Language and Platform Independent Attribution of Heterogeneous Code

Farzaneh Abazari[1], Enrico Branca[1], Evgeniya Novikova[2], and Natalia Stakhanova[1(✉)]

[1] University of Saskatchewan, Saskatoon, Canada
{faa851,enb733,natalia}@usask.ca
[2] Saint Petersburg Electrotechnical University, Saint Petersburg, Russia

Abstract. Code authorship attribution aims to identify the author of source or binary code according to the author's unique coding style characteristics. Recently, researchers have attempted to develop cross-platform and language-oblivious attribution approaches. Most of these attempts were limited to small sets of two-three languages or few platforms. However, rapid development of cross-platform malware and general language, platform and architecture diversity raises concerns about the suitability of these techniques. In this paper, we propose a unified approach that supports attribution of code irrespective of its format. Our approach leverages an image-based code abstraction that preserves the developer's coding style and lends itself to spatial analysis that reflects hidden patterns. We validate our approach on a set of Android applications achieving accuracy 82.8%–100% with source and byte code. We further explore the robustness of our approach in attributing developers' code written in 27 programming languages, compiled on 14 instruction set architectures types and 18 intermediate compiled versions. Our results on the GitHub dataset show that in the worst case scenario the proposed approach can discriminate authors of code in heterogeneous format with at least 68% accuracy.

Keywords: Source code and Binary attribution · Authorship attribution

1 Introduction

Code authorship attribution aims to identify a developer of a given code based on unique characteristics that reflect a developer's coding style. The underlying premise of the attribution techniques is the existence of inherent distinctive coding style, unique to an author and easily distinguishable from others. This style is reflected through variables, data structures, control flow logic, use of APIs, libraries, employed development tools, and other characteristics. A quantified representation of this coding style can be viewed as a developer's fingerprint. This coding style is unique to an author and invariant across all software programs written by this author.

F. Li et al. (Eds.): SecureComm 2022, LNICST 462, pp. 173–191, 2023.
https://doi.org/10.1007/978-3-031-25538-0_10

One of the main difficulties in code stylometry is compiling a fingerprint that provides efficient and accurate characterization of a coding style, remains consistent across programming languages, and survives compilation stages. This is particularly critical in security applications of code authorship attribution - software forensics [42], malware analysis [10,16,20,28], code plagiarism and theft detection [26,37] - where attribution analysis of any available code is often an essential task. Yet, the traditional attribution methods focus exclusively on the attribution of source code files expecting a homogeneous set of files written in traditional languages (typically, Java, C or Python), or attribution of binary files compiled in the same architecture. This is rarely possible in practice. Developers routinely use multiple languages for various tasks. For example, analyzing GitHub repositories Mayer and Bauer found that developers may use from 1 to 36 languages in their projects [32].

Beyond diversity of programming languages, there are other reasons that present significant challenges to attribution process in practice. Since source code is not always available, attribution is expected to be performed on a mixed set of binary files and code samples at various stages of compilation. Yet, the majority of the existing approaches almost exclusively focus on attribution of either source or binary code.

With rapidly evolving market of IoT devices, instruction set architectures (ISAs)-oblivious attribution becomes essential. Binary files might be generated on different ISAs which leads to significant differences in their instruction set even when the files are compiled from the same source code base.

Different compilers and variable compiler configurations (e.g., optimization levels) might also bring considerable changes to the resulting file structure. These reasons make it difficult or even infeasible for traditional attribution methods to attribute mixed code in a form of binary, source code, and code at different stages of compilation across architectures and compiling configurations.

To address these problems, we propose an attribution approach that supports attribution of code irrespective of its format. One of the main difficulties that the existing attribution techniques face in this context is the inherit dependence of feature engineering on the underlying nature of the code.

To address this challenge, we treat code as a binary stream, an abstraction independent of the actual code structure, and convert it to a format-oblivious gray-scale image. This approach preserves structural similarities of code segments and lends itself well to spatial analysis.

Spatial analysis is widely applied in many fields for exploratory analysis. It utilizes statistical techniques to reveal non-obvious patterns by analyzing spatial relationships of pixels. We thus analyze spatial properties of the images generated from author's code samples to derive patterns that describe a developer's style. In essence, our approach is based on the assumption that individual coding style is unique and preserved across programming languages and architectures.

We validate our approach on a set of 348 programs from 50 Android developers. Our approach can successfully attribute app's source code and byte code to its author with 82.8%–100% accuracy. We designed a set of experiments with

source code in 27 languages, binaries in 14 ISAs and 18 compilers' versions. On GitHub data, our approach was able to accurately attribute source code with 71.5%, compiled binaries with 72.7% and intermediate stage of compiled files with 73.8% accuracy.

One of the practical applications of code attribution lies in malware analysis field where analysts are often challenged to attribute malware samples to its source. Since modern malware is typically obfuscated, any author attribution approach needs to be obfuscation-resistant. We show that our approach can achieve 72.5% attribution accuracy even in a presence of complex control-flow obfuscation transformations.

Finally, we compare the performance of our attribution approach to three techniques developed binary [21] and source [14,44] code attribution. The code and datasets used in this study are publicly available:[1].

The following is a summary of our contributions:

- *Code format - oblivious attribution*: Unlike traditional attribution methods that focus exclusively on the attribution of homogeneous set of files written in traditional languages, we propose an accurate attribution approach for a mix of code in a form of binary and source code, code at different stages of compilation across architectures and compiling configurations. Our approach does not require a prior knowledge of programming language, ISA, or file format specifics.
- *Style-preserving abstraction*: We present an effective abstraction scheme optimized for detecting developer's coding characteristics in any code format.
- *Obfuscation-resilient attribution*: Our presented approach is resilient to the advanced data and control-flow obfuscation transformations applied by the off-the-shelf Tigress obfuscator.

2 Related Work

A comprehensive overview of the code attribution techniques is provided by Kalgutkar et al. [27]. The earliest studies in the field were primarily limited to an analysis of a single language, and as a result, experimented with programming language-dependent features [18]. More recently, researchers began analyzing features that represented the underlying semantics of the program behavior and moved to explore language-agnostic attribution [2,3,41]. Very few of the studies included experiments with more than one language (typically, Java, C and/or Python) [2,3,12,44]. The majority of these studies leveraged Abstract Syntax Trees(ASTs) to capture language-independent syntactical features [12,14,41,44]. AST can be helpful in representing source code. However, its construction and parsing is language-dependent and time-consuming, while the amount of generated features is unscalable for analysis. To resolve this, some employed control flow (CFG) and data flow analysis for source code attribution [42,44]. Although these approaches are programming language-agnostic,

[1] https://cyberlab.usask.ca/anycodeattribution.html.

Table 1. Related work

References	Code type		Cross-lang. characteristics		Cross-platform features		Obf.	Malware
	Src.	Bin.	Languages	#of Lang per author	Compilers	ISA		
Abuhamad et al. [2]	✓		4 (C, C++, Java, Python)	2			✓	
Abuhamad et al. [3]	✓		3 (C++, Java, Python)	1				
Kurtukova et al. [31]	✓		13				✓	
Alsulami et al. [12]	✓		2 (C++, Python)	1				
Ullah et al. [42]	✓		3 (C++, C#, Python)					
Ullah et al. [41]	✓		3 (C++, C#, Java)					
Zafar et al. [44]	✓		3 (C++, Python, Java)	3			✓	
Caliskan et al. [14]	✓		3 (C, C++, Python)				✓	
Frantzeskou et al. [19]	✓		2 (C, C++)	1				
Alrabaee et al. [7]		✓			5 with different optimization levels: gcc, g++, CLANG, ICC, MS VS 2010, 2012	3 (x86, ARM, MIPS)	✓	✓
Alrabaee et al. [6]		✓			5 with different optimization levels: gcc, g++, CLANG, ICC, MS VS 2010, 2012	1 (x86)		✓
Alrabaee et al. [11]		✓			4 with different optimization levels: gcc, g++, CLANG, ICC, MS VS 2010		✓	
Hendrikse [24]		✓			3 with different optimization levels: gcc, ICC, MS VS		✓	
Haddadpajuh et al. [21]		✓						✓
Our approach	✓	✓	27 (listed in Table 4)	1–8 (src code)	3 with different optimization levels (gcc, ICC, MS VS)	14	✓	✓

Empty table cell corresponds to the case when the data about given approach feature is not explicitly discussed or mentioned in the research paper, or not applicable to the approach.

their performance is dependent on the availability of parsers, and hence similarly support limited number of languages.

Strictly speaking, only a few studies in the field offer verifiable language agnostic approaches [2,44]. In these cases, authors with several programming languages are considered in both design and experimentation. The rest of the research studies either explicitly mention the use of homogeneous code per author or simply remain silent about this aspect.

As code authorship attribution evolved and found its application in security domain, it faced one of the main research challenges, i.e., unavailability of source code. So, several approaches have been presented in literature to attribute binaries [9,15,33,39]. Recently researchers become interested in the impact of different compilers, optimization levels and obfuscation techniques on the accuracy of authorship attribution. Most notable study was conducted by Alrabaee et al. [10]. Their study showed that attribution of a binary code is challenging, i.e., solely relying on instruction level features results in significant accuracy degradation, while utilizing only features extracted from CFG is sensitive to obfuscation. This result is consistent with the findings of other studies [9,15,39].

Several recent studies focused on adding an intermediate representation (IR), e.g., LLVM-IR, of disassembled binary code as an architecture-agnostic representation to unify files for feature extraction [6,7,11,23]. Alrabaee et al. [6,7] showed that relying on attributes extracted from lifted binaries to LLVM-IR in combina-

tion with deep learning supports multi-platform binary authorship identification and scales well to a significant number of authors. Several researchers [21,38] focused on problem of APT malware attribution and attribution of Android apks [20,25,28,29].

Table 1 presents an overview of attribution approaches that could be considered either language or platform independent. As opposed to the existing attribution approaches that focus on either source or binary code, we focus on language, platform, and architecture oblivious solution. We propose a simple yet effective method for attributing code, regardless of its format, without any prior knowledge of code specifics.

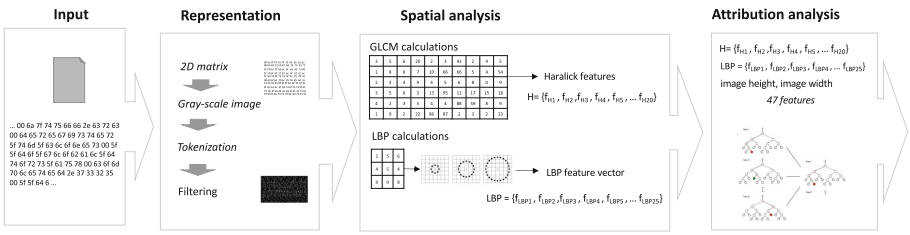


Fig. 1. The flow of the proposed approach

3 Approach

One of the main challenges that source or binary code attribution faces is engineering of features that are resistant to compilation process and persistent across programming languages and platforms.

In our work, we represent code regardless of its type as a set of consecutive bytes. This consequently allows us to generate a gray-scale image and leverage spatial analysis for deeper textural analysis of the image. Through this analysis, we can capture changes in adjacent bytes and derive characteristic byte patterns. As the last step, the characteristic patterns across all author's works are explored to derive a uniform representation of an author's coding style. Figure 1 summarizes the flow of our attribution approach.

3.1 Representation

Visualization of binary files have been used in many areas of security, e.g., for detection of obfuscation tools [25], malware detection and classification [13,34]. We leverage visualization of code as an intermediate representation to abstract the underlying format and platform specifics. The flow of the representation is shown in Fig. 2. Given that any code can be represented as a set of bytes, we read input in a raw binary format.

The input code either in human readable or machine readable format is converted into a gray-scale image following the approach introduced by Nataraj et al. [35]. The raw byte stream (or the corresponding ASCII values in case of source code) is transferred into a 2D matrix with a fixed width d, which is calculated based on the size of the original code file.

The resulting 2D matrix is treated as a 2D array of 1 byte vectors. Depending on the byte's value, each vector is then converted to a decimal value in range of $[0, 255]$ that further determines the gray-scale value of the pixel (0 for black and 255 for white). This approach preserves all patterns that exist in the original format of the input file.

The resulting images vary in size which allows us to use width and height as features in attribution analysis. These images also contain noisy and rare patterns that are irrelevant for attribution. In order to filter this noise and highlight the significant patterns, the generated image, i.e., the corresponding 2D array of bytes, is tokenized using the sliding window approach to produce n-byte consecutive grams. The sliding window iterates through the 2D matrix viewed as one consecutive sequence to extract n-grams regardless of their position in a matrix. Hence, n-grams may consist of bytes that reside at the end and the start of the row in the 2D matrix.

Filtering is based on n-gram frequency within the corresponding image and significance according to information gain (IG) value. For each file, we select the top most frequent n-grams, which results in a set of (sometime overlapping) n-grams across all samples per author. We further use Information Gain (IG) to measure each n-grams' importance for attributing samples to each author. Hence, in this process the distinctiveness of each n-gram for an author is assessed. For further analysis, we retained n-grams with IG ≥ 0.01. All occurrences of these selected n-grams are retained in their original order. The remaining n-grams are 'squeezed out' of the image, hence preserving the original order of frequent n-grams which is important for the following spatial analysis.

3.2 Spatial Analysis

Since image is a numerical representation of byte values, image texture represents the spatial organization of the gray-levels of the pixels in a code sample. Although many numeric texture analysis approaches were introduced in the past decade, statistical method is seen as the one of the most powerful image analysis techniques [22].

The deeper insight into relationships between individual pixels can be derived through second-order statistics that look at correlations between pixels. We employ two well known approaches to statistical analysis of image texture, namely, analysis based on the gray-level co-occurrence matrix (GLCM) introduced by Haralick et al. [22] and Local Binary Patterns (LBP) [36].

GLCM. GLCM characterizes image by analyzing frequency of neighbouring pixels at selected distances and orientations over the entire image. Let i and j

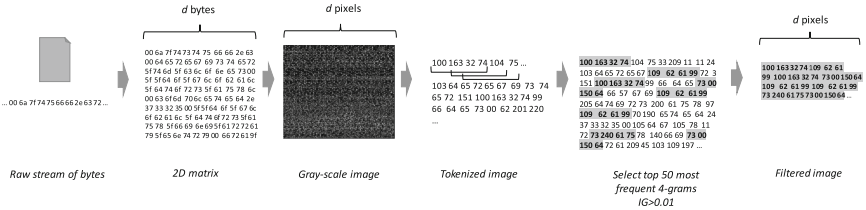


Fig. 2. The flow of the representation step.

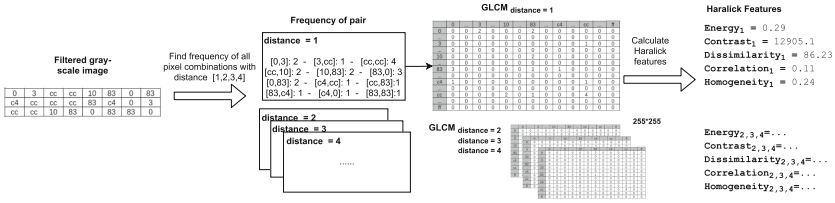


Fig. 3. An example of Haralick features' calculation.

be gray-scale values, the entry of GLCM is the probability that a pixel with value i will be found adjacent to a pixel of value j in the image separated by vector distance $\boldsymbol{d}$ which can be further expressed in terms of absolute distant d and the direction defined by the angle θ [22]. In this work, we set $d = 1, 2, 3, 4$; and $\theta = 0^{\circ}$. In other words, the neighbouring pixels located at various distances are analyzed at an angle 0° to form four different GLCMs. Each of the GLCMs serves as a basis for calculation of Haralick features [22] that describe texture features of GLCM. We derive five features to reflect specific patterns of an image (Table 2). Since we employ four directions d, each image is represented by $4 \times 5 = 20$ Haralick features which we refer to as a *Haralick vector*. Figure 3 shows the process of GLCM calculation with the corresponding Haralick features. After the GLCM matrix is calculated for d=[1..4], we calculate $p_{i,j}$ for each GLCM cell by calculating probability of combination of pair (i,j) (e.g. dividing the cell's value to the summation of all elements in GLCM).

Table 2. The Haralick features derived from GLCM

Name	Description	Formula
Energy	Shows randomness of the spatial distribution	$\sqrt{\sum_{i,j=0}^{255} P_{i,j}^2}$
Contrast	Measures gray level variations between the reference pixel and its neighbour	$\sum_{i,j=0}^{255} P_{i,j}(i-j)^2$
Dissimilarity	Shows average of differences in pixel values	$\sum_{i,j=0}^{255} P_{i,j}\lvert i-j\rvert$
Correlation	Measures the linear dependency of gray level values	$\sum_{i,j=0}^{255} P_{i,j} \frac{(i-\mu_i)(j-\mu_j)}{\sigma_i \sigma_j}$
Homogeneity	Indicates dominant values	$\sum_{i,j=0}^{255} \frac{P_{i,j}}{1+(i-j)^2}$

Local Binary Pattern (LBP). Unlike Haralick features that are based on GLCM Matrix and therefore represent global patterns, LBP conveys local patterns that are extracted directly from the image [43]. To do this, LBP measures a local representation of image texture by comparing each pixel with its surrounding neighbouring pixels located within a distance d of the reference point to test whether the surrounding points are greater or less than the central pixel value.

Figure 4 shows the process of converting patterns represented by neighbouring pixels to a binary value. Those neighbours with value less than referenced pixels are denoted as *1* and others as *0*. The resulting values are stored in an 8-bit binary array (or a corresponding decimal number), which is referred to as *LBP value*. This number reflects the texture (i.e., pattern) of the image around the referenced point. To assess the distribution of different LBP values across the image, we compute the frequency of each LBP value (i.e., each pattern) and assign it to the corresponding bin. Following the widely accepted practice in LBP analysis, we divide the range (0 to 255) into 25 equally distributed bins, each representing a set of LBP patterns.

Finally, the resulting histogram tabulates the number of times each LBP pattern occurs. The frequency of the bins is treated as *LBP feature vector* (Fig. 4). These features have highly discriminative nature that help the classifier to predict the author with higher accuracy [4,5].

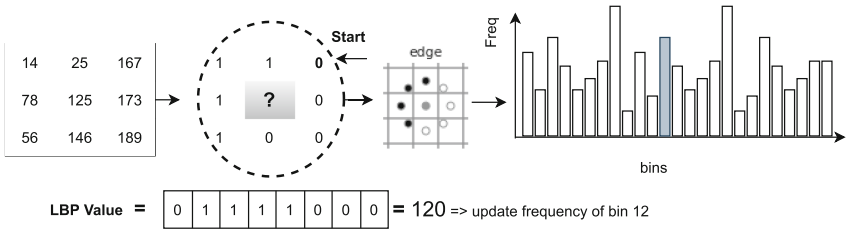


Fig. 4. An example of LBP value calculation for each pixel

3.3 Attribution

The attribution of code is based on the derived 47 features that include Haralick and LBP vectors, and image height and width.

Previous studies in source code authorship attribution employed various classification algorithms for attribution analysis [27]. Among them, Random Forest (RF) was one of the most common classifiers [1] that performs well in comparison with many standard methods.

We employ RF algorithm with 100 trees ("n_estimators" = 100) from a sample drawn with replacement from the training set with "entropy" criteria and maximum number of features is set to "sqrt". We set "min_samples_split" to 2 and "min_samples_leaf" to 1.

Table 3. The employed datasets' statistics

Filtered dataset	# of lang. or formats per author	# of authors	# of files	Range of samples per author	Range LOC or size	Avg. LOC or size	Range char. per line	Avg char. per line
Android validation set (Java src code)	1	50	7,594	5–856	3–4,253	172.1	1–1,464	36.1
Android validation set (.dex binary code)	1	50	413	5–25	12.4–11,640.7(KB)	3158.9	–	–
GCJ (src code)	2–5	3,000	63,682	10–157	1–3,039	80.1	1–95,567	23.45
GitHub (src code)	2–8	475	114,461	11–5,005	2–164,684	327	1–4,723,481	46.5
GitHub (binary code)	1–3	378	13,577	5–641	0.1–89,309(KB)	675(KB)	–	–

4 Data Corpus

The critical aspect of this work is the analysis of our approach in the presence of heterogeneous code formats. We thus ventured to collect mixed code in a form of binary, source code, and code at different stages of compilation across different architectures and compiling configurations.

All datasets were prepared to ensure presence of at least 5 samples for each format of file per author[2]. Moreover, for GitHub and GoogleCodeJam sets, the authors were selected to contain more than one type of file's format in the dataset, i.e., each author has files of at least 2 format and thus has at least 10 unique samples. The details of our collected datasets are given in Table 3.

Validation Dataset. In recent years, there has been an increasing interest for attributing Android APKs based on analysis of dex files [20], strings [28] and even specific features of the app such as permissions [45]. For our validation, we attribute APKs to their authors based on bytecode contained in dex files and original Java source files. We collected a set of 348 Android projects written by 50 authors from the open source Android application market F-Droid[3]. To ensure that our set does not include authors that use different alias for different repositories, we verified the authors' identities through the official GitHub, Gitlab, SourceForge, BitBucket platforms. Our collected set has authors with varying number of APKs ranging from 5 to 22, dex code ranging from 5 to 25 and Java source code from 5 to 856.

GitHub Repository. GitHub, an open-source software development platform. The programs in GitHub are typically more complex, include variety of programming languages, third-party libraries, several encodings and binaries focus

[2] In the rest of this work, by file's format we mean source code in a programming language, file's compiled version on some platform or architecture, or a file at different stages of compilation with various compiling configurations.

[3] https://www.f-droid.org/.

on solving diverse tasks (e.g., from game development to middleware). Performing authorship attribution on data retrieved from GitHub is more challenging due to presence of library and shared code. Due to these facts most of the previous works evaluated their approach on GoogleCodeJam dataset [2,3,42] and confirmed that the result of their paper would be different in the real-world dataset such as GitHub [44].

We collected programs from January until October 2020 by using the GitHub action logs. We consider repositories with at least one commit log, those that contain at least two different languages. Although it is difficult to guarantee sole authorship of any code posted online, we took reasonable precautions by filtering repositories marked as forks, as these are typically copies of other authors' repositories and do not constitute original work. An additional check for multiple-author repositories was performed by examining the commit logs. Repositories with logs containing more than one unique name and email address combination (potentially indicating an involvement of several authors) were also excluded. We download the latest master file. After removing duplicated files and filtering authors with less than 5 samples, 475 authors are remained with at least two languages (source code data) and 378 authors with multiple binary file formats (Table 3). For this analysis, we collected source and binary code of programs written in 25 programming languages, compiled for 14 architectures and in 18 intermediate stages of compilation from 3602 master files in GitHub. The distribution of files between languages and binary types are given in Table 4 and Table 5, respectively.

GoogleCodeJam dataset (GCJ). Since the majority of the existing studies employ data extracted from GoogleCodeJam programming competition[4], an annual international coding competition hosted by Google, for our analysis we also assembled a dataset containing code from the 2008 to 2018 competitions with authors that have code written in more than one language. We randomly selected 3000 authors with source code files written in 15 programming languages (2–5 different languages per author). The distribution of files across languages is provided in Table 4.

5 Experiments

We perform several experiments to validate our approach and examine its attribution effectiveness for various types of code. To estimate accuracy of attribution analysis, we used stratified 4-fold cross-validation that ensures that all developers are present in all folds. Note that strategy randomly partitions all author's code samples regardless of language, platform, or format, hence, different folds are likely to represent different subsets of languages. To evaluate our results, we employ a commonly used metric in attribution studies indicating the attribution accuracy or the *accuracy* in short. We use weighted-average accuracy defined as

[4] https://code.google.com/codejam/.

Table 4. GitHub and GCJ source code datasets

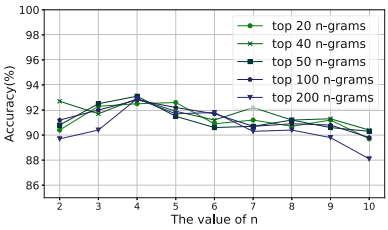
Language	# of samples GitHub	# of samples GCJ	Programming language paradigm								
			Array	Scripting	Compiled	Concurrent	Curly-bracket	Extension	Imperative	Interactive mode	Functional Impure
Python	12072	14663		✓				✓	✓	✓	✓
C	947	4668			✓		✓		✓		
C++	2500	20584			✓		✓		✓		✓
C#	65497	4494		✓		✓	✓		✓	✓	✓
Ruby	1454	1617		✓				✓	✓		✓
Golang	285	757			✓		✓		✓		
Java	40880	14062			✓	✓	✓		✓	✓	✓
Javascript	30072	612		✓			✓	✓	✓	✓	✓
LUA	186	96		✓				✓	✓	✓	
Kotlin	875	0			✓	✓	✓				✓
PHP	3943	769		✓			✓		✓	✓	✓
Perl	533	724		✓			✓	✓	✓	✓	✓
CSS	8228	0									
S	57	0	✓								
Tcl	17	0		✓				✓	✓	✓	✓
Cmake	247	0									
Dart	364	0					✓			✓	✓
Objective-c	25	0			✓						
Powershell	52	0		✓			✓	✓	✓	✓	
Verilog	123	0									
Swift	37	91			✓		✓		✓	✓	✓
Coffee	44	59									
TypeScript	848	0					✓				
Groovy	278	0		✓			✓		✓	✓	✓
Gradle	349	0									
Pascal	0	389							✓		
Lisp	0	97									✓
Total	114,461	63,682									

Table 5. GitHub dataset (binary and intermediate code)

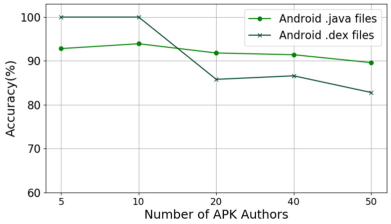
Compiled binary files:		
File Type	Architecture	# of samples
ELF	ARM 32-bit	468
	386 32-bit	174
	MIPS 32-bit	4
	AVR 32-bit	14
	68HC12 32-bit	4
	PPC 32-bit	1
	×86 64-bit	503
	AARCH64 64-bit	64
	MIPS 64-bit	2
PE32	I386 32-bit	1131
	ARMNT 32-bit	2
	AMD64 32+ -bit	630
	ARM64 32+ -bit	1
	IA64 32+ -bit	1
Files at intermediate stages of compilation:		
Compiler	Version	# of samples
Python	3.6	1178
	3.5	311
	2.7	240
	3.4	168
	2.6	3
Java	1.2	12
	1.3	11
	1.5	640
	1.6	457
	1.7	439
	1.8	5281
	1.9	645
	1.10	59
	1.11	297
	1.12	71
	1.13	199
	1.14	102
	1.15	22

a percentage of code samples correctly attributed to the corresponding authors over the total number of samples, where accuracy of each class is weighted by the number of samples from that class. All experiments were performed on an Intel server equipped with 384 GB of RAM and 32 CPU cores.

Selection of Parameters. Although n-grams are commonly applied in author attribution domain, their effect on the accuracy is often uncertain and depends on nature of the code. Since our approach leverages the top most frequent n-



(a) Selection of parameters



(b) Attribution accuracy of validation experiments (Android datasets)

Fig. 5. The experimental analysis.

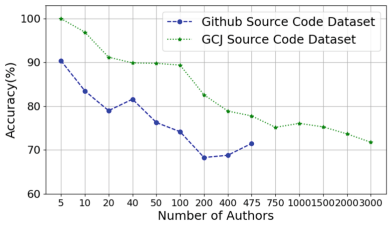
grams for analysis, we investigate the optimal quantity of n-grams and the best values of n on a subset of GCJ dataset with 50 authors and 841 source code samples. Previous studies (e.g., [30]) explored the values of n ranging between 2 and 10 concluding that 4-grams generally produce the best results. We therefore also explore this range of n-grams. As Fig. 5a shows, the attribution accuracy for varying number of n-grams and the top m most frequent n-grams is similar. This result is beneficial as voluminous feature vectors generated during the analysis may make analysis prohibitively expensive and sometimes infeasible. Hence, reducing the number of n-grams while retaining similar accuracy is beneficial in many resource-constraint environments. Although the results show low variability across different n and m values, the top 50 4-grams produce the best accuracy (93.1%). We choose these values to tokenize our input.

Validation Results. For this experiment, we leverage our validation dataset that contains APKs related files at two granularity levels: original Java source code and byte code derived from dex files. Figure 5b shows that the accuracy for all levels varies between 100% and 82.8%. The most consistent attribution is achieved with source code (92.8% for 5 authors and 89.6% for 50 authors). This is consistent with our expectations (and the previous studies) that source code inherently preserves the developer's coding style and serves as a reliable characteristic. The best accuracy (100%) was obtained with dex files for 5–10 authors, which however dropped to 82.8% for 50 authors.

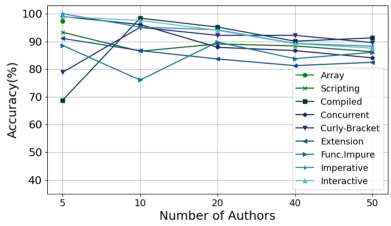
Feature Set Analysis. In our analysis, we rely on a set of 47 features that include, in addition to Haralick and LBP features, image width and height. To understand their role in the final attribution, we explore the importance of each of these features on a subset of GCJ dataset with 50 authors. The majority of features (42 features) have information gain > 0.01, i.e., their contribution to the result is meaningful. The image width and height along with Haralick features appeared to be among the top performing features.

5.1 Authorship Attribution of Source Code

To examine the performance of our approach on multi-language source code dataset, we design a set of experiments to analyze the effect of the dataset size



(a) Attribution accuracy of source code



(b) Source code attribution of language paradigms

Fig. 6. Attribution accuracy of source code

and number of languages on the accuracy of attribution. We randomly select subsets of authors from GitHub and GCJ datasets to explore behaviour of our approach on smaller sets.

Figure 6a shows that our approach can obtain attribution accuracy of more than 70% in most cases. As been noted by several studies, experiments with GitHub data in almost all cases give lower accuracy than with GCJ programs [1, 12,17]. For authorship attribution, the use of GCJ data has been extensively criticized mostly owing to its artificial setup [15,17,33]. The researchers argued the existing competition setup gives little flexibility to participants resulting in somewhat artificial and constrained program code.

The variability of languages further impact the results. The GitHub dataset has more languages (25) than GCJ set (15). For example, with 5 authors that have 2 different languages each in GCJ and 5 languages in GitHub set, our approach obtains perfect accuracy (100%) with GCJ and only 90.4% with GitHub.

Yet, with the increase in number of authors and number of languages, the performance of our approach on GCJ deteriorates. The total set of GitHub authors (475 authors) with 25 languages is attributed with 71.5% accuracy, while GCJ set with 3000 authors and 15 languages shows 71.8%. It should be also noted that the difference in performance remains somewhat consistent between two sets (around 20%).

5.2 Authorship Attribution of Binary Files

Most of the previous works focused on analysis of files compiled for a single ISA [6,8], e.g., ×86, that makes their binary authorship attribution solution ineffective for analyzing modern malware which can be designed for various ISA. The performance of our approach in attributing binaries extracted from GitHub repository is shown in Fig. 7.

As the results show, the attribution is reasonably better on compiled binaries (ELF and PE) than binaries at intermediate stages of compilation (Java class and compiled Python). Table 5 shows the distribution of GitHub binaries in 14 compiled and 18 intermediate compiled binaries. The accuracy dips from 82.7% (5 authors) to 66.2% (200 authors) for Java class files. The results on Python

compiled files are higher for smaller sets (96.5% for 10 authors) and lower for larger sets (72.9% for 50 authors). Overall, PE files have the highest accuracy (84.6%) on 50 authors.

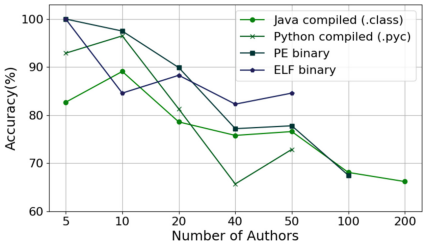


Fig. 7. Accuracy attribution of GitHub binary dataset

Table 6. Authorship attribution of GitHub dataset

Group	Number of authors	Accuracy
Compiled	142	72.7%
Inter.stage compiled	248	73.8 %
All binaries	378	68.3%
Binary and Source	761	68%

To investigate how the combination of datasets impacts the overall performance of authorship attribution, we combine GitHub files from two sets (source and binary) in 4 groups: 1) Intermediate stage compiled, 2) Compiled, 3) All binary code, and 4) Source and binary code.

The results of attribution on these four groups are shown in Table 6. The number of authors in the mixed set is less than the total number in individual sets due to overlap of authors in different groups. We are able to obtain the highest accuracy on a set of files at intermediate stages of compilation (73.8% accuracy). The least accuracy is achieved on the set of combined binary and source code (68% for 761 authors). Note that this set represents source code samples written in 25 different languages. This experiment clearly illustrates that the design of our approach allows to accurately attribute any file to a corresponding author in the worst case with at least 68% accuracy.

5.3 Authorship Attribution of Obfuscated Source Code

In this experiment, we investigate the impact of obfuscation techniques on attribution of source code. In our experiments, we use Tigress [40], an obfuscator tool designed for the C language. For this experiment, we randomly selected a subset of GCJ dataset, consisting of 50 authors with 7 samples per author on average (361 samples written in C), and obfuscate the whole set using several types of obfuscations (Table 7). The main reason behind using GCJ as opposed to GitHub is a lack of C samples within our GitHub set. Only GCJ set had enough C samples to perform experiments with 50 authors.

As expected, the accuracy of attributing obfuscated code is lower than original accuracy. However, for control obfuscation the performance drops only by 8.5% compared to the original source code (72.5% vs 81%). Note that including virtualization transformation (Advanced control obfuscation) only slightly

Table 7. Accuracy attribution of obfuscated source code (C language)

Dataset	Obfuscation methods	Accuracy	
		Zafar et al. [44]	Our approach
Original src code	–	65.7 %	81%
Data obf.	Literals encoding, Encode arithmetic	58.9%	70%
Control obf.	Opaque Predicates, Control flow Flattening, Insertion of random functions	57.5 %	72.5%
Advanced control obf.	Virtualization, Opaque Predicates, Control flow Flattening, Insertion of random functions	57.5%	70.5%

decreases our approach's accuracy (by 1.5%). Virtualization generates arbitrarily complex virtual instruction sets (i.e., customized ISA), which are then interpreted on-the-fly during program execution. The results also show that our approach is more obfuscation resilient than the recent language and obfuscation oblivious method proposed by Zafar et al. [44]. While the authors reported higher accuracy in the presence of obfuscation (on average ranging from 83% to 95% for C++ samples), their analysis only included trivial layout transformations (e.g., symbol name replacement, removal of spaces, comments, etc.). While our evaluation with advanced control obfuscation showed superior resiliency of our approach.

5.4 Comparison with the Existing Approaches

To better assess the proposed approach, we compare the performance of our attribution approach with three techniques: MVFCC, binary code attribution approach developed by Haddadpajouh et al. [21] and the state-of-the-art source code, language oblivious attribution methods proposed by Zafar et al. [44] and Caliskan et al. [14][5]

For fair comparison, we obtained an original dataset employed in Haddadpajouh et al.'s [21] study that consists of 5 APT malware groups with a total of 1463 samples. To compare with Caliskan et al.'s and Zafar et al.'s approaches, we selected subsets of the GCJ dataset with Java and C++ source code to ensure at least 7 code samples per author.

The comparison results given in Table 8 show that *our proposed approach achieves higher accuracy in most cases, while providing more flexible and broader framework for attribution of code in various formats without prior knowledge.* In case of MVFCC, we were able to attribute samples of APT malware groups with accuracy of 96.3%, which is comparable with the results reported in the original study (95.2%). Our approach performs better in terms of accuracy and efficiency than the Caliskan et al.'s approach [14]. Relying on AST n-grams, Caliskan et al.'s approach results in more than 35,000 features for 100 authors present in

[5] The authors of the other cross-platform and languages-oblivious studies [7,8,12] could not provide us with their code for comparison.

Table 8. Comparison of the attribution approaches

Java source code dataset	Accuracy		
	Caliskan et al. [14]	Zafar et al. [44]	Our approach
GCJ (5 authors)	89%	**100%**	98.7%
GCJ (10 authors)	83.3%	**100%**	97.2%
GCJ (20 authors)	85.7%	86.3%	**95%**
GCJ (40 authors)	81.4%	81.3%	**93%**
GCJ (50 authors)	80.3%	80.8%	**92%**
GCJ (100 authors)	75.2%	77.5%	**90%**
C++ source code dataset	**Caliskan et al.** [14]	**Zafar et al.** [44]	**Our approach**
GCJ (5 authors)	98.6%	86.6%	**100%**
GCJ (10 authors)	**96.4%**	93.9%	94.2%
GCJ (20 authors)	90.9%	82.8%	**92.9**%
GCJ (40 authors)	**92.3%**	73.4%	90.3%
GCJ (50 authors)	88.1%	73.4%	**90.1**%
GCJ (100 authors)	85.2%	69.1%	**86.6**%
Mixed source code dataset (50 authors)	**Zafar et al.** [44]		**Our approach**
Scripting languages	76%		**86.3%**
Compiled languages	85.6%		**91.3%**
Concurrent languages	**88.1%**		84.1%
Curly-Bracket languages	77.9%		**89.6%**
Extension languages	55.6%		8**2.5%**
Functional Impure languages	75.3%		**86.1%**
Imperative languages	77.1%		**88.3%**
Interactive languages	77.3%		**87.6%**
Binary dataset	**MVFCC** [21]		**Our approach**
APT malware set [21]	95.2%		**96.3%**

our set, in comparison, our approach generates only 47 features which as a result leads to faster attribution.

Zafar et al. approach on the other hand showed a slightly higher accuracy for Java source code compared to our approach for a small number of authors (100% for attributing 5 and 10 authors vs 98.7% and 97% with our approach). Yet, its accuracy dropped significantly for larger sets of 20 and more authors (77.5% for 100 authors). With the C++ dataset and a set with mixed languages, our approach in most cases showed better performance. With the limited applicability of Zafar et al. approach (source code only), our framework is better equipped to provide a more versatile and accurate attribution in practice.

6 Conclusion

In the field of security, code attribution finds its application in many contexts, e.g., software forensics, malware analysis, code plagiarism and intellectual property theft detection. The uncertainty of field requires the presence of suitable

techniques capable of attributing any given code. The existing authorship attribution approaches fail to provide necessary support. This work offers a unified approach for accurate code attribution. We adopt a coding style preserving abstraction scheme which enables us to accurately attribute code to its corresponding author without any knowledge of code format.

This paper's findings have important implications for developing a heterogeneous system that can relate any form of user output to the corresponding author.

References

1. Abazari, F., Branca, E., Ridley, N., Stakhanova, N., Dallapreda, M.: Dataset characteristics for reliable code authorship attribution. IEEE Trans. Depend. Secure Comput. (2021)
2. Abuhamad, M., AbuHmed, T., Mohaisen, A., Nyang, D.: Large-scale and language-oblivious code authorship identification. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 101–114 (2018)
3. Abuhamad, M., Rhim, J.S., AbuHmed, T., Ullah, S., Kang, S., Nyang, D.: Code authorship identification using convolutional neural networks. Future Gener. Comput. Syst. **95**, 104–115 (2019)
4. Ahonen, T., Hadid, A., Pietikainen, M.: Face description with local binary patterns: application to face recognition. IEEE Trans. Pattern Anal. Mach. Intell. **28**(12), 2037–2041 (2006)
5. Ahonen, T., Matas, J., He, C., Pietikäinen, M.: Rotation invariant image description with local binary pattern histogram fourier features. In: Salberg, A.-B., Hardeberg, J.Y., Jenssen, R. (eds.) SCIA 2009. LNCS, vol. 5575, pp. 61–70. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02230-2_7
6. Alrabaee, S., Debbabi, M., Wang, L.: On the feasibility of binary authorship characterization. Digital Invest. **28**, S3–S11 (2019)
7. Alrabaee, S., Debbabi, M., Wang, L.: Cpa: accurate cross-platform binary authorship characterization using lda. IEEE Trans. Inf. Forensics Secur. **15**, 3051–3066 (2020)
8. Alrabaee, S., Karbab, E.M.B., Wang, L., Debbabi, M.: `BinEye`: towards efficient binary authorship characterization using deep learning. In: Sako, K., Schneider, S., Ryan, P.Y.A. (eds.) ESORICS 2019. LNCS, vol. 11736, pp. 47–67. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29962-0_3
9. Alrabaee, S., Saleem, N., Preda, S., Wang, L., Debbabi, M.: Oba2: an onion approach to binary code authorship attribution. Digital Invest. **11**, S94–S103 (2014)
10. Alrabaee, S., Shirani, P., Debbabi, M., Wang, L.: On the feasibility of malware authorship attribution. In: Cuppens, F., Wang, L., Cuppens-Boulahia, N., Tawbi, N., Garcia-Alfaro, J. (eds.) FPS 2016. LNCS, vol. 10128, pp. 256–272. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-51966-1_17
11. Alrabaee, S., Shirani, P., Wang, L., Debbabi, M., Hanna, A.: On leveraging coding habits for effective binary authorship attribution. In: Lopez, J., Zhou, J., Soriano, M. (eds.) ESORICS 2018. LNCS, vol. 11098, pp. 26–47. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99073-6_2
12. Alsulami, B., Dauber, E., Harang, R., Mancoridis, S., Greenstadt, R.: Source code authorship attribution using long short-term memory based networks. In: Foley, S.N., Gollmann, D., Snekkenes, E. (eds.) ESORICS 2017. LNCS, vol. 10492, pp. 65–82. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66402-6_6

13. Azab, A., Khasawneh, M.: Msic: malware spectrogram image classification. IEEE Access **8**, 102007–102021 (2020)
14. Caliskan-Islam, A., Harang, R., Liu, A., Narayanan, A., Voss, C., Yamaguchi, F., Greenstadt, R.: De-anonymizing programmers via code stylometry. In: 24th {USENIX} Security Symposium ({USENIX} Security 2015), pp. 255–270 (2015)
15. Caliskan-Islam, A., et al.: When coding style survives compilation: de-anonymizing programmers from executable binaries. In: The Network and Distributed System Security Symposium (NDSS 2018) (2018)
16. Chouchane, R., Stakhanova, N., Walenstein, A., Lakhotia, A.: Detecting machine-morphed malware variants via engine attribution. J. Comput. Virol. Hack. Tech. **9**(3), 137–157 (2013). https://doi.org/10.1007/s11416-013-0183-6
17. Dauber, E., Caliskan-Islam, A., Harang, R., Greenstadt, R.: Git blame who?: stylistic authorship attribution of small, incomplete source code fragments. arXiv preprint arXiv:1701.05681 (2017)
18. Ding, H., Samadzadeh, M.H.: Extraction of java program fingerprints for software authorship identification. J. Syst. Softw. **72**(1), 49–57 (2004)
19. Frantzeskou, G., Stamatatos, E., Gritzalis, S., Chaski, C., Howald, B.: Identifying authorship by byte-level n-grams: the source code author profile (scap) method. Int. J. Digit. Evid. **6** (2007)
20. Gonzalez, H., Stakhanova, N., Ghorbani, A.A.: Authorship attribution of android apps. In: Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy, CODASPY 2018, pp. 277–286. Association for Computing Machinery, New York (2018)
21. Haddadpajouh, H., Azmoodeh, A., Dehghantanha, A., Parizi, R.M.: Mvfcc: a multi-view fuzzy consensus clustering model for malware threat attribution. IEEE Access **8**, 139188–139198 (2020)
22. Haralick, R.M., Shanmugam, K., Dinstein, I.: Textural features for image classification. IEEE Trans. Syst. Man Cybern. SMC **3**(6), 610–621 (1973)
23. Heitman, C., Arce, I.: Barf: a multiplatform open source binary analysis and reverse engineering framework. In: XX Congreso Argentino de Ciencias de la Computación (Buenos Aires 2014) (2014)
24. Hendrikse, S.: The Effect of Code Obfuscation on Authorship Attribution of Binary Computer Files. Ph.D. thesis, Nova Southeastern University (2017)
25. Jain, A., Gonzalez, H., Stakhanova, N.: Enriching reverse engineering through visual exploration of android binaries. In: Proceedings of the 5th Program Protection and Reverse Engineering Workshop, pp. 1–9 (2015)
26. Ji, J.H., Woo, G., Cho, H.G.: A plagiarism detection technique for java program using bytecode analysis. In: Third International Conference on Convergence and Hybrid Information Technology, 2008, ICCIT 2008, vol. 1, pp. 1092–1098. IEEE (2008)
27. Kalgutkar, V., Kaur, R., Gonzalez, H., Stakhanova, N., Matyukhina, A.: Code authorship attribution: methods and challenges. ACM Comput. Surv. **52**(1) (2019)
28. Kalgutkar, V., Stakhanova, N., Cook, P., Matyukhina, A.: Android authorship attribution through string analysis. In: Proceedings of the 13th International Conference on Availability, Reliability and Security. ARES 2018. Association for Computing Machinery, New York (2018)
29. Kaur, R., Ning, Y., Gonzalez, H., Stakhanova, N.: Unmasking Android obfuscation tools using spatial analysis. In: 2018 16th Annual Conference on Privacy, Security and Trust (PST), pp. 1–10. IEEE (2018)

30. Kothari, J., Shevertalov, M., Stehle, E., Mancoridis, S.: A probabilistic approach to source code authorship identification. In: Fourth International Conference on Information Technology, 2007, ITNG 2007, pp. 243–248. IEEE (2007)
31. Kurtukova, A., Romanov, A., Shelupanov, A.: Source code authorship identification using deep neural networks. Symmetry **12**(12), 2044 (2020)
32. Mayer, P., Bauer, A.: An empirical analysis of the utilization of multiple programming languages in open source projects. In: Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering, EASE 2015. Association for Computing Machinery, New York (2015)
33. Meng, X., Miller, B.P.: Binary code multi-author identification in multi-toolchain scenarios (2018)
34. Nataraj, L.: A signal processing approach to malware analysis. University of California, Santa Barbara (2015)
35. Nataraj, L., Karthikeyan, S., Jacob, G., Manjunath, B.S.: Malware images: visualization and automatic classification. In: Proceedings of the 8th International Symposium on Visualization for Cyber Security, pp. 1–7 (2011)
36. Ojala, T., Pietikainen, M., Maenpaa, T.: Multiresolution gray-scale and rotation invariant texture classification with local binary patterns. IEEE Trans. Pattern Anal. Mach. Intell. **24**(7), 971–987 (2002)
37. Prechelt, L., Malpohl, G., Philippsen, M.: Finding plagiarisms among a set of programs with jplag. J. UCS **8**(11), 1016 (2002)
38. Rosenberg, I., Sicard, G., David, E.O.: DeepAPT: nation-state APT attribution using end-to-end deep neural networks. In: Lintas, A., Rovetta, S., Verschure, P.F.M.J., Villa, A.E.P. (eds.) ICANN 2017. LNCS, vol. 10614, pp. 91–99. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68612-7_11
39. Rosenblum, N., Zhu, X., Miller, B.P.: Who wrote this code? identifying the authors of program binaries. In: Atluri, V., Diaz, C. (eds.) ESORICS 2011. LNCS, vol. 6879, pp. 172–189. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23822-2_10
40. Taylor, C., Colberg, C.: A tool for teaching reverse engineering. In: 2016 USENIX Workshop on Advances in Security Education (ASE 16). Austin, TX (2016)
41. Ullah, F., Jabbar, S., Al-Turjman, F.: Programmers' de-anonymization using a hybrid approach of abstract syntax tree and deep learning. Technol. Forecast. Social Change **159**, 120186 (2020)
42. Ullah, F., Wang, J., Jabbar, S., Al-Turjman, F., Alazab, M.: Source code authorship attribution using hybrid approach of program dependence graph and deep learning model. IEEE Access **7**, 141987–141999 (2019)
43. Wang, L., He, D.C.: Texture classification using texture spectrum. Pattern Recogn. **23**(8), 905–910 (1990)
44. Zafar, S., Sarwar, M.U., Salem, S., Malik, M.Z.: Language and obfuscation oblivious source code authorship attribution. IEEE Access **8**, 197581–797596 (2020)
45. Zhang, L., Thing, V.L., Cheng, Y.: A scalable and extensible framework for android malware detection and family attribution. Comput. Secur. **80**, 120–133 (2019)