Adversarial Authorship Attribution in Open-Source Projects

Alina Matyukhina
Canadian Institute for Cybersecurity, University of New Brunswick

Natalia Stakhanova
Department of Computer Science, University of Saskatchewan

Mila Dalla Preda
Dipartimento di Informatica, University of Verona

Celine Perley
Canadian Institute for Cybersecurity, University of New Brunswick

ABSTRACT

Open-source software is open to anyone by design, whether it is a community of developers, hackers or malicious users. Authors of open-source software typically hide their identity through nicknames and avatars. However, they have no protection against authorship attribution techniques that are able to create software author profiles just by analyzing software characteristics.

In this paper we present an author imitation attack that allows to deceive current authorship attribution systems and mimic a coding style of a target developer. Withing this context we explore the potential of the existing attribution techniques to be deceived. Our results show that we are able to imitate the coding style of the developers based on the data collected from the popular source code repository, GitHub. To subvert author imitation attack, we propose a novel author obfuscation approach that allows us to hide the coding style of the author. Unlike existing obfuscation tools, this new obfuscation technique uses transformations that preserve code readability. We assess the effectiveness of our attacks on several datasets produced by actual developers from GitHub, and participants of the GoogleCodeJam competition. Throughout our experiments we show that the author hiding can be achieved by making sensible transformations which significantly reduce the likelihood of identifying the author's style to 0% by current authorship attribution systems.

CCS CONCEPTS

• **Security and privacy** → **Software security engineering**;

KEYWORDS

Authorship attribution; obfuscation; imitation; open-source software; adversarial; attacks

CODASPY '19, March 25–27, 2019, Richardson, TX, USA

ACM ISBN 978-1-4503-6099-9/19/03…$15.00
https://doi.org/10.1145/3292006.3300032

ACM Reference Format:
Alina Matyukhina, Natalia Stakhanova, Mila Dalla Preda, and Celine Perley. 2019. Adversarial Authorship Attribution in Open-Source Projects. In *Ninth ACM Conference on Data and Application Security and Privacy (CODASPY '19), March 25–27, 2019, Richardson, TX, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3292006.3300032

1 INTRODUCTION

Consider the following scenario. Alice is an open source software developer. She contributes to different projects and typically stores her code on GitHub repository. Bob is a professional exploit developer who wants to hide his illegal activities and implicate Alice. To do this, he collects samples of Alice's code and mimics her coding style. A sample of Bob's malware ends up in the hands of law enforcement agency, where the analysis shows that a malware is written by Alice. This unfortunate scenario is possible due to the recent advances in software authorship attribution field that focuses on identification of the developer's style. In this work we explore adversarial side of software attribution and show how an adversary can confuse these techniques and conceal his identity.

The study of authorship attribution (also known as stylometry) comes from the literary domain where it is typically used for identifying the author of a disputed text based on the author's unique linguistic style (e.g., use of verbs, vocabulary, sentence length). The main premise of stylometric techniques lies in the assumption that authors unconsciously tend to use the same linguistic patterns. These patterns uniquely characterize the author's works and consequently, allow one to differentiate him/her among others.

Drawing an analogy between an author and a software developer, software authorship attribution aims to identify who wrote a program given its source or binary code. Applications of software authorship attribution are wide and include software forensics - where the analyst wants to determine the author of a suspicious program given a set of potential adversaries, plagiarism detection - where the analyst wants to identify illicit code reuse, programmer de-anonymization - where the analyst is interested in finding information on an anonymous programmer, and in general any scenario where software ownership needs to be determined. Traditionally, authorship attribution studies relied on a large set of samples to generate accurate representation of an author's style. A recent study by Dauber et al. [13] showed that this is no longer necessary and even small, and incomplete code fragments can be used to identify the developers of samples with up to 99% accuracy.

In this work, we propose an author imitation attack on authorship attribution techniques. *Author imitation* attack identifies a developer (the victim) and transforms the attacker's source code to a version that mimics the victim's coding style, while retaining the functionality of the original code. The attack success is measured by its ability to deter existing attribution techniques from recognizing this code as attacker's code and by its ability to imitate the victim author's style. The author imitation task can be considered as an extension of the authorship attribution task to a real-world scenario. Existing authorship attribution research assumes that authors are honest and do not attempt to disguise their coding style. We challenge this basic assumption and explore existing authorship methodologies in adversarial setting.

Within this context, we investigate four existing source code attribution techniques introduced by Ding et al. [14], Caliskan et al. [11], Burrows et al. [8], and Kothari et al. [19]. We explore their accuracy and their potential to be deceived by author imitation attack. Through our experiments we show that all these techniques are susceptible to author imitation and we are able to successfully imitate 73.48% of the authors on GoogleCodeJam and 68.1% of the authors on GitHub.

Finally, to subvert author imitation attack, we introduce an author hiding method and a novel coding style obfuscation approach- *author obfuscation*. The idea of author obfuscation is to allow authors to preserve the readability of their source code, while removing identifying stylistic features that can be leveraged for code attribution. Code obfuscation, common in software development, typically aims to disguise the appearance of the code making it difficult to understand and reverse engineer the code. In contrast, the proposed author obfuscation hides the original author's style by leaving the source code visible, readable and understandable. Our experiments show the effectiveness of author hiding. Indeed, we are able to reduce the accuracy of the Ding et al. attribution system from 73.84% of correctly classified authors to 1.08% by using only layout, lexical, syntactic transformations on the GoogleCodeJam dataset. We are also able to decrease the accuracy of the Caliskan et al. attribution system from 80.92% to 27.96% on the GitHub dataset. The attribution accuracy of Burrows et al. and Kothari et al. systems were decreased to 0% on the GitHub dataset. By adding control-flow obfuscation we were able further reduced the performance of Caliskan et al., Ding et al., Burrows et al. and Kothari et al. systems to 0%. Our results demonstrate that it is possible to successfully attack current authorship attribution systems with transformations which preserve readability of the code.

The rest of this paper is organized as follows. Section 2 analyses the existing attribution techniques. The author imitation attack is introduced in Section 3. The author hiding attack is presented in Section 4. The author obfuscation transformations and the evaluation of proposed attacks are described in Section 5. We conclude our work in Section 6.

2 BACKGROUND AND RELATED WORK

Authorship attribution, also known as stylometry, is a well known research subject in the literary domain. The recent interest in applying attribution techniques to software code raised a number of questions. One of them is the selection of characteristics (i.e., features) indicative of an author (i.e., software developer). Although the process of feature selection is one of the most crucial aspects of attribution, there is no guide to assist in the selection of the optimal set of features. As a result, the majority of studies venture to use features that prove to be most helpful in particular contexts. The earliest work in software forensics by Spafford and Weber [27] focused on a combination of features reflecting data structures and algorithms, compiler and system information, programming skill and system knowledge, choice of system calls, errors, choice of programming language, use of language features, comment style, variable names, spelling and grammar. Sallis et al. [26] extended this work by using additional features, such as cyclomatic complexity of the control flow and layout conventions. Krsul et al. [20], Kilgour et al. [17] and Ding et al. [14] introduced a broad classification of features according to their relevance for programming layout, programming style, programming structure, and linguistic metrics. Trivial source code obfuscation techniques can obscure part of these features, leading to a significant decrease in attribution accuracy.

A granular approach might be more effective in understanding what groups of features are beneficial in attribution and resistant to different kinds of obfuscation techniques. In this work, we classify the features into the following groups: layout, lexical, syntactic, control-flow, data-flow. Figure 1 shows such classification. These groups build on each other starting with simple and easily extractable features to more advanced ones focusing on the inner logic and structure of the program.

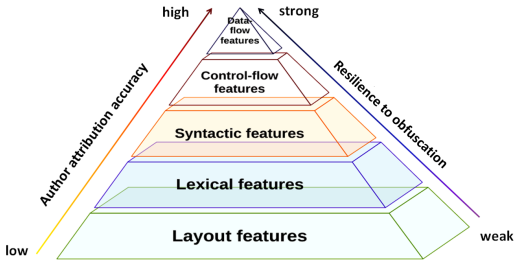

Figure 1: Feature selection levels in software authorship attribution

Layout features refer to format or layout metrics [9] that characterise the form and the shape of the code. Layout features include the length of a line of code, the number of spaces in a line of code, and the frequency of characters (underscores, semicolons, and commas) in a line of code. For the following discussion, we group these metrics as follows:

metrics that measure indentation, placement of comments, the use of white space (tab), placing of the braces.

Lexical features can be divided into programming style metrics, programming structure metrics and n-grams. Programming style metrics include character preferences, construct preferences, statistical distribution of variable lengths, and capitalisation. Programming structure metrics are assumed to be related to the programming experience and ability of a developer. For example, such metrics include the statistical distribution of lines of code per function, the ratio of keywords per line of code, the relative frequency of use of complex branching constructs and so on. A popular feature extraction technique is using n-grams to extract the frequency of sequences of n-characters from the source code.

Syntactic features are good features for authorship identification in natural language [18]. A parse tree is a convenient way to determine the syntactic structure of a sentence [23]. Baayen et al. [5] are the first to extract rewrite-rule frequencies from the parse tree for the purpose of authorship identification.

Recently, syntactic features have shown significant success also in source code authorship [4, 11]. These features represent the code structure and are invariant to changes in source code layout. Moreover, these features often describe properties of the language dependent AST (abstract syntax tree) such as code length, nesting levels, branching. AST does not include layout elements, such as unessential punctuation and delimiters (braces, semicolons, parentheses, etc.). Caliskan et al. [11, 12] investigated syntactic features to de-anonymize authors of C/C++ both at the source code and binary code level. They published the Code Stylometry Feature Set which includes layout, lexical and syntactic features. They have already achieved 94%, 96.83% and 97.67% accuracy with $1,600$, 250 and 62 class authors respectively. Recently Alsulami et al. [4] proposed Long Short-Term Memory (LSTM) and Bidirectional Long Short-Term Memory (BiLSTM) models to automatically extract relevant features from the AST representation of programmers' source code.

Control-flow features have been used in binary code attribution [3] and are not typical for source code attribution. These features are derived from control flow graph (CFG) that describes the order in which the code statements are executed as well as conditions that need to be met for a particular path of execution. Statements and predicates are represented by nodes, which are connected by directed edges to indicate the transfer of control. In binary authorship analysis graphlets (3-node subgraphs of the CFG) and supergraphlets (obtained by collapsing and merging neighbour nodes of the CFG) are used to identify the author of the code [3].

Data-flow features may indicate the author's preference in resolving a particular task through the selection of algorithms, certain data structures. These features are derived from the program dependence graph (PDG) that determines all the statements and predicates of a program that affect the value of a variable at a specific program point. It was introduced by Ferrante et al. [15] and it was originally used for program slicing [30]. In binary authorship analysis, Alrabaee et al. [3] used API data structures for this task. For binary code representation, authors analyzed the dependence between the different registers that are often accessed regardless of complexity of functions.

The arrows in Figure 1 represent our observations of feature selection influence on the authorship attribution accuracy and their strength of obfuscation. Layout features are associated with layout of programs and thus are fragile and easily alterable, for example by a code formatter. Lexical features are also related to the layout of code but are more difficult to change. Layout and lexical features alone are still less accurate (67.2% by [14]), than when used in combination with syntactic features (92.83% by [11]). Most of these features do not survive the compilation process. On the other hand, control-flow and data-flow features that retain programming ability and the experience of the programmer, are considered to provide a stronger evidence of developer's style. The existing source code attribution techniques only employ the combination of layout, lexical and syntactic features. In this work we focus on author imitation and hiding attacks at the layout, lexical, syntactic and control-flow levels.

3 AUTHOR IMITATION

The majority of previous studies show that we can successfully identify a software developer of a program. The question that naturally arises from this situation is whether it is possible to mimic someone else's coding style to avoid being detected as an author of our own software. In other words, can we pretend to be someone else?

Author imitation attack aims at deceiving existing authorship attribution techniques. The flow of the attack is shown in Figure 2 and includes three steps: (1) collecting victim's source code samples, (2) recognizing the victim's coding style, (3) imitating the victim's coding style. The pseudocode for this attack is given in Algorithm 1.

The attack starts with identifying a victim and retrieving samples of his/her source code $V_s = (s_1, s_2, ..., s_n)$. Typically, authors of open-source software hide their identity through nicknames and avatars. However, many GitHub accounts leave personal developer's information open, essentially allowing an attacker targeting a particular person collect the victim's source code samples.

Once the samples are collected, the second step is to analyze them and identify the victim's coding style. The strategy is to apply a set of transformations $M_{i,j}(A)$ to the set of source code samples $A = (t_1, t_2, ..., t_k)$ until the difference between the original victim style V and the modified attacker style is negligible.

The set of transformations $M_{i,j}$ is defined on the major feature levels i given in Figure 1, e.g., layout ($i = 1$), lexical ($i = 2$), syntactic ($i = 3$), control-flow ($i = 4$), data-flow ($i = 5$). The particular transformation j for each of the feature levels can vary. An example set of possible transformations is given in Table 2.

The distance between the feature vector extracted from the original source code of victim V and the feature vector

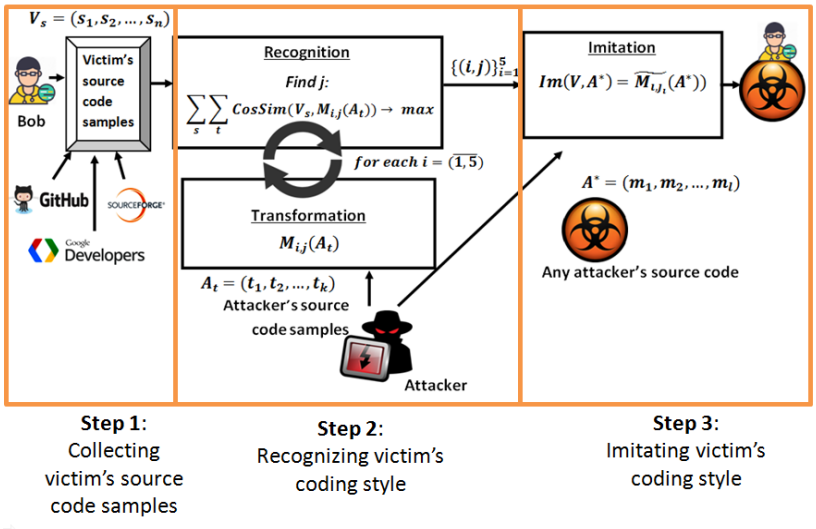

Figure 2: An overview of author imitation attack

extracted from the modified source code of A is determined based on cosine similarity[1], a widely implemented metric in information retrieval that models data as a vector of features and measures the similarity between vectors based on cosine value.

Definition 3.1 (Cosine similarity for author imitation attack). In authorship attribution, a source code can be represented as a vector of features whose dimension p depends on the considered feature set. Feature's value can refer to term frequency, average, log, or anything else depending on the features used.

Let $\vec{s_n} = \overrightarrow{(s_{n,1}, s_{n,2}, ..., s_{n,p})}$ denote the feature vector of the n-th source code of victim V, and let $\vec{t_k} = \overrightarrow{(t_{k,1}, t_{k,2}, ..., t_{k,p})}$ denote the feature vector of the k-th source code of attacker A, where $s_{n,h}$ and $t_{k,h}$ with $h \in \overrightarrow{(1,p)}$ are float numbers indicating the value of a particular feature. The cosine similarity between $\vec{s_n}$ and $\vec{t_k}$ is defined as follows:

$$CosSim(\vec{s_n}, \vec{t_k}) = \frac{\vec{s_n} \cdot \vec{t_k}}{||\vec{s_n}|| \cdot ||\vec{t_k}||} = \frac{\sqrt{\sum_{h=1}^{p} s_{n,h} \cdot t_{k,h}}}{\sqrt{\sum_{h=1}^{p} (s_{n,h})^2} \cdot \sqrt{\sum_{h=1}^{p} (t_{k,h})^2}}$$

Similarity is measured in the range 0 to 1. $CosSim = 1$ if two vectors are similar, $CosSim = 0$ if two vectors are different.

The code transformations $M_{i,j}$ that produce the maximum similarity, i.e. $CosSim(V_s, M_{i,j}(A_t)) \rightarrow max$, are the ones that the attacker should use to transform the original code in order to obtain a semantically equivalent code that mimics the victim's coding style. Note that these transformations should be calculated once per victim and can be applied on any of attacker's programs.

[1]For our analysis we experimented with a variety of similarity measures including Euclidean distance, Cosine distance, Minkovski distance, Jaccard distance, and Manhattan distance. Since Cosine similarity outperformed all other metrics, we employ it in our work.

Finally, to imitate the victim V, the attacker recursively applies the transformations identified in the previous step to A^*, i.e., $Im(V, A^*) = \widetilde{M_{i,j_i}}(A^*)$.

Algorithm 1 Author imitation attack

Input: $V_s = (s_1, s_2, ..., s_n)$ -victim's source code samples;
$A_t = (t_1, t_2, ..., t_k)$ -attacker's source code samples;
$A^* = (m_1, m_2, ..., m_l)$ -attacker's any source codes
Output: $Im(V, A^*)$ -attacker's source codes A^* with imitated victim's V coding style
# precomputation part
for all $i = 1$ to 5 **do**
  for all j **do**
    apply transformation M_{ij} to each A_t
    compute $\theta(i,j) = \sum_s \sum_t CosSim(\vec{V_s}, M_{ij}(\vec{A_t}))$
  end for
  take j such that $\theta(i,j)$ is maximum
end for
return pairs $(i, j_i)_{i=1}^{5}$ and their correspondent transformations M_i, j_i
# main part
for all A^* **do**
apply i transformations M_i, j_i to attacker's source code A^* recursively: $Im(V, A^*) = \widetilde{M_{i,j_i}}(A^*)$, where $\widetilde{M_{i,j_i}}(A^*)$ is a recursive function such as: $\widetilde{M_{i+2,j_{i+2}}}(A^*) = \widetilde{M_{i+1,j_{i+1}}}(\widetilde{M_{i,j_i}}(A^*))$
end for
return $Im(V, A^*)$

Complexity. The attack described in Algorithm 1 consists of a precomputation step and a main phase. In the precomputation step the algorithm searches for the code transformations that once applied to the attacker's source code samples transform the attacker's coding style into the victim's coding stile. The complexity of this step depends on the number of specific transformations defined for each of the five feature levels. It should be noted that these transformations only need to be determined once per victim. The main phase consists of applying selected transformations to

an attacker's source code. The time complexity of this phase grows linearly as the size of the attacker's code increases.

4 AUTHOR HIDING

To subvert author imitation attack, we propose a method that manipulates the source code to hide author's coding style while preserving code readability. The goal of author hiding is to prevent its detection by authorship attribution systems.

The author imitation attack applies transformations to the attacker's source code in order to imitate the victim's style. The most effective imitation can be generated when the distance between the source code feature vector of the victim and the modified source code feature vector of the attacker is negligible, i.e., transformations produce the maximum similarity between the two vectors.

Intuitively, to make author imitation attack unsuccessful, we should convert the original author's style to more generic less personalized version of it while fully retaining the functionality of code, i.e., these transformations should produce minimal similarity between the feature vector of the original and modified author source code. The pseudocode of author hiding is given in Algorithm 2.

In this work, we look at transformations at the layout, lexical, syntactic levels as features from these levels are commonly explored by attribution studies. Specifically, we classify all the transformations into the following groups: comment, brackets, spaces, lines, names, AST leaves, loops, and conditional statements. These transformations are low-cost and can be applied on any software with no computational overhead.

We additionally explore control flow transformation i.e. control-flow flatenning [29]. Control-flow flatenning rearranges code basic blocks (e.g., method body, loops, and conditional branch instructions) to make them appear to have the same set of predecessors and successors. Although a modified program flow is harder to follow, it is still readable for a human analyzer (Figure 3), which makes it suitable to use for our hiding approach. McCabe's complexity metric of such transformation is increased by a factor 2 to 5, which was shown by [21].

5 EVALUATION

Data. For our analysis, we collected two datasets from open source repositories. The majority of the existing attribution studies leverage programs developed during GoogleCodeJam, an annual international coding competition hosted by Google. During this competition, the contestants are presented with several programming problems and need to provide solutions to these problems within a limited time frame. We follow this practice and collect source code written in the Java programming language from the GoogleCodeJam held in 2015. Our GoogleCodeJam dataset contains 558 source code files from 62 programmers. Each program has on average 74 lines of source code. Although GoogleCodeJam is commonly used in studies, it has seen its share of criticism [12, 13, 22]. Specifically, the researchers argue that

Algorithm 2 Author hiding

Input: $V_s = (s_1, s_2, ..., s_n)$ - author's V source code samples;
$V^* = (v_1, v_2, ..., v_n)$ - code, which author V wants to hide
Output: $Hide(V^*)$ - source code without author's V style
# precomputation
for all $i = 1$ to 5 **do**
for all j **do**
apply transformation M_{ij} to each V_s
compute $\theta(i,j) = \sum_s CosSim(\vec{V_s}, M_{ij}\vec{(V_s)})$
end for
take j such that $\theta(i,j)$ minimum
end for
return pairs $(i, j_i)_{i=1}^5$ and their correspondent transformations M_i, j_i
# hiding
for all V^* **do**
apply i transformations M_i, j_i to author's source code V^* recursively: $Hide(V^*) = \widetilde{M_{i,j_i}}(V^*)$, where $\widetilde{M_{i,j_i}}(V^*)$ is a recursive function such as: $\widetilde{M_{i+2,j_{i+2}}}(V^*) = \widetilde{M_{i+1,j_{i+1}}}(\widetilde{M_{i,j_i}}(V^*))$
end for
return $Hide(V^*)$

competition setup gives little flexibility to participants resulting in somewhat artificial and constrained program code. The length of the code in GoogleCodeJam dataset is much smaller when compared with real-world programming solutions, which creates bias by making it easier to attribute programs and consequently leading to higher attribution accuracy.

To ensure the reliability of our analysis, we created second dataset with code samples from the popular open-source repository Github. Github is an online collaboration and sharing platform for programmers. Compared to the GoogleCodeJam dataset, the programs are typically more complex, can include third-party libraries and use source code from other authors. As a result, performing authorship attribution on GitHub data is more challenging. We crawled the GitHub in April 2018. Although it is difficult to guarantee a sole authorship of any code posted online, we took reasonable precautions. We filtered repositories that were marked as forks, as these are typically copies of other authors' repositories and so do not constitute original work. An additional check for multiple-author repositories was performed by examining the commit logs: if there were more than one unique name and email address combination, the repository was excluded. We further removed code from "lib" and "ext" folders and duplicate files. Overall, our final GitHub dataset included 558 source code files from 62 programmers with 303 lines of code per program on average. To understand what were the semantics of the programs in the GitHub dataset, we used a combination of automated and manual techniques. The projects in GitHub come with project description and README. We used Latent Dirichlet Allocation (LDA) [7] to analyze this textual description. By specifying a set of documents, LDA recognizes a set of topics where each topic is represented as probability of generating different words. Using existing domain names for categorization of GitHub projects [24], our final collection contains Application (352), Database (0), CodeAnalyzer (32), MiddleWare (0), Library (11), Framework (132), Other (31).

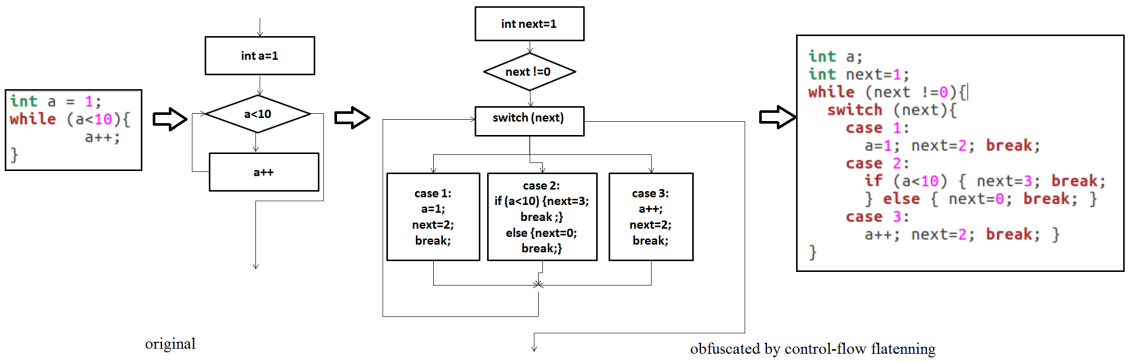

Figure 3: Control-flow flatenning obfuscation

Features. In spite of the wealth of attribution studies in the literary domain, throughout the years only a few source code attribution techniques were proposed. The majority of these studies experiment with diverse sets of features that capture developers' stylistic traits. These feature sets often range in size and level of analysis which makes it difficult to compare them. Burrows et al. [10] suggested grouping attribution systems by the type of features used in analysis: strings of n tokens/bytes (n-grams) or software metrics.

In this work we aim at exploring the accuracy of four prominent attribution systems from both categories and their potential to be attacked. In particular, we investigate the Ding et al. [14] and Caliskan et al. [11] systems that use feature sets based on software metrics, and the Burrows et al. [8] and Kothari et al. [19] attribution approaches based on n-grams features.

The study conducted by Ding et al. [14] is state-of-art research in Java source code attribution. The feature set and the obtained results serve as a reference for many studies [10]. Ding et al. collected and analysed 56 features and considered two datasets containing samples of 46 authors; the highest classification accuracy they obtained is 67.2%. Although not all features contributed to this result, the authors never provided the final subset or ranking of features. This was corrected by the follow-up study by Burrows et al. [10] that summarized previous classification techniques, and provided a final feature set of 56 metrics from the Ding et al. study. In our work, we use this feature set and refer to it as *Ding features*.

Another feature set that we explore is the one recently proposed by Caliskan et al. [11]. The study experimented with syntactic features (specifically, features derived from the AST) in an attribution context and published the Code Stylometry Feature Set. The results significantly outperformed all previously proposed attribution methods. For convenience we refer to this set as *Caliskan features*. With the dataset containing 250 authors and 9 samples per author, the study reported an attribution accuracy of 96.83%. Another dataset with 62 authors with 9 samples per author gave them an accuracy of 97.67% with Random Forest classification.

To give a fine granularity to our analysis, we created one more feature set. The Code Stylometry Feature Set (CSFS) analysed by Caliskan et al. [11] also includes term frequency of word unigrams. The authors tokenized the source file to obtain the number of occurrences of each token. Since these features constitute a significant portion of the original CSFS set (nearly half of the whole feature set), we consider them separately and refer to these features as *TFunigrams*. TFunigrams consist of term frequency of variable names, methods, classes, strings, comments, import names, etc. For each of the datasets we parsed the source codes to extract necessary sets of features. Since the study of Caliskan et al. [11] considers C++ and Python source code, we reimplemented their attribution model for Java source code. To produce AST, we use the external JavaParser library [28]. We created a parser to extract all the AST features specified in the Caliskan feature set. Following their method we reduced the total size and sparsity of the feature vector, by retaining only those features that individually have non-zero information gain. Information gain considers the difference between the entropy of the distribution classes and the entropy of conditional distribution of classes [31].

The study by Burrows et al. [8] uses indexed n-grams of tokens extracted from the parsed program source code. Their study identifies 6-grams as the most accurate n-gram size. In their follow up study [10], they explored normalised counts of n-gram occurrences as features with machine learning classifiers. Since the number of features to process increases exponentially for n-grams of features, they truncate the feature space to the most commonly occurring n-grams. In our work we use this feature set and refer to it as *Burrows features*.

Kothari et al [19] consider two sets of metrics. The first set of metrics consists of layout metrics, for example, distributions of leading spaces, line length, etc. The second metric set measures occurrences of byte-level n-grams. The n-gram length $n = 4$ is derived empirically. They use entropy to identify the fifty most discerning metrics for each author. In our work we use this feature set and refer to it as *Kothari features*. Burrows et al. [10] in their analysis of several attribution

Feature set	Number of authors	Samples/ author	L O C	Total number	Selected features	Layout	Lexical	Syntactic	Random Forest	Naive Bayes	J48	IBk
Original work												
Ding features	76	6	250	56	56	16	40	0	-	64.05%	66.17%	39.75%
Caliskan features	62	9	70	-	-	6	-	-	97.67%	-	-	-
Burrows features	76	6	250	-	1000	0	1000	0	-	73.24%	56.61%	37.01%
Kothari features	76	6	250	-	168	-	-	0	-	67.45%	74.70%	49.26%
GoogleCodeJam data set												
Ding features	62	9	74	56	56	16	40	0	**73.84%**	58.42%	59.86%	51.25%
Caliskan features	62	9	74	38630	607	6	187	414	**97.31%**	95.88%	91.03%	97.13%
TFunigrams features	62	9	74	20544	190	0	190	0	**96.95%**	95.69%	90.5%	93.01%
Burrows features	62	9	74	54267	1000	0	1000	0	**73.29%**	58.96%	64.87%	70.25%
Kothari features	62	9	74	34308	147	0	147	0	**86.56%**	79.92%	70.96%	81.99%
GitHub data set												
Ding features	62	9	303	56	56	16	40	0	**67.25%**	57.11%	54.62%	52.14%
Caliskan features	62	9	303	224478	687	5	203	479	**80.92%**	74.56%	78.12%	79.56%
TFunigrams features	62	9	303	87229	199	0	199	0	**75.08%**	67.79%	69.93%	66.01%
Burrows features	62	9	303	230735	1000	0	1000	0	**69.56%**	61.23%	66.34%	65.67%
Kothari features	62	9	303	116517	325	0	325	0	**80.23%**	72.78%	77.54%	79.52%

Table 1: The details of our datasets and feature sets employed by previous studies.

studies identified the Kothari features as the best in terms of classification accuracy; therefore the authors claimed that the n-gram approaches are more effective than the ones utilizing software metrics (note that at that time the work by Caliskan et al. [11] was not published yet). Since these studies work with source code developed with different programming languages (Java[14], C++, Python [11], and C, C++, Java [10]), in our work we bring everything to a common enumerator and employ Java programming language source code.

Table 1 shows statistics for the extracted features: number of authors, number of samples per author, average samples size in lines of code (LOC), the total number of features, the number of selected features, the number of layout, lexical, syntactic features on different datasets and original classification accuracy results reported by authors.

Classification. The previous approaches to source code authorship attribution employ various classification algorithms for attribution analysis while providing no justification of their algorithm's selection. Caliskan et al. [11] utilized Random Forest classifier, Burrows et al. [10] used Naive Bayes, Decision Tree, k-nearest neighbour classification, neural network, regression analysis, support vector machine, and voting feature interval.

For our analysis we employ Weka 3.8.2 platform [16]. Since our datasets do not have extensively large number of instances, we use the Weka implementation of Random Forest (RandomForest), Naive Bayes (NaiveBayes), J48 decision tree implementation of ID3 and IBk implementation of k-nearest neighbour algorithm. Since Burrows et al. [10] retained default parameters, for proper comparison we follow the same practice and do not change configuration for NaiveBayes, J48, IBk. For RandomForest algorithm, we chose 300 as the number of trees following the configuration used by Caliskan et al. [11]. All our experiments are performed with 9 fold cross validation.

5.1 Evaluation of existing authorship attribution methods

The results reported in Table 1 allow us to compare different authorship attribution methods on the GoogleCodeJam and GitHub datasets. The results that we obtain with the Caliskan feature set for the Java programs from GoogleCodeJam are similar to the ones originally obtained for C/C++ programs (97.31%) by Caliskan et al. [11].

Interestingly, the accuracy of this technique drops significantly (80.92%) on the GitHub dataset even though the number of features used in the analysis is 5.8 times bigger compared to the GoogleCodeJam dataset. We observe the same tendency with the TFunigrams feature set (that represents a significant amount of the Caliskan feature set) for which we have an accuracy of 96.95% on the GoogleCodeJam dataset and of 75.08% on the GitHub dataset. For the rest of the feature sets, Ding, Burrows and Kothari, the accuracy varies depending on the employed classification algorithm.

The difference in classification accuracy might be caused by the size of the source code in the dataset. The work of Caliskan et al. [11] considers only the GoogleCodeJam dataset, where each sample has on average 70 lines of code, while in real-world applications the programs are typically much larger. Indeed, the average number of lines of the samples in our GitHub dataset is 303 (4 times larger than the ones in the GoogleCodeJam considered by [11]).

An interesting observation came from the nature of the GoogleCodeJam competition that essentially forces authors to reuse their code written for previous tasks. As a result individual authors' style is derived from a set of very similar programs which significantly simplifies the attribution task.

The Code Stylometry Feature Set developed by Caliskan et al. was reported to "significantly outperform" other methods [11]. Yet our analysis does not agree with this; for example, with Kothari features we were able to achieve very similar results on Github data (80.23%) with a significantly smaller number of features. Since the Code Stylometry Feature Set of Caliskan et al. was designed solely based on the experiments on the GoogleCodeJam data, its suitability for real-world attribution is not definitive.

These results show that applying the attribution methods on different datasets leads to significantly different classification rates. Yet the majority of studies in attribution domain tend to only use the GoogleCodeJam data [3, 11, 25]. In spite of GoogleCodeJam data criticisms, an alternate dataset does not readily exist. We offer GitHub data to research

community in the hope of diversifying and strengthening experiments in this field[2].

Our results in Table 1 show that RandomForest classifier performs the best for both datasets. For this reason we use the RandomForest classifier in the rest of our experiments.

5.2 Author imitation evaluation

For the evaluation of the author imitation attack, we consider the features sets and datasets detailed in Table 1. We focus on layout (comments, brackets, spaces, lines), lexical (names), syntactic (AST leaves), control-flow feature groups in this analysis. Specific transformations that are considered for these groups are given in Table 2.

Type	Name
Comments	1. Transform all comments to Block comments; 2. Transform all comments to Javadoc comments; 3. Transform all comments to Line comments; 4. Transform all comments to pure comment lines 5. Delete all inline comments; 6. Delete all pure comment lines; 7. Delete all comments; 8. Add pure comments on each line; 9. Add inline comments on each line; 10. Add pure comments on each line and one inline comments
Brackets	1. Transform all brackets to Allman style; 2. Transform all brackets to Java style; 3. Transform all brackets to Kernighan and Ritche style; 4. Transform all brackets to Stroustup style; 5. Transform all brackets to Whitesmith style; 6. Transform all brackets to VTK (Visualization toolkit) style; 7. Transform all brackets to Banner style; 8. Transform all brackets to GNU style; 9. Transform all brackets to Linux style; 10. Transform all brackets to Horstmann style; 11. Transform all brackets to "One True Brace Style"; 12. Transform all brackets to Google style; 13. Transform all brackets to Mozilla style; 14. Transform all brackets to Pico style; 15. Transform all brackets to Lisp style
Spaces	1. Indent using (from 2 to 20) number of spaces per indent; 2. Indent using tabs for indentation and spaces for continuation line alignment; 3. Indent using all tab characters, if possible;4. Indent using mix of tabs and spaces; 5. Insert space padding around operators; 6. Insert space padding after only commas; 7. Insert space padding around parenthesis on both outside and the inside; 8. Insert space padding around parenthesis on the outside only; 9. Insert space padding around parenthesis on the inside only; 10. Insert space padding between 'if', 'for', 'while'; 11. Remove extra space padding around parenthesis; 12. Remove all space/tabs padding
Lines	1. Delete empty lines within a function or method; 2. Delete all emty lines; 3. Write each statement in one line; 4. Write several statements in one line; 5. Write one declaration per line; 6. Write several declarations in one line; 7. Add emty line after each nonempty line
Names	1. Change all names on extremely short name identifiers (one-two characters); 2. Use dictionary to change names to long names (8-10 characters); 3. Change the first letter in identifiers to uppercase; 4. Change the first letter to lowcase; 5. The first sign is underscore/dollarsign
AST leaves	1. Copy and insert all comments from imitated author; 2. To imitate the author B, change all name identifiers (methods names, variable names, class names) to the same names used in author B' source code; 3. All names are unigue for every author (use different dictionaries to change the names); 4. Use the same dictionary to change names for every author
Control-flow	1. Change "for" to "while" loop 2. Change "while" to "for" loop 3. Change "else if" to "switch case" 4. Change "switch case" to "else if" 5. Control-flow flatenning (for author hiding only)

Table 2: Applied transformations

The idea of the evaluation is simple: we consider each author in our dataset as a potential victim and we mount an author imitation attack on the chosen victim from all other authors (aka attackers). If the attack is successful, all attackers should be recognized as the chosen victim author.

The methodology of the evaluation of the attack is presented in Algorithm 3.

Let n be the number of authors in the dataset $A_1, A_2, ..., A_n$, for every author we collect m samples of code: $s(A_1)$, $s(A_2)$, ..., $s(A_n)$ and extract feature vectors from every source code: $\overrightarrow{s(A_1)}, \overrightarrow{s(A_2)}, ..., \overrightarrow{s(A_n)}$. In the transformation, recognition and imitation steps, we use only feature vectors of source code that belongs to the testing set. We then apply Algorithm 1 to imitate each author in the dataset.

Specifically, we take one author (victim V), e.g. $V = A_1$, leaving the remaining $n-1$ authors $A^* = (A_2, A_3, ..., A_n)$ to represent adversaries who want to imitate the victim's style. After applying Algorithm 1, we obtain $n-1$ samples of attackers' source code with the imitated victim's style: $Im(V, A^*) = Im(A_1, A_2), ..., Im(A_1, A_n)$. We use a similar method to imitate each author/victim in the dataset: A_2,...A_n.

Algorithm 3 Imitation attack evaluation

Input: Dataset of n authors A_w with their source code samples $s(A_w)$, where $w \in \overrightarrow{(1,n)}$, k-number of folds for cross-validation method
Output: accuracy of correctly classified authors ξ

```
# precomputation
for all w do
    find feature vectors s(A_w)→ extracted from each source code samples s(A_w) by using any known authorship attribution feature extraction methods (i.e. [10], [11], etc.)
    return feature vectors s(A_w)→
end for
# main part
for all k do
    divide dataset on training and testing set:
    TRAIN = (s_{i,k}(A_1)→, s_{i,k}(A_2)→, ..., s_{i,k}(A_n)→))
    TEST = (s_{j,k}(A_1)→, s_{j,k}(A_2)→, ..., s_{j,k}(A_n)→))
    apply imitation attack on every author n's source code from TEST set to get TEST_Im
    fold(k) = NN(TRAIN, TEST_Im)
end for
```

$$TEST_{Im} = \begin{pmatrix} \overrightarrow{A_1} & \overrightarrow{Im(A_1,A_2)} & \overrightarrow{Im(A_1,A_3)} & \dots & \overrightarrow{Im(A_1,A_n)} \\ \overrightarrow{Im(A_2,A_1)} & \overrightarrow{A_2} & \overrightarrow{Im(A_2,A_3)} & \dots & \overrightarrow{Im(A_2,A_n)} \\ \dots & \dots & \dots & \dots & \dots \\ \overrightarrow{Im(A_n,A_1)} & \overrightarrow{Im(A_n,A_2)} & \overrightarrow{Im(A_n,A_3)} & \dots & \overrightarrow{A_n} \end{pmatrix}$$

$$\xi = \sum_k fold(k)/k$$

At the end we move feature vectors from $Im(A_i, A_j)$ to the testing set $TEST_{Im}$, so that we have $m_2 * n * n$ feature vectors in the testing set $TEST_{Im}$. The evaluation then proceeds to classification to find the closest match among all authors for a given source code. In an ideal situation, we expect the closest match to be the imitated author. The accuracy is calculated as an average attribution rate after k fold cross validation. This approach allows us to test the author imitation attack on different scenarios, using different n coding styles for imitation and n styles to be imitated.

We employ all or nothing approach. Consider an example: we have 3 authors in the dataset: A_1, A_2, A_3, with 5 samples each. Take A_1 - if all A_2 and A_3 samples (i.e. all 10 samples) were attributed to A_1, then accuracy of imitating A_1 is 100%. Take A_2 - if all samples from A_3 were recognized as A_2,

[2]https://cyberlab.usask.ca/authorattribution.html

Dataset	Ding features	Caliskan features	Burrows features	Kothari features	TFunigrams features
GCJ	53.4%	73.48%	100%	99.8%	100%
GitHub	40.86%	68.1%	100%	97.85%	100%

Table 3: Percentage of successfully imitated authors

but one sample from A_1 was still attributed to A_1, this means that only one author (A_3) was successfully imitating A_1, hence the accuracy is 50%. Take A_3 - if all samples from A_1 were recognized as A_3, and only 3 samples from A_2 were recognized as A_3, the accuracy of imitation of A_3 is 50%. Finally, we average the results across all authors: $(100+50+50)/3 = 66.67\%$ the accuracy after author imitation attack.

Table 3 presents the result of our evaluation. With Ding features we could imitate 40.86% of the Github authors and 53.4% of the GoogleCodeJam (GCJ). This result is expected as Ding's set is very small, only 56 features, thus they do not use any feature selection algorithm. With transformations from Table 2 we were able to imitate 32 features of 56. The rest of features are dependent to data-flow for example Java's primitive and user-defined types, fields, methods, generic parameters, and exceptions.

Using Caliskan features, more than half of the authors in Github dataset (68.1%) and in GoogleCodeJam (73.48%) are imitated successfully.

Better results were obtained with features of TFunigrams, and Burrows features: 100% for Github users and 100% for GoogleCodeJam. Since all features in this set are lexical, and require only undergo transformation, the result is nearly perfect, i.e., we are able to successfully mount an imitation attack on all users.

5.3 Author hiding evaluation

To evade author imitation attack, one could possibly use code formatting to produce a more generic less personal version of code. Alsulami at al. [4] stated that "Modern IDEs format source code file content based on particular formatting conventions. This results in consistent coding style across all source code written using the same development tools. This reduces the confidence of using format features to identify the authors of source code." Indeed, many software development standards dictate formatting style that developers have to adhere to. We hypothesize that the use of a particular style alone is not sufficient to avoid attribution. Figure 4 shows our preliminary experiments with several different types of formatting: Beautify, Java Code Convention, PrettyPrinter using the Eclipse Code Formatter. It indicates that attribution accuracy does not produce significant change after using such formatting tools. We could still attribute nearly half of the authors with the Ding feature set and more than 70% of the authors with other feature sets. After manual analysis of our data, we found that near 80% authors in our datasets were already following Java Code Convention.

Our goal is to offer a source code obfuscation technique that preserves the readability of the code and misleads existing

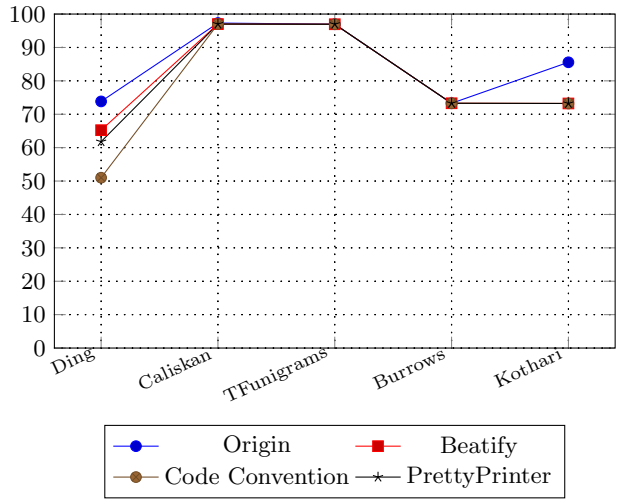

Figure 4: Percentage of correctly classified authors after using the following formatting: Beatify, Code Convention and PrettyPrinter on original source code

authorship attribution tools. We have seen that the existing methods for authorship attribution leverage layout and lexical features (Ding and Kothari) or layout and lexical, syntactic (Caliskan) or just lexical features (Burrows and TFunigrams).

In addition to these features, we also look at transforming control flow of a program. Although specific transformation can vary depending on the goal of the analysis, in this study we experimented with the transformations specified in Table 2.

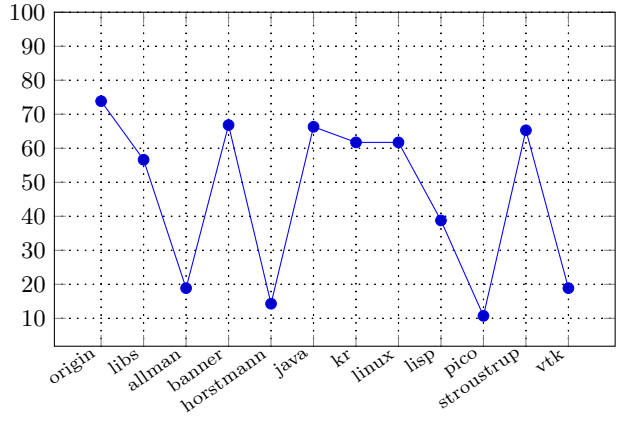

Figure 5: Percentage of correctly classified authors after brackets transformation for Google Code Jam dataset

For example, in the group brackets, we explore all possible styles of curly brackets (parentheses) that a developer can use. Figure 5 shows the results of experiments with only transformation of brackets in the code. We use 11 different

styles of brackets. By changing the bracket style to pico (the most rarely used by developers), we are able to reduce the attribution rate for the Ding feature set from 73.84% to 11.29%. The worst result is obtained with Java style (66.84%), this implies that most authors in our dataset use Java style. When considering the Caliskan feature set, the accuracy results do not change much and are still around 97%. This is due to the fact that the layout features make up only 1% of the Caliskan set after information gain feature selection.

We define "HideByCosine" obfuscation as following:

Definition 5.1 ("HideByCosine" obfuscation). Let $P \rightarrow P'$ be a transformation of a source program P into a target program P'. In order for $P \rightarrow P'$ to be a "HideByCosine" obfuscation the following steps should be applied:

- Apply a set of transformations T on program P
- Select those transformations so that $CosSim(\vec{a}, \vec{b}) \rightarrow min$, where $\vec{a}$ is the feature vector of the original program and $\vec{b}$ is the feature vector of the modified program.

The setup of our experiment for "HideByCosine" is similar to that of an imitation attack; we train RandomForest classifier on the non-obfuscated code and test it on the obfuscated one.

The results of the evaluation process for "HideByCosine" are reported in Table 5. When using only layout, lexical, syntactic and control-flow obfuscation we are able to achieve 1.43% attribution rate after applying author hiding attack for the Ding feature set. The transformations are unique for each author. With the Caliskan feature set we drop accuracy from 97.31% to 39.43% for the GoogleCodeJam dataset and from 80.92% to 27.96% for the GitHub dataset. This is due to the fact that the Caliskan et al. system uses information gain to select the most informative features, most of which are term frequency of unigrams and leaves (near 87% for GoogleCodeJam and GitHub) which can be easily obfuscated by name obfuscation. Table 4 shows the statistics of Caliskan features after information gain feature selection. The next group is AST nodes and AST node bigrams features (two AST nodes that are connected to each other), which only represent 12%, and are part of control-flow features. A portion of them can be obfuscated with simple transformations, e.g., changing "for" to "while" loops, and "else if" to "switch case", and do not guarantee author style's hiding. Therefore we applied control-flow flatenning (+CFF) on the top. After such transformations 0% of authors can be successfully attributed by Caliskan et al. and Ding et al. attribution systems.

Note that our result differs from what what reported in the Caliskan et al. work. [11]. The authors claimed that their method is resistant to simple obfuscation such as provided by Stunnix [1] with the reported accuracy of 98.89%, and to more sophisticated obfuscation (such as function virtualization by Tigress obfuscator [2]) with 67.22% on GoogleCodeJam data. Our results showed significantly lower accuracy. This is due to two facts. First, the authors used a much smaller dataset with only 20 authors making the task of attribution easier. Second, the experimental setup offered by their study assumed that

Features	Layout	Lexical		Syntactic		All
		unigrams	non-unigrams	leaves	non-leaves	
GCJ						
Original	6	20544	20	17393	667	38630
After info-gain	6	187	0	342	72	607
GitHub						
Original	6	87229	20	136415	808	224478
After info-gain	5	197	6	405	76	689

Table 4: Effect of information gain feature selection using Caliskan et al. [11] approach

the adversary is manipulating the training data and thus training of the classifiers is performed on selected and already obfuscated features.

We however followed a more realistic scenario commonly used in adversarial machine learning, i.e., the adversary aims to evade detection by manipulating test samples only [6]. We thus trained the RandomForest classifier on non-obfuscated code and tested it on obfuscated samples (imitated samples in the case of imitation attack). We were able to hide the coding style of the author with lexical and syntactic features and decrease the attribution accuracy dramatically to 39.43% for GoogCodeJam and to 30.29% for GitHub. After adding control-flow flatenning, no authors were recognized correctly.

Generalization of author hiding method for any author style. The proposed author hiding method "HideByCosine" requires that every time authors want to hide their identity they should first identify the code transformations that generate the most distinctive style with respect to their own style. The goal now is to define transformations which will be unique for all authors. We are doing this by considering one transformation from each group given in Table 2. We transform the source code of all authors by using this transformation and then perform classification again. In this way, for example, we are able to identify the bracket styles that are used the least in a considered dataset and then modify each author's brackets style by using this type of brackets. This gives us the opportunity to find an unique anonymous style that hides the coding style of all the authors in the dataset. We refer to this obfuscation methods as "MaxiCode" and "MiniCode". The methodolody of finding such transformations presents in Algorithm 4.

Algorithm 4 Author hiding "MaxiCode" or "MiniCode"

Input: Dataset V_s with number of authors s and number of source code samples m for each author; List of possible transformations T

Output: Set of transformations T_k

for all V_s **do**
apply each transformation T_k untill authorship attribution will fail identify all authors s in V_s
end for
return T_k

After applying such algorithm to our datasets, we define "MaxiCode" obfuscation as following:

Definition 5.2 ("MaxiCode" obfuscation). Let $P \rightarrow P'$ be a transformation of a source program P into a target program P'. In order for $P \rightarrow P'$ to be a "MaxiCode" obfuscation the following transformations should be applied to original program P: Comments (add pure line comments on each line); Brackets (transform all brackets to pico style); Spaces (indent using all tab characters, if possible); Lines (add empty line after each nonempty); Names and AST leaves: (use dictionary to change all names to long names (8-10 characters), the first sign is underscore, all names are unique for every author); Control-flow (change "while" to "for", change "else if" to "switch case")

One of the obvious concerns with this method is the size of source code since this transformation almost triples the code size. This makes this transformation impractical to use for developers in open-source projects. A possible solution is to shrink the code instead of expanding it; we refer to this method as "Minicode".

Definition 5.3 ("MiniCode" obfuscation). Let $P \rightarrow P'$ be a transformation of a source program P into a target program P'. In order for $P \rightarrow P'$ to be a "MiniCode" obfuscation the following transformations should be applied to original program P: Comments (no comments); Brackets (transform all brackets to pico style); Spaces (remove all spaces or tabs padding); Lines (delete all empty lines; write several statements in one line); Names and AST leaves (change all names on extremely short name indentifiers, the first sign is underscore, all names are unique for every author); Control-flow (change "for" to "while", change "switch case" to "else if")

The difference between "HideByCosine" method and "MaxiCode" ("MiniCode") method that we proposed is that the first one is more about modifying the individual author's style to the most dissimilar one; thus for every author the method identifies different transformations, while the second method is about finding an unique style that protects the style of all authors. The main advantage of using the obfuscation "MiniCode" or "MaxiCode" for author hiding is that authors do not need to make any precomputations beforehand.

Table 5 shows how the accuracy of attribution changes after applying the "MiniCode" approach to each feature set. This transformation decreases the size of the program by almost 8 times, while still preserving readability. After MiniCode obfuscation, the Kothari feature set achieves 1.88% attribution accuracy (GoogleCodeJam dataset) and 0% accuracy (GitHub). As the Kothari feature set is composed of layout and lexical features, they can be obfuscated by our transformations. As expected, Burrows and TFunigrams give us 0% attribution, since these are n-grams features and can be easily obfuscated by transformations in Table 2. After applying control-flow flatenning (+CFF) on the top of MiniCode and MaxiCode obfuscation 0% of authors can be successfully attributed by Caliskan et al., Ding et al., and Kothari et al. attribution systems.

Author Hiding	**GoogleCodeJam data set**				
	Ding	Caliskan	Burrows	Kothari	TFunigrams
Origin	73.84%	97.31%	73.29%	85.56%	96.95%
HideByCosine	**1.08%**	**39.43%**	**0%**	**1%**	**0%**
MaxiCode	4.12%	41.04%	**0%**	1.97%	**0%**
MiniCode	4.28%	43.90%	**0%**	1.88%	**0%**
HideByCosine+CFF	**0%**	**0%**	**0%**	**0%**	**0%**
MaxiCode+CFF	**0%**	**0%**	**0%**	0%	**0%**
MiniCode+CFF	**0%**	**0%**	**0%**	**0%**	**0%**
Author Hiding	**GitHub data set**				
	Ding	Caliskan	Burrows	Kothari	TFunigrams
Origin	67.25%	80.92%	69.56%	80.23%	75.08%
HideByCosine	**0%**	**27.96%**	**0%**	**0%**	**0%**
MaxiCode	2.37%	28.67%	**0%**	**0%**	**0%**
MiniCode	2.49%	30.29%	**0%**	**0%**	**0%**
HideByCosine+CFF	**0%**	**0%**	**0%**	**0%**	**0%**
MaxiCode+CFF	**0%**	**0%**	**0%**	**0%**	**0%**
MiniCode+CFF	**0%**	**0%**	**0%**	**0%**	**0%**

Table 5: Results of author hiding methods on both datasets

6 CONCLUSION

In this paper, we explored the accuracy of attribution using currently existing authorship attribution techniques in the presence of deception. We introduced an author imitation attack and investigated its feasibility on real-world software repositories. We used low-cost and easily implementable obfuscation techniques.

Also, we proposed several author hiding attacking techniques: "HideByCosine", "MaxiCode" and "MiniCode". The first method, "HideByCosine", works individually with each author by obfuscating source code to a style that is the most different from that of the the user. This method has better performance with respect to the other two. The other two methods work by finding one style which is unique for all authors. "MaxiCode" modifies the source code by extending it which triples its size. In the meantime, "Minicode" significantly reduces the size of the code making it almost 7 times smaller. In addition we applied control-flow flatenning on the top of this techniques to be able fully hide the coding style of the author, which gives 0% attribution at the end.

With both author imitation and author hiding methods we could significantly decrease the accuracy of current authorship attribution techniques.

REFERENCES

[1] 2014. Stunnix. Retrieved November 2014 from http://www.stunnix.com/prod/cxxo/

[2] 2014. Tigress. http://tigress.cs.arizona.edu

[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 (2014), S94–S103.

[4] Bander Alsulami, Edwin Dauber, Richard Harang, Spiros Mancoridis, and Rachel Greenstadt. 2017. Source Code Authorship Attribution Using Long Short-Term Memory Based Networks. In *European Symposium on Research in Computer Security*. Springer, 65–82.

[5] Harald Baayen, Hans Van Halteren, and Fiona Tweedie. 1996. Outside the cave of shadows: Using syntactic annotation to enhance authorship attribution. *Literary and Linguistic Computing* 11, 3 (1996), 121–132.

[6] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Šrndić, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. 2013. Evasion attacks against machine learning at test time. In *Joint European conference on machine learning and knowledge discovery in databases*. Springer, 387–402.

[7] David M Blei, Andrew Y Ng, and Michael I Jordan. 2003. Latent dirichlet allocation. *Journal of machine Learning research* 3,

Jan (2003), 993–1022.
[8] Steven Burrows and Seyed MM Tahaghoghi. 2007. Source code authorship attribution using n-grams. In *Proceedings of the Twelth Australasian Document Computing Symposium, Melbourne, Australia, RMIT University*. Citeseer, 32–39.
[9] Steven Burrows, Alexandra L Uitdenbogerd, and Andrew Turpin. 2009. Application of information retrieval techniques for source code authorship attribution. In *International Conference on Database Systems for Advanced Applications*. Springer, 699–713.
[10] Steven Burrows, Alexandra L Uitdenbogerd, and Andrew Turpin. 2014. Comparing techniques for authorship attribution of source code. *Software: Practice and Experience* 44, 1 (2014), 1–32.
[11] Aylin Caliskan-Islam, Richard Harang, Andrew Liu, Arvind Narayanan, Clare Voss, Fabian Yamaguchi, and Rachel Greenstadt. 2015. De-anonymizing programmers via code stylometry. In *24th USENIX Security Symposium (USENIX Security), Washington, DC*.
[12] Aylin Caliskan-Islam, Fabian Yamaguchi, Edwin Dauber, Richard Harang, Konrad Rieck, Rachel Greenstadt, and Arvind Narayanan. 2015. When coding style survives compilation: De-anonymizing programmers from executable binaries. *arXiv preprint arXiv:1512.08546* (2015).
[13] Edwin Dauber, Aylin Caliskan-Islam, Richard Harang, and Rachel Greenstadt. 2017. Git Blame Who?: Stylistic Authorship Attribution of Small, Incomplete Source Code Fragments. *arXiv preprint arXiv:1701.05681* (2017).
[14] Haibiao Ding and Mansur H Samadzadeh. 2004. Extraction of Java program fingerprints for software authorship identification. *Journal of Systems and Software* 72, 1 (2004), 49–57.
[15] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 3 (1987), 319–349.
[16] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. 2009. The WEKA data mining software: an update. *ACM SIGKDD explorations newsletter* 11, 1 (2009), 10–18.
[17] RI Kilgour, AR Gray, PJ Sallis, and SG MacDonell. 1998. A fuzzy logic approach to computer software source code authorship analysis. (1998).
[18] Sangkyum Kim, Hyungsul Kim, Tim Weninger, Jiawei Han, and Hyun Duk Kim. 2011. Authorship classification: a discriminative syntactic tree mining approach. In *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*. ACM, 455–464.
[19] Jay Kothari, Maxim Shevertalov, Edward Stehle, and Spiros Mancoridis. 2007. A probabilistic approach to source code authorship identification. In *Information Technology, 2007. ITNG'07. Fourth International Conference on*. IEEE, 243–248.
[20] Ivan Krsul and Eugene H Spafford. 1997. Authorship analysis: Identifying the author of a program. *Computers & Security* 16, 3 (1997), 233–257.
[21] Thomas J McCabe. 1976. A complexity measure. *IEEE Transactions on software Engineering* 4 (1976), 308–320.
[22] Xiaozhu Meng and Barton P Miller. [n. d.]. Binary Code Multi-Author Identification in Multi-Toolchain Scenarios. ([n. d.]).
[23] Arvind Narayanan, Hristo Paskov, Neil Zhenqiang Gong, John Bethencourt, Emil Stefanov, Eui Chul Richard Shin, and Dawn Song. 2012. On the feasibility of internet-scale author identification. In *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 300–314.
[24] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. 2014. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 155–165.
[25] Nathan Rosenblum, Xiaojin Zhu, and Barton Miller. 2011. Who wrote this code? identifying the authors of program binaries. *Computer Security–ESORICS 2011* (2011), 172–189.
[26] Philip Sallis, Asbjorn Aakjaer, and Stephen MacDonell. 1996. Software forensics: old methods for a new science. In *Software Engineering: Education and Practice, 1996. Proceedings. International Conference*. IEEE, 481–485.
[27] Eugene H Spafford and Stephen A Weeber. 1993. Software forensics: Can we track code to its authors? *Computers & Security* 12, 6 (1993), 585–595.
[28] Danny van Bruggen. 2017. JavaParser. Retrieved November 15, 2017 from https://javaparser.org/index.html
[29] Chenxi Wang and John Knight. 2001. *A security architecture for survivability mechanisms*. University of Virginia.
[30] Mark Weiser. 1981. Program slicing. In *Proceedings of the 5th international conference on Software engineering*. IEEE Press, 439–449.
[31] Ian H Witten, Eibe Frank, Mark A Hall, and Christopher J Pal. 2016. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann.