On the Feasibility of Malware Authorship Attribution

Saed Alrabaee(✉), Paria Shirani, Mourad Debbabi, and Lingyu Wang

Concordia University, Montreal, Canada
s_alraba@encs.concordia.ca

Abstract. There are many occasions in which the security community is interested to discover the authorship of malware binaries, either for digital forensics analysis of malware corpora or for thwarting live threats of malware invasion. Such a discovery of authorship might be possible due to stylistic features inherent to software codes written by human programmers. Existing studies of authorship attribution of general purpose software mainly focus on source code, which is typically based on the style of programs and environment. However, those features critically depend on the availability of the program source code, which is usually not the case when dealing with malware binaries. Such program binaries often do not retain many semantic or stylistic features due to the compilation process. Therefore, authorship attribution in the domain of malware binaries based on features and styles that will survive the compilation process is challenging. This paper provides the state of the art in this literature. Further, we analyze the features involved in those techniques. By using a case study, we identify features that can survive the compilation process. Finally, we analyze existing works on binary authorship attribution and study their applicability to real malware binaries.

1 Introduction

Authorship attribution comprises an important aspect of many forensic investigations, which is equally true in the computer world. When a malware attacks computer systems and leaves behind a malware corpus, an important question to ask is '*Who wrote this malware?*'. By narrowing down the authorship of a malware, important insights may be gained to indicate the origin of the malware, to correlate the malware to previously known threats, or to assist in developing techniques for thwarting future similar malware. Considering the fact that humans are creatures of habit and habits tend to persist, therefore, various patterns may be embedded into malware when their creators follow their habitual styles of coding.

Although significant efforts have been made to develop automated approaches for source code [18,34,41], such techniques typically rely on features that will likely be lost in the strings of bytes representing binary code after the compilation process (e.g., variable and function renaming, comments, and code organization, or the development environment, such as programming languages and

F. Cuppens et al. (Eds.): FPS 2016, LNCS 10128, pp. 256–272, 2017.
DOI: 10.1007/978-3-319-51966-1_17

text editors). Identifying the author of a malware binary might be possible but challenging. Such identification must be based on features of the malware binary that are considered to be author specific, which means those features must show only small variations in the writing of different programs by the same author and large such variations over the writing by different authors [41]. That is, authorship identification requires stylistic features that depend on authorship of the code, instead of any other properties, such as functionality. This fact implies that most existing malware analysis techniques will not be directly applicable to authorship attribution. On the other hand, several papers show that the stylistic features are abundant in binaries [13,19,38], and it may be practically feasible to identify the authorship with acceptable accuracy. Another challenge unique to malware authorship attribution is that, while software code may take many forms, including sources files, object files, binary files, and shell code, the malign nature of a malware usually dictates the focus on binary code due to the lack of source code.

In this paper, we investigate the state of the art on binary code authorship techniques and analyze them. More specifically, we first present the survey of existing techniques that are related to the analysis of authorship attribution. This paper covers related work on different representations of malware, including both source files and binaries. Second, we also look at a broader range of work on general purpose malware analysis in order to study which features are involved. Such a comprehensive study of features will allow us to consider a rich collection of features before selecting those which potentially survive the compilation process and are present in the binary code. Third, we analyze and compare binary authorship attribution systems [13,19,38]. Besides, we study their applicability to real malware binaries. Based on our analysis, we provide many important steps that should be considered by reverse engineers, security analysts, and researchers when they deal with malware authorship attribution.

2 Authorship Attribution

In this section, we review the state of the art in the broad domain of authorship attribution, including some techniques proposed for malware analysis. An important goal of this study is to collect a rich list of features that are potentially relevant to malware authorship attribution.

2.1 Source Code Authorship Attribution

Investigating source code authorship attribution techniques can help us understand the features that are likely preserved during the compilation process. Several studies have shown that certain programmers or types of programmers usually hold some features of programming. Examples are layout (spacing, indentation and boarding characters, etc.), style (variable naming, choice of statements, comments, etc.) and environment (computer platform, programming language, compiler, text editor, etc.). The authorship identification of source

codes has been gaining momentum since the initial empirical work of Krsul [34]. Krsul et al. described different important applications of source code authorship techniques and found that style-related features could be extracted from malicious code as well. Burrows [16] and Frantzeskou et al. [26] use n-grams with ranking methods. Burrows and Frantzeskou have both proposed information retrieval approaches with n-grams for source code authorship attribution.

Kothari et al. [33] first collected sample source code of known authors and created profiles by using metrics extraction and filtering tools. In addition, they used style-based and character sequences metrics in classifying the particular developer. Chen et al. [22] proposed a semantic approach for identifying authorship by comparing program data flows. More specifically, they computed the program dependencies, program similarities, and query syntactic structure and data flow of the program. Burrows et al. [17] introduced an approach named Source Code Author Profile (SCAP) using byte level n-gram technique. The author claimed that the approach is language independent and n-gram profiles would represent a better way than traditional methods in order to find the unique behavioral characteristics of a specific source code author. Jang et al. [28] performed experiments to find a set of metrics that can be used to classify the source code author. They worked on extracting the programming layout, style, structure, and fuzzy logic metrics to perform the authorship analysis. Yang et al. [43] performed experiments to support the theory that a set of metrics can be utilized to classify the programmer correctly within the closed environment and for a specific set of programmers. With the help of programming metrics, they suggested developing a signature of each programmer within a closed environment. They used two statistical methods, cluster and discriminant analysis. They did not expect that metrics gathered for a programmer would remain an accurate tag for a long time. It is obvious that a one-time metrics gathering is not enough, as this should be a continuous task. The practice of authorship analysis includes metrics extraction, data analysis and classification.

A separate thread of research focuses on plagiarism detection, which is carried out by identifying the similarities between different programs. For example, there is a widely-used tool called Moss that originated from Stanford University for detecting software plagiarism [12]. More recently, Caliskan-Islam et al. [18] investigated methods to de-anonymize source code authors of C++ using coding style. They modeled source code authorship attribution as a machine learning problem using natural language processing techniques to extract the necessary features. The source code is represented as an abstract syntax tree, and the properties are driven from this tree.

2.2 Binary Code Authorship Attribution

In contrast to source code, binary code has drawn significantly less attention with respect to authorship attribution. This is mainly due to the fact that many salient features that may identify an author's style are lost during the compilation process. In [13,19,38], the authors show that certain stylistic features can indeed survive the compilation process and remain intact in binary code,

which leads to the feasibility of authorship attribution for binary code. The methodology developed by Rosenblum et al. [38] is the first attempt to automatically identify authors of software binaries. The main concept employed by this method is to extract syntax-based and semantics-based features using predefined templates, such as idioms (sequences of three consecutive instructions), n-grams, and graphlets. Machine learning techniques are then applied to rank these features based on their relative correlations with authorship. A subsequent approach to automatically identify the authorship of software binaries is proposed by Alrabaee et al. [13]. The main concept employed by this method is to extract a sequence of instructions with specific semantics and to construct a graph based on register manipulation, where a machine learning algorithm is applied afterwards. A more recent approach to automatically identify the authorship of software binaries is proposed by Caliskan et al. [19]. They extract syntactical features present in source code from decompiled executable binary. Though these approaches represent a great effort on authorship attribution, it should be noted that they were not applied to real malware. Further, some limitations could be observed including weak accuracy in the case of multiple authors, being potentially thwarted by light obfuscation, and their inability to decouple features related to functionality from those which are related to authors' styles.

3 Study of Features

In this section, we present a more elaborated study of features collected during the literature review.

3.1 Features of Source Files

Program source code provides a far richer basis for writer-specific programming features. Our goal is to determine which features may survive the compilation process and be helpful for authorship identification of binary code.

Linguistic Features: Programming languages allow developers to express constructs and ideas in many ways. Differences in the way developers express their ideas can be captured in their programming styles, which in turn can be used for author identification [40]. The linguistic style is used to analyze the differences in the literary techniques of authors. Researchers have identified over 1,000 characteristics, or style markers, such as comments, to analyze literary works [20]. Moreover, it has been used to identify the author by capturing, examining, and comparing style markers [27].

Formatting: The source code formatting shows a very personal style. Formatting is also considered as a good way for programmers to make it easier when reading what was written. These factors indicate that the formatting style of code should yield writer-specific features [34]: Placement of statement delimiters, Multiple statements per line, Format of type declarations, Format of function arguments, and Length of comment lines.

Bugs and Vulnerabilities: A written program might have errors or bugs such as buffer overflow, or a pointer to an undefined memory address. These kinds of issues could be an indicator of the author.

Execution Path: The execution path may indicate the author's preference in how resolving a particular task through the selection of algorithms, as well as certain data structures, or using specific keywords such as `while` or `for`.

Abstract Syntax Tree (AST): AST is an intermediate representations produced by code parsers of compilers, and thus forms the basis for the generation of many other code representations. Such tree forms how statements and expressions are nested to produce programs. More specifically, it encompasses inner nodes representing operators (e.g., additions or assignments) and leaf nodes correspond to operands (e.g., constants or identifiers).

Control Flow Graph (CFG): It describes the order in which code statements are executed as well as conditions that need to be met for a particular path of execution to be taken. Statements and predicates are represented by nodes, which are connected by directed edges to indicate the transfer of control. For each edge, there is a label of true, false or unconditional control.

Program Dependence Graph (PDG): It is introduced by Ferrante et al. [24], which has been originally developed to perform program slicing [42]. This graph determines all statements and predicates of a program that affect the value of a variable at a specified statement.

3.2 Features of Binary Files

Compiler and System Information: A unique sequence of instructions might be an indicator of the compilers. The code may contain different system calls found only in certain operating systems. The analysis of binary code may reveal that it was written in a specific source language such as C++. This can be determined based on support routines and library calls in the binary code.

System Call: It is considered as programmatic way in which a computer program requests a service from the kernel of the operating system it is executed on, for instance, process scheduling with integral kernel services. Such system calls capture intrinsic characteristics of the malicious behavior and thus are harder to evade [21].

Errors: The binary code might have errors or bugs such as buffer overflow, or a pointer to an undefined memory address. These kinds of bugs could be an indicator of the author.

Idioms: An idiom is not really a specific feature, but rather a feature template that captures low-level details of the sequence underlying a program. Idioms are short sequences of instructions. A grammar for idiom feature follows the Backus-Naur [32] form.

Graphlet: A graphlet is an intermediary representation between the assembly instructions and the Control Flow Graph, which represents the details of a program structure [36], and is represented as a small connected non-isomorphic induced sub-graph of a large network [23]. Graphlets were first introduced by Prvzulj et al. [36] for designing two new highly sensitive measures of network locality, structural similarities: the relative graphlet frequency distance [23], and the graphlet degree distribution agreement [38].

n-grams: The n-gram feature was first used by an IBM research group [30]. An n-gram is an n-character slice of a longer string. A string is simply split into substrings of fixed length n. For example, the string 'MALWARE' can be segmented into the following 4-grams: 'MALW', 'ALWA', 'LWAR', and 'WARE'.

Opcode: An opcode is the portion of an assembly instruction that specifies the action to be performed, for instance, `jmp`, `lea`, and `pop`. Opcode sequences have recently been introduced as an alternative to byte n-grams [35]. Some of the opcodes (e.g. `push` or `mov`) have a high frequency of appearance within an executable file. In [39] is shown that the opcodes by themselves were capable to statistically explain the variability between malware and legitimate software.

Strings and Constants: The type of constants that used in the literature is integers, which are used in computation, as well as integers used as pointer offsets. The strings are ANSI single-byte null-terminated strings [31].

Register Flow Graph: This graph captures the flow and dependencies between the registers that annotated to `cmp` instruction [13]. Such graph can capture an important semantic aspects about the behavior of a program, which might indicate the author's skills or habits.

4 Implementation

This section shows the setup of our experiments and provides an overview of the collected data.

4.1 Implementation Environment

The described binary feature extractions are implemented using separate python scripts for modularity purposes, which altogether form our analytical system. A subset of the python scripts in our evaluation system is used in tandem with IDA Pro disassembler [4]. The Neo4j [10] graph database is utilized to perform complex graph operations such as k-graph (graphlet) extraction. Gephi [9] is used for all graph analysis functions (e.g., page rank) that are not provided by Neo4j. The PostgreSQL database is used to store extracted features according to its efficiency and scalability. For the sake of usability, a graphical user interface in which binaries can be uploaded and analyzed is implemented.

4.2 Dataset

The utilized dataset is composed of several files from different sources, as described below: (i) GitHub [3], where a considerable amount of real open-source projects are available; (ii) Google Code Jam [2], an international programming competition, where solutions to difficult algorithmic puzzles are available; and (iii) a set of known malware files representing a mixture of nine different families [7] provided in Microsoft Malware Classification Challenge. According to existing works, we only examine code written in C/C++. These programs are either open-source or publicly available, in which case the identities of the authors are known. Statistics about the dataset are provided in Table 1.

Table 1. Statistics about the binaries used in the evaluation

Source	# of authors	# of programs	# of functions
GitHub	50	150	40000
Google Code Jam	120	550	1065
Malware	9	36	15000
Total	179	736	46065

4.3 Dataset Compilation

To construct our experimental datasets, we compile the source code with different compilers and compilation settings to measure the effects of such variations. We use GNU Compiler Collection's gcc, Xcode, ICC, as well as Microsoft Visual Studio (VS) 2010, with different optimization levels.

4.4 Implementation Phases

The original binaries are passed to the pre-processing component, where are disassembled with IDA Pro disassembler. The second component contains two processes: (1) ASMTODB, which extracts some specific features (e.g., idioms) from the assembly files, and (2) BINTODB, which extracts the features directly from the binary files. The result of this stage is a set of features stored in the database. This phase also implements the feature ranking which is a pre-processing phase for classification.

4.5 Feature Ranking

Feature ranking is a pre-processing phase for classification. We assume that there exists a known set of program authors and a set of programs written by each of them. The task of the feature ranking algorithm is to associate the identity

of the most likely author of a feature. We extract features from the program assemblies and binaries as described in the previous section in order to obtain the feature list associated with a specific author. We apply mutual information and information gain applied in Rosenblum et al. [38] and Islam et al. [19], respectively.

4.6 SQL Schema to Store All Features

Storing, ranking and processing the features in the classification phase require an appropriate SQL schema. We have chosen the PostgreSQL database system, and designed our SQL tables, the relations between them, together with the Features-to-DB APIs, so that our software modules minimize their interaction with the database.

4.7 Authorship Classification

The authorship classification technique assumes that a known set of authors with their program samples are collected. After extracting and ranking features, as described in the previous subsection, a classifier is built based on the top-ranked features, producing a decision function that can assign a label (authorship) to any given new program based on the given set of known authors. More specifically, the typical steps for authorship classification are the following:

1. Each program is first represented as an integral-valued feature vector describing those features that are present in the program.
2. Those features are ordered using the aforementioned ranking algorithm based on the mutual information between the features and the known author labels. A given number of top-ranked features are selected, and others filtered out in order to reduce both the training cost and risk of overfitting the data.
3. A cross-validation is performed on those highly-ranked features. Those features would jointly produce a good decision function for the authorship classifier.
4. The LIBLINEAR support vector machine for the actual classification is employed for the actual classification.

5 Evaluation

In this section, we present the evaluation results for the existing works on binary authorship attribution. Subsequently, we evaluate the identification and the scalability of existing works. The impact of evading techniques is then studied. Finally, binary features are applied to real malware binaries and the results are discussed.

5.1 Accuracy

The main purpose of this experiment is to demonstrate how to evaluate the accuracy of author identification in binaries.

Evaluation Settings. The evaluation of existing works is conducted using the datasets described in Sect. 4. The data is randomly split into ten sets, where one set is reserved as a testing set, and the remaining sets are used as training sets to evaluate the system. The process is then repeated 15 times (according to existing works). Furthermore, since the application domain targeted by binary authorship attribution works is much more sensitive to false positives than false negatives, we employ an F-measure as follows:

$$F_{0.5} = 1.25 \cdot \frac{P \cdot R}{0.25P + R} \tag{1}$$

Existing Works Comparison. We evaluate and compare the existing authorship attribution methods [13,18,38]. For this purpose the source code and the used database are needed. The source code of the authorship classification techniques presented by Rosenblum et al. [38] and Caliskan-Islam et al. [18] are available at [5,8], respectively; however the datasets are not available. For the system proposed by Alrabaee et al. (OBA2) [13], we have asked the authors to provide us the source code.

Caliskan-Islam et al. present the largest scale evaluation of binary authorship attribution in related work, which contains *600* authors with *8* training programs per author. Rosenblum et al. present a large-scale evaluation of *190* authors with at least *8* training programs, while Alrabaee et al. present a small scale evaluation of 5 authors with *10* programs for each. Since the datasets used by the aforementioned techniques are not available, we compare our results with these methods using the same datasets mentioned in Table 1. The number of features used in Caliskan-Islam et al., Rosenblum et al., and Alrabaee et al. systems are *4500*, *10000*, and *6500*, respectively.

Figure 1 details the results of comparing the accuracy between existing methods. It shows the relationship between the accuracy ($F_{0.5}$) and the number of authors present in all datasets, where the accuracy decreases as the size of author population increases. The results show that Caliskan-Islam et al. approach achieves better accuracy in determining the author of binaries. Taking all three approaches into consideration, the highest accuracy of authorship attribution is close to *90%* on the Google Code Jam dataset with less than *20* authors, while the lowest accuracy is *45%* when *179* authors are involved.

As can be seen in Fig. 1, OBA2 achieves good accuracy when it deals with small scale of authors. For instance, the accuracy is approximately *84%* on GitHub dataset when the number of authors is *30*, while the accuracy drops to *58%* on the same dataset when the number of authors increases to *50*. The main reason is due to the fact that the authors of projects in Github have no restrictions when developing projects. The lower accuracy obtained by OBA2

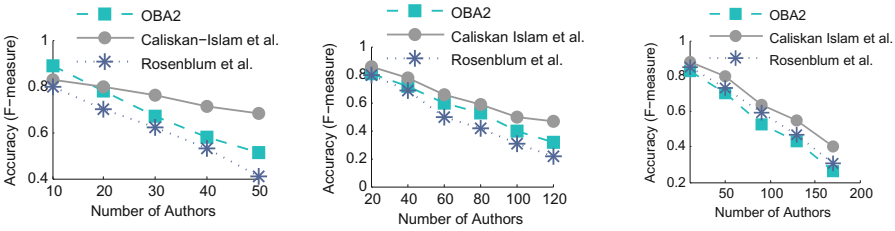

Fig. 1. Accuracy results of authorship attribution obtained by Caliskan-Islam et al. [18], Rosenblum et al. [38], and OBA2 [13], on (a) Github, (b) Google Code Jam, and (c) All datasets.

is approximately *28%* on all datasets when the number of authors is *179*. The accuracy of Rosenblum et al. approach drops rapidly to *43%*, whereas Caliskan-Islam et al. system accuracy remains greater than *60%*, if the *140* authors on all datasets are considered.

5.2 Scalability

We evaluate how well existing works scale up to *1000* authors. Since in the case of malware, an analyst may be dealing with an extremely large number of new samples on a daily basis. For this experiment, we work with *1000* users, of which are authors from the Google Code Jam. First, we extract the top-ranked features as described in Rosenblum et al. and Caliskan-Islam et al. approaches, while the features used by OBA2 are not ranked.

The results of large-scale author identification are shown in Fig. 2. As seen in Fig. 2, by increasing the number of authors, all the existing works accuracy drops significantly. For instance, Rosenblum et al. approach accuracy drops to approximately *5%* when the number of authors is greater than *600* authors. While the accuracy of OBA2 approach drops to *15%* when the number of authors reaches to *500*. However, Caliskan-Islam et al. approach accuracy drops to *20%* with an increase to over *700* authors.

Through our experiments we have observed that top-ranked features used in the Rosenblum et al. approach are mixture of compiler and user features, where leads to higher rate in false positives. OBA2 identifies author according to the way of branch handling. Therefore, when the number of authors is largely increased, distinguishing the author based on handling branches becomes limited and hard. Finally, Caliskan-Islam et al. approach relies on the features extracted from AST of compiled binary; so with the large number of authors, these features became common and similar which make the authorship attribution harder.

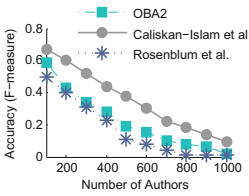

Fig. 2. Large-scale author identification results

5.3 Impact of Evading Techniques

In this subsection, we apply different techniques to evade the existing systems in order to study their stability. For this purpose, we randomly choose *50* authors and *8* programmes for each author. The accuracy results without applying any evading technique, and with applying evading techniques are shown in Table 2.

Refactoring Techniques. The adversary may use existing refactoring techniques to prevent authorship attribution. Hence, we use chosen dataset for the C^{++} refactoring process [1,11]. We consider the techniques of (i) Renaming a Variable (RV), (ii) Moving a Method from a superclass to its subclasses (MM), and (iii) extracting a few statements and placing them into a New Method (NM). Depth explanations of these techniques are detailed in [25]. We obtain an accuracy of *81%* in correctly classifying authors for OBA2 system, which drops to *62%* when RV is applied. The reason of this dropping in accuracy is that variable renaming affects the features used by OBA2, while OBA2 can tolerate NM, and MM. The accuracy of Caliskan-Islam et al. approach drops not greatly from *79%* to *70%*. This is due to the fact that their approach decompiles the binary into source code, and then extracts the features. Hence, the aforementioned refactoring techniques do not change much in the abstract syntax tree. However, the approach can tolerate renaming variables. Finally, Rosenblum et al. approach is the one that is mostly affected by Refactoring techniques, where the accuracy drops from *66%* to *40%*. Since their approach extracts idioms from assembly files, any of these techniques will change the idioms (sequence of assembly instructions) which cause a drop in accuracy.

The Impact of Obfuscation. We are interested in determining how existing works handle simple binary obfuscation techniques intended for evading detection, as implemented by tools such as Obfuscator-LLVM [29]. These obfuscators could apply Instruction Replacement (IR): replacing instructions by other semantically equivalent instructions, Dead Code Insertion (DCI), Register Renaming (RR), spurious control flow insertion, and can even completely Flatten Control Flow graphs (FCF). Obfuscation techniques implemented by Obfuscator-LLVM are applied to the samples prior to classifying the authors. Caliskan-Islam et al. approach is the most affected approach by FCF technique; since control flow flattening makes the decompilation process hard, which means the features cannot be extracted correctly.

Table 2. Accuracy results before and after applying refactoring techniques, obfuscation techniques, and different compilers. (AbET): Accuracy before Evading Techniques, (∼): The accuracy has not affected.

System	AbET	*Refactoring*				*Obfuscation*					*Compiler*		
		RV	NM	MM	All	RR	IR	DCI	FCF	All	GCC	Xcode	ICC
OBA2	0.81	0.62	∼	∼	0.62	0.64	0.74	∼	∼	0.58	0.74	0.60	0.54
Caliskan-Islam	0.79	∼	0.72	0.71	0.70	∼	∼	∼	0.24	0.24	0.66	0.64	0.54
Rosenblum	0.66	0.60	0.58	0.55	0.4	0.62	∼	∼	0.27	0.25	0.15	0.55	0.29

The Impact of Compilers. To create experimental datasets for this purpose, we first compile the source code with GCC, VS, ICC, and Xcode compilers. Next, the effect of different compilation options, such as the source of compiler, is measured. The results show that the approach which is mostly affected by changing the compiler is Rosenblum et al.'s approach; since this approach does not distinguish between user functions or compiler functions. For instance, the accuracy observed through our experiments is 15%, when the binaries are compiled with GCC, because the GCC compiler inserts many compiler functions.

5.4 Applying Existing Works to Malware Binaries

We apply existing works to different sets of real malware: `Ramnit`, `Lollipop`, `Kelihos`, `Vundo`, `Simda`, `Tracur`, `Obfuscator.ACY`, and `Gatak`. These malware are selected due to their availability [7]. These samples contain different variants of the same malware so we assume that these variants are written by the same author or the same group of authors. Due to the lack of ground truth, we compare outputs of each approach manually to verify that they belong to same family. Details about the malware dataset are shown in Table 3. The number of compiler functions are obtained based on [37], while the fifth column shows the number of library functions acquired by F.L.I.R.T technology [4]. According to Table 3, we can observe that the percentage of compiler functions is quite high, so a pre-processing step before applying authorship attribution approaches would be demanding. For instance, the percentage of compiler functions in `Lollipop` family is *30%*. We apply existing works and cluster functions according to their features by using standard k-mean. Then we manually analysis the obtained clusters to classify them to correct/wrong clusters as shown in Table 4.

Table 3. Characteristics of malware datasets. (BF): binary functions, (CF): compiler functions, (LF): library function.

Malware	# of variants	# of BF	# of CF	# of LF
`Ramnit`	4	5285	1601	50
`Lollipop`	3	3510	1054	100
`Kelihos`	2	1924	847	74
`Vundo`	4	7923	2410	219
`Simda`	2	2100	689	105
`Tracur`	2	1657	787	100
`Obfuscator.ACY`	3	2762	986	310
`Gatak`	2	2054	860	174

Table 4. Clustering results based on the features used in existing systems. (TC): the total number of clusters, (CC): the percentage of correct clusters, (WC): the percentage of wrong clusters.

Malware	*OBA2*			*Caliskan-Islam*			*Rosenblum*		
	TC	CC	WC	TC	CC	WC	TC	CC	WC
`Ramnit`	145	60%	30%	110	47%	50%	208	18%	70%
`Lollipop`	90	75%	14%	185	59%	38%	220	21%	67%
`Kelihos`	41	88%	8%	17	90%	4%	75	34%	55%
`Vundo`	200	62%	14%	89	28%	68%	384	39%	48%
`Simda`	52	49%	50%	41	92%	5%	109	42%	51%
`Tracur`	44	89%	9%	53	83%	12%	124	51%	40%
`Obfuscator.ACY`	30	78%	21%	45	74%	24%	89	29%	70%
`Gatak`	29	57%	34%	51	87%	12%	79	38%	62%

6 Learnt Lessons and Concluding Remarks

Functionality or styles: During the evaluation, we have observed that the features selected by existing techniques are more closely related to the functionality of the program rather than the author's style. This argument may be supported by the evidence that a basic short program has less features than comparatively bigger, functionality-oriented programs. This shows that features are directly related to the size of the program, which usually depicts functionality but not style [13,18,38]. In order to avoid this, some existing systems could be used as preprocessing stage [14,15] applies different steps.

Feature pre-processing: We have encountered top-ranked features related to the compiler (e.g., stack frame set-up operation). It is thus necessary to filter irrelevant functions (e.g., compiler functions) in order to better identify author-related portions of code [38]. To avoid this, a filtration method based on the FLIRT technology for library identification as well as a system for compiler functions filtration such as BinComp [37] should be used. Successful distinction between the two groups of functions (library/compiler and user functions) will lead to considerable savings in time and will help shift the focus of analysis to more relevant functions.

Application type: We find that the accuracy of existing methods [13,38] depends highly on the application's domain. For example, in Fig. 1, superior accuracy is observed for the Google Code Jam dataset where the accuracy is 77% in average. This is because the approach used by Rosenblum et al. extracts SysCalls, which are more useful in the case of academia/competition code than in other cases. This can be explained by the authors' choice to systematically rely on external libraries and to implement, for instance, MFC APIs. The results also show that Alrabaee et al. rely on the application because their approach

extracts the manner by which the author handles branches; for instance, the accuracy drops from *82%* to *57%* when Google Code Jam is used. After investigating the source code, we notice that the number of branches is not big, which makes the attribution even more difficult.

The source of features: Caliskan et al. [18] use a decompiler to translate the program into C-like pseudo code via Hex-Ray [6]. They pass the code to a fuzzy parser, thus abstract syntax tree is obtained, which is the source of feature extraction. In addition to Hex-Ray limitations [6], the C-like pseudo code is also different from the original code to the extent that the variables, branches, and keywords are different. For instance, we find that a function in the source code consists of the following keywords: (`1-do, 1-switch, 3-case, 3-break, 2-while, 1-if`) and the number of variables is 2. Once we check the same function after decompilation, we find that the function consists of the following keywords: (`1-do, 1-else/if, 2-goto, 2-while, 4-if`) and the number of variables is 4. This will evidently lead to misleading features.

Misleading Features: To make things worse, our re-evaluation results show that many top-ranked features are in fact completely unrelated to authors' styles. For example, many source code-level functions do not have their names identified at binary level, i.e., IDA Pro assigns a name prefixed with "sub" and postfixed with randomly generated numbers by the compiler. Experiments show that these functions with random numbers play a vital role for features to be ranked high by calculating the mutual information. This discovery shows that this technique may select many features unrelated to author styles but rather some other properties, such as compiler-generated functions.

Concluding Remarks: Binary code authorship attribution is a less explored problem compared with source code level authorship attribution due to many facts (e.g., the reverse engineering is time consuming, having limited features preserved during the compilation process). In this paper, we have first presented a literature review relevant to authorship identification of binary and source code. Subsequently, we introduce the way of extracting binary features. Then, we deeply analysis and evaluate the existing works on different scenarios such as scalability. Finally, we applied them to real set of malware binaries. It is clear that there exist many features that could potentially be useful to determine malware authorship. However, the harder part is to verify their applicability through experimental studies. We must pay special care to the following issues when we deal with binary authorship attribution:

- Dataset Size: A small amount of training set code might not be sufficient to make a good identification and a precise comparison unless very unusual indicators are present.
- Multiple Authors: The identification of authors in the case of multiple authors will be more challenging, since we have to first identify code fragments that are written by the same author.

Acknowledgments. The authors thank the anonymous reviewers for their valuable comments. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsoring organizations.

References

1. Refactoring tool. https://www.devexpress.com/Products/CodeRush/
2. The Google Code Jam (2008–2015). http://code.google.com/codejam/
3. GitHub-Build software better (2011). https://github.com/trending/cpp
4. IDA pro Fast Library Identification and Recognition Technology (2011). https://www.hex-rays.com/products/ida/tech/
5. The materials supplement for the paper: Who Wrote This Code? Identifying the Authors of Program Binaries (2011). http://pages.cs.wisc.edu/~nater/esorics-supp/
6. Hex-Ray decompiler (2015). https://www.hex-rays.com/products/decompiler/
7. Microsoft Malware Classification Challenge (BIG 2015) (2015). https://www.kaggle.com/c/malware-classification/data
8. Programmer De-anonymization from Binary Executables (2015). https://github.com/calaylin/bda
9. The Gephi plugin for nneo4j (2015). https://marketplace.gephi.org/plugin/neo4j-graph-database-support/
10. The Scalable Native Graph Database (2015). http://neo4j.com/
11. C++ refactoring tools for visual studio (2016). http://www.wholetomato.com/
12. Aiken, A., et al.: Moss: a system for detecting software plagiarism. University of California–Berkeley (2005). www.cs.berkeley.edu/aiken/moss.html 9
13. Alrabaee, S., Saleem, N., Preda, S., Wang, L., Debbabi, M.: Oba2: an onion approach to binary code authorship attribution. Digit. Invest. **11**, S94–S103 (2014)
14. Alrabaee, S., Shirani, P., Wang, L., Debbabi, M.: Sigma: a semantic integrated graph matching approach for identifying reused functions in binary code. Digit. Invest. **12**, S61–S71 (2015)
15. Alrabaee, S., Wang, L., Debbabi, M.: Bingold: towards robust binary analysis by extracting the semantics of binary code as semantic flow graphs (sfgs). Digit. Invest. **18**, S11–S22 (2016)
16. Burrows, S., Tahaghoghi, S.M.: Source code authorship attribution using n-grams. Citeseer (2007)
17. Burrows, S., Uitdenbogerd, A.L., Turpin, A.: Application of information retrieval techniques for source code authorship attribution. In: Zhou, X., Yokota, H., Deng, K., Liu, Q. (eds.) DASFAA 2009. LNCS, vol. 5463, pp. 699–713. Springer, Heidelberg (2009). doi:10.1007/978-3-642-00887-0_61
18. 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)
19. Caliskan-Islam, A., Yamaguchi, F., Dauber, E., Harang, R., Rieck, K., Greenstadt, R., Narayanan, A.: When coding style survives compilation: de-anonymizing programmers from executable binaries. arXiv preprint arXiv:1512.08546 (2015)
20. Can, F., Patton, J.M.: Change of writing style with time. Comput. Humanit. **38**(1), 61–82 (2004)

21. Canali, D., Lanzi, A., Balzarotti, D., Kruegel, C., Christodorescu, M., Kirda, E.: A quantitative study of accuracy in system call-based malware detection. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis, pp. 122–132. ACM (2012)
22. Chen, R., Hong, L., Lü, C., Deng, W.: Author identification of software source code with program dependence graphs. In: 2010 IEEE 34th Annual Computer Software and Applications Conference Workshops (COMPSACW), pp. 281–286. IEEE (2010)
23. Edwards, N., Chen, L.: An historical examination of open source releases and their vulnerabilities. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, pp. 183–194. ACM (2012)
24. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. (TOPLAS) **9**(3), 319–349 (1987)
25. Fowler, M.: Refactoring: Improving the Design of Existing Code. Pearson Education India, New Delhi (2009)
26. Frantzeskou, G., Stamatatos, E., Gritzalis, S., Katsikas, S.: Source code author identification based on n-gram author profiles. In: Maglogiannis, I., Karpouzis, K., Bramer, M. (eds.) AIAI 2006. IIFIP, vol. 204, pp. 508–515. Springer, Heidelberg (2006). doi:10.1007/0-387-34224-9_59
27. Holmes, D.I.: Authorship attribution. Comput. Humanit. **28**(2), 87–106 (1994)
28. Jang, J., Brumley, D., Venkataraman, S.: Bitshred: feature hashing malware for scalable triage and semantic analysis. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, pp. 309–320. ACM (2011)
29. Junod, P., Rinaldini, J., Wehrli, J., Michielin, J.: Obfuscator-llvm: software protection for the masses. In: Proceedings of the 1st International Workshop on Software Protection, pp. 3–9. IEEE Press (2015)
30. Kephart, J.O., et al.: A biologically inspired immune system for computers. In: Artificial Life IV: Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems, pp. 130–139 (1994)
31. Khoo, W.M., Mycroft, A., Anderson, R.: Rendezvous: a search engine for binary code. In: Proceedings of the 10th Working Conference on Mining Software Repositories, pp. 329–338. IEEE Press (2013)
32. Knuth, D.E.: Backus normal form vs. backus naur form. Commun. ACM **7**(12), 735–736 (1964)
33. Kothari, J., Shevertalov, M., Stehle, E., Mancoridis, S.: A probabilistic approach to source code authorship identification. In: Fourth International Conference on Information Technology, ITNG 2007, pp. 243–248. IEEE (2007)
34. Krsul, I., Spafford, E.H.: Authorship analysis: identifying the author of a program. Comput. Secur. **16**(3), 233–257 (1997)
35. Kruegel, C., Kirda, E., Mutz, D., Robertson, W., Vigna, G.: Polymorphic worm detection using structural information of executables. In: Valdes, A., Zamboni, D. (eds.) RAID 2005. LNCS, vol. 3858, pp. 207–226. Springer, Heidelberg (2006). doi:10.1007/11663812_11
36. Pržulj, N., Corneil, D.G., Jurisica, I.: Modeling interactome: scale-free or geometric? Bioinformatics **20**(18), 3508–3515 (2004)
37. Rahimian, A., Shirani, P., Alrbaee, S., Wang, L., Debbabi, M.: Bincomp: a stratified approach to compiler provenance attribution. Digit. Invest. **14**, S146–S155 (2015)
38. 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). doi:10.1007/978-3-642-23822-2_10

39. Santos, I., Penya, Y.K., Devesa, J., Bringas, P.G.: N-grams-based file signatures for malware detection. In: Proceedings of the ICEIS, vol. 2(9), pp. 317–320 (2009)
40. Shevertalov, M., Kothari, J., Stehle, E., Mancoridis, S.: On the use of discretized source code metrics for author identification. In: 2009 1st International Symposium on Search Based Software Engineering, pp. 69–78. IEEE (2009)
41. Spafford, E.H., Weeber, S.A.: Software forensics: can we track code to its authors? Comput. Secur. **12**(6), 585–595 (1993)
42. Weiser, M.: Program slicing. In: Proceedings of the 5th International Conference on Software Engineering, pp. 439–449. IEEE Press (1981)
43. Yang, K.-X., Hu, L., Zhang, N., Huo, Y.-M., Zhao, K.: Improving the defence against web server fingerprinting by eliminating compliance variation. In: 2010 Fifth International Conference on Frontier of Computer Science and Technology (FCST), pp. 227–232. IEEE (2010)