opy [38]
	PHP	Yakpro-po [39] PHP Obfuscator [40]
Compiled	C++ C	Cpp Guard [41] Analyse C [42]

The JavaScript (JS) Obfuscator Tool was used to obfuscate source codes in the JS language, performing lexical transformations, replacing variable and function names, deleting comments and whitespaces, and converting lines into hexadecimal sequences and encoding them further to base64. The second tool was JS-obfuscator, which, in addition to transformations similar to the first one, obfuscates embedded HTML, PHP, and other code.

Pyarmor was used for the Python language, which obfuscates the code of functions during their work. Its principle of operation is to obfuscate the bytecode of each object and clean up local variables immediately after the end of the function. An alternative tool was Opy, which produces simple lexical obfuscation: Changing the names of variables and functions on a sequence of "1" and "l" characters and converting strings into random character sets.

For PHP two lexical obfuscators, Yakpropo and PHP Obfuscator were used. Both tools aim to remove whitespaces and comments. In addition, the tools allow scrambling the names of variables, functions, classes, etc., as well as replacing conditional statements with the "if... go" construction.

The obfuscators for C and C++ compiled languages were also considered. Cpp Guard removes single-line and multi-line comments, adds pseudo-complex trash code that does not affect the computational complexity of the entire program, removes spaces and line breaks, processes all preprocessor directives using declarations, and single-line "else" conditions where brackets are missing and substitutes lines for their hexadecimal representation. AnalyseC replaces variable names with asequence of random characters.

Table 5 presents the results of experiments with interpreted programming languages. The accuracy of HNN model identification, both with and without obfuscation, is given here.

Table 5. Results with interpretable languages (obfuscation case).

Language	Obfuscation	5 Authors	10 Authors	20 Authors
JavaScript	Without obfuscation	92 ± 2.78	84 ± 2.35	76 ± 3.88
	JS Obfuscator Tool	86 ± 3.52	80 ± 4.16	70 ± 3.84
	JS-obfuscator	86 ± 4.08	70 ± 4.08	63 ± 3.77
Python	Without obfuscation	95 ± 4.16	92 ± 4.75	92 ± 4.75
	Opy	87 ± 2.57	80 ± 3.99	48 ± 2.98
	Pyarmor	70 ± 1.76	54 ± 4.1	38 ± 3.24
PHP	Without obfuscation	92 ± 2.99	89 ± 2.76	86 ± 2.89
	Yakpro-po	89 ± 5.18	76 ± 3.87	63 ± 4.76
	PHP Obfuscator	82 ± 4.42	74 ± 5.01	61 ± 3.98

It can be noted that simple lexical obfuscation has a minor effect on identification, while more complex code modifications (as in the case of PyArmor) reduce accuracy by an average of 30%.

Table 6 presents the results of experiments with compiled programming languages (C, C++). It also shows the identification accuracy of the HNN model both with and without obfuscation.

Table 6. Results with compiled languages (obfuscation case).

Language	Obfuscation	5 Authors	10 Authors	20 Authors
C++	Without obfuscation	92 ± 4.16	92 ± 3.84	90 ± 3.95
	C++ Obfuscator	71 ± 4.11	67 ± 4.09	41 ± 4.32
C	Without obfuscation	96 ± 2.92	95 ± 2.76	94 ± 3.91
	C Obfuscator	90 ± 3.54	81 ± 3.31	76 ± 4.2

Similar to the PyArmor case, the C++ obfuscator significantly reduces identification efficiency, while lexical conversions that do not work at the bytecode level are minor.

The results of the experiments allow us to conclude that the developed HNN model is resistant to lexical obfuscation: Removing whitespace characters, transforming strings, converting and encoding them, etc. The accuracy of identifying the author of a lexically obfuscated source code is on average 7% lower than that of the original one, which is acceptable for this type of task.

In the case of more complex obfuscation performed by tools such as PyArmor and C++ Obfuscator, with bytecode of objects, the addition of pseudo-complex code, etc., the difference in the accuracy of identification reaches 30%.

8. Evaluation of the Technique Based on HNN on Source Codes, Designed in Accordance with Coding Standards

A programmer's compliance with coding standards is also a factor that makes it difficult to identify the author.

Source code quality is one of the keynote aspects of commercial software development. The importance of this aspect is supported by studies [43,44] aimed at determining the time spent by the ordinary developer. They confirm that the ratio of time spent reading the code to the time spent writing is on average 10:1. This value, the developer's time, can be reduced by improving the quality of the software code developed in the company. The main way to achieve this goal is through a coding standard. This is a set of rules and conventions that describe how to design source code and is shared by the development team. This improves the readability of the code and reduces the load on the memory and vision of the developer.

The coding standard for software development can imply both the use of basic rules and restrictions and detailed recommendations for the design of interfaces, implementation of methods and classes, formatting conventions, naming of variables, etc. An illustrative example of the latter is the implementation of the Linux kernel [45]. This project is being jointly implemented by a large number of developers.

Coding standards are the subject of research [46,47]. However, their authors do not pay enough attention to how standards influence the author's style, in particular, on his professional techniques and habits, which include many unique informative features of a programmer. An evaluation of this impact is essential for the task of identifying the source code author. The evaluation will make it possible to make conclusions about the applicability of identification algorithms to software codes written according to internally defined rules and corporate agreements.

Data from the Linux Kernel project developers' repositories were collected to conduct experiments to identify the source code author written following coding standards. The total number of source codes in the dataset was over 250,000 written in C, C++, and Assembler programming languages by the five most active developers. The data obtained from Github was transformed into training sets of different sizes 10, 20, and 30 files with lengths from 1000 to 10,000 characters. The histogram in Figure 4 shows a dependence on the accuracy of the training corpus volume in cases when programmers follow the coding standards and when they do not.

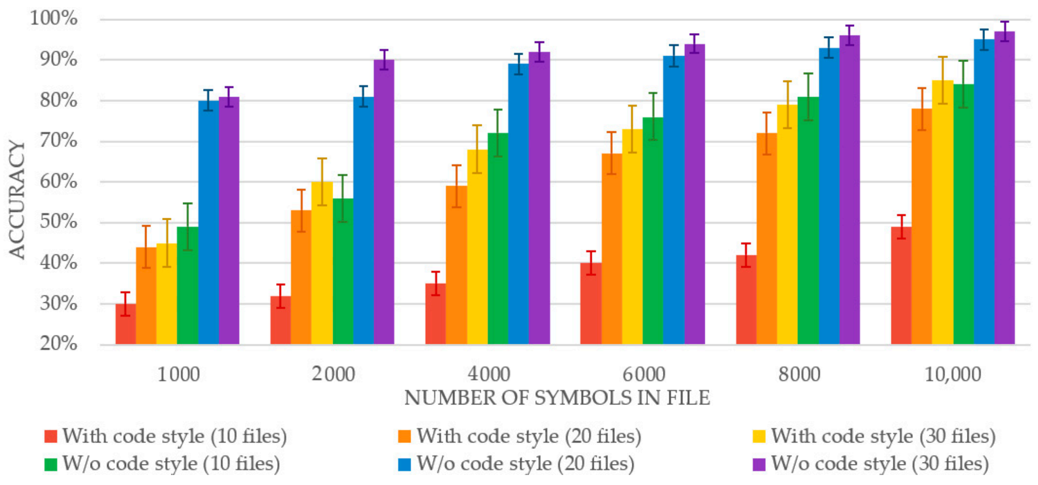

Figure 4. Graph of the dependence of the identification accuracy on the volume of the training corpus (coding standards case).

This graph demonstrates a direct dependence of accuracy on the volume of the training set—an increase in the number of files in the set and/or the maximum length of the samples entails an increase in the identification accuracy.

The results of the experiments demonstrate the negative impact of source code samples written in accordance with coding standards on the author's identification process. This is due to the unification of the code design among developers in one company. Its consequence is the disappearance of such informative features as:

- author's levelling of logical blocks of the program, including the use of a certain type of spaces;
- style of naming variables, classes, methods, etc., e.g., CamelCase, snake_case;
- formatting comments on source lines;
- "code smells" [11].

Thus, the aim of HNN is to the search for unobvious informative features that allow find difference between the authors-programmers. Based on the obtained graph (Figure 4), it is obvious that such searching can be effective only in the case of a sufficient amount of training data and requires more computing resources. If these conditions are met, the author's technique can be used to identify the author of the source code in all mentioned cases.

9. Conclusions and Future Works

As part of a study, the problem of identifying the source code author was defined, and approaches to its solution were analyzed. The HNN-based technique developed by the authors solves a number of the previously identified problems, namely: Dependence on the programming language, instability to obfuscation, and inefficiency to source codes written according to coding standards.

The technique shows an average 95% accuracy in the simple case of identifying the source code author. Other researchers, engaged in the same problem also tested their methods on source codes from github.com, selecting data at random. However, there are no publicly available training corpora. A direct comparison of results is incorrect since our corpus most likely does not intersect with their ones. Moreover, there are no program implementations of their methods in the public domain, so it is impossible to evaluate them on our corpus. However, judged only on published results our ones exceed competitors. In addition, the developed technique allows achieving high accuracy in cases of obfuscation and using of coding standards, which are more difficult tasks. Competitors did not conduct such research. The author's technique demonstrated resistance stability to lexical obfuscation of source codes—the loss in accuracy for all obfuscators did not exceed 10%. The programmer's commitment to certain coding standards and source code formatting reduced the identification accuracy. However, an increase in the volume of training data is improved accuracy from 30% to 85%, which is an excellent result for such a complex case.

It should be taken into account that the technique does not require any prior data preparation: The HNN model is able to find new patterns and dependencies in the data that are unobvious to the human, as well as indirectly taking into account the intellectual content and implementation of the program code.

The developed technique is universal, i.e., independent of the programming language preferred by the author—the distinctive features of programming languages have not had a critical impact on the accuracy of the classification. The difference in accuracy for various programming languages is caused by several factors. The history of language development and the original philosophy [48] have a great influence on the program structure and semantics of the language. Long-standing programming languages (C, C++, Perl, Python, Java, PHP, and C#) have already passed a phase of active development and contain built-in tools for solving specific tasks. Recently appeared languages (Groovy, Go, Swift, and Kotlin) were created for fast and efficient solving of absolutely new classes of tasks. Now they are still in the active development phase. New modules are released frequently and immediately started to be actively used by the community. Authors often have to invent new solutions, relying only on their own skills. Due to these findings, the program code contains more author's distinctive features. Furthermore, the difference in accuracy can be explained by the creativity thinking in a process of code writing. For example, whitespaces can be placed arbitrarily when using some languages, and in others, they indicate the boundaries of logical blocks. The paradigm of a specific language also limits the author's style. For example, using associative arrays in a programming language makes the text search easier, because the search algorithm is encapsulated in the array implementation. If there is no such data type, so the programmer has to implement it by himself. These facts can partly explain the low accuracy for the JS and Ruby languages. Thus, for different programming languages, the number of informative features extracted by the NN will be different. Accuracy depends on the number of such features.

There is no problem in the performance degradation on a large number of authors. High generalization ability is not required when solving real-life tasks. The number of possible authors can be reduced with source code filtration on functionality, used frameworks and libraries, etc. After that, the author identification procedure is performed. The authors from the obtained set are used as target classes. This approach can improve the accuracy of author identification without necessity of additional training data.

Further research is planned. In particular, it is planned to solve the complicated task of identifying the source code author in the case when multiple programmers working on the same file or project. Moreover, the task needs to be generalized. It is necessary to consider the case when it is unclear whether the code is written by one programmer or more. We are already working on this problem and are developing a deep-learning based approach for one-class classification.

Author Contributions: Supervision, A.S.; writing—original draft, A.K. and A.R.; writing—review and editing, A.R.; Conceptualization, A.K. and A.R.; methodology, A.K.; software, A.K.; validation, A.K. and A.R.; formal analysis, A.K.; resources, A.S.; data curation, A.K. 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 2020–2022, project no. FEWM-2020-0037 (TUSUR).

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. Kurtukova, A.; Romanov, A.; Fedotova, A. De-Anonymization of the Author of the Source Code Using Machine Learning Algorithms. In Proceedings of the 2019 International Multi-Conference on Engineering, Computer and Information Sciences (SIBIRCON), Yekaterinburg, Russia, 25–27 October 2019.
2. Kurtukova, A.V.; Romanov, A.S. Identification author of source code by machine learning methods. *Trudy SPIIRAN* **2019**, *18*, 741–765. [CrossRef]
3. Rakhmanenko, I.A.; Shelupanov, A.A.; Kostyuchenko, E.Y. Automatic text-independent speaker verification using convolutional deep belief network. *Comput. Opt.* **2020**, *44*, 596–605. [CrossRef]
4. Kostyuchenko, E.Y.; Viktorovich, I.; Renko, B.; Shelupanov, A.A. User Identification by the Free-Text Keystroke Dynamics. In Proceedings of the 3rd Russian-Pacific Conference on Computer Technology and Applications (RPC), Vladivostok, Russia, 18–25 August 2018; pp. 1–4.
5. Nikerov, D.M.; Khokhlova, O.M. Crimes in the field of high technologies in modern Russia. *Bull. East-Sib. Inst. MIA Russ.* **2019**, *2*, 82–93.
6. Yang, X.; Li, Q.; Guo, Y.; Zhang, M. Authorship attribution of source code by using backpropagation neural network based on particle swarm optimization. *PLoS ONE* **2017**, *12*, e0187204.
7. Alsulami, B.; Dauber, E.; Harang, R.; Mancoridis, S.; Greenstadt, R. Source Code Authorship Attribution using Long Short-Term Memory Based Networks. In Proceedings of the 22nd European Symposium on Research in Computer Security 2017, Oslo, Norway, 11–15 September 2017; pp. 65–82.
8. Frantzeskou, G.; Stamatatos, E.; Gritzalis, S. Identifying authorship by byte-level n-grams: The source code author profile (SCAP) method. *Int. J. Digit. Evid.* **2007**, *1*, 1–18.
9. Wisse, W.; Veenman, C.J. Scripting DNA: Identifying the JavaScript Programmer. *Digit. Investig.* **2015**, *15*, 61–71. [CrossRef]
10. Osovetskiy, L.G.; Stremoukhov, V.D. Determining the authorship of malicious code using the data compression method. *Softw. Prod. Syst.* **2013**, *3*, 167–169.
11. Zia, T.; Ilyas, M.I.J. Source Code Author Attribution Using Author's Programming Style and Code Smells. *Intel. Syst. Appl.* **2017**, *5*, 27–33.
12. Caliskan-Islam, A.; Harang, R.; Liu, A. Deanonymizing programmers via code stylometry. In Proceedings of the 24th USENIX Security Symposium 2015, Washington, DC, USA, 12–14 August 2015; pp. 255–270.
13. Caliskan-Islam, A.; Dauber, E.; Harang, R. Git blame who? *arXiv* **2017**, arXiv:1701.05681.
14. Burrows, S.; Uitdenbogerd, A.; Turpin, A. Application of information retrieval techniques for source code authorship attribution. In Proceedings of the 14th International Conference on Database Systems for Advanced Applications 2009, Brisbane, Australia, 21–23 April 2009; pp. 699–713.
15. Wang, N.; Ji, S. Integration of Static and Dynamic Code Stylometry Analysis for Programmer De-anonymization. In Proceedings of the 11th ACM Workshop on Artificial Intelligence and Security 2018, Toronto, ON, Canada, 19 October 2018; pp. 74–84.
16. 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.
17. Kim, Y. Convolutional Neural Networks for Sentence Classification. *arXiv* **2014**, arXiv:1408.5882.
18. Zhang, X.; Zhao, J.; LeCun, Y. Character-level Convolutional Networks for Text Classification. *arXiv* **2016**, arXiv:1509.01626.
19. Jin, Y.; Wu, D.; Guo, W. Attention-Based LSTM with Filter Mechanism for Entity Relation Classification. *Symmetry* **2020**, *12*, 1729. [CrossRef]
20. Nowak, J.; Taspinar, A.; Scherer, R. LSTM Recurrent Neural Networks for Short Text and Sentiment Classification. In Proceedings of the International Conference on Artificial Intelligence and Soft Computing 2017, Zakopane, Poland, 11–15 June 2017; pp. 553–562.
21. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.; Kaiser, L.; Polosukhin, I. Attention Is All You Need. *arXiv* **2017**, arXiv:1706.03762.
22. Lai, S.; Xu, L.; Liu, K.; Zhao, J. Recurrent Convolutional Neural Networks for Text Classification. In Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence 2015 (AAAI'15), Austin, TX, USA, 25–30 January 2015; pp. 2267–2273.

23. Apaydin, H.; Feizi, H.; Sattari, M.T.; Colak, M.S.; Shamshirband, S.; Chau, K.-W. Comparative Analysis of Recurrent Neural Network Architectures for Reservoir Inflow Forecasting. *Water* **2020**, *12*, 1500. [CrossRef]
24. Mangal, S.; Joshi, P.; Modak, R. LSTM vs. GRU vs. Bidirectional RNN for script generation. *arXiv* **2020**, arXiv:1908.04332.
25. Xue, X.; Feng, J.; Gao, Y.; Liu, M.; Zhang, W.; Sun, X.; Zhao, A.; Guo, S. Convolutional Recurrent Neural Networks with a Self-Attention Mechanism for Personnel Performance Prediction. *Entropy* **2019**, *21*, 1227. [CrossRef]
26. Github. Available online: https://github.com/ (accessed on 9 November 2020).
27. Szegedy, C.; Liu, W.; Jia, Y. Going Deeper with Convolutions. *arXiv* **2014**, arXiv:1409.4842.
28. Zeiler, M.D. Adadelta: An adaptive learning rate. *arXiv* **2012**, arXiv:1212.5701.
29. Nwankpa, C.; Ijomah, W.; Gachagan, A.; Marshall, S. Activation Functions: Comparison of trends in Practice and Research for Deep Learning. *arXiv* **2018**, arXiv:1811.03378.
30. Popa, M. Techniques of Program Code Obfuscation for Secure Software. *J. Mob. Embed. Distrib. Syst.* **2011**, *3*, 205–219.
31. Buintsev, D.N.; Shelupanov, A.A.; Shevtsova, O.O. Analysis of the use of obfuscating transformations for software. *Inform. Secur. Is.* **2015**, *3*, 38–43.
32. Ceccato, M.; Di Penta, M.; Nagra, J.; Falcarin, P.; Ricca, F.; Torchiano, M.; Tonella, P. The Effectiveness of Source Code Obfuscation: An Experimental Assessment. In Proceedings of the IEEE 17th International Conference on Program Comprehension 2009, Vancouver, BC, Canada, 17–19 May 2009; pp. 178–187.
33. Anckaert, B.; Madou, M.; Sutter, B.; Bus, B.; Bosschere, K.; Preneel, B. Program Obfuscation: A Quantitative Approach. In Proceedings of the 2007 ACM Workshop on Quality of Protection (QoP 2007), Alexandria, VA, USA, 29 October 2007; pp. 15–20.
34. The Tigress Diversifying c Virtualizer. Available online: http://tigress.cs.arizona.edu (accessed on 9 November 2020).
35. JS Obfuscator Tool. Available online: https://obfus-cator.io/ (accessed on 9 November 2020).
36. JS-Obfuscator. Available online: https://github.com/cai-guanhao/js-obfuscator (accessed on 9 November 2020).
37. Pyarmor. Available online: https://github.com/da-shingsoft/pyarmor (accessed on 9 November 2020).
38. Opy. Available online: https://github.com/QQuick/Opy (accessed on 9 November 2020).
39. Yakpro-po. Available online: https://github.com/pkfr/-yakpro-po (accessed on 9 November 2020).
40. PHP Obfuscator. Available online: https://github.com/-naneau/php-obfuscator (accessed on 9 November 2020).
41. Cpp Guard. Available online: https://github.com/te-chtocore/Cpp-Guard (accessed on 9 November 2020).
42. AnalyseC. Available online: https://github.com/ryarn-yah/AnalyseC (accessed on 9 November 2020).
43. Martin, R.C. *Clean Code: A Handbook of Agile Software Craftsmanship*; Prentice Hall: Upper Saddle River, NJ, USA, 2009; 448p.
44. Wang, Y.; Zheng, B.; Huang, H. Complying with Coding Standards or Retaining Programming Style: A Quality Outlook at Source Code Level. *JSEA* **2008**, *1*, 88–91. [CrossRef]
45. Linux Kernel. Available online: https://github.com/torvalds/linux (accessed on 9 November 2020).
46. Li, X.; Prasad, C. Effectively teaching coding standards in programming. In Proceedings of the 6th Conference on Information Technology Education—SIGITE 2005, Newark, NJ, USA, 20–22 October 2005; pp. 239–244.
47. Gorshkov, S.; Nered, S.; Ilyushin, E.; Namiot, D. Using Machine Learning Methods to Establish Program Authorship. *Int. J. Open Inf. Technol.* **2019**, *7*, 2307–8162.
48. Fourment, M.; Gillings, M.R. A comparison of common programming languages used in bioinformatics. *BMC Bioinf.* **2008**, *9*, 82. [CrossRef] [PubMed]