ІНТЕЛЕКТУАЛЬНІ ІНФОРМАЦІЙНІ ТЕХНОЛОГІ
INTELLECTUAL INFORMATION TECHNOLOGIES

https://doi.org/10.15407/intechsys.2025.02.081
UDC 004.91

A.H. ADAMCHUK, Student,
National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute",
37, Beresteyskyi ave., Kyiv, 03056, Ukraine
ann.adamchuk2002@gmail.com

V.I. SUSHCHUK-SLUSARENKO, Senior Lecturer,
National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute",
37, Beresteyskyi ave., Kyiv, 03056, Ukraine
https://orcid.org/0000-0002-6096-3832
Sushchuk.Viktoriia@lll.kpi.ua

A.I. DYCHKA, PhD (Engineering), Assistant,
National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute",
37, Beresteyskyi ave., Kyiv, 03056, Ukraine
https://orcid.org/0000-0003-0578-2788
andriydychka@gmail.com

AUTOMATED AUTHORSHIP IDENTIFICATION OF PROGRAM CODE BASED ON A METRIC SYSTEM

The paper reviews existing methods for automated program code authorship attribution and then proposes an original method based on a system of metrics. The proposed method uses a metric system grounded in the "fingerprinting" technique. The metrics reflect the individual stylistic features of a programmer, regardless of the programming language.

Keywords: *metrics, attribution, source code, authorship identification, information system..*

Terminology

Metric — a numerical indicator used to measure and analyze the characteristics of a specific object, system, or process.

Author Profile — a set of characteristics (metrics) that define an individual's coding style.

Cite: Adamchuk A.H., Sushchuk-Slusarenko V.I., Dychka A.I. Automated Authorship Identification of Program Code Based on a Metric System. *Information Technologies and Systems*, Київ, 2025, Том 2 (2), 81–89. https://doi.org/10.15407/intechsys.2025.02.081

Cyclomatic Complexity — a software metric that evaluates the number of independent paths through a program.

Introduction and Problem Statement

Determining the authorship of program code is an extremely important aspect of detecting unlawful activities in various domains. In particular, it applies to identifying the authorship of malicious code that has infiltrated an information system through unauthorized access, as well as detecting information theft using specially designed software. Given the continuously growing number of cyber threats, the ability to accurately attribute authorship to malicious programs is critically important for ensuring the security of information systems. Additionally, program code authorship identification plays a key role in detecting copyright violations and ensuring academic integrity, which is especially relevant in the digital era.

Currently, automated methods for determining the authorship of texts written in natural languages are considered well-developed. However, attempts to apply these methods to artificial languages, particularly programming languages, do not yield the desired results. This is due to the fact that artificial languages have a significantly smaller vocabulary and much stricter syntactic rules for constructing statements. Thus, developing a new method for automated authorship identification of program code is a relevant challenge today.

The goal of this study is to improve the accuracy of program code authorship identification by developing a method that can be adapted to any programming language and focuses on the individual coding style of different authors. To achieve this goal, it is necessary to review existing solutions for program code authorship attribution, analyze their advantages and disadvantages, and, based on the findings, develop a new method and implement its software realization.

Literature Review

The most common method for determining the authorship of program code is the n-gram attribution method, which analyzes sequences of n elements consisting of adjacent letters, syllables, or other lexical units in the examined text [1]. This method is independent of a specific programming language since it operates on low-level data. However, it has a significant drawback: the core elements of this method are susceptible to subjective manipulation. Specifically, if an attacker knows the frequency of an author's characteristic n-grams, they can easily forge that author's style.

Metric-based methods are also used to determine the authorship of texts written in artificial languages. These methods differ in the number and type of indicators used to construct metrics. For example, one of the earliest code attribution methods, Holsted's metrics [2], relies on only 16 indicators, which is insufficient for accurate authorship identification.

Another method, *IDENTIFIED* [3, 4], developed for the C++ programming language, utilizes cyclomatic complexity and graph-based metrics. Additionally, this method takes into account comments in the code.

A more recently developed method, program "fingerprinting" [5], includes 48 different metrics. This method works specifically for *Java* programs and achieves an accuracy of approximately 73%. The research suggests that source code metrics can indeed help in identifying authorship. However, a major limitation of such metrics is their dependence on the programming language.

To this review, it is worthwhile to add an analysis of contemporary achievements in this field. In particular, study [6] presents an innovative approach that achieves 93–95% accuracy through:

- an expanded set of metrics including: stylistic formatting features, patterns of language construct usage, and abstract syntax tree characteristics;
- a language-agnostic approach that works with various programming languages, unlike most existing methods focused on specific languages (e.g., Java in the "fingerprinting" method);
- deep analysis of structural code features that are more resistant to intentional modification, significantly complicating attempts to forge authorial style.

It is important to note that this method demonstrates substantially higher accuracy (93–95%) compared to the aforementioned "fingerprinting" method, while also having broader applicability due to its language independence. However, like other metric-based approaches, it requires significant computational resources for analyzing large codebases.

Other modern approaches to code authorship attribution should also be analyzed. For instance, Graph-Based methods (e.g., GraphCodeBERT [7]) utilize analysis of code dependency graphs (DFG) combined with textual features, enabling detection of unique function call patterns. A drawback of such methods is the requirement for preliminary graph construction for each code sample, which complicates analysis of large projects.

Another employed solution is Ensemble Learning [8]. This approach involves combining predictions from multiple classifiers (SVM, Random Forest, CNN), improving accuracy to 94%. Its disadvantages include high computational complexity and the need for repeated model training.

Thus, the aforementioned methods cover scenarios where classical approaches are ineffective (binary analysis, small datasets). However, many of them require specialized computational resources or deep ML expertise.

A Metric System for Automated Authorship Identification of Program Code

For further modification in this study, we selected the metric system defined in the program "fingerprinting" method. This system was expanded by adding new indicators, such as the frequency of logical operators, com-

parison operators, increments and decrements, the presence of prefixes and suffixes in variable names, and others. The number of metrics in the expanded system increased to 65.

Next, all metrics were divided into two groups: basic metrics — adaptable to any programming language, and specialized metrics — those that differ depending on the programming language. For convenience, the basic metrics were further categorized into four groups: keywords, operators, delimiters, identifier names.

The specialized metrics include language-specific keywords and operators for Java, Python, C++, and JavaScript. Below, we examine these metric groups in more detail.

Keywords. This category measures the frequency of different keywords, revealing a programmer's specific preferences. For instance, some developers favor *"if"* statements over *"switch/case"* even when both options are interchangeable, or they might prefer *"for"* loops over *"while"* loops. Another example is the preference for *"try + catch"* blocks instead of handling errors with conditional statements. Analyzing such patterns helps track a developer's unique stylistic choices, particularly their inclination toward certain conditional statements and constructs.

Operators. This category includes metrics that reflect the frequency of operators such as "+", "-", "*", "/", "%", and others. It also examines how a developer uses comparison operators ("==", "!=", ">", "<") and logical operators (&&, ||). Additionally, the use of increment ("++") and decrement ("--") operators versus more explicit expressions (e.g., "x = x + 1") is considered. This category helps determine how a programmer structures mathematical operations, conditions, and logical checks in their code.

Delimeters. This category analyzes the frequency and style of using various delimiters, such as spaces, tabs, and newlines, which affect the structure and readability of the code. These symbols help organize text blocks and highlight nested structures (e.g., conditional statements or loops). The placement of spaces around operators is also taken into account. Some developers prefer writing *"x = y + z"*, while others use a more compact style like *"x=y+z"*.

Additionally, this category considers the use of optional symbols in some programming languages, such as semicolons (";"), as well as different types of quotes and commenting styles. Metrics in this category reveal how carefully a programmer formats their code, ensuring readability and adherence to stylistic conventions.

Identifier names. This group includes metrics that analyze variable, class, and method naming styles. First, it considers naming conventions (*camelCase*, *PascalCase*, etc.). Second, some programmers prefer short names (*i, tmp*) in small functions, while others use descriptive names (*indexCounter, temporaryStorage*) for clarity. Third, it accounts for cultural differences, such as using transliterated words from the developer's native language (*knyga* instead of *book*). Finally, the use of standard prefixes and

suffixes is assessed, such as is, get, and others to indicate boolean variables or getter methods (e.g., *isAvailable*, *getName*). The correct use of suffixes for data structures (*list* for arrays/lists, *map* for dictionaries) is also evaluated. These characteristics can reveal a programmer's experience, education, and habits.

Specialized metrics. Specialized metrics account for unique keywords and operators in each programming language. This is crucial because different languages have unique constructs that reflect their design philosophy. For example, *null* in Java has analogous representations in other languages: *None* in Python, *nullptr* in C++, *null* in JavaScript.

Automated Method for Determining Source Code Authorship

The previously described metrics are applied in the developed method for automated authorship identification of source code to establish an author's stylistic characteristics. The method consists of six stages.

Stage 1. Collecting source code from projects

At this stage, software projects with known authorship are selected to form a reference database, along with one project under investigation that needs analysis. The source code of each selected project serves as the basis for calculating metric values, which will be used for further analysis.

Stage 2. Removing auxiliary files from the project

To ensure accurate analysis and prevent bias in the results, all auxiliary files that do not contain core source code are removed. These may include configuration files (XML, JSON, YAML), compiled files (.class or .o), test files, or other files unrelated to the program's primary functionality.

Stage 3. Calculating basic metrics.

For each project, the values of basic metrics are computed.

Stage 4. Calculating special metrics.

If the project contains source code files in C++, Java, Python, or JavaScript, the special metrics are also computed.

Stage 5. Comparing the investigated project's metrics with other projects' metrics

The basic metrics are compared across projects. If both projects contain special metrics, those are also included in the comparison. The deviation of the investigated project's metric values from those of other projects is calculated.

Stage 6. Presenting the results.

The analysis results are presented as a percentage similarity between the investigated project's metric values and those of other projects. The project with the highest similarity percentage should belong to the same author as the investigated project.

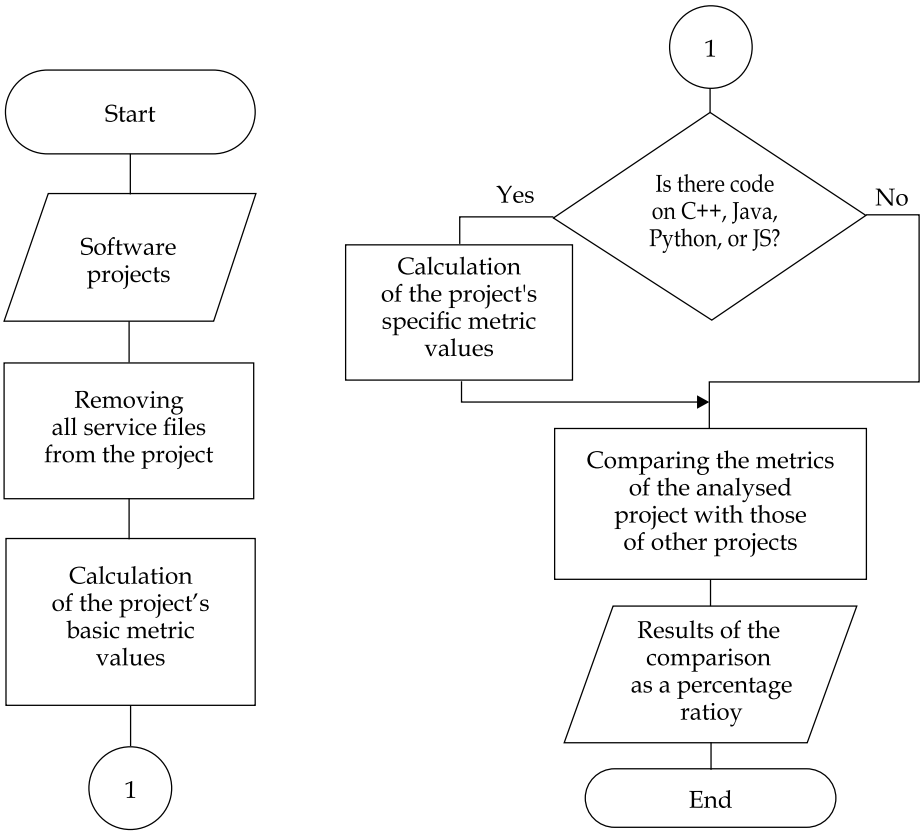

Fig. 1. Sequence of stages of the proposed method

Software Implementation of the Method

In the software implementation of the proposed method, metric values of the investigated project are compared not with the metric values of other projects but with the metric values from author profiles. Initially, for each author's projects — there may be one or more — the method's stages 2 to 4 are applied, meaning auxiliary files are removed, and either basic or both basic and special metrics are calculated. Based on these metric values, an author profile is created: the metric values of all projects by a single author are averaged.

For the development of the software project implementing the proposed method, Python was chosen due to its powerful data processing capabilities. The following libraries were used: NumPy, pandas, and SQLAlchemy. The database management system PostgreSQL is used for database interaction. The server side of the project is built using the Flask framework, which handles request processing, routing, database interaction, and API implementation. The front-end interface is developed using Vue.js, enabling the creation of a dynamic and user-friendly interface.

The web application follows a modular architecture.

The *client-side* includes the following modules:

• API interaction module — responsible for communication between the user interface and the server, sending requests to the server API and receiving responses to be displayed in the graphical interface;

• graphical interface components — implement user interaction features.

The *server-side* includes an API controller module, which processes requests from the client side and calls appropriate modules to perform tasks, along with separate functional modules.

The API controller module contains the following submodules:

• module for working with data about authors, which provides creation, deletion and update of information about project authors;

• module for working with data about projects, which provides addition, deletion of information about projects;

• module for working with metrics, which provides addition, deletion and update of metrics;

• module for working with results, which provides storage of results of determining authorship.

Additional server-side modules:

• a module for calculating the values of basic metrics;

• a module for calculating the values of special metrics;

• a module for calculating the results, in which the deviations of the metric values are calculated and the results are calculated in percentages;

• a module for interacting with the database, which performs the operations of saving, updating and retrieving data from the PostgreSQL database.

The PostgreSQL *database* serves as a storage system for all necessary information, including author data, project data, analysis results, and metric values.

Conclusions

This study reviews existing methods for automated authorship attribution of source code and proposes a new method based on an extended system of metrics.

The proposed method utilizes a metric system based on the "fingerprint» approach to code analysis. These metrics capture the individual stylistic characteristics of a programmer, regardless of the programming language used. By computing metric values for projects, author profiles can be generated, forming the basis for further authorship identification.

As a next step in the project, the authors consider it appropriate to conduct experiments to evaluate the effectiveness of the proposed method using different input data across various programming languages. Additionally, they plan to compare the results obtained with the proposed method against those derived using the Halstead metrics-based approach.

REFERENCES

1. Frantzeskou G., Stamatatos E., Gritzalis S., Chaski C. Identifying Authorship by Byte-Level N-Grams: The Source Code Author Profile (SCAP) Method. *International Journal of Digital Evidence,* Trier, Germany, 2007, Vol. 6 (1), 139–148. URL: https://www.researchgate.net/publication/220542545_Identifying_Authorship_by_Byte-Level_N-Grams_The_Source_Code_Author_Profile_SCAP_Method [Accessed 12 Nov. 2024].
2. Frantzeskou G., MacDonell S., Stamatatos E., Georgiou S., Gritzalis S. The significance of user-defined identifiers in Java source code authorship identification *Computer Systems Science and Engineering*. Samos, Greece, 2011.
3. Gray A., Sallis P., MacDonell S. *IDENTIFIED (Integrated Dictionary-based Extraction of Non-language-dependent Token Information for Forensic Identification, Examination, and Discrimination): a dictionary-based system for extracting source code metrics for software forensics.* IEEE Computer Society Press, Dunedin, New Zealand, 1998, 252–259 pp. https://doi.org/10.1109/SEEP.1998.707658
4. Sallis P., Aakjaer A., MacDonell S. *Software Forensics: Old Methods for a New Science.* IEEE Computer Society Press, Dundin, New Zealand, 1998, 367–371 pp. https://doi.org/10.1109/SEEP.1996.534037
5. Ding H., Samadzadeh M.H. Extraction of Java program fingerprints for software authorship identification. *The Journal of Systems and Software,* 2004, Vol. 72 (1), 49–57. https://doi.org/10.1016/S0164-1212(03)00049-9
6. Abuhamad M., AbuHmed T., Mohaisen A., Nyang, D.H. Large-scale and language-oblivious code authorship identification. *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security,* 2018, 101–114. https://doi.org/10.1145/3243734.3243738
7. Daya G. et al. GraphCodeBERT: Pre-training Code Representations with Data Flow *Proceedings of the ICLR 2021*. URL: https://www.researchgate.net/publication/344294734_GraphCodeBERT_Pre-training_Code_Representations_with_Data_Flow [Accessed 14 Nov. 2024]
8 Abbasi A., Javed A.R., Iqbal F. et al. Authorship identification using ensemble learning. *Scientific Reports, 2022, Issue 12.* https://doi.org/10.1038/s41598-022-13690-4

Received 06.03.2025

А.Г. Адамчук, студентка,
Національний технічний університет України
«Київський політехнічний інститут ім. Ігоря Сікорського»,
Берестейський просп., 37, м. Київ, 03056, Україна
ann.adamchuk2002@gmail.com

В.І. Сущук-Слюсаренко, старш. викладач,
Національний технічний університет України
«Київський політехнічний інститут ім. Ігоря Сікорського»,
Берестейський просп. 37, м. Київ, 03056, Україна
https://orcid.org/0000-0002-6096-3832
Sushchuk.Viktoriia@lll.kpi.ua

А.І. Дичка, д-р філософії (техн.), асистент каф.,
Національний технічний університет України
«Київський політехнічний інститут ім. Ігоря Сікорського»,
Берестейський просп. 37, м. Київ, 03056, Україна
https://orcid.org/0000-0003-0578-2788
andriydychka@gmail.com

АВТОМАТИЗОВАНЕ ВИЗНАЧЕННЯ АВТОРСТВА ПРОГРАМНОГО КОДУ НА ОСНОВІ СИСТЕМИ МЕТРИК

Вступ. Визначення авторства програмного коду набуває важливого значення в сучасних умовах, коли зростає кількість кіберзагроз, актуалізується захист авторських прав і забезпечення академічної доброчесності. У статті продемонстровано метод автоматизованого визначення авторства програмного забезпечення шляхом аналізу вихідного коду. Метод розширює систему метрик, що застосована в методі «відбитків» програми, що дозволяє працювати з мультимовними проєктами. Ці метрики використовуються для створення «профілів авторів» на основі наявного коду програміста, що дозволяє перевіряти авторство в інших програмних проєктах.

Мета статті. Метою дослідження є підвищення точності визначення авторства програмного коду шляхом розроблення методу, адаптованого під будь-які мови програмування.

Методи. Метод автоматизованого визначення авторства програмного коду розроблено на основі системи метрик, що включають чотири категорії базових метрик: ключові слова, операційні символи, розділові символи та імена ідентифікаторів, а також спеціальні метрики. Для визначення авторства збираються програмні проєкти з достовірною інформацією про авторство, обчислюються значення базових та спеціальних метрик цих проєктів, після чого формуються профілі авторів на основі середніх значень цих метрик. Подальше порівняння значень метрик досліджуваного коду з метриками з профілей авторів дозволяє ідентифікувати автора коду.

Результат. В роботі проведено огляд існуючих методів автоматизованого визначення авторства програмного коду, після чого запропоновано власний метод на основі використання системи метрик. Запропонований метод використовує систему метрик, в основу якої покладено метод «відбитків». Метрики відображають індивідуальні стилістичні особливості програміста незалежно від мови програмування.

Висновки. Матеріали статті будуть корисними при вирішенні задач автоматизованого визначення авторства програмного коду.

Ключові слова: *метрики, атрибуція, вихідний код, визначення авторства, інформаційна система.*