On Leveraging Coding Habits for Effective Binary Authorship Attribution

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

Security Research Center, Concordia University, Montreal, Canada
s_alraba@encs.concordia.ca

Abstract. We propose *BinAuthor*, a novel and the first compiler-agnostic method for identifying the authors of program binaries. Having filtered out unrelated functions (compiler and library) to detect user-related functions, it converts user-related functions into a canonical form to eliminate compiler/compilation effects. Then, it leverages a set of features based on collections of authors' choices made during coding. These features capture an author's coding habits. Our evaluation demonstrated that *BinAuthor* outperforms existing methods in several respects. First, when tested on large datasets extracted from selected open-source C/C++ projects in GitHub, Google Code Jam events, and Planet Source Code contests, it successfully attributed a larger number of authors with a significantly higher accuracy: around 90% when the number of authors is 1000. Second, when the code was subjected to refactoring techniques, code transformation, or processing using different compilers or compilation settings, there was no significant drop in accuracy, indicating that *BinAuthor* is more robust than previous methods.

1 Introduction

Binary authorship attribution refers to the process of discovering information related to the author(s) of anonymous binary code on the basis of stylometric characteristics extracted from the code. It is especially relevant to security applications, such as digital forensic analysis of malicious code [30] and copyright infringement detection [33] because the source code is seldom available in these cases. However, in practice, authorship attribution for binary code still requires considerable manual and error-prone reverse engineering analysis, which can be a daunting task given the sheer volume and complexity of today's malware. Although significant efforts have been made to develop automated approaches for authorship attribution for source code [19,25,37], such techniques typically rely on features that will likely be lost in the binary code after the compilation process, for example, variable and function naming, original control and data

J. Lopez et al. (Eds.): ESORICS 2018, LNCS 11098, pp. 26–47, 2018.
https://doi.org/10.1007/978-3-319-99073-6_2

flow structures, comments, and space layout. Nonetheless, at the recent Black-Hat conference, the feasibility of authorship attribution for malware binaries was confirmed [5], though the process still requires considerable human intervention.

Most existing approaches to binary authorship attribution employ machine learning methods to extract unique features for each author and subsequently match the features of a given binary to identify the authors [15,19,32]. These approaches were studied and analyzed in our previous work [16], and we uncovered several issues that affect them all. Notably, a considerable percentage of the extracted features are related to compiler functions rather than to author styles, which causes a high false positive rate. Moreover, the extracted features are not resilient to code transformation methods, refactoring techniques, changes in the compilation settings, and the use of different compilers. We implemented a system that improved the accuracy obtained by Caliskan et al. [19] in attributing 600 authors from 83% to 90%, and then we scaled the results to 86% accuracy for 1500 authors.

Key Idea: We present *BinAuthor*, a system designed to recognize author *coding habits* by extracting author's choices from binary code. *BinAuthor*[1] performs a series of steps in order to capture coding habits. First, it filters unrelated functions such as compiler-related functions by proposing a method that is discussed in Sect. 2.1. Second, it labels library-related functions and free open-source related functions using our previous works, BinShape [35], SIGMA [17], and FOSSIL [18], respectively. The results of filtering process would be a set of user-related functions. Third, to eliminate the effects of changes in the compiler or the compilation settings, code transformation, and refactoring tools, *BinAuthor* converts the code into a canonical form that is robust against heavy obfuscation [38]. However, conversion is extremely slow, so we apply it only to the set of user-related functions remaining after filtering. Then we collect a set of author choices frequently made during coding (e.g., preferring to use either `memcopy` or `bcopy`). To capture the choices, we examined a large collection of source code and the corresponding assembly instructions to determine which coding habits may be preserved in the binary. Next, we designed features based on these habits and integrated them into *BinAuthor*. To verify that the features capture coding habits, we investigated the ground truth source code in a controlled experiment (using debug information) to determine if the choices are based on functionality or habit.

Contributions: The main contributions of this study are described below.

1. To the best of our knowledge, *BinAuthor* is the first effort that leverages author *coding habits* extracted from binary code for effective binary authorship attribution. This enables *BinAuthor* to work on programs with different functionalities.
2. *BinAuthor* achieves higher accuracy and survives refactoring techniques and code transformation techniques. This shows its potential for use as a practical tool that can assist reverse engineers in many security-related tasks.

[1] The code is available at https://github.com/g4hsean/BinAuthor.

3. *BinAuthor* is among the first approaches that performs automated authorship attribution on real-world malware binaries. When we applied it to `Zeus-Citadel`, `Stuxnet-Flame`, and `Bunny-Babar` malware binaries, it automatically generated evidence of coding habits shared by each malware pair, matching the findings of antivirus vendors [3,12] and reverse engineering teams [5].

2 BinAuthor

We propose a system encompassing different components, each of which is meant to achieve a particular purpose, as illustrated in Fig. 1. The first component (*Filtration*), isolates user functions from compiler functions, library functions, and open-source software packages. For this purpose, we employ BinShape, and FOSSIL tools developed by our team beside our proposed method to identify compiler functions. Hence, additional outcome of this component could be considered as a choice (e.g., the preference in using specific compiler or open-source software packages). The second component (*Canonicalization*), adapts the existing framework angr [36] for lifting function into LLVM-IR, then optimizes the lifted LLVM-IR, and finally converts the optimized IR into a canonical form. The third component (*Choices*), analyzes user-related functions to extract possible features that represent stylistic choices and then converts the extracted choices into vectors. The vector of choices are used by the attribution probability function in the last component (*Classification*). The aforementioned components are explained in depth in the remainder of this section.

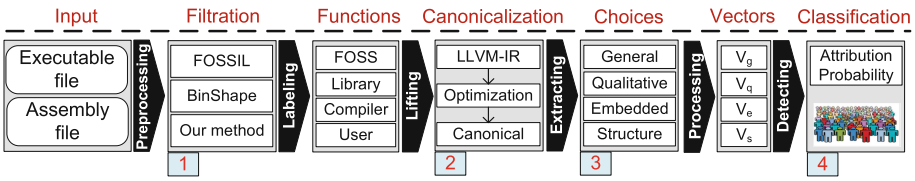

Fig. 1. *BinAuthor* architecture

2.1 Filtration Process

An important initial step in most reverse engineering tasks is to distinguish between user functions and library/compiler functions. This step saves considerable time and helps shift the focus to more relevant functions. The filtration process consists of three steps. First, Binshape [35] is used to label library functions. Second, FOSSIL [18] is leveraged to label the functions that are related to specific FOSS libraries, such as `libpng`, `zlib`, and `openssl`. The last step filters compiler-related functions, which the details are given below.

The idea is based on the hypothesis that compiler/helper functions can be identified through a collection of static signatures that are created in the training phase (e.g. opcode frequencies). We analyze a number of programs with different functionalities, ranging from a simple "Hello World!" program to programs fulfilling complex tasks. Through the intersection of these functions combined with manual analysis, we collect about 240 functions as compiler/helper functions related to two GCC and VS compilers. The opcode frequencies are extracted from these functions, after which the mean and variance of each opcode are calculated.

In other words, each disassembled program P, after passing IDA Pro, consists of n functions $\{f_1, \cdots, f_n\}$. Each function f_k is represented as m pairs of opcodes o_i, where m is the number of distinct opcodes in function f_k. Each opcode $o_i \in O$ has a pair of values (μ_i, ν_i), which represents the mean and variance values of that specific opcode. Each opcode in the target function is measured against the same opcode of all compiler functions in the training set. If the measured distance $D_{i,j}$ (i.e., i represents the training function and j represents the target function) is less than a predefined threshold value $\alpha = 0.005$, the opcode is considered as a match. A function is labeled as *compiler-related* if the matched opcodes ratio is greater than a predefined threshold value learned from experiments to be $\gamma = 0.75$; otherwise, the target function is labeled as *user-related*. Dissimilarity measurements are performed based on distance calculations as per the following equation [39]:

$$D_{i,j} = \frac{(\mu_j - \overline{\mu_j})^2}{\left(\nu_i^2 + \overline{\nu_i^2}\right)}$$

where $(\overline{\mu}_j, \overline{\nu}_j)$ represents the opcode mean and variance of the target function, respectively. This dissimilarity metric detects functions, which are closer to each other in terms of types of opcodes. For instance, logical opcodes are not available in *compiler-related* functions. Finally, a score is given to every distance that is below a predefined threshold α.

2.2 Canonicalization

We use a strategy similar to that applied in the recent work by [21] when lifting the resulting user-related functions.

Lifting Binaries to Intermediate Representation (IR): We adopt the existing framework angr [36] for lifting function into LLVM-IR. We first convert the disassembled binary to the VEX-IR [29] using angr, and then implement a translator to convert the VEX-IR to LLVM-IR.

Optimizing Intermediate Representation to Optimized IR: To achieve this goal, we employ the extended version of Peggy tool [38] to optimize LLVM-IR. It performs the following tasks: dead code elimination, global value numbering, partial redundancy elimination, sparse conditional constant propagation,

loop-invariant code motion, loop deletion, loop unswitching, dead store elimination, constant propagation, and basic block placement. In this way, we prevent such changes from affecting our extracted choices. For more details, we refer the reader to [38].

Canonical Form: Canonicalization offers several benefits [21]. Lifting the instructions according to LLVM may impose changes such as redundant loads, but these changes will now be reverted. Moreover, in the case of writing dependencies, canonicalization of the expression makes it possible to perform the addition with the constant first, and the result is put in the register before the subtraction is performed. Furthermore, with canonicalization, the comparison becomes simple addition with a positive constant, instead of subtraction with a negative. Note that this last step serves to reoptimize code which might not have been previously optimized [21].

2.3 Choices Categorization

Determining a set of characteristics that remain constant for a significant portion of a program written by a particular author is analogous to discovering human characteristics that can later be used to identify an individual. Accordingly, our aim is to automate the identification of program characteristics, but with a reasonable computational cost. To capture coding habits at different levels of abstraction, we consider a spectrum of habits, assuming that an author's habits can be reflected in a preference for choosing certain keywords or compilers, a reliance on the main function, or the use of an object-oriented programming paradigm. The manner in which the code is organized may also reflect the author's habits. All possible choices are stored as a template in this step. We provide a detailed description of each category of author choices in the following subsections.

2.3.1 General Choices

General choices are designed to capture an author's general programming preferences, for example, preferences in organizing the code, terminating a function, the use of particular keywords, or the use of specific resources.

(1) **Code organization:** We capture the way code is organized by measuring the reliance on the `main` function using statistical features, since it is considered a starting part for managing user functions. We define a set of ratios, shown in Table 1, that measures the actions used in the `main` function. We thus capture the percentage usage of keywords, local variables, API calls, and calling user functions, as well as the ratio of the number of basic blocks in the `main` function to the number of basic blocks in other user functions. These percentages are computed relative to the length of the `main` function, where the length signifies the number of instructions in the function. The results are represented as a vector of ratios, which is used by the detection component.

Table 1. Features extracted from the `main` function

Ratio equation	Description
#`push`/l	Ratio of accessing the stack to length
#`push`/#`lea`	Ratio of accessing the stack to local variables
#`lea`/l	Ratio of local variables to length
#calls/l	Ratio of function calls to length
#callees/l	Ratio of the calls to `main` function to length
#indirect calls/l	Ratio of API calls to length
#`BBs`/total # all `BBs`	Ratio of the number of basic blocks of the `main` function to that of other user functions
#calls/#user functions	Ratio of function calls to the number of user functions

length(l) represents number of instructions in the `main` function

(2) **Function termination:** *BinAuthor* captures the way in which an author terminates a function. This could help identify an author since programmers may favor specific ways of terminating a function. *BinAuthor* considers not just the last statement of a function as the terminating instruction; rather, it identifies the last basic block of the function with its predecessor as the terminating part. This is a realistic approach since various actions may be required before a function terminates. With this in mind, *BinAuthor* not only considers the usual terminating instructions, such as `return` and `exit`, but also captures related actions that are taken prior to termination. For instance, a function may be terminated with a display of messages, a call to another function, the release of some resources, or communication over networks. Table 2 shows examples of what is captured in relation to the termination of a function. Such features could be captured by extracting the strings and opcodes. Each feature is set to *1* if it is used to terminate a function; otherwise, it is set to *0*. The output of this component is a binary vector that is used by the detection component.

Table 2. Examples of actions taken in terminating a function

Features	
Printing results to memory	Printing results to file
Using system ("pause")	User action such as `cin`
Calling user functions	Calling API functions
Closing files	Closing resources
Freeing memory	Flushing buffer
Using network communication	Printing clock time
Releasing semaphores or locks	Printing errors

(3) **Keyword and resource preferences:** *BinAuthor* captures an author's preferences in the use of keywords or resources. We consider only groups of

preferences with equivalent or similar functionality to avoid functionality-dependent features. These include keyword type preferences for inputs (e.g., using `cin`, `scanf`), preferences for particular resources or a specific compiler (we identify the compiler by using PEiD[2]), operation system (e.g., Linux), CPU architecture (e.g., ARM), and the manner in which certain keywords are used, which can serve as further indications of an author's habits. Some of these features are identified through binary string matching, which tracks the strings annotated to `call` and `mov` instructions. For instance, excessive use of `fflush` will cause the string ``fflush'' to appear frequently in the resulting binary.

2.3.2 Quality-Related Choices

We investigate code quality in terms of compliance with C/C++ coding standards and security concerns. The literature has established that code quality can be measured using different indicators, such as testability, flexibility, and adaptability [31]. *BinAuthor* defines rules for capturing code that exhibits either relatively low or high quality. For any code that cannot be matched using such rules, the code is labelled as having regular quality, which indicates that the code quality feature is not applicable. Such rules are extracted by defining a set of signatures (sequence of instructions) for each choice. An example is introduced in Appendix A.

Examples of low-quality coding styles are reopening already opened files, leaving files open when they are no longer in use, attempting to modify constants through pointers, using float variables as loop counters, and declaring variables inside a switch statement. Such declarations, which can be captured through the structure matching of code, could be considered a structural choice, possibly resulting in unexpected/undefined behavior due to jumped-over instructions. It is for this reason that we put them in the low-quality category. Examples of high-quality coding styles are handling errors generated by library calls (i.e., examining the value returned by `fclose()`); avoiding reliance on side effects (e.g., the ++ operator) within calls such as `sizeof` or `_Alignof`; avoiding particular calls to some environments or using them with protective measures (since invoking the `system()` in Linux may lead to shell command injection or privilege escalation, using `execve()` instead is indicative of high-quality coding); and the implementation of locks and semaphores around critical sections.

2.3.3 Embedded Choices

We define embedded choices as actions that are related to coding habits present in the source code, which are not easily captured at the binary level by traditional features such as strings or graphs. Examples are initializing member variables in constructors and dynamically deleting allocated resources in destructors. Since it is not feasible to list all possible features, *BinAuthor* relies on the fact that opcodes reveal actions, expertise, habits, knowledge, and other author's

[2] https://www.aldeid.com/wiki/PEiD.

characteristics, and then analyzes the distribution of opcode frequencies. Our experiments showed that this distribution can effectively capture the manner in which an author manages code. Since every action in source code can affect the frequency of opcodes, *BinAuthor* targets embedded choices by capturing the distribution of opcode frequencies.

2.3.4 Structural Choices

Programmers usually develop their own structural design habits. They may prefer to use a fully object-oriented design, or they may be more accustomed to procedural programming. Structural choices can serve as features for author identification. To avoid functionality, we consider the common subgraphs for each user function and then intersect them among different user functions to identify those subgraphs that are unique and those that are common. These types of subgraphs are defined as k-graphs, where k is the number of nodes. The common k-graphs form author's signatures since they always appear, regardless of the program functionality. In addition, we consider the longest path in each user function because it reflects the way in which an author tends to use deep or nested loops. An author may organize classes either ad hoc or hierarchically by designing a driver class to contain several manager classes, where each manager is responsible for different processes (collections of threads running in parallel). Both ad hoc and hierarchical systems of organization can create a common structure in an author's programs.

2.4 Feature Vectors

General Choice Computation: To consider the reliance on the `main` function, a vector v_{g1}, representing related features, is constructed according to the equations shown in Table 1. These equations indicate the author's reliance on the `main` function as well as the actions performed by the author. *Function termination* is represented as a binary vector, (v_{g2}), which is determined by the absence or existence of a set of features for function termination. *Keyword and resource preferences* are identified through binary string matching. We extract a collection of strings from all user functions of a particular author, then intersect these strings in order to derive a persistent vector (v_{g3}) for that author. Consequently, for each author, a set of vectors representing the author's signature is stored in our repository. Given a target binary, *BinAuthor* constructs the vectors from the target and measures the distance/similarity between these vectors and those in our repository. The v_{g1} vector is compared using Euclidean distance, whereas v_{g2} vector is compared using the Jaccard similarity. For v_{g3}, the similarity is computed through string matching. Finally, the three derived similarity values are averaged in order to obtain λ_g, which is later used in Sect. 4.6 for author classification.

Quality-Related Choice Computation: We build a set of idiom templates to describe high or low quality habits. Idioms are sequences of instructions with wild-card possibility [24]. We employ the idioms templates in [24] according to

our qualitative-related choice. In addition, such templates carry a meaningful connection to the quality-related choices. Our experiments demonstrate that such idiom templates may effectively capture quality-related habits. *BinAuthor* uses the Levenshtein distance [40] for this computation due to it's efficiency. The similarity is represented by λ_q as follow:

$$\lambda_q = 1 - \frac{L(C_i, C_j)}{max(|C_i|, |C_j|)}$$

where $L(C_i, C_j)$ is the Levenshtein distance between the qualitative-related choices C_i (sequence of instructions) and C_j, $max(|C_i|, |C_j|)$ returns the maximum length between two choices C_i and C_j in terms of characters.

Embedded Choice Computation: The Mahalanobis distance [26] is used to measure the dissimilarity of opcode distributions among different user functions, which is represented by λ_e. The Mahalanobis distance is chosen because it can capture the correlation between opcode frequency distributions.

Structural Choice Computation: *BinAuthor* uses subgraphs of size k in order to capture structural choices ($k = 4$, 5, and 6 through our experiments). Given a k-graph, the graph is transformed into strings using Bliss open-source toolkit [23]. Then, a similarity measurement is performed over these strings using the normalized compression distance (NCD) [20]. The reason of our choice for NCD is threefold: (i) it enhances the search performance; (ii) it allows to concatenate all the common subgraphs that appear in author's programs; and (iii) it allows to perform inexact matching between the target subgraphs and the training subgraphs. *BinAuthor* forms a signature based on these strings. The similarity obtained from this choice is represented by λ_s.

2.5 Classification

As previously described, *BinAuthor* extracts different types of choices to characterize different aspects of author coding habits. Such choices do not equally contribute to the attribution process, since the significance of these indicators are not identical. Consequently, a weight is assigned to each choice by applying logistic regression to them in order to predict class probabilities (e.g., the probability of identifying an author). For this purpose, we use the introduced dataset in Sect. 3.2; to prevent the overfitting, we test each dataset separately and then compute the average of weights. The weights are calculated as follows:

$$w_i = rnd\Big((p_i/p_s)/\sum_{i=1}^{4}(p_i/p_s)\Big)$$

where p_s is the smallest probability value (e.g. 0.39 in Table 3), p_i is the probability outcome from logistic regression of each choice, and the rnd function rounds the final value. The probability outcomes of logistic regression prediction is illustrated in Table 3.

Table 3. Logistic regression weights for choices

Choice	Probability (P_i)	$P_i/(P_s = 0.39)$	Weight $w_i = rnd\Big((p_i/p_s)/\sum_{i=1}^{4}(p_i/p_s)\Big)$
General	0.83	2.128205	0.35
Qualitative	0.63	1.615385	0.27
Structural	0.52	1.333333	0.22
Embedded	0.39	1	0.16
		$\sum_{i=1}^{4}(p_i/p_s) = 6.076923$	

After extracting features, we define a probability value P based on obtained weights. The attribution probability is defined as follows:

$$P(A) = \sum_{i=1}^{4} w_i * \lambda_i$$

where w_i represents the weight assigned to each choice, as shown in Table 3, and λ_i is the distance metric value obtained from each choice ($\lambda_g, \lambda_q, \lambda_e$, and λ_s) as described in Sect. 2.4. We normalize the probabilities of all authors, and if $P \geq \zeta$, where ζ represents predefined threshold values, then the author is labeled as a matched author. Through our experiments, we find that the best value of ζ is 0.87. If more than one author has probability larger than the threshold value, then *BinAuthor* returns the set of those authors.

3 Evaluation

3.1 Implementation Setup

The described stylistic choices are implemented using separate Python scripts for modularity purposes, which altogether form our analytical system. A subset of the python scripts in the *BinAuthor* system is used in tandem with IDA Pro disassembler. The final set of the framework scripts perform the bulk of the choice analysis functions that compute and display critical information about an author's coding style. With the analysis framework completed, a graph database is utilized to perform complex graph operations such as k-graph extraction. The graph database chosen for this task is Neo4j. Gephi [8] is employed for all graph analysis functions, which are not provided by Neo4j. MongoDB database is used to store our features for efficiency and scalability purposes.

3.2 Dataset

Our dataset is consisted of several C/C++ applications from different sources, as described below: (i) GitHub [2]; (ii) Google Code Jam [1], an international

programming competition; (iii) Planet Source Code [9]; (iv) Graduate Student Projects at our institution. Statistics about the dataset are provided in Table 4. In total, we test 800 authors from different sets in which each author has two to five software applications, resulting in a total of 3150 programs. To compile these datasets, we use GNU Compiler Collection (version 4.8.5) with different optimization levels, as well as Microsoft Visual Studio (VS) 2010.

3.3 Experimental Setup

In our experimental setup, we split the collected program binaries into ten sets, reserving one as a testing set and using the remaining nine sets as the training set. We repeat this process 100 times. In order to evaluate *BinAuthor* and to compare it with existing methods, the precision (P) and recall (R) metrics are applied as $Precision = \frac{TP}{TP+FP}$, $Recall = \frac{TP}{TP+FN}$, where the true positive (TP) indicates number of relevant authors that are correctly retrieved; true negative (TN) returns the number of irrelevant authors that are not detected; false positive (FP) indicates the number of irrelevant authors that are incorrectly detected; and false negative (FN) presents the number of relevant authors that are not detected.

3.4 Accuracy

The main purpose of this experiment is to evaluate the accuracy of author identification in binaries. The evaluation of *BinAuthor* is conducted using the datasets described in Sect. 3.2.

Results Comparison. We compare *BinAuthor* with the existing authorship attribution methods [15,19,32]. The source code and dataset of our previous work, OBA2 [15], is available which performs authorship attribution on a small scale of 5 authors with 10 programs for each. The source code of the two other approaches presented by Caliskan-Islam et al. [19] and Rosenblum et al. [32] are available at [7] and [4], respectively. Both Caliskan-Islam et al. and Rosenblum et al. present a largest-scale evaluation of binary authorship attribution, which contains 600 authors with 8 training programs per author, and 190 authors with at least 8 training programs, respectively. However, since the corresponding

Table 4. Statistics about the dataset used in the evaluation of *BinAuthor*

Source	# of authors	# of programs	# of functions
GitHub	150	600	110000
Google Code Jam	500	2000	23650
Planet Source Code	100	300	12080
Graduate Student Projects	50	250	9823

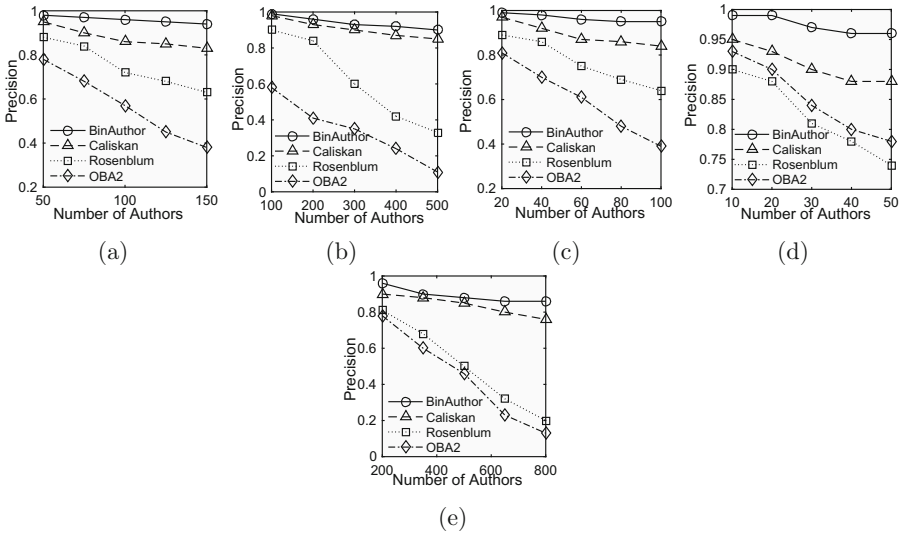

Fig. 2. Precision results of authorship attribution obtained by *BinAuthor*, Caliskan-Islam et al., Rosenblum et al., and OBA2, on (a) Github, (b) Google Code Jam, (c) Planet Source Code, (d) Graduate Student Projects, and (e) All datasets

datasets are not available, we compare *BinAuthor* with these methods by using the datasets mentioned in Table 4.

Figure 2 details the results of comparing the precision between *BinAuthor* and the aforementioned methods. It shows the relationship between the precision and the number of authors present in all datasets, where the precision decreases as the size of author population increases. The results show that *BinAuthor* achieves better precision in determining the author of binaries. Taking all four approaches into consideration, the highest precision of authorship attribution is close to 99% on the Google Code Jam with less than 150 authors, while the lowest precision is 17% when 800 authors are involved on all dataset together. We believe the reason behind Caliskan-Islam et al. approach that achieves high precision on Google Jam Code is that this dataset is simple and can be easily decompiled to source code. *BinAuthor* also identifies the authors of Github dataset with an average precision of 92%. The main reason for this is due to the fact that the authors of projects in Github have no restrictions when developing projects. In addition, the advanced programmers of such projects usually design their own class or template to be used in the projects. The lowest precision obtained by *BinAuthor* is approximately 86% on all datasets together. We have observed that *BinAuthor* achieves lower precision when it is applied on Graduate student projects. When the number of authors is 400 on the mixed dataset, the precision of Rosenblum et al. and OBA2 approaches drop rapidly to 40% on all datasets, whereas our system's precision remains greater than 86% while Caliskan-Islam et al. approach remains greater than 73%. This provides evidence for the stability of using coding habits in identifying authors. In total,

the different categories of choices achieve an average precision of 98% for ten distinct authors and 86% when discriminating among 800 authors. These results show that author habits may survive the compilation process.

Observations. Through our experiments, we have noticed the following observations:

(1) *Feature Pre-processing.* We have encountered that in the existing methods, the top-ranked features are related to the compiler (e.g., stack frame setup operation). It is thus necessary to filter irrelevant functions (e.g., compiler functions) in order to better identify author-related portions of code. To this end, we utilize a more elaborate method for filtration to eliminate the compiler effects and to label library, compiler, and open-source software related functions. Successful distinction between these functions leads to considerable time savings and helps shift the focus of analysis to more relevant functions.
(2) *Source of Features.* Existing methods use disassembler and decompilers to extract features from binaries. Caliskan-Islam et al. use a decompiler to translate the program into C-like pseudo code via Hex-Ray [6]. They pass the code to a fuzzy parser for C, thus obtain an abstract syntax tree from which features can be extracted. In addition to Hex-Ray limitations [6], the C-like pseudo code is 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 decompiling its binary, 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, thus increasing the rate of false positives.

3.5 Scalability

Security analysts or reverse engineers may be interested in performing large-scale author identification, and in the case of malware, an analyst may deal with an extremely large number of new samples on a daily basis. With this in mind, we evaluate how well *BinAuthor* scales. To prepare the large dataset required for large-scale authorship attribution, we obtain programs from three sources: Google

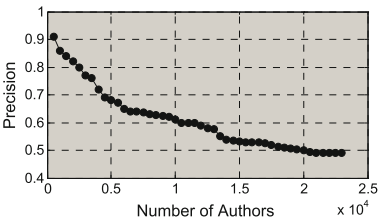

Fig. 3. Large-scale author attribution precision

Code Jam, GitHub, and Planet Source Code. We eliminate from the experiment programs that could not be compiled because they contain bugs and those written by authors who contributed only one or two programs. The resulting dataset comprised 103,800 programs by 23,000 authors: 60% from Google Code Jam, 25% from Planet source code, and 15% from GitHub. We modified the script[3] used in [19] to download all the code submitted to the Google Code Jam competition. The programs from the other two sources were downloaded manually. All the programs were compiled with the Visual Studio and GCC compilers, using the same settings as those in our previous investigations (Sect. 3). The experiment evaluate how well the top-weighted choices represent author habits.

The large-scale author identification results are shown in Fig. 3. Figure 3 shows the precision with which *BinAuthor* identifies the author, and its scaling behavior as the number of authors increases is satisfactory. Among almost 4000 authors, an author is identified with 72% precision. When the number of authors is doubled to 8000, the precision is close to 65%, and it remains nearly constant (49%) after the number of authors reaches 19,000. Additionally, we test *BinAuthor* on the programs obtained from each of the sources. The precision was high for samples from the GitHub dataset (88%) and also for samples from the Planet dataset (82%), however it was low for samples from Google Code Jam (51%). The results suggest that it is easier to perform attribution for authors who wrote code for difficult tasks than for those addressing easier tasks.

We have also investigated the impact of false positives (Appendix B), and impact of code transformation techniques (Appendix C).

3.6 Applying *BinAuthor* to Real Malware Binaries

The malware binary authorship attribution is very challenging due to the following main reason: the lack of ground truth concerning the attribution of authorship due to the nature of malware. Such limitation explains the fact that few research efforts have been seen on manual malware authorship attribution. In fact, to the best of our knowledge, *BinAuthor* is the first attempt to apply automated authorship attribution to real malware. We describe the application of *BinAuthor* to some well-known malware binaries. Details of malware dataset are shown in Table 5. Given a set of functions, *BinAuthor* clusters them based on the number of common choices.

A. Applying *BinAuthor* to `Bunny` and `Babar`: We apply *BinAuthor* to `Bunny` and `Babar` malware samples and cluster the functions based on the choices. *BinAuthor* is able to find the following coding habits automatically: the preference for using Visual Studio 2008 and the use of a common approach to managing functions (general choices); the use of one variable over a long chain (structural choice); the choice of methods for accessing freed memory, dynamically deallocating allocated resources, and reopening resources more than once in the same function (quality choices). As shown in Table 6, *BinAuthor* found functions common to `Bunny` and `Babar` that share the aforementioned coding habits:

[3] https://github.com/calaylin/CodeStylometry/tree/master.

Table 5. Characteristics of malware dataset

Malware	Packed	Obfuscated	Source code	Binary code	Type	# binary function	Source of sample
`Zeus`	✗	✗	✓	✓	PE	557	Our security lab
`Citadel`	✗	✗	✓	✓	PE	794	Our security lab
`Flame`	✗	✓	✗	✓	ELF	1434	Contagio [13]
`Stuxnet`	✗	✓	✗	✓	ELF	2154	Contagio [13]
`Bunny`	✓	✗	✗	✓	PE	854	VirusSign [14]
`Babar`	✓	✗	✗	✓	PE	1025	VirusSign [14]

494 functions share qualitative choices; 450 functions share embedded choices; 372 functions share general choices; and 127 functions share structural choices. Among these, *BinAuthor* found 340 functions that share 4 choices, 478 functions that share 3 choices, 150 functions that share 2 choices, and 290 functions that share 1 choice. Considering the 854 and 1025 functions in `Bunny` and `Babar`, respectively, *BinAuthor* found that 44% $((340+478)/(854+1025))$ are likely to have been written by a single author (same common choices), and 23% are likely to have been written by multiple authors (contradictive different choices inside the same function). No common choices were identified in the remaining 33%, likely because different segments or code lines within the same function were written by different authors, a common practice in writing complex software.

Table 6. Statistics of applying *BinAuthor* to malware binaries

Malware	Number of functions with common choices				Number of common functions with			
	General	Qualitative	Structural	Embedded	1 choice	2 choices	3 choices	4 choices
`Bunny` and `Babar`	372	494	127	450	290	150	478	340
`Stuxnet` and `Flame`	725	528	189	300	689	515	294	180
`Zeus` and `Citadel`	655	452	289	370	600	588	194	258

B. Applying *BinAuthor* to `Stuxnet` and `Flame`: *BinAuthor* found the following coding habits automatically: the use of global variables, `Lua` scripting language, a specific open-source package `SQLite`, and heap sort rather than other sorting methods (general choices); the choice of opening and terminating processes (qualitative choices); the presence of recursion patterns and the use of POSIX socket API rather than BSD socket API (structural choices); and the use of functions that are close in terms of the Mahalanobis distance, with distance close to 0.1. As shown in Table 6, *BinAuthor* identified functions common to `Stuxnet` and `Flame` that share the aforementioned coding habits. *BinAuthor* clustered the functions and found that 13% $((180+294)/(1434+2154))$ were

written by one author, while 34% ((515 + 689)/(1434 + 2154)) were written by multiple authors. No common choices were found in the remaining 53% of the functions. The fact that these malware packages follow the same rules and set the same targets suggests that `Stuxnet` and `Flame` are written by an organization.

C. Applying *BinAuthor* to `Zeus` and `Citadel`: *BinAuthor* identified the following coding habits: the use of network resources rather than file resources, creating configurations using mostly config files, the use of specific packages such as `webph` and `ultraVNC` (general choices); the use of `switch` statements rather than `if` statements (structural choices); the use of semaphores and locks (qualitative choices); and the presence of functions that are close in terms of the Mahalanobis distance, with distance = 0.0004 (embedded choices). As listed in Table 6, *BinAuthor* found functions common to `Zeus` and `Citadel` that share the aforementioned coding habits. After *BinAuthor* clustered the functions, it appears that 33% were written by a single author, while 53% were written by the same team of multiple authors. No common choices were found for the remaining 14% of the functions. Our findings clearly support the common belief that `Zeus` and `Citadel` were written by the same team of authors.

D. Comparison with Technical Reports: We compare *BinAuthor*'s findings with those made by human experts in technical reports.

- For `Bunny` and `Babar`, our results match the technical report published by the Citizen Lab [5], which demonstrates that both malware packages were written by a set of authors according to common implementation traits (general and qualitative choices) and infrastructure usage (general choices). The correspondence between the *BinAuthor* findings and those in the technical report is the following: 60% of the choices matched those mentioned in the report, and 40% did not; 10% of the choices found in the technical report were not flagged by *BinAuthor* as they require dynamic extraction of features, while *BinAuthor* uses a static process.
- For `Stuxnet` and `Flame`, our results corroborate the technical report published by Kaspersky [12], which shows that both malware packages use similar infrastructure (e.g., Lua) and are associated with an organization. In addition, *BinAuthor*'s findings suggest that both malware packages originated from the same organization. The frequent use of particular qualitative choices, such as the way the code is secured, indicates the use of certain programming standards and strict adherence to the same rules. Moreover, *BinAuthor*'s findings provide much more information concerning the authorship of these malware packages. The correspondence between *BinAuthor*'s findings and those in the technical report is as follows: all the choices found in the report [12] were found by *BinAuthor*, but they represent only 10% of our findings. The remaining 90% of *BinAuthor*'s findings were not flagged by the report.
- For `Zeus` and `Citadel`, our results match the findings of the technical report published by McAfee [3], indicating that `Zeus` and `Citadel` were written by the same team of authors. The correspondence between the findings of *BinAuthor* and those of McAfee are as follows: 45% of the choices matched

those in the report, while 55% did not, and 8% of the technical report findings were not flagged by *BinAuthor*.

4 Related Work

Binary Authorship Attribution: 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 [15,19,32], the authors show that certain stylistic features can indeed survive the compilation process and remain intact in binary code, thus showing that authorship attribution for binary code should be feasible. The methodology developed by Rosenblum et al. [32] is the first attempt to automatically identify authors of software binaries. The main concept employed by this method is to extract syntax-based features using predefined templates such as idioms, n-grams, and graphlets. A subsequent approach (OBA2) to automatically identify the authorship of software binaries is proposed by Alrabaee et al. [15]. 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. A more recent approach to automatically identify the authorship of software binaries is proposed by Caliskan-Islam et al. [19]. The authors extract syntactical features present in source code from decompiled executable binaries. Most recently, Meng et al. [27] introduce new fine-grained techniques to address the problem of identifying the multiple authors of binary code by determining the author of each basic block. The authors extract syntactic and semantic features at a basic level, such as constant values in instructions, backward slices of variables, and width and depth of a function control flow graph (CFG). Table 7 compares our approach with the aforesaid approaches. Please note that the results of code transformation (CT) section are based on conducted experiment. When we found the accuracy is dropped by 1–3%, we considered as "Not affected", while 4–14% gives "Partially affected", and finally if it was above 15%, we considered as "Affected".

Malware Authorship Attribution: Most existing work on malware authorship attribution relies on manual analysis. In 2013, a technical report published by FireEye [28] discovered that malware binaries share the same digital infrastructure and code, such as the use of certificates, executable resources, and development tools. More recently, the team at Citizen Lab attributed malware authors according to the manual analysis exploit type found in binaries and the manner by which actions are performed, such as connecting to a command and control server. The authors in [5] presented a novel approach to creating credible links between binaries originating from the same group of authors. Their goal aimed to add transparency in attribution and to supply analysts with a tool that emphasizes or denies vendor statements. The technique is based on features derived from different domains, such as implementation details, applied evasion techniques, classical malware traits, or infrastructure attributes, which are leveraged to compare the handwriting among binaries.

Table 7. Comparing different existing solutions with *BinAuthor*.

Effort	Features				Compiler				CT				Binaries	
	Syntax	Semantic	Structural	Statistical	VS	GCC	Clang	ICC	DCI	IR	IRO	RT	ELF	PE
OBA2	✗	✓	✗	✗	✓	✗	✗	✗	◐	◐	○	●	✗	✓
Caliskan	✗	✓	✓	✓	✗	✓	✗	✗	○	○	○	●	✓	✗
Rosenblum	✓	✓	✓	✗	✗	✓	✗	✗	●	●	●	○	✓	✗
Meng	✗	✓	✓	✓	✗	✓	✗	✗	◐	◐	◐	○	✓	✓
BinAuthor	✓	✓	✓	✓	✓	✓	✓	✓	○	○	○	○	✓	✓

Note: The (✓) symbol indicates that the proposal solution offers the corresponding feature. (CT) stands for code transformation. (DCI) stands for dead code insertion. (IR) stands for instruction replacement. (IRO) stands for instruction reordering. (RT) stands for refactoring techniques. (○): Not affected by the code transformation method. (●): Affected by the code transformation method. (◐): Partially affected by the code transformation method.

5 Limitations

Our work has a few important limitations.

Advanced Obfuscation: Our tool fails to handle most of the advanced obfuscation techniques, such as virtualization and jitting, since our system does not deal with bytecode.

IR: Through our experiments, we notice that optimizing IR would remove some author styles, e.g., loop deletion. We left this issue for future work by leveraging some existing work [34].

Functionality: There are some choices appear when an author implements a specific functionality. For instance, if the functionality does not have a multiple-branch logic, there is no choice between `if` and `switch`.

6 Conclusion

To conclude, we have presented the first known effort on decoupling coding habits from functionality. Previous research has applied machine learning techniques to extract stylometry styles and can distinguish between 5–50 authors, whereas we can handle up to 150 authors. In addition, existing works have only employed artificial datasets, whereas we included more realistic datasets. Our findings indicated that the precision of these techniques drops dramatically to approximately 45% at a scale of more than 50 authors. We also applied our system to known malware samples (e.g., `Zeus and Citadel`) as a case study. We realized that authors with advanced expertise are easier to attribute than authors who have less expertise. Authors of realistic datasets are easier to attribute than authors of artificial datasets. Specifically, in the GitHub dataset, the authors of a sample can be identified with greater than 90% precision. In summary, our system demonstrates superior results on more realistic datasets.

Acknowledgements. The authors thank the anonymous reviewers for their valuable comments. We also appreciate the help we received from Perry Jones in implementing *BinAuthor*. This research is the result of a fruitful collaboration between the Security Research Center (SRC) of Concordia University, Defence Research and Development Canada (DRDC) and Google under a National Defence/NSERC Research Program.

Appendix

A Example of Qualitative Choices

Consider a template of dynamic memory allocation presented in Listing 1.1. As shown in, we have a call to `malloc`, followed by checking whether or not it is Null.

Listing 1.1. A fragment of assembly instruction that captures a bad habit of dynamic memory allocation

```
...
call ds:malloc
...
or eax, 0FFFFFFFF  // -1 if text_buffer is Null
...
xor eax, eax // 0 if text_buffer is not Null
```

The Listing 1.2 shows how the bad habit in Listing 1.1 could be considered as a good habit at the assembly level.

Listing 1.2. A fragment of assembly instruction that captures a good habit of dynamic memory allocation

```
...
call ds:malloc
...
or eax, 0FFFFFFFF  // -1 if text_buffer is Null
...
push eax      // memory address of text_buffer
call ds:free
...
xor eax, eax // 0 if text_buffer is not Null
```

B False Positives

We investigate the false positives in order to understand the situations where *BinAuthor* is likely to make incorrect attribution decisions. For this experiment, we consider 5 programs for each author. For instance, when we have 500 authors ($5 * 500 = 2500$ programs), *BinAuthor* misclassifies 49 programs. Also, when the number of authors is 2000 ($2000 * 4 = 8000$ programs), the number of false positives is 336. We have 2000 authors from dataset used in Sect. 3.2. After

investigation, we have found that the false positives rate for student dataset is the highest rate and we believe the reason behind this is that the students should follow the standard coding instructions which restrict them to have their own habits.

C Impact of Code Transformation Techniques

Refactoring Techniques. We consider a random set of 50 files from our dataset which we use for the C++ refactoring process [10,11]. We ignore the variable renaming since it will have no effect in binary code, we consider the following techniques of, (i) moving a method from a superclass to its subclasses, and (ii) extracting a few statements and placing them into a new method. We obtain a Precision of 91.5% in correctly classifying authors, which is only a mild drop in comparison to the 95% precision observed without applying refactoring techniques.

Impact of Obfuscation. We are interested in determining how *BinAuthor* handles simple binary obfuscation techniques intended for evading detection, as implemented by tools such as Obfuscator-LLVM [22]. These obfuscators replace instructions by other semantically equivalent instructions, introduce spurious control flow, and can even completely flatten control flow graphs. Obfuscation techniques implemented by Obfuscator-LLVM are applied to the samples prior to classifying the authors. We proceed to extract features from obfuscated samples. We obtain a precision of 92.9% in correctly classifying authors, which is only a slight drop in comparison to the 95% precision observed without obfuscation.

Impact of Compilers and Compilation Settings. We are further interested to study the impact of different compilers and compilation settings on the precision of our proposed system. We perform the following tasks: (i) testing the ability of *BinAuthor* when identifying the author from binaries compiled with the same compiler, but different compiler optimization levels. Specifically, we use binaries that were compiled with GCC/VS on x86 architecture using optimization levels O2 and O3. In this test, the precision remains same (95%). (ii) We use a different configuration to identify the author of program compiled with both a different compiler and different compiler optimization levels. Specifically, we use programs compiled for x86 with VS -O2 and GCC -O3. In this test, the precision slightly drops to 93.9%. We also redo the test for the same binaries compiled with ICC and Clang compilers. The precision remains almost the same 93.8%. This stability in the accuracy is due to the canonicalization process.

References

1. The Google Code Jam (2008–2015). http://code.google.com/codejam/
2. GitHub-Build software better (2011). https://github.com/trending?l=cpp
3. Technical report: McAfee (2011). www.mcafee.com/ca/resources/wp-citadel-trojan-summary.pdf

4. 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/
5. Big Game Hunting: Nation-state malware research, BlackHat (2015). https://www.blackhat.com/docs/us-15/materials/us-15-MarquisBoire-Big-Game-Hunting-The-Peculiarities-Of-Nation-State-Malware-Research.pdf
6. Hex-Ray decompiler (2015). https://www.hex-rays.com/products/decompiler/
7. Programmer De-anonymization from Binary Executables (2015). https://github.com/calaylin/bda
8. The Gephi plugin for neo4j (2015). https://marketplace.gephi.org/plugin/neo4j-graph-database-support/
9. The planet source code (2015). http://www.planet-source-code.com/vb/default.asp?lngWId=3#ContentWinners
10. C++ refactoring tools for visual studio (2016). http://www.wholetomato.com/
11. Refactoring tool (2016). https://www.devexpress.com/Products/CodeRush/
12. Technical report, Resource 207: Kaspersky Lab Research proves that Stuxnet and Flame developers are connected, May 2012. http://www.kaspersky.com/about/news/virus/2012/
13. Contagio: malware dump, May 2016. http://contagiodump.blogspot.ca
14. VirusSign: Malware Research & Data Center, Virus Free, May 2016. http://www.virussign.com/
15. Alrabaee, S., Saleem, N., Preda, S., Wang, L., Debbabi, M.: OBA2: an onion approach to binary code authorship attribution. Digit. Investig. **11**, S94–S103 (2014)
16. 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
17. Alrabaee, S., Shirani, P., Wang, L., Debbabi, M.: SIGMA: a semantic integrated graph matching approach for identifying reused functions in binary code. Digit. Investig. **12**, S61–S71 (2015)
18. Alrabaee, S., Shirani, P., Wang, L., Debbabi, M.: FOSSIL: a resilient and efficient system for identifying FOSS functions in malware binaries. ACM Trans. Priv. Secur. (TOPS) **21**(2), 8 (2018)
19. Caliskan-Islam, A., et al.: When coding style survives compilation: de-anonymizing programmers from executable binaries. Netw. Distrib. Syst. Secur. Symp. (NDSS) (2018)
20. Cilibrasi, R., Vitanyi, P.: Clustering by compression. IEEE Trans. Inf. Theory **51**(4), 1523–1545 (2005)
21. David, Y., Partush, N., Yahav, E.: Similarity of binaries through re-optimization. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 79–94. ACM (2017)
22. 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)
23. Junttila, T.A., Kaski, P.: Engineering an efficient canonical labeling tool for large and sparse graphs. In: ALENEX, vol. 7, pp. 135–149. SIAM (2007)
24. Knuth, D.E.: Backus normal form vs. Backus Naur form. Commun. ACM **7**(12), 735–736 (1964)
25. Krsul, I., Spafford, E.H.: Authorship analysis: identifying the author of a program. Comput. Secur. **16**(3), 233–257 (1997)

26. Mahalanobis, P.C.: On the generalized distance in statistics. Proc. Natl. Inst. Sci. (Calcutta) **2**, 49–55 (1936)
27. Meng, X., Miller, B.P., Jun, K.-S.: Identifying multiple authors in a binary program. In: Foley, S.N., Gollmann, D., Snekkenes, E. (eds.) ESORICS 2017. LNCS, vol. 10493, pp. 286–304. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66399-9_16
28. Moran, N., Bennett, J.: Supply Chain Analysis: From Quartermaster to Sunshop, vol. 11. FireEye Labs, Milpitas (2013)
29. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: ACM SIGPLAN Notices, vol. 42, pp. 89–100. ACM (2007)
30. Palmer, G., et al.: A road map for digital forensic research. In: First Digital Forensic Research Workshop, Utica, New York, pp. 27–30 (2001)
31. Rajlich, V.: Software evolution and maintenance. In: Proceedings of the Future of Software Engineering, pp. 133–144. ACM (2014)
32. 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
33. Schleimer, S., Wilkerson, D.S., Aiken, A.: Winnowing: local algorithms for document fingerprinting. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, pp. 76–85. ACM (2003)
34. Shirani, P., et al.: B**INARM**: scalable and efficient detection of vulnerabilities in firmware images of intelligent electronic devices. In: Giuffrida, C., Bardin, S., Blanc, G. (eds.) DIMVA 2018. LNCS, vol. 10885, pp. 114–138. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-93411-2_6
35. Shirani, P., Wang, L., Debbabi, M.: BinShape: scalable and robust binary library function identification using function shape. In: Polychronakis, M., Meier, M. (eds.) DIMVA 2017. LNCS, vol. 10327, pp. 301–324. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-60876-1_14
36. Shoshitaishvili, Y., et al.: SOK: (state of) the art of war: offensive techniques in binary analysis. In: 2016 IEEE Symposium on Security and Privacy, SP, pp. 138–157. IEEE (2016)
37. Spafford, E.H., Weeber, S.A.: Software forensics: can we track code to its authors? Comput. Secur. **12**(6), 585–595 (1993)
38. Tristan, J.-B., Govereau, P., Morrisett, G.: Evaluating value-graph translation validation for LLVM. ACM SIGPLAN Not. **46**(6), 295–305 (2011)
39. Wang, J.T.-L., Ma, Q., Shasha, D., Wu, C.H.: New techniques for extracting features from protein sequences. IBM Syst. J. **40**(2), 426–441 (2001)
40. Yujian, L., Bo, L.: A normalized Levenshtein distance metric. IEEE Trans. Pattern Anal. Mach. Intell. **29**(6), 1091–1095 (2007)