Code Authorship Attribution: Methods and Challenges

VAIBHAVI KALGUTKAR, RATINDER KAUR, HUGO GONZALEZ,
NATALIA STAKHANOVA, and ALINA MATYUKHINA, University of New Brunswick

Code authorship attribution is the process of identifying the author of a given code. With increasing numbers of malware and advanced mutation techniques, the authors of malware are creating a large number of malware variants. To better deal with this problem, methods for examining the authorship of malicious code are necessary. Code authorship attribution techniques can thus be utilized to identify and categorize the authors of malware. This information can help predict the types of tools and techniques that the author of a specific malware uses, as well as the manner in which the malware spreads and evolves. In this article, we present the first comprehensive review of research on code authorship attribution. The article summarizes various methods of authorship attribution and highlights challenges in the field.

CCS Concepts: • **General and reference** → **Surveys and overviews**; • **Software and its engineering** → *Language features*; • **Security and privacy** → *Malware and its mitigation*;

Additional Key Words and Phrases: Authorship analysis, programming style, malware attribution, software forensics

ACM Reference format:
Vaibhavi Kalgutkar, Ratinder Kaur, Hugo Gonzalez, Natalia Stakhanova, and Alina Matyukhina. 2019. Code Authorship Attribution: Methods and Challenges. *ACM Comput. Surv.* 52, 1, Article 3 (February 2019), 36 pages.
https://doi.org/10.1145/3292577

1 INTRODUCTION

Since the development of the computer virus, the security community has been interested in ways to expose the identity of an adversary. In the past, this was often possible owing to the relative simplicity of malicious software. In some cases, a manual analysis of code even revealed personal and identifiable information embedded by the authors themselves [51]. However, with the growing use of complex obfuscation techniques and the availability of malware code generators, which allow malware developers to create variants of malware at unprecedented rates, this process has become significantly more challenging and requires advanced methodologies. The relevant methods often found in authorship attribution research are referred to as stylometry. Authorship attribution in the literary domain offers a wide spectrum of techniques to identify an author of a document based on a set of textual features that quantify writing style [83]. The assumption underlying attribution

This work was supported by a grant from the New Brunswick Innovation Foundation.
Authors' addresses: V. Kalgutkar, R. Kaur (corresponding author), H. Gonzalez, N. Stakhanova, and A. Matyukhina, Canadian Institute for Cybersecurity, Faculty of Computer Science, University of New Brunswick, 3 Bailey Drive, Fredericton, NB, E3B 5A3, Canada; emails: {vkalgutk, rkaur1, hugo.gonzalez, natalia.stakhanova, amatyukh}@unb.ca.

0360-0300/2019/02-ART3 $15.00
https://doi.org/10.1145/3292577

is the existence of a distinctive writing style, unique to an author and easily distinguishable from others. A quantified representation of this style can be viewed as a fingerprint.

Dating back to the 19th century and the first analysis of the sonnets of Shakespeare, authorship attribution techniques have leveraged developments in machine learning and natural language processing (NLP) for electronic texts (e.g., email messages, blog posts, and Web pages), software, and source and binary codes. Beyond the traditional focus on the analysis of literary works, authorship attribution techniques have been applied to the software field, such as in code plagiarism detection [84], biometric research [40], source code authorship attribution [36, 64], and malware analysis [23]. Code authorship attribution can help in cybercrime investigations because it can attribute attacks to particular adversaries. In this context, attribution often pursues the following objectives:

- *Authorship identification*—to find the most likely author of a specific work from a set of candidate authors. Authorship identification can help identify adversaries behind zero-day attacks or variants in the wild. For instance, WannaCry (ransomware) and its new variant, WannaMine (cryptominer) are built on EternalBlue, an exploit developed by the U.S. National Security Agency (NSA) [7].
- *Authorship clustering*—to group the authors based on stylistic similarities from a set of candidates. This can help identify groups with which the author has collaborated, and in some cases identify different adversary groups targeting similar organizations.
- *Authorship evolution*—to analyze the evolution of the author's programming skills, preferences, and writing style over a period of time. Authorship evolution can help researchers understand ever-changing trends in the underground economy: the way adversary groups evolve their attack tools, techniques, and tactics.
- *Authorship profiling*—to analyze details of an author's profile, such as gender, nationality, and ethnicity, based on stylistic features extracted from the given work. Such profiling can help law enforcement agencies understand the global landscape of individual adversaries and adversary groups around the world.
- *Authorship verification*—to determine the author of a given sample code. An authorship verification problem may occur if an adversary modifies a victim's code to add some illegal functionality and the victim is blamed for it. In this case, the task is to identify the author of the modified code.

The uniqueness of code attribution lies in the characteristics of a fingerprint and its ability to quantify an author's programming style. One of the main difficulties in the field is in compiling a fingerprint that efficiently and accurately characterizes the author's style. In the traditional setting, authorship attribution relies heavily on information that allows deep linguistic analysis of works (e.g., richness of vocabulary, tense of verbs, semantic analysis of sentences). In software, emphasis is often placed on surface characteristics, such as variable naming, program layout, and spacing, that reflect the textual nature of the source code. Such an approach is dictated by the nature of the field that, in many cases, fails to provide the original source code for software (e.g., malware analysis and commercial software theft), leaving researchers with the binary representation. As such, binary code retains very few of the surface characteristics of the original, and the focus of research in recent years has shifted toward the analysis of stylistic features of program binaries [23, 74, 76].

This shift was also driven by different objectives of code attribution. In contrast to assigning authorship to one of the candidate authors whose identities are known and well described, the goal of code attribution is much broader, and it often seeks insight into anonymous authors throughout the entire development and usage cycle. Similar to that of literary works that reflect an author's

stylistic habits, the analysis of software may unveil the digital identity of a programmer reflected through variables and structures, the programming language, development tools, settings, and how and for what these tools are being used.

In this article, we survey research advances in code authorship attribution. Figure 1 shows a timeline of publications related to authorship attribution. We trace the historical development of the field since the first appearance of code attribution research. Although a number of general surveys discuss the attribution of literary works and offer excellent reviews of the classifications of linguistic style [50, 54, 83], there is a lack of research on the attribution of the source and binary code, which we aim to cover here. The contribution of this work is threefold:

- —We offer a comprehensive review of studies focusing on code authorship attribution. This is the first effort of its kind to the best of our knowledge.
- —We review traditional stylometric features and their representation in light of source and binary code attribution. We also discuss code attribution models and methods emphasizing challenges in the field.
- —We present a comparative summary of research in the field of code authorship attribution.

The remainder of the article is organized as follows. Section 2 presents the history of the evolution of code authorship, and Section 3 describes the overall process of code authorship attribution, starting from sources of input, feature extraction, feature representation, and separate embeddings that are then applied to attribution models and methods. Section 4 discusses research challenges, and we offer the conclusions of our review in Section 5.

2 CODE AUTHORSHIP ATTRIBUTION FROM A HISTORICAL PERSPECTIVE

2.1 Origin

Although traditional research on authorship attribution dates back to the 19th century, the first studies on code attribution appeared only in the late 1970s [30, 70]. Initial efforts revolved around the automatic evaluation of algorithms in programming assignments to understand algorithmic complexity and, consequently, measure the student's capability [65].

Past attempts on authorship attribution are rooted in the theory of software science developed by Halstead [45] in the early 1970s. The theory states that only four basic metrics are needed to reflect the implementation and structure of any algorithm. These four metrics represent the "internal quality" of an algorithm and are unlikely to be the same among programs written by independent authors. These metrics became known as Halstead's metrics or software science metrics, and are listed in Table 1.

The theory of software science has been reviewed by many researchers [2, 12, 24, 35, 46, 47, 79]. Most have confirmed its benefits while offering improvements. Bulut et al. [12] proposed the counting algorithm for FORTRAN programs to reduce the time required to measure software science metrics. Albrecht and Gaffney [2] demonstrated the relation between the size of a programming system, its development efforts, and Halstead's software science metrics. Fitzsimmons and Love [35] concluded that the metrics yield higher accuracy than other measurements when applied to a sufficiently large collection of programs.

Despite the significant support for Halstead's theory, several objections were raised by some researchers [47, 79]. According to these critics, in many cases, the empirical results of the theory failed to support its own postulates; hence, any practical use of the theory in its current state is questionable.

Despite disagreements, the theory set a precedent in software assessment and triggered research focusing on software similarity detection. Initially, the focus of these efforts was to find similarities

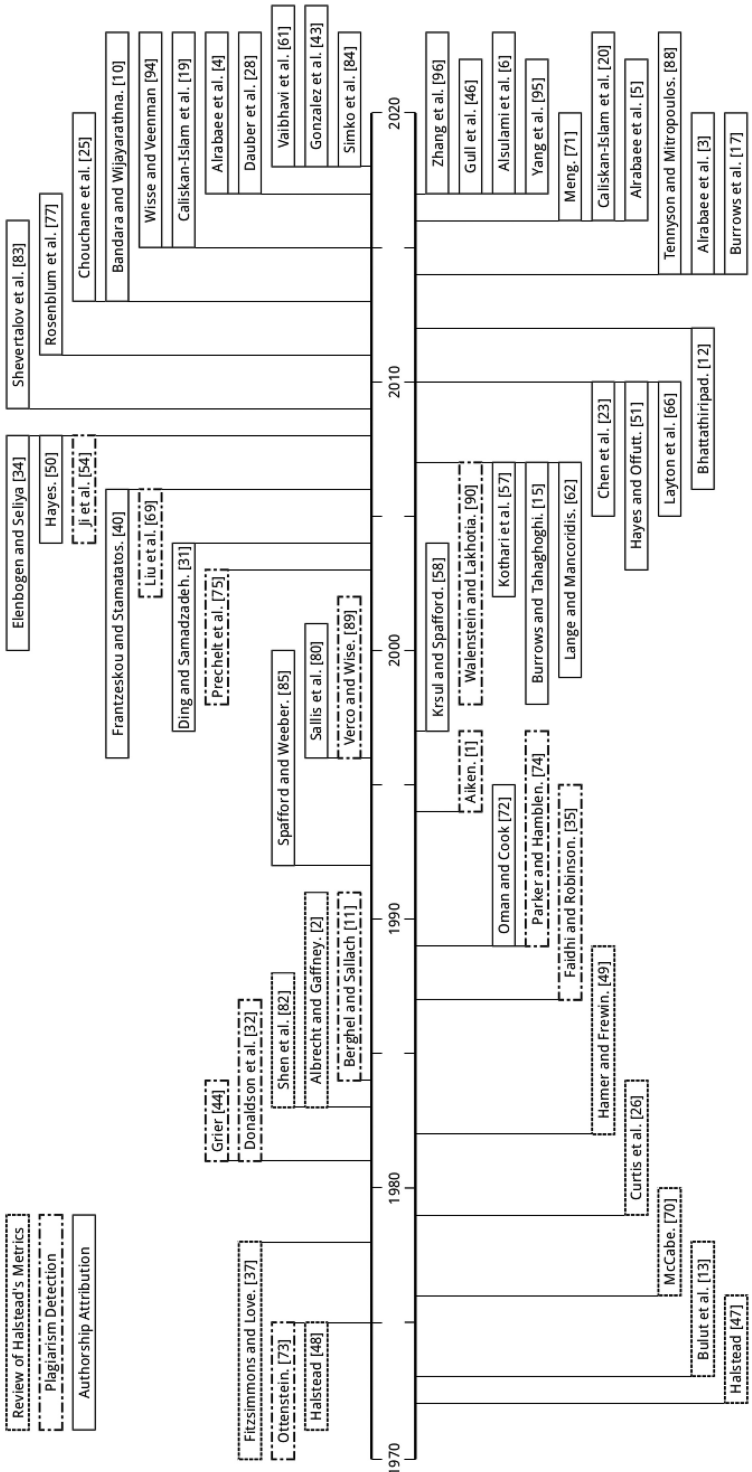


Fig. 1. Timeline of publications related to authorship attribution.

Table 1. The Basic Elements of Software Science Theory [45]

Metric	Definition
n1	Number of unique operators
n2	Number of unique operands
N1	Total occurrences of operators
N2	Total occurrences of operands

between programming assignments submitted by students, and to detect the original and the plagiarized documents. This research, known as plagiarism detection, established the ground for code authorship attribution and later spawned research along several other directions, such as software cloning and software evolution.

Plagiarism detection is often used interchangeably with code authorship attribution. Authorship attribution focuses on an author and is thus concerned with the writing style reflected in code. In case of plagiarism detection, however, the author of the document is known, and the aim is to identify unoriginal content extracted from another known author. In other words, plagiarism detection is an example of a closed-world problem, when authors are enumerated and their styles are well known, with the objective of correctly choosing one of the candidates, whereas authorship attribution goes beyond this problem and aims to identify the original author's style when the author's identity is not necessarily known and readily available. Therefore, the goals of the two are different. However, both fields are based on the same underlying principle of feature extraction to detect similarities between programs written by an author. Thus, techniques employed for authorship attribution and plagiarism detection often overlap. To study authorship attribution, however, it is important to review research in plagiarism detection as well.

2.2 Plagiarism Detection

A well-known study in plagiarism detection was conducted by Ottenstein [70]. He employed the software science metrics to find similarities between students' assignments written in FORTRAN. He noted that cosmetic transformations, such as renaming variables and reordering independent statements, that are often used to bypass plagiarism detection tools are not reflected in software science metrics. Thus, he argued, software science metrics alone are not sufficient for the detection of plagiarism in FORTRAN source code.

Following this study, several researchers aimed to redress the situation. Common methods used to disguise plagiarism, such as variable renaming, do not modify the internal structure of a program. Based on this, Donaldson et al. [30] developed a system for similarity detection based on the structure of a program focusing on FORTRAN, COBOL, and BASIC programming languages. He expanded software science metrics to include other structural features, such as the number of loops, assignment statements, and the number of calls to subprograms. Although this approach was found to be accurate only in detecting simple plagiarism (e.g., format modification and renaming), it introduced structural analysis for exposing program similarity.

Subsequently, Berghel and Sallach [9] performed a comparative analysis of 15 common complexity metrics to detect similarities between FORTRAN programs. The features included Halstead's metrics and other surface characteristics of programs, such as the total number of lines of code and number of integer variables. Using the results of a large experiment involving 700 programs, the authors concluded that Halstead's metric system is ineffective because it identifies non-existent similarities while missing obvious resemblances between programs. The study confirmed earlier

results showing that the components of Halstead's metrics have no unique practical value. Despite such criticisms of Halstead's metrics, researchers continued to use them for analysis [42, 65].

Each study at that time tried to develop larger sets of features than past work. In most cases, features were selected in an ad hoc fashion, and there was no theoretical basis for their effectiveness. Over time, it became clear that attribute-based features (e.g., lines of code and the frequency of integer variables) are only effective when simplistic plagiarism techniques are used, such as comment alteration, identifier modification, and statement reordering. Yet, the use of a wide spectrum of methods for disguising plagiarism was becoming common. Instead of limiting the analysis to the identification of trivial disguises, research efforts shifted to the detection of complex modifications. These modifications required exposing features of the underlying logic, structure, and flow of the code.

Several plagiarism detection systems based on program structure were introduced. Of them were the structure metric–based system by Faidhi and Robinson [33], Plague [89], MOSS (Measure of Software Similarity) [78], Sim [43], YAP3 [90], and GPLAG [66]. All of these studies highlighted the effectiveness of structure-based metrics (e.g., choice of data structure or control structure) over simple attribute counting systems. Yet, a study conducted by Verco and Wise [86] concluded that structure metric systems perform equally well, and only in some cases are better than attribute-counting systems. Although this study compared only the attribute-counting system–based method "Accuse" [42] with the structure metric–based method proposed by Faidhi and Robinson [33], no follow-up study to disprove this result was conducted.

Over the years, it became clear that the basic foundation of software science metrics cannot be scientifically supported. What operators or operands should be counted, and how, was left to each researcher. As a result, many studies started implementing programming language–dependent features and exploring features characterizing the structure and syntax of a language. The most popular software system of this nature is JPlag [72]. Research by Ji et al. [52] added another level of abstraction to language-dependent features. They extracted token sequences from Java bytecode and evaluated the similarity of Java programs using adaptive local alignment. The important result was that source codes and bytecodes are equally good at discovering program similarities.

Over time, interest in Halstead's metrics and plagiarism detection diminished. As software theft became a more prominent topic in the industry, the research community turned its attention to binary code analysis. In essence, the detection of software theft and plagiarism are very similar, with one caveat—the availability of source code. Traditionally, plagiarism studies experimented with program source code; however, in the case of software theft, we typically deal only with the binary version of software that does not retain most of the author's characteristics. Beyond confirming whether a given program is a copy of another, software theft detection cannot determine if both programs have been created by the same author. This work focuses on determining the authorship of code.

2.3 Evolution of Authorship Attribution

Along the lines of plagiarism detection, researchers started investigating the relation between the authors and the programs written by them. Similarly to plagiarism research, this topic started with the analysis of source code and evolved into binary code analysis.

Oman and Cook [69] were pioneers in the study of programming style for authorship attribution. They focused on typographical or layout characteristics that can produce a wide range of style markers. Their proposed method employed a statistical analysis of these markers and showed highly accurate results. This study failed to consider that programs can be easily modified by code formatters.

A new era of authorship attribution started with *software forensics*, a term coined by Spafford and Weeber [82] in their seminal 1993 work. The authors claimed that every programmer has a unique style. In investigating a security breach, they connected the manual writing recognition technique used in law enforcement to identify people with the task of the analysis of residual code in a security incident to identify an adversary. They suggested that attack traces can be found in software of any form, including source files, object files, executable code, and shell scripts. Although Spafford and Weeber did not provide any statistical evidence to support their theory, the work became a springboard for research on adversarial author attribution. Considering that the presence of source code in the adversarial domain is rare, their work sparked considerable interest in binary code attribution.

A 1994 study by Krsul and Spafford [57] argued that authorship attribution in computer software is a much more difficult task than in the literary domain. Software developers reuse code, work in teams, format programs by code formatters, and use third-party libraries and tools that introduce new stylistic features. Despite their critical view of code attribution, Krsul and Spafford highlighted the effectiveness of authorship analysis to enhance real-time intrusion detection.

In 1996, Sallis et al. [77] revisited software forensics to discuss the challenges of using software attribution methods and techniques available in natural language processing. Noting the variability of methods, they emphasized the core problem—that none of the prevalent approaches alone can solve the problem. However, methods from NLP in combination with conventional software metrics can be a good fit for the software attribution process.

Yet, the main challenge of authorship attribution for source and binary codes at the time was the ultimate set of features that can be used for analysis. In 2004, Ding and Samadzadeh [29] focused on the metric selection procedure for the attribution of Java programs. They used two ways of selecting the most relevant features: by manual selection using one-way ANOVA and automatic selection by stepwise discriminant analysis. Canonical discriminant analysis was used to evaluate the effectiveness of selected metrics. The study reported a classification accuracy of 85.8% using canonical variates and, the authors claimed, produced a set of features more diverse than in any previous study [58, 77].

A different approach to identifying an effective combination of features useful for authorship attribution was proposed by Lange and Mancoridis [60]. A set of normalized histograms illustrating the distribution of code features was used to identify the style of a developer. The authors employed genetic algorithms to test different combinations of 17 features. Unfortunately, no combination was optimal and required recalibration for every new dataset.

Still, all of the preceding studies focused on meaningful and often human-readable features. In 2006, a team of researchers explored the benefits of raw N-gram features for binary code attribution. This novel approach that involved generating the most frequent byte-level N-grams for author profiles was proposed by Frantzeskou et al. [38, 39]. The similarity between known author profiles and an unidentified code sample guides the attribution decision process. This method, known as the source code author profile (SCAP) approach, had previously been evaluated for natural language authorship attribution by Kešelj et al. [53]. The main benefit of using the approach in software is its independence of programming language, computational simplicity, and resistance to compilation. Considering that many previous studies had examined the characteristics of the layout of source code (e.g., comments, brackets) typically eliminated in the compilation process, using raw N-grams is beneficial in preserving potentially useful author characteristics.

Owing to these advantages, N-gram analysis was applied in many subsequent studies [13, 31, 63]. Layton et al. [63] evaluated the unsupervised SCAP method for the analysis of phishing scams. Burrows and Tahaghoghi [13] explored the N-gram approach to attribution based on keywords

and operators. Dubba and Pujari [31] examined the effectiveness of N-grams for computer virus detection and Burrows et al. [15] confirmed the effectiveness of byte-level N-gram analysis in the SCAP method for the general source code attribution task.

The problem with N-gram–based approaches is the selection of n. Typically, the best value of n depends on experimental data. Considering that most studies rely on datasets generated in a controlled environment, the optimal value of n varies depending on the application and settings. For example, a study by Kothari et al. [55] concluded that character-level 4-grams analysis provides satisfactory accuracy because it can capture an author's style better than any other style-based feature.

Moving away from the analysis of syntactic similarity, research has focused on identifying the author of source code through the analysis of code logic. Eventually, researchers began analyzing features that represent the underlying semantics and dynamic behavior of the program [10, 17, 21, 49].

2.4 State of the Art in Authorship Attribution

Code authorship attribution faces many research challenges that are similar to those in the domain of malware analysis. Issues such as the unavailability of source code, reuse of existing code, and prevalence of obfuscation are common to both fields. As pointed out in the theoretical study by Walenstein and Lakhotia [87], similar challenges might also lead to similar solutions. Considering that malware analysis is typically performed at the binary level, attribution might overcome many difficulties by also leveraging binary analysis.

The study by Rosenblum et al. [74] is one of the first that attempted authorship attribution of binary code. By converting binary code into a different representation, the work explored N-grams, idioms (recurring code constructs in all programs), control flow graph (CFG), and other representations to capture the internal structure of the software.

The idea for this work came from previous research on the provenance of the tool chain. A tool chain refers to a set of tools employed in the software development process. Earlier, the same authors had investigated the compiler provenance—for instance, characterizing details of program compilers and their effect on program binaries. They found it feasible to identify the original compiler used to generate a particular binary code [76]. Follow-up studies [73, 75] further investigated tool chain provenance with the aim of attributing program binaries to the production tool chain. Once again, they confirmed that it is possible to distinguish between complete tool chains used to generate a program. In a similar vein, Chouchane et al. [23] showed the feasibility of attributing malware to the metamorphic engine or kit used to generate it. The results obtained from these studies prove that the ideas of authorship attribution can be applied to discover different levels of the origin of the relevant software.

A critical facet of any attribution study is the availability of reliable datasets. The existence of representative, properly verified, and labeled datasets of the authors' source/binary code samples is a major challenge in the attribution domain. This typically drives researchers to use customized and manually selected sets that are often too small, or imbalanced, and contain few samples for some authors and significantly more for others. For example, Lange and Mancoridis [60] experimented with a dataset containing code by 20 authors with only 3 samples per author, whereas Rosenblum et al. [74] employed a dataset with code by almost 200 authors with programs ranging from 4 to 16 per author. Similarly, the size of code samples used for analysis varies drastically. Bandara and Wijayarathna [8] collected code samples ranging in size from 28 lines to more than 15,000 lines of code, whereas a number of studies have experimented with code samples sometimes containing only a single line [14, 26].

The lack of sufficient data creates biased or even inaccurate results. This was illustrated in a study conducted by Chatzicharalampous et al. [20], which employed an imbalanced set of source code samples. The results decreased in dependability and consistency as the level of imbalance in authors' samples increased. Noting the difficulties with a large number of complete programs, Dauber et al. [26] studied the feasibility of authorship attribution of segmented code samples.

Although attribution is typically performed on authors of benign software and their data, a few studies examined the challenges of attributing malware. Layton and Azab [62] investigated the authorship of the Zeus botnet source code and concluded that most of the code had been written by one author, whereas some modules that represented additional functionality appeared to have been written by others.

Attributing a binary file is difficult because it undergoes various changes by means of some compilation or obfuscation techniques. Obfuscation transforms a binary into a form that is difficult to analyze or reverse engineer while preserving its original behavior. With the increasing popularity of obfuscation in both legitimate and malware applications, the task of attribution is becoming more challenging. Various compilation and obfuscation techniques modify the original code, rendering the characteristic markers of an author's style even more elusive.

Several studies investigated features and methods resilient to obfuscation. Alrabaee et al. [3] proposed the Onion approach to improve the accuracy of binary authorship attribution. By removing and filtering unnecessary data and irrelevant code, they applied syntactic and semantic attribution to yield promising results. Following this intuition, Caliscan-Islam et al. [18] proved that certain syntactic features extracted from decompiled binaries survive various compilation techniques. Yet, in another study, Alrabaee et al. [5] showed that the features most robust to obfuscation are the ones derived from the data and control flow.

Thus far, work on code authorship attribution has been based on the assumption that the code is developed by a single author, which is not the case in modern software development, where a group of authors generally works on it. In such cases, authorship attribution is a challenge because the underlying software does not represent the style of any one author. Understanding whether an unknown binary has been developed by one or more authors comes before attribution can be applied. An option in this scenario is to study code chunks instead of complete samples [26]. Meng [68] proposed a method to identify the multiple authors of a binary by analyzing each basic block. This technique can discriminate among the authors of each basic block with an accuracy of 52%.

As the field of authorship attribution evolves, more complex features are being introduced based on application programming interface (API) calls and CFGs, giving us stronger results and more flexibility in analysis. However, the main difficulty (i.e., the unavailability of source code) persists. Tools such as code formatters, beautifiers, and obfuscators are prevalent and only help authors hide their identities. In such situations, binary code becomes the primary target of analysis.

3 PROCEDURE FOR AUTHORSHIP ATTRIBUTION

Despite the variability of studies in the field, a typical workflow of code authorship attribution is the same. As Figure 2 shows, the process of attribution starts with the extraction of features at the source or binary code level and their translation into a suitable form. Depending on how the features should be processed further, a proper attribution model is chosen. Finally, based on the ultimate goal, a method of attribution is used along with machine learning techniques.

3.1 Features

In the history of authorship attribution research, the employed features have always played a critical role. Over the years, the features have evolved significantly from simple counting metrics in

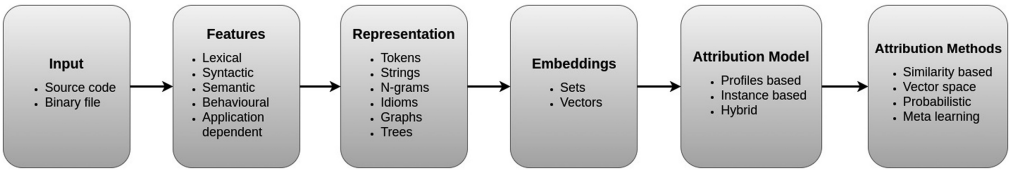


Fig. 2. Conceptual schema of the authorship attribution process.

software science to semantic metrics such as CFGs. As in the literary domain, features in code attribution can be classified into three broad categories: lexical, syntactic, and semantic [83]. Unlike the literary domain, however, where only plain text is available, software consists of many more components, including resources (e.g., image, audio files), metadata, and libraries. Therefore, application-specific information and runtime behavior can complement the attribution process. We thus also introduce two new categories of features: behavioral and application dependent.

3.1.1 Lexical Features. The basic form of a feature that can be extracted from the source or binary file is the lexical feature. It is a simple sequence of tokens or characters, or even bytes, and can be counted in many ways, such as term frequency (TF), term frequency-inverse document frequency (TF-IDF), and ratio and average. While extracting lexical features, the semantic characteristics of tokens are not considered. Lexical features include the length of lines, number of lines in a program, number of operands, number of variables, word frequencies, token frequencies, character N-gram, function or method names, and identifiers. Stylistic and layout features such as the use of indentation, number of spaces, variable naming, and number of blank lines also fall into this category.

Lexical features work on the idea that authors are prone to choose the same or similar words to name functions, variables, or other resources in all of their programs. The main advantage of lexical features is that they are basic and simple. They are language independent; hence, techniques used to extract such features can be easily applied to different programming languages and environments. Lexical features can be obtained from source code and other forms of programs. Their simplicity and ease of implementation has made them the focus of many research studies.

Regardless of their advantages, lexical features can be inadequate to the task of authorship attribution. They heavily depend on the number of training samples available per author—for example, with a greater number of samples, one can better determine the choice of preferred function names. Over the years, a wide variety of lexical features have been proposed; however, selecting the most relevant and contributory features is a challenge. Many stylistic and layout-related features can be easily altered by code formatters, thus affecting the reliability of such features. Moreover, proprietary software and malware writers can employ techniques such as variable renaming, addition/removal of redundant code, splitting sentences, transposing statements, string obfuscation, and string encryption to evade attribution systems. Despite these challenges, lexical features have continued attracting many researchers. Studies [13, 37–39] have used byte sequences and produced good results.

3.1.2 Syntactic Features. Syntactic features characterize the external structural organization of code. In contrast to lexical features, syntactic features work with the organization or use of tokens. Syntactic features can also be counted in different ways, including TF, TF-IDF, average, and percentage. Examples cover the numbers of functions, numbers of keywords, choice of data structure, use of special macros, use of language features, number of inputs, number of conditional/assignment statements, choice of control structures, number of calls to functions, order of statements, number of specific delimiters, average function size, and frequency of overloaded operators.

The main goal of syntactic features is to exploit the idea that authors are comfortable staying within formed habits that are hard to change and tend to use code routines unconsciously. For example, some authors might prefer using the for loop over the while loop. Syntactic features are reliable and can produce stronger author profiles than those obtained using only lexical features because they are more resistant to code formatting and obfuscation techniques, such as variable renaming and string encryption.

However, syntactic features are language dependent. Therefore, a tool developed to extract and analyze features for one programming language cannot be directly applied to another. Several syntactic features have been proposed in various studies over the years, and selecting the best is a challenge. Sophisticated proprietary software and malware authors can still mask their identities by changing the structure of the programs, such as through function modification by changing its type, return type, use of inline functions, reordering of methods, using different control structures (e.g., replacing a for loop with a while loop, or replacing case statements with if statements), reducing the number of subprograms, addition of redundant code, and class merging [81]. The use of advanced obfuscators can also affect the performance of these features. However, syntactic features remain promising and can provide a lot of information about the author. Studies need to explore the right combination of modification-resistant features that can withstand various obfuscation and code optimization techniques.

3.1.3 Semantic Features. Semantic features express the logical flow of the code and give meaningful insight into its inner workings. Semantic features can be extracted irrespective of the syntax and structure of the program; names of the function or variables; and other stylistic, layout-related, or structural features. For example, a programming loop can be expressed in different syntactical ways, such as with a for loop, do while, repeat until, and for each, but at the semantic level, they all represent a loop.

The main goal of semantic features is to exploit the idea that authors tend to use the same logic while creating programs. The programming style of the author for various programming languages may differ; however, the analytical skills of the author reflected in the problem-solving approach are difficult to change. Semantic features try to capture this aspect of the author's programming style. These features include various sophisticated, high-level features, such as the number of loops, algorithms implemented, control flow analysis, dataflow analysis, and procedure-dependence analysis. Semantic features are much more robust than the previous two categories of features and are resistant to many semantics-preserving transformation techniques such as code optimization, string obfuscation/encryption, data transformation [22], and various compilation techniques.

Despite the robustness and effectiveness of semantic features, their extraction is a difficult process. The author's logic across several programming languages may remain constant; however, tools to extract this logical flow for every language require additional effort. For example, a tool to extract the CFG in Java cannot be directly used for C programs. Proprietary software and malware authors can still hide their identity by various techniques, such as addition of redundant code, including API calls and additional functions, to modify the control flow [22, 81]. Advanced obfuscation techniques including control flow obfuscation and control flow flattening can significantly degrade the performance of these features [61]. Regardless of these challenges and difficulties, semantic features can generate strong author profiles. More research is needed to make them more resistant to code obfuscation. Choi et al. [22] suggested techniques to tackle issues like control flow modification and the addition of unnecessary API calls. Semantic features for authorship attribution have not been explored fully by researchers. More studies focusing on exploiting the author's semantic style are needed.

3.1.4 Behavioral Features. Unlike in literary authorship attribution analysis, source code can be transformed into binary form that can then be executed for further analysis. While executing the program as a black box, there exists the possibility for additional code to be loaded dynamically "on-the-fly" from external sources, previously encrypted functions to be decrypted or deobfuscated, or a data structure (e.g., string literal containing shellcode) to be executed as code.

Behavioral features can be obtained from dynamic information generated by executing a program binary. Runtime features include system calls, files accessed, network connections, created mutex, visited URLs, and dynamic values generated. These runtime behavioral features cannot be captured simply by analyzing code statically. Moreover, they play an important role when the source code is not available.

The main goal of behavioral features is to reveal the hidden functionality of the software that might contribute to the attribution process. The major advantage of behavioral features is that they are resilient to most techniques commonly used for software protection, even advanced obfuscation (e.g., code flattening).

Although behavioral features can provide good results, incorporating them into the authorship attribution system is challenging. Behavioral analysis techniques and tools differ depending on the software being analyzed. Employing behavioral features across a large dataset can be problematic because, to capture them, each binary has to be executed in a controlled environment, which is time consuming and often requires special settings to trigger hidden functionality. Hayes and Offutt [49] showed that system calls do not perform well if programs have few or no unique system call. Proprietary software and malware authors can use more advanced techniques, such as tuning the system for a virtual environment and active debuggers, to deceive analysis based on behavioral features. Despite these challenges, behavioral features are promising and deserve further research attention.

3.1.5 Application-Dependent Features. The preceding features are application independent, as they can be extracted from any code given the availability of appropriate tools and techniques. Application-dependent features are defined to better distinguish an author's style in a given application development realm.

These features are static application features that can be extracted from various application-specific binary components, such as linked libraries, resources (e.g., images and sound files), property files (e.g., plist files in case of iOS mobile applications), log files, and permissions files (AndroidManifest.xml in the case of Android mobile applications). These features also contain various items of metadata, such as the compiler version used to create the executable files and packaging tools.

In contrast to behavioral features, these features can be extracted without executing the application itself. Their main goal is to reveal the authors' preferences with regard to the development context, such as external resources, development tools, and libraries.

Application-dependent static features are unique, and their extraction is cost efficient. They can play a crucial role in cases where the source code is not available and the extraction of behavioral features is prohibitively expensive. Despite their potential, these features have received little attention from the research community. Table 2 briefly summarizes the feature types along with their advantages and disadvantages.

3.2 Representations

In the domain of traditional linguistics, sample text is viewed mainly as a plain sequence of tokens, whereas software, either in the form of source code or binary, can be represented in many different ways for the attribution process. Owing to the unique characteristic features of software,

Table 2. Summary of Feature Types Based on Techniques Surveyed

Feature Type	Examples	Advantages	Disadvantages
Lexical	Lines of code Operands Variables Spaces Word frequencies Token frequencies Character N-gram Function names	Basic feature Language independent Extracted from source code and byte code	Dependent on number of training samples Relevant feature selection is a challenge Easily evaded with code formatters or obfuscation
Syntactic	Average function size Use of special macros Choice of data structure Choice of control structure Input statements Conditional statements Assignment statements	Builds strong author profiles Robust against code formatters and obfuscation Captures problem-solving approach of author	Language dependent Feature selection is a challenge Evaded by changing structure of code
Semantic	Loops Dataflow analysis Control flow analysis Algorithms implemented Procedure-dependent analysis	Develops a strong author profile Robust against code/data transformation techniques Language independent	Difficult to extract features Evaded by advanced identity-hiding techniques
Behavioral	System calls Files accessed Created mutex Visited URLs Dynamic values Network connections	Robust against software protection and advanced obfuscation Useful when source code is not available Reveals hidden functionality	Challenging to use Dependent on size of data Easily evaded by anti-analysis techniques Large numbers of false positives and false negatives
Application-dependent	Strings from: —Log files —Smali files —Properties files —Error message files	Cost efficient Works even if source code is not available Augments author profile	Requires advanced tools to extract features Not obfuscation resilient

a variety of representations is possible. For example, source code can be viewed as a simple sequence of tokens or a control flow in the form of a graph. Different representations can be employed throughout the authorship attribution process to extract the features and represent the extracted features in a form suitable for automated analysis. This section lists the most commonly used representations in the code attribution domain derived from previous studies [3, 4, 74] and the nature of the software used.

Tokens. The basic representation of a feature is the token. A token by itself represents a single extracted feature. Lexical and syntactic features can be obtained by representing source/binary code files in the form of tokens. For example, a token can be a word, character, symbols, function names, keywords, operators, operands, and basic blocks. Specific tools are not required to extract this representation, because it can be obtained by applying simple tokenization techniques on source/binary code files. However, this feature does not capture the semantics of the code or any context-specific information. It focuses only on plain tokens irrespective of their functions.

Owing to its ease of implementation, it has been widely used by researchers, starting from the first basic software science metrics of Halstead [45]. Researchers often use frequency or existence to measure features represented in the form of tokens. Donaldson et al. [30] used the frequency of token-based features, such as number of variables, subprograms, input statements, and conditional statements. Oman and Cook [69] used Boolean measurement based on the existence of token-based features, like blank lines, spacing, and comment format.

Strings. A sequence of tokens composes a string. Instead of analyzing tokens separately, they can be combined as strings that can be processed further by applying different string analysis techniques. Many lexical and syntactic features, such as keywords, operators, operands, and variables, can be combined together to form strings. However, certain features (e.g., number of blank lines, spacing, and indentation) do not form meaningful strings. A string analysis of the tokens can provide a better idea of the author's writing style than tokens embedding some context-specific information. Similarly to tokens, however, they cannot capture code semantics. String representation allows the leveraging of string analysis algorithms for code attribution. A few studies have used this (e.g., JPlag [72]). Overall, however, string representation has not been used widely in authorship attribution research.

N-grams. An N-gram is a sequence of n terms combined together to form a given token sequence. N-grams have been used extensively for detecting software similarity and in authorship attribution systems. The advantages of this representation are clear: N-grams preserve contextual information, lend themselves easily to both source and binary code analysis, produce computationally simple author profiles, and are program language independent. This representation is promising and has been widely used by researchers [13, 37, 63, 78].

Although experiments have shown that the N-gram representation is reliable, some factors need to be considered while incorporating it. Researchers generally select the size of the N-gram arbitrarily, because there is no proven method to determine the best choice of n. In practice, this typically leads to extensive experiments with various values of n on a given dataset to determine the one that produces the desired results of attribution. In such cases, features extracted as N-grams might produce biased results depending on the value of n. Thus, the validation of various sizes of N-gram on different datasets is needed while designing the attribution system [37].

Idioms. An idiom is a short sequence of instructions with possible wild cards [76]. This representation reflects low-level details of code instructions and can be extracted from assembly language code by disassembling the binary. Idioms represent instruction/statements unique to the particular language and thus are used to express language-dependent syntactic features. This representation has not been used widely by researchers; however, it can be explored for attribution on programming languages separately.

Graphs. A graph is a collection of nodes representing basic blocks of a program connected with directed edges. This representation is used to model the flow and structure of the program. Unlike token-based representation, graph representation helps reveal the underlying semantics of the program. Graphs can be used for attribution in different forms and can be constructed from both the source and the binary code (e.g., a program dependence graph (PDG) [66], CFG [49], system call dependence graph [88], and API call graphs [19, 22]). Typically, features represented or extracted by using graph representation are obfuscation resilient. Indeed, avoiding certain system calls while retaining overall functionality is difficult, but, still, the resulting features are more reliable compared to other representations.

A more restricted form of the graph is a tree, an undirected acyclic graph that can be used to represent the structure of the data. Trees can capture only the underlying structure and syntax

of the program. Abstract syntax trees (ASTs) and parse trees are examples of this representation [17, 25].

3.3 Embeddings

A single representation might not be enough to express the diverse nature of a software program. In these situations, researchers leverage embeddings (i.e., combinations of different representations). Sets, collections of objects irrespective of their order, and vectors, and ordered lists with a fixed number of objects, are common examples of embeddings.

Unlike all of the representations discussed earlier, sets and vectors cannot be directly extracted from the source code or binary. Rather, they are used to combine features represented in any form. A set can be a unique collection of N-grams, tokens, strings, and API calls. Vector representation is often used to leverage the complementary nature of different types of features, such as combining lexical and syntactic features [9, 30, 42, 70].

Examples of the discussed representations are given in Table 3.

3.4 Attribution Models

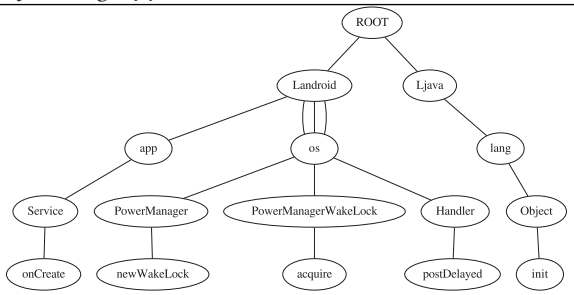
Once features have been extracted and represented in the desired format, the next step is to select the attribution model most suitable for the given context. Typical authorship attribution is a multi-class single-label categorization task where a number of training samples of work by a fixed number of known candidate authors are available. A classification model is trained based on this dataset, where each class represents a candidate author. The model is then employed to classify the unknown sample as one of the learned classes. Depending on whether the training samples are treated cumulatively or separately, profile- or instance-based attribution models can be employed.

3.4.1 Profile-Based Models. A profile-based attribution model produces a single unique style of representation per author. The author's style is represented through a set of common characteristics derived through the analysis of all available samples of the author's work. Differences in samples belonging to the same author are not considered. In this way, a unique author profile, also known as author fingerprint or author signature, is generated. Typically, the author fingerprint is a vector or a set of feature representations (e.g., N-grams, strings). However, other presentations are also possible. For example, an author fingerprint comprising a histogram distribution of features has been proposed [60, 80]. For every candidate author, the classifier first produces normalized interpolated histograms illustrating the distribution of features. This set of an author's histograms is then compared to the histogram distribution of the unknown sample.

A profile-based approach, SCAP, proposed by Frantzeskou et al. [37], is one of the most commonly used. The SCAP approach generates a profile using the most frequent byte-level N-grams extracted from the source code samples of an author. This language-independent method has proved to be effective even with a limited number of training samples and the absence of comments from the code.

3.4.2 Instance-Based Models. An instance-based attribution model produces separate styles of representation per sample. Every composition belonging to the same author is considered an independent unit. This approach allows for the consideration of differences between samples of the same author. The unknown sample whose authorship is to be determined is then compared to every sample in the corpus. This approach has been found to be especially effective in cases when some authors have limited numbers of samples [20]. Considering that finding a reliable dataset for code attribution is challenging, many studies in this domain have employed this approach, including one of the earliest attempts to analyze the programming style of an author by Oman and

Table 3. Examples of Feature Representation

Raw features extracted from Android API calls	
	Landroid/app/Service;->onCreate, Landroid/os/PowerManager;->newWakeLock, Landroid/os/PowerManager$WakeLock; ->acquire, Ljava/lang/Object;-><init>, Landroid/os/Handler;->postDelayed
Representation	**Example**
Tokens	Landroid {4} app {1} Service {1} onCreate {1} os {3} PoWerManager {2} ...
Strings	Landroid/app/Service;->onCreate, Landroid/os/PowerManager;->newWakeLock, Landroid/os/PowerManager$WakeLock;->acquire, Ljava/lang/Object;-><init>, Landroid/os/Handler;->postDelayed
N-grams	<>-Landroid Landroid-app app-Service Service-OnCreate OnCreate-<> Landroid-os os-PowerManager PowerManager-newWakeLock ...
Idioms	Landroid/*/* {4} Ljava/*/* {1} Landroid/app/* {1} Landroid/os/* {3} Ljava/lang/* {1}
Graphs	

Cook [69], where author clusters were analyzed by calculating the similarity between each pair of training samples.

The most recent studies in the field [17, 18, 74] have also evaluated the performance of the instance-based authorship attribution model on attribution tasks.

3.4.3 Selecting an Attribution Model. The main difference between profile- and instance-based models is the way in which they handle samples for attribution tasks. A profile-based model is a classification model of a generative nature that produces one cumulative representation of all training samples by an author. It emphasizes the modeling of the distribution of the author class and classifies by computing the likelihood of a new sample. However, an instance-based model is a classification model of a discriminative nature that treats each sample related to an author as a separate instance. It emphasizes learning the class boundaries without attempting to model the entire underlying class density. To design an efficient authorship attribution system, it is important to understand the performance of both models in different scenarios. When deciding on the best approach for analysis, the following points should be considered:

— *Author style*: Profile- and instance-based models attempt to handle different kinds of author styles. Considering that a profile-based model represents the stylistic features of each author and does not consider differences between samples by the same author, it deals with the generic style of each author and is thus more suitable for dealing with consistent styles (e.g., experienced programmers [49]). However, an instance-based model represents the stylistic features of each training sample for each author, and the differences among the samples are considered. It thus tackles the separate style of each sample by an author.
— *Number of available samples*: Profile-based models consider all training samples cumulatively. Hence, this approach is fairly insensitive for small datasets, whereas instance-based models require a sufficient number of training samples of moderate size.
— *Length of the profile*: The length of the author profile (i.e, the number of most frequently occurring features extracted from an author's code) plays a crucial role in the profile-based attribution model. Frantzeskou et al. [37] proposed selecting the L most common features extracted from the source code, where L represents the size of the profile. However, this can produce biased results as features are removed arbitrarily depending on the value of L while creating the author profile. The study evaluated the performance of the system by varying the profile length parameter L. Although it failed to provide the optimal value for parameter L, the authors concluded that the accuracy of the profile-based model increases with the size of the profile. Based on this observation, Burrows et al. [15], in a comparative analysis, suggested using the maximum value of parameter L (i.e., by including all features in the profile). A follow-up study by Tennyson and Mitropoulos [85] concluded that author profiles generated by excluding features appearing less frequently provide better accuracy than those generated by selecting the L most common features. In case of instance-based models, the length of all training samples should be uniform, because all samples must be adequately represented to produce reliable stylistic features.
— *Imbalanced representation of authors*: Ideally, all authors should be represented by similar numbers of code samples. Considering that this is often impossible, the attribution model should compensate for it. A systematic study conducted by Chatzicharalampous et al. [20] showed that both attribution models are negatively affected by the imbalanced datasets. Profile-based models suffer when the total size of all training samples for each author is not uniform across authors. This is because the non-uniform distribution of total training samples for each author can produce biased author profiles. Other factors, such as the number of training samples per author and the size of each sample, do not affect profile-based

models. In case of instance-based models, as each sample plays a role in the classification, the number and the size of samples per author need to be uniform for a balanced dataset. In case of balanced or nearly balanced datasets, both models yield stable performance, although the instance-based model provides better results because it considers differences between samples for classification. In case of a moderately balanced dataset (i.e, a dataset where half the authors have insufficient training samples), instance-based models still outperform profile-based models. Finally, for heavily imbalanced datasets, the performance of the instance-based model degrades rapidly owing to insufficient training samples for each author, whereas author profiles can still be created even if the number of training samples for each author is small. Hence, the profile-based model is the best solution for imbalanced datasets.

Profile- and instance-based methods are complementary in nature; therefore, in certain cases, their combinations can be beneficial—for example, by training over instances, and then using the results to create a profile from the average values of the extracted features.

3.5 Attribution Methods

Once the attribution model has been designed, the next step is to select a suitable method to compare author profiles or samples to identify the author of an unknown sample. Depending on how the comparison is performed, the methods of attribution can be categorized as follows.

3.5.1 Similarity-Based Methods. Similarity-based methods are the simplest and most common and widely used methods for attribution tasks. They attempt to calculate pairwise similarity between unobserved code and all samples used for training or all author profiles in the training set to make a final decision. The author with the highest similarity with an unknown sample is identified as its probable author. These methods can be used with profile- and instance-based attribution models.

Many code attribution studies have used similarity-based methods of attribution. The exact method for calculating similarity depends on the nature of the authorship task and the attribution model. The most common examples of such methods include clustering techniques [63], nearest-neighbor algorithms [60], distance-based algorithms [37, 53, 69], and ranked-based techniques [80].

Frantzeskou et al. [37] compared the performance of two similarity measures: the relative distance measure based on comparing the normalized frequency of N-grams proposed by Kešelj et al. [53], and the simplified profile intersection (SPI) measure based on the number of common N-grams between an unseen code sample and the author profile. The study concluded that the SPI measure is simpler and better than the relative distance measure for authorship attribution.

Burrows and Tahaghoghi [13] assessed the performance of attribution with five similarity measures: Okapi BM25, cosine, pivoted cosine, language modeling with Dirichlet smoothing, and a custom similarity measure designed by them. They showed that 6-gram representation yields the most accurate results, whereas Okapi BM25 is the most accurate similarity measure. With the exception of the Dirichlet measure, other similarity measures performed with similar efficiency.

3.5.2 Vector Space Methods. One of the most efficient representations in code authorship attribution is vector representation. Vector space methods consider each sample code as a vector in a multivariate space. Feature vectors extracted from the sample codes can be applied to a variety of machine learning algorithms, including support vector machines (SVMs), principal component analysis, decision trees, neural networks, genetic algorithms, and classifier ensemble methods.

These methods can usually handle noisy, high-dimensional, and sparse data, permitting expressive representation for the codes. Instance-based attribution models use these methods.

Vector space methods have gained popularity in recent studies. The most common approach is to represent metrics in the vector format followed by the use of different machine learning techniques (e.g., discriminant analysis [29, 48, 49], SVM [68, 74], and random forest classification [17, 18]).

3.5.3 Probabilistic Methods. Probabilistic methods compute the probability that an unknown sample belongs to a candidate author from a given set of authors. The author with the maximum probability measure is identified as the probable author. These methods determine the probability $P(C|A)$ of code C belonging to candidate author A. They are mostly used with profile-based attribution models owing to their probabilistic nature. However, we can use probabilistic techniques as a similarity measure to calculate the similarity between unseen code and all training samples as well. The probabilistic algorithms employed by researchers include the naive Bayes classifier [55] and logistic regression [8]. In some cases, the SVM algorithm, which is generally used as a vector space method, can be transformed into a probabilistic classifier. A relevance vector machine functionally identical to an SVM with a probabilistic sparse kernel model can be used for such probabilistic classification.

3.5.4 Meta-Learning Methods. General-purpose classification algorithms might not always be effective on the authorship attribution problem. In these situations, researchers have the option of developing a custom algorithm designed for a particular task by modifying machine learning techniques, such as deep learning, or creating an ensemble of several methods to improve accuracy. Of these options, the ensemble approach has been used most often.

Krsul and Spafford [58] used a combination of discriminant and graphical analysis to discard irrelevant features followed by classification using a neural network algorithm. Layton et al. [63] relied on the calculation of similarity between documents based on the byte-level N-gram approach followed by document partitioning using fuzzy c-means MST clustering (FMC). Rosenblum et al. [74] leveraged the distance metric generated by the large-margin nearest-neighbors algorithm to cluster unlabeled samples using the k-means algorithm.

Meta-learning methods are not common but have been used in both profile- and instance-based models.

3.5.5 Selecting a Machine Learning Algorithm for Attribution. A classical problem in code authorship attribution is to attribute an unknown sample to a known author based on a training dataset of samples of known authorship. This classification is conducted while keeping key points in mind, such as the objective of attribution, the author's style, availability of a sufficient number of known samples, and sample length. The two most important factors from a machine learning perspective affecting the classification process are the (i) choice of appropriate features and (ii) selection of an effective classification technique. In terms of feature selection (as discussed in Section 3.1), lexical, syntactic, and semantic features have been extensively used [3, 8, 13, 17, 18, 29, 32, 39, 48, 49, 55, 58, 60, 63, 68, 69, 74, 80, 91]. With regard to feature representation, researchers commonly use tokens [8, 13, 17, 18, 29, 48, 58, 60, 80], with a growing consensus on the use of N-grams for consistent results [17, 39, 41, 55, 59, 63, 74, 91]. However, there is no common agreement among researchers regarding the best machine learning classifier for code authorship attribution.

Prevalent studies have focused on testing the classification accuracy of separate machine learning methods, and rarely in comparison with other methods. This, together with the variation in the experimental environment and the evaluation methodology, makes it challenging to draw any conclusions about the performance of these methods. Only a few studies in the code authorship attribution domain have attempted to compare the accuracy and efficiency of classifiers [6, 44, 55,

92, 93]. Yet, none of the studies has performed a comparative analysis of classifiers with respect to their suitability across attribution problems.

In general, there are two high-level ways to approach a classification problem in machine learning: supervised learning and unsupervised learning. In the context of code authorship attribution, supervised learning relies on ground truth from known sources, such as the GitHub repository. The collected samples are then used to learn a model (profile- or instance based) for each author's programming style. In unsupervised learning, the analysis is conducted without the ground truth. In this method, the new code sample is analyzed to find subsets that appear to have been written by the same author [63, 69, 74]. Unsupervised machine learning techniques, such as clustering, work best in cases where only unlabeled data are available (e.g., malware author attribution). Unsupervised machine learning has other benefits as well–it does not need training and performs well even with a small and imbalanced dataset.

Most work surveyed in this article uses supervised learning methods. In this task, most profile-based approaches use a nearest-neighbor classifier to compare an author's profile to those of all candidate authors based on a similarity measure [39, 60, 80]. A nearest-neighbor classifier is simple to implement and is flexible in terms of choice of distance. Frantzeskou et al. [39] proposed a dissimilarity measure that showed that the classifier performed well in cases where only a limited number of very short samples for each author were available for training.

Lange and Mancoridis [60] also used the nearest-neighbor algorithm with a general distance measure to rank a list of authors according to similarity of style. The authors used the entire code at a time to train the classifier with the aim of reducing the number of comparisons that the classifier needed to make. However, nearest neighbor did not yield the best performance when large datasets with numerous features were employed, because it took longer to run. However, Shevertalov et al. [80] focused on improving classification using metric discretization and performed experiments using IB1 (a simple nearest-neighbor classifier) to achieve a moderate accuracy of 75% but with improved performance. In a similar way, data discretization optimization can be applied to advanced classification algorithms to produce better results.

Other algorithms used in profile-based approaches are naive Bayes and logistic regression. Kothari et al. [55] compared naive Bayes and voting feature intervals (VFI) classifiers. Similar to the findings of Shevertalov et al. [80], they showed that the VFI classifier that partitions a metric into intervals produces better classification results and requires relatively less time to classify authors. Gull et al. [44] also performed a comparative analysis of naive Bayes, the decision tree (J48), SVM, and *k*-nearest neighbors (KNN) on authorship attribution. Their results showed that naive Bayes converges more quickly than other discriminative algorithms on a moderate-sized dataset. Bandara and Wijayarathna [8] applied logistic regression. Being a binary classifier, it requires *k* classifiers to train, where *k* is the number of authors in a given dataset. Hence, logistic regression can be a good choice for closed-world problems with a limited number of authors.

However, instance-based approaches use SVM, discriminative analysis, random forests, decision trees, and neural networks. Of these, the SVM classifier has the ability to effectively handle large numbers of training samples [59, 68, 74, 91]. It can easily process hundreds and thousands of features, thus allowing the use of all features from a code sample as input, instead of having to carefully select specific features. Moreover, the training time and accuracy are better than those of other methods. Another method of choice among researchers is discriminant analysis [29, 48, 58]. It is similar to regression analysis and works within a closed environment. In case of a limited number of samples for each author, cross-validation is specified in discriminant analysis to obtain a realistic estimation of performance [29, 48].

To date, only Elenbogen and Seliya [32] have utilized the C4.5 decision tree model to extract data patterns in the form of rules to predict plagiarized student programming assignments. The

random forest, which is an ensemble of decision trees, is gaining more popularity in code authorship attribution [16, 17, 41, 81]. It is useful in several scenarios, such as for simple classifications (i.e., selecting the exact author), rank-based classifications when performing relaxed attribution (i.e., with a reduced number of suspected authors), and when dealing with a code sample written by multiple authors. The main objective in these scenarios is to identify the most likely author of a code fragment. Further, once trained, it provides the best tradeoff between accuracy and processing time, hence outperforming other models, including neural networks.

Alsulami et al. [6] and Yang et al. [92] applied different forms of neural networks to authorship attribution and compared their models to classical machine learning approaches. The evaluation illustrated that with a moderate problem size (number of authors), adequate training data, reasonable training time, repeated experiments to adjust the control parameters, neural network models can achieve comparable accuracy with a tolerable overhead. Attributing code authorship by using neural networks and deep learning is an emergent area that requires more research and optimization in classification.

3.5.6 Dataset Selection. Reliability (i.e., consistency of experiments in a study) depends heavily on the employed experimental practices and evaluation data. The high quality of a dataset that properly represents practical programming practices not only shows a method's ability to attribute code but also provides insight into its potential effectiveness in the deployed operating environment.

With a lack of benchmark datasets in the late 1990s, researchers resorted to using archives of programming assignments by students [15, 39, 58]. A large amount of time was spent only on systematizing and categorizing the data. Sharing these data was challenging owing to privacy concerns, and generally required anonymization, which removed valuable information (i.e., comments) commonly used for authorship attribution.

Hence, the availability of publicly assessable data is crucial. Tennyson and Mitropoulos [85] explored the use of programs written by textbook authors for authorship attribution. Although the samples were freely available, their authorship was unclear because there was no evidence that a single code had been authored by one person and all samples were written by the same programmer. The detection of multiple authors having written a single block of code is a challenging problem [17, 68], because such researchers typically use datasets with single authors or discard samples from multiple authors.

With the growing popularity of open source repositories, researchers started leveraging this data. Frantzeskou et al. [39] downloaded source code samples from freshmeat.net.[1] Lange and Mancoridis [60] and Sheverlatov et al. [80] used free software projects hosted on SourceForge.[2] Bandara and Wijayarathna [8] considered planet-source-code.com[3] as a data source.

Although these data repositories still do not offer a clear distinction among programs written by multiple authors, they offer a rich and diverse pool of source and binary code.

Most recent attribution studies [6, 18, 81] have used programs developed during the Google Code Jam (GCJ),[4] an annual international coding competition hosted by Google. Given a set of problems, the contestants need to provide solutions in a restricted time. The availability of statistical information, such as popularity of programming language, contestants' skill levels, and their nationalities make data from the GCJ useful for authorship attribution and authorship profiling.

[1] http://freshmeat.sourceforge.net/.
[2] https://sourceforge.net/.
[3] https://www.planet-source-code.com/.
[4] https://code.google.com/codejam/.

Although there are many benefits to using these data, this practice has been extensively criticized mostly owing to its artificial setup [27].

With a limited number of samples per author and limited variety of programs of some languages (e.g., JavaScript), the GCJ is not a comprehensive source of data for attribution.

The authors [27, 91, 92] who approached the authorship attribution problem "in the wild" mainly used repositories found on GitHub.[5] Github is an online collaboration and sharing platform for programmers. As of June 2018, it reported nearly 30 million users[6] and 57 million repositories [34] (including 34 million public repositories[7]). Similar to other repositories, GitHub does not offer reliable facilities to differentiate code written by multiple authors and has significant noise in the data (e.g., junk code).

Furthermore, some authors used multiple sources to construct their datasets. For examples, Burrows et al. [15] used student data from school assignments and open source programs for evaluation.

For datasets of binary code, researchers have used compiled versions of programs from similar data sources, such as GCJ and GitHub [3, 5, 18, 68, 74]. Thus, challenges applicable to the selection of a dataset of source codes applies directly to that of binary code. Some authors [41, 59] focusing on authorship attribution in Android apps have faced challenges unique to Android application development. One such challenge is dealing with APKs signed by open or publicly available certificates. These certificates can be used by anyone, thus calling into question the known authorship of the APK. Further, generating an obfuscated dataset requires compiling Android application source code and employing obfuscation tools, which, in some cases, is not successful for reasons such as incompatible versions of the Android plugin, SDK version collision, and improper environment parameters set in the configuration files, resulting in some APK source code files being discarded. For Android malware datasets, both authors collected malicious Android APKs from the Koodous system,[8] which is an open source, collaborative, Web-based Android malware analysis platform, and verified the malicious nature of those APKs by VirusTotal.[9]

An overview of studies on authorship along with their classifications is shown in Table 4.

4 CHALLENGES

The presented spectrum of representations, features, and models for code authorship attribution clearly shows a diversity of solutions in code authorship attribution. However, over the years, these studies have also revealed a number of challenges that the area faces:

— **Closed-world problem:** In the software domain, attribution is a closed-world problem that has only one solution leading to one answer. The solution is restricted to known software authors whose codes are analyzed for authorship. For example, if profiles are built for n number of authors, the solution leads to one of them only. The solution is not generic and, thus, cannot be valid for other authors whose profiles do not exist in the system. One solution to this challenge is the use of unsupervised learning methods for attribution [63]. Adopting approaches like clustering allows new clusters to form whenever a new author's program sample is encountered by the system. Such an approach does not require training and performs well even with a small and incomplete dataset.

[5] https://github.com/.
[6] https://github.com/search?q=type:user&type=Users.
[7] https://github.com/search?q=is:public.
[8] https://koodous.com/.
[9] https://www.virustotal.com/.

Table 4. Classification of Code Authorship Attribution Studies

Author	Feature Type	Representation	Attribution Model	Attribution Method	Feature Selection Method	Classifier	Dataset
				Input type is Source Code			
Oman and Cook [69]	Lexical, Syntactic	Tokens, Vector	Instance based	Similarity based	Boolean measurement	SPSS-X Cluster Analysis	PASCAL code for three algorithms, each from six computer science textbooks
Krsul and Spafford [58]	Lexical, Syntactic	Tokens, Vector	Instance based	Meta-learning	Discriminant Analysis	Neural Networks	88 C programs with different problem domains from 29 known authors
Ding and Samadzadeh [29]	Lexical, Syntactic	Tokens, Vector	Instance based	Vector based	One-way ANOVA	Canonical Discriminant Analysis (CDA)	Java source code from different known groups
Frantzeskou et al. [39]	Lexical	Set of N-grams	Profile based	Similarity based	Most frequent N-gram	Nearest Neighbor using a proposed dissimilarity measure	Java and C++ programs from books and commercial programmers
Lange and Mancoridis [60]	Lexical, Syntactic	Tokens, Set of Histograms	Profile based	Similarity based	Genetic algorithm	Nearest Neighbor using a general distance	60 free software projects from SourceForge and three academic projects per 20 developers
Burrows and Tahaghoghi [13]	Lexical, Syntactic	Tokens, Set of N-grams	Instance based	Similarity based	—	—	1640 C programs by 100 authors
Kothari et al. [55]	Lexical	N-grams, Set, Histograms	Profile based	Probabilistic	Shannon's information entropy	Bayes and Voting Feature Intervals (VFI) Classifier	2110 code samples from open source projects and 220 from student assignments
Hayes [48]	Lexical	Tokens, Vector	Instance based	Vector space	ANOVA and Principal Component Analysis	Linear Discriminant Analysis (LDA)	C programs from professional programmers and students
Elenbogen and Seliya [32]	Lexical, Syntactic	Tokens, Vector	Instance based	Vector based	—	Decision Tree	83 programming assignments from 12 students
Shevertalov et al. [80]	Lexical	Tokens, Set of histograms	Profile based	Vector space	—	Nearest Neighbor	60 open source Java projects by 20 developers
Layton et al. [63]	Lexical	Set of N-grams	Instance based	Meta-learning	Most frequent N-gram	Fuzzy c-means MST Clustering (FMC)	over 700 phishing Website source codes

(Continued)

Table 4. Continued

Author	Feature Type	Representation	Attribution Model	Attribution Method	Feature Selection Method	Classifier	Dataset
Hayes and Offutt [49]	Lexical, Syntactic, Behavioral	Vector, Graph	Instance based	Vector space	—	—	15 C programs by professionals and 60 by students
Bandara and Wijayarathna [8]	Lexical, Syntactic	Tokens, Vector	Profile based	Probabilistic	Sparse auto-encoder	Logistic Regression	Five Java source code datasets from different known sources
Caliskan-Islam et al. [17]	Lexical, Syntactic	Tokens, Abstract syntax tree (AST)	Instance based	Vector space	Information gain	Random Forest	C/C++ programs by 1,850 authors from Google Code Jam (GCJ)
Wisse and Veenman [91]	Lexical, Syntactic	N-gram, AST	Instance based	Vector space	Most frequent node N-grams	Support Vector Machine	JavaScript source code samples from GitHub
Yang et al. [92]	Lexical, Syntactic	Tokens, AST, Vector	Instance based	Vector space	—	Back-propagation neural network based on particle swarm optimization (PSOBP)	3,022 Java source code files with 40 authors from GitHub
Alsulami et al. [6]	Structural syntactic features of AST	AST, Vector	Hybrid model	Vector space	—	Deep Neural Networks	700 Python files by 70 authors (GCJ) and 200 C++ files by 10 authors (GitHub)
Zhang et al. [93]	Lexical, Syntactic, Semantic	Word-level and Character-level N-gram	Instance based	Vector space	—	Sequential Minimal Optimization (SMO)	8,000 Java programs by eight programmers (GitHub), 502 Java programs by 53 programmers (planet source code)
Simko et al. [81]	Lexical, Syntactic	Tokens, N-grams, AST	Instance based	Vector space	—	Random Forest	214 C programs by five authors from GCJ
Input type is binary file							
Rosenblum et al. [74]	Syntactic, Semantic	Idiom, N-grams, Graphs	Instance based	Vector space for classification, Meta-learning model for clustering	Mutual information between the feature	SVM for author identification and *k*-means clustering for author clustering	Program binaries from GCJ and from an undergraduate course
Alrabaee et al. [3]	Syntactic, Semantic	Vector, Graph	Profile based	-	-	-	C/C++ programs from GCJ

(Continued)

Table 4. Continued

Author	Feature Type	Representation	Attribution Model	Attribution Method	Feature Selection Method	Classifier	Dataset
Caliskan-Islam et al. [18]	Syntactic, Semantic	Tokens, N-gram, Graph	Instance based	Vector space	Information gain	Random Forest	Compiled C++ code from GCJ, C/C++ repositories from GitHub and Nulled.IO hacker forum
Alrabaee et al. [5]	Semantic	Graph	Instance based	Similarity based	—	—	30 C/C++ programs complied in debug mode and from GCJ
Meng [68]	Syntactic, Semantic	Graph	Instance based	Vector space	Mutual information between major authors	Linear SVM	831,372 basic blocks, 170 binaries, and 282 authors
Gonzalez et al. [41]	Syntactic, Application dependent	Strings, Byte-level N-grams	Instance based	Vector space	—	Random forest	196,385 unique Android apps from various sources in 10 markets
Kulgutkar et al. [59]	Application dependent	Word N-grams	Instance based	Vector space	—	Linear SVM	1,559 benign Android apps by 40 authors, 262 malicious apps by 10 authors, 96 obfuscated apps by 9 GitHub authors

—**Lack of source code:** This challenge is more common in the malware domain. A majority of research, both academic and industrial, on malware defense and malware analysis has focused on binary because malware samples are distributed in binary form. Very rarely do researchers and malware analysts have access to source code. The lack of the availability of malware source code has resulted in a poor understanding of the evolution and properties of malware, such as origin and authorship. Therefore, malware code authorship attribution is a challenging task where only binary code is available, as the compilation process strips the code of certain surface characteristics specific to the malware author [74]. However, a recent study focused on features that survive the compilation process [18]. There are some syntactical features in source code that can be recovered by decompiling the binary. These features are robust against basic obfuscation and optimization techniques. More research is required to extract such features directly from the compiled binaries.

—**Use of automatic code generation tools:** Numerous tools are available online for automatically generating code that the authors otherwise have to write themselves. Code from such tools does not provide any useful information regarding a particular author because code thus generated cannot capture the exact programming style of that author. Moreover, in case of malware, authors extensively use numerous code generators and toolkits to produce a large number of malware variants. One solution to answer this challenge is to utilize software clone-detection techniques to first filter out duplicate code from known code generation tools. The clone-detection technique should be able to filter fragments that differ in variable names, parameters, and rearranged statements.

—**Code obfuscation:** Code obfuscation has the ability to lower the accuracy of traditional code authorship attribution methods. The authors of benign software commonly utilize code obfuscation for legal purposes, i.e., software protection, while making use of various obfuscation tools to make their code willfully ambiguous and harder to understand. Various obfuscation techniques like renaming, string encryption, control flow obfuscation, and code morphing can be applied to hide identity. One of the best ways to extract features from an obfuscated code is to first deobfuscate it—identify the obfuscation technique applied and remove the obfuscating patterns from code. Another way is to extract obfuscation-resilient features. These features have not been fully explored by researchers. More work is required to identify features resistant to various obfuscation techniques and tools. Such features should be selected carefully; otherwise, this could result in more false positives.

—**Tool dependence:** Feature extraction is a crucial step in authorship attribution. The features capture distinctive aspects of an author's programming style. Authorship attribution techniques are dependent on the accuracy of tool(s) for feature extraction and classification. Tools can be used to generate call graphs from binaries, which can be then used to extract specific features like API calls. Thus, the feature extraction process is dependent on the accuracy and efficiency of a given tool. If the tool cannot return correct results, this affects the overall accuracy of the authorship attribution technique. To address this problem, the accuracy of each dependent tool can be measured and compared before using it for attribution.

—**Code reuse:** Reusing existing code, components, and third-party libraries is common practice among authors of both malware and legitimate software. Code reuse is intended to save time and resources by taking advantage of assets already created. However, every new malware is created at a faster pace than a new, legitimate software. The new malware is created either by modifying existing malware with a few tweaks or reusing existing code components. Code reuse techniques are also used to hide malicious program behavior and author identity from analysis [28]. This eventually leads to false alarms in systems that

aim to detect code or author characteristics. Experts believe that most novice programmers make use of code reusability. To identify code reuse, clone detection techniques or common code reused every time by an author can be extracted and profiled.

- **Scarcity of datasets:** Research in code authorship attribution suffers from a lack of open-benchmark datasets and uncertain labels like misclassified or unattributed software. Researchers have to manually collect software for specific authors and obfuscate them to create a small dataset for testing. No published data are available for training and testing authorship attribution models. Academic and industry researchers working in code authorship attribution should publish their datasets for the research community.
- **No prior knowledge:** Another challenge that is more common with malware is that there is no way to baseline the methods, tools and techniques that malware authors might utilize to write their code. While the benign software authors are obligated to follow quality code writing guidelines and utilize standard tools and methods, this is not the case with malware authors. Purely unknown malware are rare and attributing them is a challenge. Since the majority of unknown malware are variants of existing malware, the common functionality derived could be baselined.
- **Collaborative environment:** Akin to legitimate software development, the underground economy works in groups, and it is possible that the same malicious code is written by multiple authors. In this case, it is not easy to classify the author. Rather, attributing code to a particular group of authors is more feasible. Studying the segmented pieces of a code sample instead of the entire sample is another approach to solve such a problem [68].
- **Evolving author's programming style:** The author's programming style keeps evolving over time. This evolution comes from the level of expertise and use of different programming languages. With more experience in programming, the programmers writing style develops naturally. Many programming tools also impose naming conventions, parameter passing, and commenting styles [58]. All these factors influence the author's program writing style, rendering it varied. One feasible solution is to collect and analyze the latest piece of work from the author or employ a self-learning feature in the system to learn the evolving programming style of the author.

The above questions remain open for code authorship attribution. From our perspective, the most important issues in code authorship attribution are a lack of source code and obfuscation. An analysis of source code is easy, and gives more details on the style of writing, which leads to better fingerprinting, whereas extracting author fingerprinting details through binary analysis is more resource intensive. The analysis becomes more difficult when obfuscation techniques are applied on the code. These obfuscation techniques are used to hide the identity of the author and, thus, affect the accuracy of solutions to code authorship attribution. Another important concern is a lack of acknowledged benchmark datasets. This is the main obstacle to the development of new methods and for the comparison with existing methods in malware authorship attribution. Apart from these, other important research questions to address are the types of information or features to be extracted to clearly identify the author of malware, the number of code samples sufficient to build an author profile, the requisite size of unseen code to predict its author, the definition of noise (like reuse or similar code) in the dataset, dealing with noisy samples, the extent of training needed, and the accepted confidence level for accuracy.

5 CONCLUSION

Code authorship attribution is necessary to identify the authorship of a given software code. It has practical implications for the detection of software theft, digital forensics, and malware analysis.

In this review paper, various code attribution features, their representations, and the models used to analyze them are presented. This is the first ever attempt to provide a comprehensive review of this topic. Although plagiarism research is historically related to authorship attribution research and shares in some methods, the goals are different. Code authorship attribution has evolved from a basic software science theory to more complex methods based on API calls and dependency graphs. Prevalent studies extract software metrics based on requirements, datasets, and programming environments. However, extracting software metrics is not easy as the source code is not always available. Moreover, code formatters and obfuscators are used to hide author identity. Thus, binary code analysis plays an important role. Most research has focused on identifying well-known authors but more work is required to identify unknown authors.

A comparative summary of existing work is presented that highlights the main features and contributions of research in the area. This article also discussed challenges or open questions in the area of code authorship attribution in general, and in the malware domain. The survey identified issues inherent to code authorship attribution and revealed the need for more research in authorship attribution for malware.

There are no "silver bullet" solutions yet for authorship attribution analysis of malware. However, many researchers have proposed multiple solutions at different levels of abstraction. More research methods focusing on the binary level need to be explored.

APPENDIX

A COMPARATIVE ANALYSIS

In Table 5, we provide a comparative analysis of the surveyed techniques on the basis of the extracted features and the contribution made by each to code authorship attribution.

Table 5. Summary of Reviewed Research

Author	Year	Features	Contribution
		Software Measurement	
Halstead [45]	1972	Software science metric: —Number of unique operators —Number of unique operands —Total number of occurrences of operators —Total number of occurrences of operands	First to propose the measurable properties of software and the relations between them
Bulut and Halstead [11]	1973	Software science metric	First to propose the counting algorithm to count the software metric for FORTRAN programs
Halstead [46]	1975	Software science metric	Experimental validation of software science
McCabe [67]	1976	Number of independent control path segments	First to propose a metric to measure the computational complexity of a program based on its structure
Fitzsimmons and Love [35]	1978	Software science metric	Review of major studies related to software science theory
Curtis et al. [24]	1979	Three software complexity measures	Evaluated performance of three metrics: Software science, McCabe, and number of statements to measure psychological complexity

(Continued)

Table 5. Continued

Author	Year	Features	Contribution
Hamer and Frewin [47]	1982	Software science metric	Experimental validation and a critical analysis of software science theory
Albrecht and Gaffney [2]	1983	Software science metric	Analyzed the relation among the size of a programming system, its development efforts, and Halstead's software science metric
Shen et al. [79]	1983	Software science metric	A critical analysis of Halstead's software science theory and review of related studies
		Plagiarism	
Ottenstein [70]	1976	Software science metric	Developed plagiarism detection system for FORTRAN based on the software science metric
Donaldson et al. [30]	1981	Variables, subprograms, input statements, conditional statements, loop statements, assignment statements, calls to subprograms, order of the statements	A system based on structural analysis to detect similarities between programs written in FORTRAN, COBOL, and BASIC
Grier [42]	1981	Total lines, code lines, comments, multiple statements, constants and types, unused variables, used variables, procedures and functions, control flow statements such as for repeat, while, goto, and software science metrics	Developed system "Accuse" for plagiarism detection in PASCAL
Berghel and Sallach [9]	1984	Lines of code, integer variables, variables, keywords, continuation statements, initialization statements, assignment statements, declaration statement, assignment along with software science metric	Investigated 15 complexity metrics to detect similarities between FORTRAN programs—confirmed that the soft science theory has no unique practical value
Faidhi and Robinson [33]	1987	Number of characters per line, comments, average function length, reserved word count, average identifier length, number of labels, goto statements, program intervals, User-defined identifiers, program structure, program impurity, etc.	Proposed six levels of program modifications for plagiarism detection
Parker and Hamblen [71]	1989	Not applicable	A survey of plagiarism detection systems
Aiken [1]	1994	Hash values of documents *k*-grams	An advanced system for detecting plagiarism in languages like C, C++, Java, and PASCAL
Verco and Wise [86]	1996	Not applicable	Compared the performance of attribute-counting- and structure metric–based systems
Prechelt et al [72]	2002	Token strings extracted from source code	An advanced plagiarism detection system for Java, C#, C, C++, Scheme, and natural language text
Liu et al. [66]	2006	Program dependence graph	GPLAG: A system to detect plagiarism by using program dependence graph analysis of source code
Walenstein and Lakhotia [87]	2007	Not applicable	Analysis of challenges in detecting similarities between malware samples
Ji et al. [52]	2008	Token sequences extracted from Java bytecode	Java plagiarism detection system using bytecode based on adaptive local alignment

(Continued)

Table 5. Continued

Author	Year	Features	Contribution
		Authorship Attribution	
Oman and Cook [69]	1989	Typographical or characteristics od layout style such as blank lines, spacing, line length, indentation, keywords, identifiers, and comment format	First to propose statistical analysis of style markers and clustering approach for authorship attribution
Spafford and Weeber [82]	1993	Proposed features for binary and source code	Described the theory of software forensics
Sallis et al. [77]	1996	Not applicable	Review of authorship attribution studies
Krsul and Spafford [56]	1997	Proposed more than 50 metrics classified into program layout, style, and structure metrics	Clarified the difference between different authorship tasks; proposed using techniques of authorship analysis to enhance real-time intrusion detection systems
Ding and Samadzadeh [29]	2004	Set of 56 metrics-based program layout, style, and structure	Investigated the role of 56 extracted metrics for Java authorship attribution; concluded that layout metrics contribute most to authorship attribution
Frantzeskou et al. [38]	2006	Most frequent byte-level N-grams	Introduced source code author profile approach based on byte-level N-gram and examined the role of comments for the task of authorship attribution
Lange and Mancoridis [60]	2007	Histogram distribution of 17 metrics with a combination of text-based and high-level syntactic features	First to use the set of normalized histogram distributions of code metrics as the author fingerprint
Burrows and Tahaghoghi [13]	2007	Keywords and operators (function words of a source code)	Performed N-gram analysis of keywords and operators extracted from source code; used information retrieval model useful for plagiarism detection as well
Kothari et al. [55]	2007	Style based and character sequences	Evaluated performance of character-based metrics for developing author profiles
Hayes [48]	2008	Number of program lines, number of unique operators and operands, average occurrence of operators and operands, number of comments per line	Evaluation of a consistent programmer hypothesis and authorship attribution using five features
Elenbogen and Seliya [32]	2008	Number of lines of code, number of comments, average length of variable names, number of variables, number of for loops/number of total loops, and number of bits in the compressed program	Presented a data mining–based approach to identify outsourced student assignment programs
Shevertalov et al. [80]	2009	Leading spaces and tabs, line length, number of words per line	First to evaluate performance of the genetic algorithm for the task of metric discretization
Layton et al. [63]	2010	Most frequent byte-level N-grams	Introduced unsupervised source code author profile (SCAP) method for determining phishing campaigns by applying clustering techniques
Hayes and Offutt [49]	2010	Number of lint warnings, unique constructs, constants and operands, average occurrence of operators and constructs, number of semicolons per comment, and dynamic measure of testability	Evaluation of consistent programmer hypothesis by analyzing different static and dynamic features

(Continued)

Table 5. Continued

Author	Year	Features	Contribution
Chen et al. [21]	2010	Program dependence graphs	Proposed a semantic approach to authorship identification and program theft
Rosenblum et al. [74]	2011	Stylistic features based on instruction sequence, control flow graph, external library calls	First to develop a binary code authorship attribution and clustering method by analyzing different representations of the structure of the instruction and control flow
Bhattathiripad [10]	2012	Dead code, programming blunders	First to propose use of dead codes and other programming blunders for authorship analysis
Bandara and Wijayarathna [8]	2013	Line length, word length, access level, frequency of comment types, identifiers length, white space and tabs, frequency of use of underscore, indentation	First to propose an unsupervised feature learning technique using a sparse auto-encoder
Chouchane et al. [23]	2013	Opcode N-grams	First to propose a malware attribution system based on metamorphic engine attribution
Burrows et al. [15]	2014	Not applicable	First to compare source code authorship attribution techniques
Alrabaee et al. [3]	2014	Data structures, algorithms, application programming interface, input/output functions, type of encryption, programming choices, register flow	First to propose a three-layered approach for binary authorship task: Preprocessing by filtering library and irrelevant code followed by syntactic and semantic attribution
Tennyson and Mitropoulos [85]	2014	Not applicable	Evaluated profile length parameter for the SCAP method
Caliskan-Islam et al. [17]	2015	Word unigram, nesting depth, branching factor, average parameters of functions, logarithmic ratio of each keyword, ternary operators, words, comments, literals, unique keywords, functions, macros to the file length in characters	Proposed the use of lexical and syntactic features extracted from the abstract tree representation of the source code; introduced the fuzzy parser to extract abstract syntax trees even from incomplete code
Wisse and Veenman. [91]	2015	Node expressions, statements, number of descendants of nodes, length of lists defined in nodes, string patterns on identifiers, type of comments, type of parent node of comments, length of comments, literal data types, number of tabs, spaces, return at a position	Proposed and evaluated a language-specific programmer identification technique for JavaScript source code
Caliskan-Islam et al. [18]	2015	Not applicable	First to conclude that syntactic features extracted from decompiled binaries survive various compilation, obfuscation, and optimization techniques
Alrabaee et al. [5]	2016	Semantic flow graph analysis by combining data and control flow analysis of functions	Developed a robust system—BinGold—for evaluating similarity between binary files based on semantic graph analysis
Meng [68]	2016	Features based on instructions, control flow, dataflow, context of the block	First to propose identifying multiple authors of a program by analyzing basic blocks

(Continued)

Table 5. Continued

Author	Year	Features	Contribution
Dauber et al. [26]	2017	Same as Caliskan-Islam et al. [18], with the addition of word bigram and trigram	First to study authorship of partial, small, segmented code samples
Alsulami et al. [6]	2017	Number of children in abstract syntax trees (ASTs), subtrees, AST depth, etc.	Proposed AST-based source code authorship attribution using deep neural network models
Zhang et al. [93]	2017	Import, if statements, loop statements, leading whitespaces of lines, percentage of blank lines, comments, keywords, add operations, average length of methods, operators, use-defined identifiers, punctuation	Proposed an approach to construct author profiles based on a source code logic model and a multi-level context model
Yang et al. [92]	2017	Ratio of blank lines to code lines, ratio of comment lines to code lines, percentage of block comments in comment lines, open braces alone in a line, variable naming without uppercase letters, average variable name length, average number of methods per class, max AST depth, etc.	First to introduce a back-propagation neural network based on particle swarm optimization to authorship attribution of source code
Alrabaee et al. [4]	2017	Not applicable	A survey of recent advances in binary authorship attribution
Simko et al. [81]	2018	Variable name, API calls, macros, literals, keywords	Evaluated a state-of-the-art attribution system against adversaries' tactics and capabilities
Gonzalez et al. [41]	2018	Number of methods, classes, fields, strings, usage of data structures, and array operation opcodes extracted from .dex file	First to attribute unlabeled Android apps by automatic learning and creating new profiles
Kulgutkar et al. [59]	2018	Unreferenced strings, DEX strings, application strings, and all strings	First to design an Android authorship attribution system by leveraging different string components of apps

REFERENCES

[1] Alex Aiken. 1994. MOSS: A system for detecting software similarity. Retrieved August 29, 2017 from https://theory.stanford.edu/~aiken/moss/.

[2] Allan J. Albrecht and John E. Gaffney. 1983. Software function, source lines of code, and development effort prediction: A software science validation. *IEEE Transactions on Software Engineering* SE-9, 6 (Nov. 1983), 639–648.

[3] Saed Alrabaee, Noman Saleem, Stere Preda, Lingyu Wang, and Mourad Debbabi. 2014. OBA2: An onion approach to binary code authorship attribution. *Digital Investigation* 11, 1 (May 2014), S94–S103.

[4] Saed Alrabaee, Paria Shirani, Mourad Debbabi, and Lingyu Wang. 2017. On the feasibility of malware authorship attribution. arXiv:1701.02711.

[5] Saed Alrabaee, Lingyu Wang, and Mourad Debbabi. 2016. BinGold: Towards robust binary analysis by extracting the semantics of binary code as semantic flow graphs (SFGs). *Digital Investigation* 18 (Aug. 2016), S11–S22.

[6] Bander Alsulami, Edwin Dauber, Richard Harang, Spiros Mancoridis, and Rachel Greenstadt. 2017. Source code authorship attribution using long short-term memory-based networks. In *Proceedings of the European Symposium on Research in Computer Security*. 65–82.

[7] Ionut Arghire. 2018. WannaMine malware spreads via NSA-linked exploit. Retrieved July 13, 2018 from https://www.securityweek.com/wannamine-malware-spreads-nsa-linked-exploit.

[8] Upul Bandara and Gamini Wijayarathna. 2013. Source code author identification with unsupervised feature learning. *Pattern Recognition Letters* 34, 3 (Feb. 2013), 330–334.

[9] Hal L. Berghel and David L. Sallach. 1984. Measurements of program similarity in identical task environments. *ACM SIGPLAN Notices* 19, 8 (Aug. 1984), 65–76.

[10] P. Vinod Bhattathiripad. 2012. Software piracy forensics: A proposal for incorporating dead codes and other programming blunders as important evidence in AFC test. In *Proceedings of the 36th Annual Computer Software and Applications Conference Workshops*. IEEE, Los Alamitos, CA, 206–212.

[11] Necdet Bulut and Maurice H. Halstead. 1973. Invariant properties of algorithms. *ACM SIGPLAN Notices* 8, 6 (1973), 12–13.

[12] Necdet Bulut, Maurice H. Halstead, and Rudolf Bayer. 1974. Experimental validation of a structural property of fortran algorithms. In *Proceedings of the 1974 Annual Conference*. ACM, New York, 207–211.

[13] S. Burrows and S. M. M. Tahaghoghi. 2007. Source code authorship attribution using N-grams. In *Proceedings of the 12th Australasian Document Computing Symposium*. 32–39.

[14] Steven Burrows, Alexandra L. Uitdenbogerd, and Andrew Turpin. 2009. Application of information retrieval techniques for source code authorship attribution. In *Proceedings of the 14th International Conference on Database Systems for Advanced Applications (DASFAA'09)*. 699–713.

[15] Steven Burrows, Alexandra L. Uitdenbogerd, and Andrew Turpin. 2014. Comparing techniques for authorship attribution of source code. *Software: Practice and Experience* 44, 1 (Aug. 2014), 1–32.

[16] Aylin Caliskan, Fabian Yamaguchi, Edwin Dauber, Richard Harang, Konrad Rieck, Rachel Greenstadt, et al. 2018. When coding style survives compilation: De-anonymizing programmers from executable binaries. In *Proceedings of the 2018 Network and Distributed System Security Symposium (NDSS'18)*.

[17] Aylin Caliskan-Islam, Richard Harang, Andrew Liu, Arvind Narayanan, Clare Voss, Fabian Yamaguchi, et al. 2015. De-anonymizing programmers via code stylometry. In *Proceedings of the 24th USENIX Security Symposium*. 255–270.

[18] Aylin Caliskan-Islam, Fabian Yamaguchi, Edwin Dauber, Richard Harang, Konrad Rieck, Rachel Greenstadt, et al. 2015. When coding style survives compilation: De-anonymizing programmers from executable binaries. arXiv:1512.08546.

[19] Dong-Kyu Chae, Sang-Wook Kim, Jiwoon Ha, Sang-Chul Lee, and Gyun Woo. 2013. Software plagiarism detection via the static API call frequency birthmark. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. ACM, New York, NY, 1639–1643.

[20] Evangelos Chatzicharalampous, Georgia Frantzeskou, and Efstathios Stamatatos. 2012. Author identification in imbalanced sets of source code samples. In *Proceedings of the 24th International Conference on Tools With Artificial Intelligence*. IEEE, Los Alamitos, CA, 790–797.

[21] Rong Chen, Lina Hong, Chunyan Lu, and Wu Deng. 2010. Author identification of software source code with program dependence graphs. In *Proceedings of the IEEE 34th Annual Computer Software and Applications Conference Workshops*. IEEE, Los Alamitos, CA, 281–286.

[22] Seokwoo Choi, Heewan Park, Hyun il Lim, and Taisook Han. 2009. A static API birthmark for Windows binary executables. *Journal of Systems and Software* 82, 5 (May 2009), 862–873.

[23] Radhouane Chouchane, Natalia Stakhanova, Andrew Walenstein, and Arun Lakhotia. 2013. Detecting machine-morphed malware variants via engine attribution. *Journal of Computer Virology and Hacking Techniques* 9, 3 (Sept. 2013), 137–157.

[24] B. Curtis, S. B. Sheppard, P. Milliman, M. A. Borst, and T. Love. 1979. Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics. *IEEE Transactions on Software Engineering* SE-5, 2 (March 1979), 96–104.

[25] Charles Curtsinger, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. 2011. Zozzle: Fast and precise in-browser JavaScript malware detection. In *Proceedings of the 20th Usenix Security Symposium*. 33–48.

[26] Edwin Dauber, Aylin Caliskan, Richard Harang, and Rachel Greenstadt. 2017. Git blame who?: Stylistic authorship attribution of small, incomplete source code fragments. arXiv:1701.05681.

[27] Edwin Dauber, Aylin Caliskan, Richard Harang, and Rachel Greenstadt. 2018. Git blame who?: Stylistic authorship attribution of small, incomplete source code fragments. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings (ICSE'18)*. ACM, New York, NY, 356–357. DOI: https://doi.org/10.1145/3183440.3195007

[28] Lucas Davi and Ahmad-Reza Sadeghi (Eds.). 2015. Code-reuse in malware. In *Building Secure Defenses Against Code-Reuse Attacks*. Springer, Cham, Switzerland, 22–26.

[29] Haibiao Ding and Mansur H. Samadzadeh. 2004. Extraction of Java program fingerprints for software authorship identification. *Journal of Systems and Software* 72, 1 (June 2004), 49–57.

[30] John L. Donaldson, Ann-Marie Lancaster, and Paula H. Sposato. 1981. A plagiarism detection system. In *Proceedings of the 12th SIGCSE Technical Symposium on Computer Science Education*. ACM, New York, NY, 21–25.

[31] Krishna S. R. Dubba and Arun K. Pujari. 2006. N-gram analysis for computer virus detection. *Journal in Computer Virology* 2, 3 (Dec. 2006), 231–239.

[32] Bruce S. Elenbogen and Naeem Seliya. 2008. Detecting outsourced student programming assignments. *Journal of Computing Sciences in Colleges* 23, 3 (2008), 50–57.

[33] J. A. W. Faidhi and S. K. Robinson. 1987. An empirical approach for detecting program similarity and plagiarism within a university programming environment. *Computers and Education* 11, 1 (Jan. 1987), 11–19.

[34] Brandi Firestine. 2017. Celebrating nine years of GitHub with an anniversary sale. Retrieved July 17, 2018 from https://blog.github.com/2017-04-10-celebrating-nine-years-of-github-with-an-anniversary-sale/.
[35] Ann Fitzsimmons and Tom Love. 1978. A review and evaluation of software science. *ACM Computing Surveys* 10, 1 (March 1978), 3–18.
[36] Georgia Frantzeskou, Stefanos Gritzalis, and Stephen G. Macdonell. 2004. Source code authorship analysis for supporting the cybercrime investigation process. In *Proceedings of the 1st International Conference on E-Business and Telecommunication Networks*. 85–92.
[37] Georgia Frantzeskou, Efstathios Stamatatos, and Stefanos Gritzalis. 2007. Identifying authorship by byte-level N-grams: The source code author profile (SCAP) method. *International Journal of Digital Evidence* 6, 1 (2007), 1–18.
[38] Georgia Frantzeskou, Efstathios Stamatatos, Stefanos Gritzalis, and Sokratis Katsikas. 2006. Effective identification of source code authors using byte-level information. In *Proceedings of the 28th International Conference on Software Engineering*. ACM, New York, NY, 893–896.
[39] Georgia Frantzeskou, Efstathios Stamatatos, Stefanos Gritzalis, and Sokratis Katsikas. 2006. *Source Code Author Identification Based on N-gram Author Profiles*. IFIP International Federation for Information Processing, Vol. 204. Springer, Boston, MA. DOI: https://doi.org/10.1007/0-387-34224-9_59
[40] Marina L. Gavrilova and Roman V. Yampolskiy. 2010. Applying biometric principles to avatar recognition. In *Proceedings of the International Conference on Cyberworlds*. IEEE, Los Alamitos, CA, 179–186.
[41] Hugo Gonzalez, Natalia Stakhanova, and Ali A. Ghorbani. 2018. Authorship attribution of Android apps. In *Proceedings of the 8th ACM Conference on Data and Application Security and Privacy*. ACM, New York, NY, 277–286.
[42] Sam Grier. 1981. A tool that detects plagiarism in Pascal programs. In *Proceedings of the 12th SIGCSE Technical Symposium on Computer Science Education*. ACM, New York, NY, 15–20.
[43] Dick Grune. 1991. *Concise Report on Algorithms in Sim*. Report distributed with Sim software.
[44] Muqaddas Gull, Tehseen Zia, and Muhammad Ilyas. 2017. Source code author attribution using author's programming style and code smells. *International Journal of Intelligent Systems and Applications* 9, 5, 27.
[45] Maurice H. Halstead. 1972. Natural laws controlling algorithm structure? *ACM SSIGPLAN Newsletter* 7, 2 (Feb. 1972), 19–26.
[46] Maurice H. Halstead. 1975. Toward a theoretical basis for estimating programming effort. In *Proceedings of the ACM Annual Conference*. ACM, New York, NY, 222–224.
[47] Peter G. Hamer and Gillian D. Frewin. 1982. MH Halstead's software science-a critical examination. In *Proceedings of the 6th International Conference on Software Engineering*. 197–206.
[48] Jane Huffman Hayes. 2008. Authorship attribution: A principal component and linear discriminant analysis of the consistent programmer hypothesis. *International Journal on Computers and Their Applications* 15, 2 (2008), 79–99.
[49] Jane Huffman Hayes and Jeff Offutt. 2010. Recognizing authors: An examination of the consistent programmer hypothesis. *Software Testing, Verification and Reliability* 20, 4 (Dec. 2010), 329–356.
[50] David I. Holmes. 1998. The evolution of stylometry in humanities scholarship. *Literary and Linguistic Computing* 13, 3 (Sept. 1998), 111–117.
[51] Mikko Hypponen. 2011. BRAIN: Searching for the first PC virus in Pakistan. Retrieved July 4, 2017 from http://campaigns.f-secure.com/brain/virus.html.
[52] Jeong-Hoon Ji, Gyun Woo, and Hwan-Gue Cho. 2008. A plagiarism detection technique for Java program using bytecode analysis. In *Proceedings of the 3rd International Conference on Convergence and Hybrid Information Technology*. IEEE, Los Alamitos, CA, 1092–1098.
[53] Vlado Kešelj, Fuchun Peng, Nick Cercone, and Calvin Thomas. 2003. N-gram-based author profiles for authorship attribution. In *Proceedings of the Pacific Association for Computational Linguistics*, Vol. 3. 255–264.
[54] Moshe Koppel, Jonathan Schler, and Shlomo Argamon. 2009. Computational methods in authorship attribution. *Journal of the American Society for Information Science and Technology* 60, 1 (Jan. 2009), 9–26.
[55] Jay Kothari, Maxim Shevertalov, Edward Stehle, and Spiros Mancoridis. 2007. A probabilistic approach to source code authorship identification. In *Proceedings of the 4th International Conference on Information Technology*. IEEE, Los Alamitos, CA, 243–248.
[56] Ivan Krsul and Eugene Spafford. 1997. Authorship analysis: Identifying the author of a program. *Computers and Security* 16, 3 (Oct. 1997), 233–257.
[57] Ivan Krsul and Eugene H. Spafford. 1994. *Authorship Analysis: Identifying the Author of a Program*. Technical Report 96-052. Purdue University, West Lafayette, IN.
[58] Ivan Krsul and Eugene H. Spafford. 1997. Authorship analysis: Identifying the author of a program. *Computers and Security* 16, 3 (Jan. 1997), 233–257.
[59] Vaibhavi Kulgutkar, Natalia Stakhanova, Paul Cook, and Alina Matyukhina. 2018. Authorship attribution through string analysis. In *Proceedings of the 13th International Conference on Availability, Reliability and Security (ARES'18)*. ACM, New York, NY.

[60] Robert Charles Lange and Spiros Mancoridis. 2007. Using code metric histograms and genetic algorithms to perform author identification for software forensics. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*. 2082–2089.

[61] Tımea László and Ákos Kiss. 2009. Obfuscating C++ programs via control flow flattening. *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica* 30, 1 (Aug. 2009), 3–19.

[62] Robert Layton and Ahmad Azab. 2014. Authorship analysis of the Zeus botnet source code. In *Proceedings of the 5th Cybercrime and Trustworthy Computing Conference*. IEEE, Los Alamitos, CA, 38–43.

[63] Robert Layton, Paul Watters, and Richard Dazeley. 2010. Automatically determining phishing campaigns using the USCAP methodology. In *Proceedings of the 2010 eCrime Researchers Summit*. 1–8.

[64] Robert Layton, Paul Watters, and Richard Dazeley. 2012. Unsupervised authorship analysis of phishing Webpages. In *Proceedings of the International Symposium on Communications and Information Technologies*. IEEE, Los Alamitos, CA, 1104–1109.

[65] Ronald J. Leach. 1995. Using metrics to evaluate student programs. *ACM SIGCSE Bulletin* 27, 2 (June 1995), 41–43.

[66] Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. 2006. GPLAG: Detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, New York, NY, 872–881.

[67] Thomas J. McCabe. 1976. A complexity measure. *IEEE Transactions on Software Engineering* SE-2, 4 (Dec. 1976), 308–320.

[68] Xiaozhu Meng. 2016. Fine-grained binary code authorship identification. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, New York, NY, 1097–1099.

[69] Paul W. Oman and Curt R. Cook. 1989. Programming style authorship analysis. In *Proceedings of the 17th ACM Annual Computer Science Conference (CSC'89)*. 320–326.

[70] Karl J. Ottenstein. 1976. An algorithmic approach to the detection and prevention of plagiarism. *ACM SIGCSE Bulletin* 8, 4 (Dec. 1976), 30–41.

[71] Alan Parker and James O. Hamblen. 1989. Computer algorithms for plagiarism detection. *IEEE Transactions on Education* 32, 2 (May 1989), 94–99.

[72] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. 2002. Finding plagiarisms among a set of programs with JPlag. *Journal of Universal Computer Science* 8, 11 (Nov. 2002), 1016–1038.

[73] Nathan Rosenblum, Barton P. Miller, and Xiaojin Zhu. 2011. Recovering the toolchain provenance of binary code. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, New York, NY, 100–110.

[74] Nathan Rosenblum, Xiaojin Zhu, and Barton P. Miller. 2011. *Who Wrote This Code? Identifying the Authors of Program Binaries*. Lecture Notes in Computer Science, Vol. 6879. Springer, 172–189. DOI: https://doi.org/10.1007/978-3-642-23822-2

[75] Nathan E. Rosenblum. 2011. *The Provenance Hierarchy of Computer Programs*. Ph.D. Dissertation. University of Wisconsin, Madison.

[76] Nathan E. Rosenblum, Barton P. Miller, and Xiaojin Zhu. 2010. Extracting compiler provenance from program binaries. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. ACM, New York, NY, 21–28.

[77] Philip J. Sallis, Asbjorn Aakjaer, and Stephen G. Macdonell. 1996. Software forensics: Old methods for a new science. In *Proceedings of the International Conference Software Engineering: Education and Practice*. IEEE, Los Alamitos, CA, 481–485.

[78] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. 2003. Winnowing: Local algorithms for document fingerprinting. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, 76–85.

[79] V. Y. Shen, S. D. Conte, and H. E. Dunsmore. 1983. Software science revisited: A critical analysis of the theory and its empirical support. *IEEE Transactions on Software Engineering* SE-9, 2 (March 1983), 155–165.

[80] Maxim Shevertalov, Jay Kothari, Edward Stehle, and Spiros Mancoridis. 2009. On the use of discretized source code metrics for author identification. In *Proceedings of the IEEE 1st International Symposium on Search Based Software Engineering*. IEEE, Los Alamitos, CA, 69–78.

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

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

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

[84] Benno Stein, Nedim Lipka, and Peter Prettenhofer. 2011. Intrinsic plagiarism analysis. *Language Resources and Evaluation* 45, 1 (March 2011), 63–82.

[85] Matthew F. Tennyson and Francisco J. Mitropoulos. 2014. Choosing a profile length in the SCAP method of source code authorship attribution. In *Proceedings of the 2014 IEEE SoutheastCon*. IEEE, Los Alamitos, CA, 1–6.
[86] Kristina L. Verco and Michael J. Wise. 1996. Software for detecting suspected plagiarism: Comparing structure and attribute-counting systems. In *Proceedings of the 1st Australasian Conference on Computer Science Education*. ACM, New York, NY, 81–88.
[87] Andrew Walenstein and Arun Lakhotia. 2007. The software similarity problem in malware analysis. In *Proceedings of the Conference on Duplication, Redundancy, and Similarity in Software*. 1–10.
[88] Xinran Wang, Yoon-Chan Jhi, Sencun Zhu, and Peng Liu. 2009. Behavior based software theft detection. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*. ACM, New York, NY, 280–290.
[89] Geoff Whale. 1990. Identification of program similarity in large populations. *Computer Journal* 33, 2 (April 1990), 140–146.
[90] Michael J. Wise. 1996. YAP3: Improved detection of similarities in computer program and other texts. *ACM SIGCSE Bulletin* 28, 1 (March 1996), 130–134.
[91] Wilco Wisse and Cor Veenman. 2015. Scripting dna: Identifying the JavaScript programmer. *Digital Investigation* 15 (2015), 61–71.
[92] Xinyu Yang, Guoai Xu, Qi Li, Yanhui Guo, and Miao Zhang. 2017. Authorship attribution of source code by using back propagation neural network based on particle swarm optimization. *PloS One* 12, 11 (2017), e0187204.
[93] Chunxia Zhang, Sen Wang, Jiayu Wu, and Zhendong Niu. 2017. Authorship identification of source codes. In *Proceedings of the Asia-Pacific Web (APWeb) and Web-Age Information Management (WAIM) Joint Conference on Web and Big Data*. 282–296.

Received December 2017; revised August 2018; accepted September 2018