Decoupling Coding Habits from Functionality for Effective Binary Authorship Attribution

Saed Alrabaee[a,*], Paria Shirani[b], Lingyu Wang[b], Mourad Debbabi[b] and Aiman Hanna[c]
[a] *Information Systems and Security, United Arab Emirates University, UAE*
E-mail: salrabaee@uaeu.ac.ae
[b] *CIISE, Concordia University, Montreal, QC, Canada*
E-mails: p_shira@encs.concordia.ca, wang@ciise.concordia.ca, debbabi@ciise.concordia.ca
[c] *Computer Science, Concordia University, Montreal, QC, Canada*
E-mail: contact@aimanhanna.com

Abstract. Binary authorship attribution refers to the process of identifying the author of a given anonymous binary file based on stylistic characteristics. It aims to automate the laborious and error-prone reverse engineering task of discovering information related to the author(s) of a binary code. Existing works typically employ machine learning methods to extract features that are unique for each author and subsequently match them against a given binary to identify the author. However, most existing works share a common critical limitation, i.e., they cannot distinguish between features representing program functionality and those representing authorship (e.g., authors' coding habits). Such distinction is crucial for effective authorship attribution because what is unique in a particular binary may be attributed to either author, compiler, or function. In this study, we present BINAUTHOR a system capable of decoupling program functionality from authors' coding habits in binary code. To capture coding habits, BINAUTHOR leverages a set of features that are based on collections of functionality-independent choices made by authors during coding. Our evaluation demonstrates that BINAUTHOR outperforms existing methods in several aspects. First, it successfully attributes a larger number of authors with a significantly higher accuracy (around 90%) based on the large datasets extracted from selected open-source C++ projects in GitHub, Google Code Jam events, Planet Source Code contests, and several programming projects. Second, BINAUTHOR is more robust than previous methods; there is no significant drop in accuracy when the code is subjected to refactoring techniques, simple obfuscation, and processed with different compilers. Finally, decoupling authorship from functionality allows us to apply BINAUTHOR to real malware binaries (`Citadel`, `Zeus`, `Stuxnet`, `Flame`, `Bunny`, and `Babar`) to automatically generate evidence on similar coding habits.

Keywords: Binary Authorship Attribution, Malware Analysis, Coding Habits, Assembly Instructions

1. Introduction

Binary authorship attribution aims to automate the arduous and error-prone reverse engineering task of extracting information related to the author(s) of a program binary and further attribute the author(s). The ability to conduct such analyses at the binary level is especially important for security applications because the source code for malware is not always available. In automating binary authorship attribution compared to source code analysis, two main challenges are typically encountered: the binary code lacks

*Corresponding author. E-mail: salrabaee@uaeu.ac.ae.

many abstractions (i.e., variable names) that are present in the source code, and the time and space complexities of analyzing binary code are greater than those of the corresponding source code. Although significant efforts have been made to develop automated approaches for source code authorship attribution [1], these often rely on features that will likely be lost in the strings of bytes representing binary code after the compilation process, such as variable and function naming, original control and data flow structures, comments, and space layout. To this end, a few approaches to binary authorship attribution have been proposed; typically, they employ machine learning methods to extract unique features for each author and then match a given binary against such features to identify the authors [2–4]. These approaches share the following limitations: (i) the chosen features are generally not related to author's style but rather to functionality; (ii) significantly lower accuracy is observed in the case of multiple authors [5]; (iii) the approaches are easily defeated by refactoring techniques or simple obfuscation methods; and (iv) the approaches handle only one platform (e.g., x86). Dealing with the binary authorship attribution problem requires novel features that can capture author's style characteristics, such as a preference for using specific compilers, keywords, reused packages, implementation frameworks, and binary timestamps. More recently, the feasibility of authorship attribution on malware binaries was discussed at the BlackHat conference [6]. A set of features are employed to group malware binaries according to authorship. However, the process is not automated and requires a considerable human intervention.

Key idea: We present BINAUTHOR, a system designed to recognize authors' *coding habits* by decoupling them from program functionality in binary code. Instead of using generic features (e.g., n-grams or small subgraphs of a CFG [4]), which may or may not be related to authorship, BINAUTHOR captures coding habits based on a collection of functionality-independent choices frequently made by authors during coding (e.g., preferring to use either `if` or `switch`, and relying more on either object-oriented modularization or procedural programming). We first investigate a large collection of source code and their mapping to assembly instructions in order to determine what coding habits may be preserved in the binary, and consequently design and integrate features based on such habits into BINAUTHOR. Then, we apply it to a series of problems related to binary authorship attribution, namely, attributing the author of a given binary from a set of author candidates inside a repository, identifying the presence of multiple authors of the same binary on the basis of function-level analysis, and automatically extracting authorship-related evidences from real malware binaries.

Summary of results: Our evaluation shows that BINAUTHOR outperforms existing approaches [2–4] in several aspects. Specifically, the system attributes a larger number of authors with a significantly higher accuracy (approximately 90%) based on large-scale datasets extracted from selected open-source C++ projects on GitHub [7], Google Code Jam events [8], Planet Source Code contests [9], and students' programming projects. Furthermore, BINAUTHOR is more robust than previous methods in the sense that the system can still attribute authors with an acceptable accuracy after the code is subjected to refactoring and simple obfuscation techniques. Finally, applying BINAUTHOR to real malware binaries (`Zeus-Citadel`, `Stuxnet-Flame`, and `Bunny-Babar`) automatically generates evidence of similar coding habits shared by each pair of malware. These types of evidence match the findings of the technical reports published by antivirus vendors [10, 11] and reverse engineering team [6].

Contributions: The main contributions of this study are:

(1) BINAUTHOR yields a higher accuracy that survives refactoring techniques and simple obfuscation. This shows the potential of BINAUTHOR as a practical tool to assist reverse engineers in a num-

ber of security-related tasks, such as identifying the author of a malware sample, and clustering malware samples based on common authors.
(2) BINAUTHOR is amongst 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 [10, 11] and reverse engineering teams [6].
(3) The filtration component of BINAUTHOR (which labels functions as compiler-related, library-related, or user-related functions) is a stand-alone component. In addition, it can be employed as pre-processing steps in other applications, such as clone detection, to allow for the reduction of false positives and the enhancement of performance. We made the code available to the community[1].

In this paper, we significantly extend our previous work, by studying three models: BINAUTHOR; BINAUTHOR$_1$, which combines BINAUTHOR with a compiler identification tool BINCOMP [12]; and BINAUTHOR$_2$, which combines BINAUTHOR with GITZ, a tool [13] for lifting assembly instructions to a canonical form. Specifically, our major extensions are as follows: i) we introduce a set of illustrative examples to answer the following question: *Does each programmer have a distinctive coding style reflected in his/her collection of functionality-independent choices?* (in Section 2.1); ii) we present use cases to highlight the interest of BINAUTHOR (in Section 2.3); iii) Since the filtration component of BINAUTHOR (which labels functions as compiler-related, library-related, or user-related functions) is a stand-alone component, we have added a detailed algorithm to make it possible for the researchers to leverage it with any binary analysis tool (in Section 3.1); iv) we propose a new choice called "variable choice" and then re-compute the significance of BINAUTHOR choices and accordingly re-conducted the results comparison in (Sections 3.2.2, 3.4, and 4.3, respectively); (v) we conduct new experiments to study the impact of training dataset and test size (in Section 4.7); (vi) we investigated the effect of choices on large-scale author identification and further showed the percentage of choices found in large-scale dataset (in Section 4.9); (vii) we provide a comparison between the three proposed models (in Section 4.10); (viii) finally, we have added more details to highlight the findings of BINAUTHOR when it is applied to real malware binaries (in Section 5) as well as new verifying method has been investigated in order to verify the correctness of BINAUTHOR findings (in Section 5.4).

2. Preliminaries

In this section, we first provide a general overview of the main idea of choosing and extracting the coding habit styles in BINAUTHOR followed by an illustrative example. Then, we provide the threat model. Finally, we introduce the use cases in which BINAUTHOR can be employed.

2.1. Stylistic Choices

Programmers may develop their own coding habits over time. For instance, a programmer may prefer to write a code that takes advantage of object-oriented modularization over procedural programming, or s/he may have a tendency to utilize more global variables than local variables and more `while` loops

[1]https://github.com/g4hsean/BinAuthor

rather than `for` loops, etc. Profiling such functionality-independent programming choices may help to capture a programmer's coding habits and stylistic characteristics. However, although many stylistic characteristics may exist in source code and can be used to distinguish authors' styles, source code is not always available, e.g., in the case of malware analysis. This leads to the following questions: *How many stylistic characteristics found in the source code would survive the compilation process, and could consequently be extracted from the binary? To what extent would the extracted characteristics represent the author's coding habits?*

Table 1
Examples of stylistic choices that can be captured at binary level

Category	Sample Choices	Captured?
Keyword	`printf` vs. `cout`	✓
	`printf` vs. `printf` with new line	✓
	`cout` vs. `cout` with new line	✓
	`scanf` vs. `cin`	✓
	`void` vs. `int`	✗
General	`if/else` vs. `switch/cases`	✓
	`for` loop vs. `while` loop	✗
	Files vs. memory	✓
	API calls	✓
Unusual	Empty methods	✗
	Flushing buffer unnecessarily and excessively	✓
	Excessive printing of system clock time	✓
Action	How a function is ended	✓
	How parameters are provided	✓
	How exceptions are handled	✓
	How directives are defined	✗
	Use of recursion	✓
Quality	Using system() calls	✓
	Using semaphores	✓
	Handling standard library errors	✓
	Reopening of already opened files	✓
	Code reuse	✓
	Use of infinite loops	✓
	Modifying constants through pointers	✓
Structure	Vectors vs. arrays	✓
	Global variables vs. local variables	✓
	Procedural vs. object-oriented	✓

To answer above questions, we investigate a large collection of source code and its mapping to assembly instructions to determine which stylistic characteristics may be preserved in binary. Additionally, we aim to isolate functionality-independent characteristics from those that are more likely driven by functionality requirements. We investigate various programs performing different tasks but written by the same author, for the purpose of *averaging out* the effect of functionality. To capture stylistic characteristics at different abstraction levels, BinAuthor employs many different types of features, such as statistical and structural features, opcode instructions, and specific template of instructions.

Table 2

Example of C++-keywords choices and options

CPP Keyword	Options	Alternative(s)
`printf`	`d` or `i` for signed integer. `f` or `F` for decimal floating point. `e` or `E` for scientific notation lower case or upper case. `a` or `A` for Hexadecimal floating point lower case or upper case. With or without new line.	`cout`
`cout`	With or without new line. Overloading «...«.... Using it with other keywords such as `put(cout,"x = ");`	`printf`
`cin`	Using the » operator. Using `cin.get`. Using `cin.getline`. Using `cin.ignore`.	`scanf`
`scanf`	`d` or `u` for decimal integer. `f`, `e`, `a`, or `g` for floating point. Number of parameters `%`. Scan a set or an individual element.	`cin`
`sscanf`	`d`, `i`, or `n` for integer. `o`, `u`, or `x` for unsigned integer. `e`, `f`, or `g` for long double.	
`fscanf`	`e`, `E`, `f`, `g`, `G` specifiers for floating point. `x`, `X` specifiers for hexadecimal integer.	`scanf`
`put`	Insert character to stream.	operator «
`getc`	Get character from stream.	`fgetc`
`fgetc`	Get character from stream.	`getc`
`ferror`	Check error indicator.	`perror`
`perror`	Print error message.	`ferror`
`strlen`	Get string length.	`sizeof ()`
`sizeof`	How many elements are in an array.	`strlen`

Examples of stylistic choices. Table 1 provides examples of the functionality-independent choices, and indicates whether they can be captured at binary level. For clarity, these choices are categorized into basic groups. For instance, the *Keyword* group indicates the preference of using a particular keyword over another. The preference can also be based on one variation of a keyword over another variations of the same keyword. For instance, calling a particular method often allows different options that can be indicative of style. A `printf`, for example, allows various format specifiers such as `%e` and `%E` for scientific notation. BINAUTHOR considers these different variations of keywords as defined by the C/C++ language reference [14]. Unusual coding styles, such as the implementation of empty methods, or making excessive use of particular calls unnecessarily, are grouped in the *Unusual* group. Table 2 provides examples of keywords that can be captured at binary level, their options, and their alternatives. For instance, `scanf` and `cin` are alternatives that can also be captured at binary level. The output of this preference is a set of strings that is used by detection component.

2.2. Threat Model

BINAUTHOR is specifically designed to assist reverse engineers in discovering information from a binary that may be related to its author(s). As such, it is not intended for general-purpose reverse engineering tasks, such as unpacking or deobfuscating malware samples. We investigate refactoring and obfuscation techniques later in this paper in order to demonstrate possible evading countermeasures that may be used by future malware authors in order to circumvent detection. It is worth to mention that since we are not employing machine learning techniques to identify the authors, adversarial machine learning models cannot be used by the adversaries to evade detection. More specifically, the threat model and scope of this paper are further clarified in what follows.

In-Scope Threats. Since the research on binary authorship attribution is still in its infancy, BINAUTHOR is certainly not meant as a bullet-proof solution. As such, strengthening it against possible countermeasures will be an ongoing and continuous battle. Nonetheless, we dedicated special care to evading techniques while designing and implementing BINAUTHOR. We have taken into consideration some potential evading techniques. First, we assume that the adversaries might apply refactoring techniques to evade detection. Second, the adversaries might apply obfuscation techniques to source and binary files. Third, a program binary can be significantly changed by simply choosing a different compiler or by altering the compilation settings. Finally, the adversaries may intentionally avoid or fake some of their coding habits.

We show how BINAUTHOR survives the first three aforementioned threats. As for the last threat, the features of BinAuthor have been carefully designed to capture coding habits at multiple abstraction levels, which makes it harder for adversaries to evade detection even if they are aware of the habits being looked for [15]. In addition, an operational solution is to customize and enrich the list of features based on the actual use case and learning data, which will both improve accuracy and make it more difficult for adversaries to hide all their habits.

Out of Scope Threats. As previously mentioned, BINAUTHOR is not intended for general-purpose reverse engineering tasks, such as unpacking, de-obfuscation, or decryption. The evading techniques we investigate are not intended to be exhaustive. Also, if adversaries possess sufficient knowledge about the exact settings (e.g., the exact list of choices) of a particular BINAUTHOR user, they may potentially apply BINAUTHOR to their malware in exactly the same manner and consequently tweak the code to evade detection. Further hardening of BinAuthor against such threats is an interesting and challenging research direction for future work.

2.3. Use Cases

The interest of BINAUTHOR is to determine the authorship of an anonymous piece of binary code based on coding style. Additionally, given a code that is written by multiple authors, it required to determine which part of the code is attributed to which author. It is assumed that a set of binary code samples is available, where the code is either labelled with the author(s) from a set of known candidate authors, or from other unknown authors. Given an anonymous piece of binary code, BINAUTHOR converts the code into a set of coding habits that are consequently used to either attribute the author(s), or conclude that the code is attributed to some author(s) not belonging to BINAUTHOR repository. This is useful for the following two applications: software infringement and forensic investigation.

In software infringement, a set of candidate authors is assembled based on previously collected malware samples, online code repositories, etc. There are also no guarantees that the anonymous author is one of the candidates since the test sample may not belong to any known authors. Finally, it may be suspected that a piece of code is not written by the claimed author, but yet there are no leads to who the actual author might be. This is the authorship verification problem.

In forensic investigation, FireEye, discovered that malware binaries share the same digital infrastructure and code, for instance, the use of certificates, executable resources and development tools. FireEye investigators eventually noticed that malware binaries of the same previously discovered infrastructures are written by the same group of authors. In such cases, training on such binaries and some random authors' code may offer a vital help to forensics investigators. In addition, testing recent pieces of malware binary code using some confidence metrics would verify if a specific author is the actual author.

In each of these applications, the adversary may try to actively modify the coding style of the anonymous code. In the case of software forensics, the adversary may modify his/her own code to hide his/her style. In the copyright and authorship verification applications, the adversary attempts to modify code written by another author to match his/her own style. In forensic applications, two of the parties may collaborate to modify the style of code written by one to match the other's style. BinAuthor emphasizes coding habits that are robust to adversarial manipulation. Since these habits are driven by the adversary's own coding style, they cannot easily/completely be modified.

3. BinAuthor

Our goal is to automatically identify the author(s) of binary code. We approach this problem using different distance metrics; that is, we generate a list of functionality-independent choices from training data of sample binaries with known authors. Hence, we propose a system encompassing different components, each of which is meant to achieve a particular purpose. Figure 1 illustrates the architecture of BinAuthor. The first component (*Filtration*), isolates user functions from compiler functions, library functions, and free open-source software packages. Hence, the outcome of this component is considered as a habit (e.g., the preference in using specific compiler or open-source software packages). For instance, using `GCC` compiler rather than `visual studio`, or utilizing `SQLite` rather than `MongoDB`, etc. The second component (*Feature Categorization*), analyzes binary code to extract possible features that represent stylistic choices. The third component (*Author Habits Profiling*), constructs a repository of habits of known authors. The last component (*Authorship Attribution*), performs matching to BinAuthor's repository for author classification attribution. The aforementioned components are explained in the remainder of this section.

3.1. Filtration Process

An important initial step in most reverse engineering tasks is to distinguish between user functions and library/compiler functions [2]; this saves considerable time and helps shift the focus to more relevant functions. As shown in Figure 1, the filtration process consists of three steps. First, FLIRT [16] (Fast Library Identification and Recognition Technology) is used to label library functions. Then, a set of signatures is created for specific FOSS libraries, such as `SQLite3`, `libpng`, `zlib`, and `openssl`, using Fast Library Acquisition for Identification and Recognition (FLAIR) [16]; this set is added to the existing signatures of the IDA FLIRT list. In the last step, compiler function filtration is performed. The underlying hypothesis is that compiler/helper functions can be identified based on a collection of static signatures that are created in the training phase (e.g., opcode frequencies). We analyzed a number of programs with different functionalities, ranging from a simple "Hello World!" program to programs

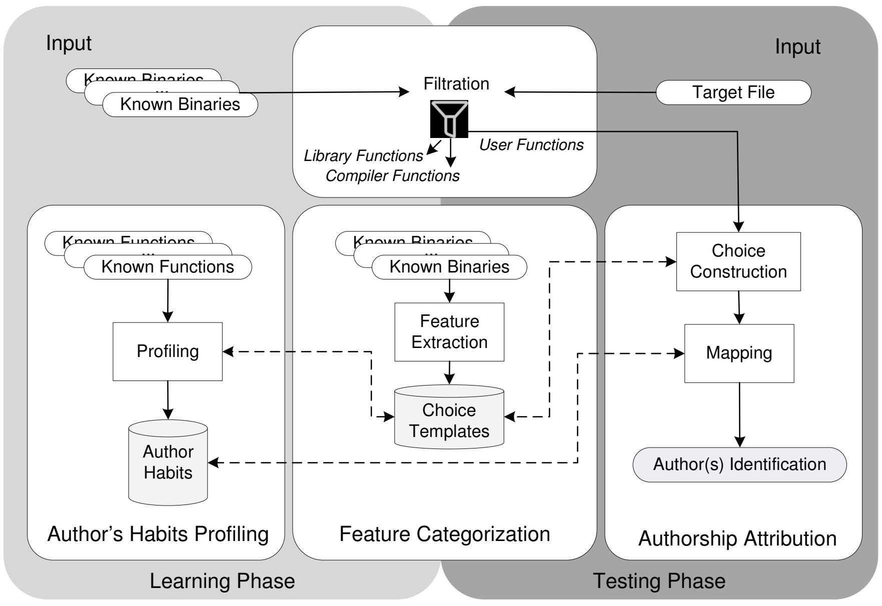


Fig. 1. BINAUTHOR Architecture.
Accesses to the databases are indicated by dashed lines.

executing complex tasks. Through the intersection of these functions combined with manual analysis, we collected *120* functions as compiler/helper functions. The opcode frequencies are extracted from these functions, and the mean and variance of each opcode are calculated.

After passing through FLIRT, each disassembled program P consists of n functions $\{f_1, \cdots, f_n\}$. Each function f_k is represented as set of m opcodes associated with the (mean, variance) pair of their frequency of occurrence in function f_k. More specifically, each opcode $o_i \in O$ has a pair of values (μ_i, ν_i), which represent the mean and variance values for that opcode.

Example. We introduce an example in Table 3 to show how we compute (μ_i, ν_i) for each opcode. For instance, suppose a compiler function has the following opcodes and corresponding frequencies: `push(19)`, `mov(31)`, `lea(13)`, and `pop(7)`. We compute the μ_i as $\mu_i = \frac{x_i}{\sum_{j=1}^{n} x_j}$, where x_i represents the frequency of opcode i and n is the number of opcodes. Similarly, ν_i is computed as $\nu_i = \frac{(x_i - \bar{x})^2}{n}$, where $\bar{x}$ represents the average of the frequencies. As shown in the table, μ for `push` is $19/(19 + 31 + 13 + 7) = 0.271$, and ν_i is $(19 - 17.5)^2/4 = 0.563$.

Table 3
An example of computing (μ_i, ν_i)

Opcode	`push`	`mov`	`lea`	`pop`
Frequency	19	31	13	7
μ	0.271	0.442	0.186	0.1
ν	0.563	45.563	5.063	27.563

Dissimilarity measurement $D_{i,j}$ is performed based on distance between the target function j and the training function i as per the following equation [17]:

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

where (μ_i, ν_i) and (μ_j, ν_j) represent the opcode mean and variance of the training function i and target function j, respectively.

Each opcode o_i in the target function is measured against the same opcode of all compiler functions in the training set. If the distance $D_{i,j}$ is less than a predefined threshold value $\alpha = 0.005$, the opcode is considered as a match. 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 function is labelled as *compiler-related* if the number of matched opcodes over the total number of opcodes (m) is greater than a predefined threshold value learned from experiments to be $\gamma = 0.75$. Otherwise, the target function is labeled as *user-related*.

3.2. Feature Categorization

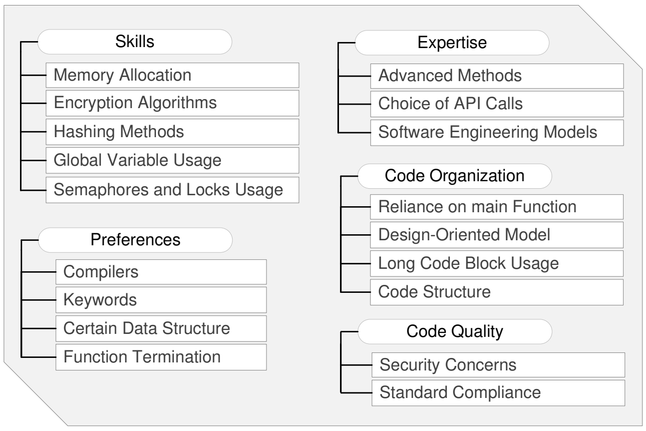


Fig. 2. Coding Habits Taxonomy

Determining a set of characteristics that remain constant for a significant portion of programs, written by an author, is analogous to finding human characteristics that can later be used for the identification of a specific person. To this end, our aim is to automate the finding of such program characteristics, and with reasonable computational cost. To capture coding habits at different abstraction levels, we consider a spectrum of such habits, as illustrated in Figure 2. As shown in Figure 2, an author's habits can be reflected in the preference of choosing certain keywords or compilers, the reliance on the main function, or the use of an object oriented programming paradigm. In addition, the manner in which the code is organized by the author may also reflect author habits. All possible choices are stored as a template in this step. Moreover, we introduce a novel taxonomy of coding habits in Figure 2. We provide details of each category of functionality-independent choices in the following subsections.

3.2.1. General Choices

General choices are designed to capture the author's general programming preferences; for example, preferences in organizing the code, terminating a function, use of particular keywords, or use of specific resources. In this subsection, we first introduce the three proposed general choices and then we provide the details on computing the final similarity score.

1) Code Organization: We capture how code is organized by measuring the reliance on the `main` function since it is considered a starting part for managing user functions. We define a set of ratios, as shown in Table 4, that measures the actions used in the `main` function. We thus capture the percentage of usage of keywords, local variables, API calls, and calling user functions, as well as the ratio between the number of basic blocks of the `main` function to the number of basic blocks of other user functions. These percentages are computed in relation 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 ratio values, which is used by the detection component.

Table 4

Features extracted from the `main` function: length(l): Number of instructions in the `main` function

Ratio Equation	Description
# of `push` / l	Ratio of local variables to length
# of `push` / # of `lea`	Ratio of local variables to memory address locations
# of `lea` / l	Ratio of memory address locations to length
# of calls / l	Ratio of function calls to length
# of indirect calls / l	Ratio of API calls to length
# of `BBs` / total # of all `BBs`	Ratio of the number of basic blocks of the `main` function to that of other user functions
# of calls / # of user functions	Ratio of function calls to the number of user functions

Table 5

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

2) Function Termination: BinAuthor captures how an author terminates a function. This could help identify an author since programmers may be used to specific ways of terminating a function. BinAuthor does not only consider the last statement of a function as the terminating instruction; rather, it considers the last basic block of the function with its predecessor as the terminating part. This is a realistic consideration since various actions may be required before a function terminates. To this end, BinAuthor not only considers the usual terminating instructions, such as `return` and `exit`, but also captures other related actions that are taken prior to termination. For instance, a function may be terminated with a display of messages, calling another function, releasing some resources, communication over networks, etc. Table 5 shows examples of what is captured in relation to the termination of a function. 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.

3) Keyword and resource preferences: BINAUTHOR captures the author's preference of using different keywords or resources. We consider only groups of such preferences with equivalent or similar functionality in order 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.

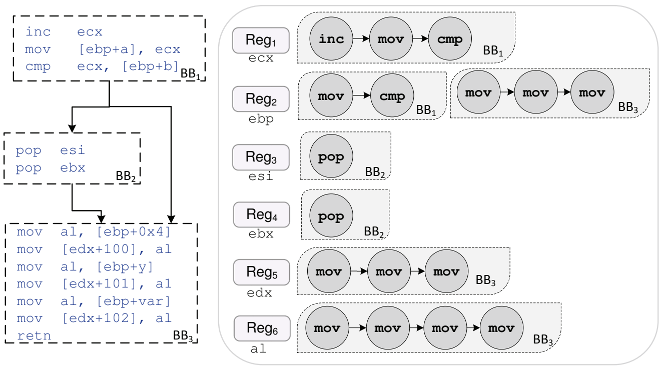
General Choice Computation: We compute a set of vectors (v_{gi}), where g represents the general category and i represents the sub-category number. To consider the reliance on the `main` function, a vector v_{g1}, representing related features, is constructed according to the equations shown in Table 4. 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, which tracks the annotations to `call` and `mov` instructions. For instance, excessive use of `fflush` will cause the string `"_imp_fflush"` to appear frequently in the resulting binary. 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 distance. 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 for the purpose of author classification.

3.2.2. *Variable Choices*

Developers often have their own habits for defining local and global variables, which may originate from the author's experiences or skills. The variable chain has been shown to greatly improve author attribution of source code [18]. It has been defined as the variable usage among different functions. Inspired by this work, we introduce a register chain to capture authors' habits in using variables. We define the register chain concept as the states of using a particular register through all basic blocks in a user function. To avoid compilation setting effects, we normalize the registers to general names such as Reg_1, Reg_2, etc. and keep their occurrence order. Useful characteristics of such chains include the longest chain, the shortest chain, the number of existing chains, the liveness of registers among basic blocks, etc.

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

Example: In what follows, we illustrate how a register chain is extracted. Part of the Control Flow Graph (CFG) of the `RC4` function in `Citadel` is shown in Figure 3(a). Figure 3(b) shows the construction of the register liveness [19] for the indicated registers. As illustrated in Figure 3(b), the used registers `ecx`, `ebp`, `esi`, `ebx`, `edx`, and `al` are normalized to $Reg_1, \cdots, Reg_6$. The first, second, and third basic blocks manipulate $\langle Reg_1, Reg_2 \rangle$, $\langle Reg_3, Reg_4 \rangle$, and $\langle Reg2, Reg5, Reg6 \rangle$ registers, respectively. BinAuthor captures the register liveness by storing the set of basic blocks where the register is alive. For instance, the Reg_2 register appears in the first and third basic blocks and does not appear in the second basic block, so we represent the liveness of the Reg_2 register as $\{BB_1, BB_3\}$. A summary of the registers liveness is given in Table 6.



```
inc   ecx
mov   [ebp+a], ecx
cmp   ecx, [ebp+b]
```

```
pop  esi
pop  ebx
```

```
mov  al, [ebp+0x4]
mov  [edx+100], al
mov  al, [ebp+y]
mov  [edx+101], al
mov  al, [ebp+var]
mov  [edx+102], al
retn
```

Fig. 3. (a) Part of the CFG of `RC4` (b) Register chain

Table 6

Register liveness (✓indicates that the register is alive in a `BB`)

Register	BB_1	BB_2	BB_3
Reg_1	✓	-	-
Reg_2	✓	-	✓
Reg_3	-	✓	-
Reg_4	-	✓	-
Reg_5	-	-	✓
Reg_6	-	-	✓

Variable Choice Computation: Since each function may have a large number of register chains, BinAuthor employs locality-sensitive hashing (LSH) for feature reduction. The hash is calculated over all sets of chains, and only those with similar hash values are clustered (hashed) to the same bucket. In the case of register chain similarity, similar register chains will be hashed to the same bucket. Once register chains have been hashed to a corresponding bucket, any bucket containing more than one similar hash value is identified and a list of candidate register chains is extracted. Finally, similarity analysis is performed to rank the candidate pairs obtained from the previous steps. The similarity score obtained from this choice is λ_v.

3.2.3. Quality-Related Choices

We investigate code quality in terms of standard compliance with C/C++ coding standards and security concerns. In the literature, code quality can be measured with different indicators, such as testability, flexibility, adaptability [20], etc. BinAuthor defines rules for capturing code that exhibits either relatively high or low quality. For any code that cannot be matched using such rules, the code is labeled as 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.

Rules: Examples of low-quality coding styles include reopening already opened files, leaving files open when they are no longer in use, attempting to modify constants (i.e., through pointers), using float variables as loop counters, and declaring variables inside of a switch statement, which can result in unexpected/undefined behavior due to jumped-over instructions. Examples of high-quality coding styles include handling errors generated by library calls (i.e., examining the returned value by `fclose()`), avoiding reliance on side-effects (i.e., $++$ operator) within particular calls such as `sizeof` or `_Alignof`, averting the use of particular calls on some environments or using them with protective measures (i.e., the use of `system()` in Linux may lead to shell command injection or privilege escalation; hence, using `execve()` instead is indicative of high-quality coding), and the use of locks and semaphores around critical sections.

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 [21]. We employ the idioms templates in [21] 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 [22] for this computation due to its efficiency. The similarity is represented by λ_q, which is used for author classification purpose.

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

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.

3.2.4. Embedded Choices

We define embedded choices by actions that are related to coding habits present in the source code that are not easily captured at the binary level by traditional features such as strings or graphs. For instance, initializing member variables in constructors and dynamically deleting allocated resources in destructors are examples of embedded choices. As it is not feasible to list all possible features, BinAuthor relies on the fact that opcodes reveal actions, expertise, habits, knowledge, and other author characteristics, and analyzes the distribution of opcode frequencies. Our experiments show that such a distribution can effectively capture the manner by which the author manages the code. As every single action in source code can affect the frequency of opcodes, BinAuthor targets embedded choices by capturing the distribution of opcode frequencies.

Embedded Choice Computation: For measuring the distance between distributions of opcode frequencies, the Mahalanobis distance [23] is used to measure the similarity of opcode distributions among different user functions. The Mahalanobis distance is chosen because it can capture the correlation between opcode frequency distributions, and this correlation represents the embedded choices. The similarity returned is represented by λ_e.

3.2.5. Structural Choices

Programmers usually develop their own habits in terms of structural design of an application. They may prefer to use a fully object-oriented design or they may be more accustomed to procedural programming. Such structural choices can serve as features for author identification. To avoid functionality, we consider the common subgraphs and the longest path for each user function, and then intersect them among different user functions. These subgraphs are defined as k-graphs, where k is the number of nodes. These common k-graphs form author signatures since these graphs always appear regardless of the program functionality. In addition, we consider the longest path since it reflects how an author tends to use deep or nested loops.

Example: An author may organize different classes in an ad hoc manner, or organize them in a hierarchical way by designing a driver class to contain several manager classes, where each manager is responsible for different processes (a collection of threads running in parallel). Both ad hoc and hierarchical organizations will create a common structure in the author's programs.

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 [24]. Then, a similarity measurement is performed over these strings using the normalized compression distance (NCD) [25], which enhances search performance. We have chosen NCD since it allows us to concatenate all the common subgraphs that appear in author's programs. Additionally, it allows for 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.

3.3. Discussion

Our main intuition is based on the following hypothesis that the coding characteristics of programmer/developer are resulted according to their educational background, level of expertise, and coding traits and preferences. To achieve this, we defined a set of coding habits based on the following intuitions for each category.
General Choice: The developers may use different keywords according to their preferences, experience, and knowledge. For instance, the data types operated by author could reveal data access patterns used by authors. Moreover, using specific resources (e.g., file over memory) or terminating the function could reveal authors' habits.
Structural Choice: This choice captures the author's habits in organizing and structuring the code, for instance, creating deeply nested code, unusually long functions or long chains of assignments, or there are some authors who prefer object oriented style or spaghetti oriented style. Moreover, it captures the use of API calls and recursion. We do believe this habit is derived from the author's experience.
Variable choice: This choice captures the method by which variables are manipulated, such as identifying aliases of the stack pointer. Further, developers often have their own habits for defining local and

global variables, which may originate from the author's experiences or skills. For example, PC-relative addressing often represents accessing a global variable, while scaled indexing often represents accessing an array. Moreover, the variable chain has been shown to greatly improve author attribution of source code [1].

Quality-Related Choice: One of the characteristics that can reflect the authors' expertise and knowledge is the code quality. Therefore, we investigate the code quality in terms of standard compliance with C/C++ coding standards and security concerns.

Embedded Choice: We define embedded choices to capture the actions which are related to coding habits (present in the source code) that are not easily captured at the binary level. This choice can reveal several characteristics such as: expertise, habits, and knowledge. As every single action in source code can affect the frequency of opcodes, BinAuthor targets embedded choices by capturing the distribution of opcode frequencies.

3.4. 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 each choice individually in order to predict class probabilities (e.g., the probability of identifying an author). For this purpose, we use the introduced dataset in Section 4.2. To prevent the overfitting, we test each dataset individually then we compute the average of the weights. The probability outcomes of logistic regression prediction is illustrated in Table 7. We calculate the weights as follows.

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

where p_s is the smallest probability value (e.g. 0.39 in Table 7), p_i is the probability outcome from logistic regression of each choice, and the *rnd* function rounds the final value. For instance, the weight for general choice is calculated as $2.128205/(2.128205 + 1.615385 + 1.333333 + 1 = 6.07692) = 0.35$. The probability outcomes of logistic regression prediction is illustrated in Table 7.

Table 7

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 author attribution (A) 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 7, and λ_i is the distance metric value obtained from each choice ($\lambda_g, \lambda_q, \lambda_e$, and λ_s) as described in Section 3.2. We normalize the probabilities of all authors, and if $P \geqslant \zeta$, 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.

4. Evaluation

4.1. 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 [26]. Gephi [27] 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.

4.2. Dataset

Our dataset is consisted of several C/C++ applications from different sources, as described below: (i) GitHub [7], where a considerable amount of real open-source projects are available; (ii) Google Code Jam [8], an international programming competition, where solutions to difficult algorithmic puzzles are available; (iii) Planet Source Code [9], a web-based service that offers a large amount of source code written in different programming languages; (iv) Graduate Student Projects at our institution. Statistics about the dataset are provided in Table 8. We compile the source code with different compilers and compilation settings to measure the effects of such variations. We use GNU Compiler Collection (version

Table 8
Statistics about the dataset used in the evaluation of BINAUTHOR

Source	**# of authors**	**# of programs**	**# of functions**	**average # of code lines**	**# of files**
GitHub	5	10	400	5000	754
Google Code Jam	50	250	650	80	250
Planet Source Code	44	168	580	250	400
Graduate Student Projects	25	125	875	250	450

4.8.5) with different optimization levels, as well as Microsoft Visual Studio (VS) 2010. We study the impact of Clang and ICC compilers, as described in Section 4.11.

We perform two sets of experiments. In the first phase, we use a dataset for different authors with the same functionalities. The purpose of this experiment is to ensure that we are not attributing the functionality. This shows that our tool segregates between the binaries according to the authorship attribution not functionality. Afterwards, we move to the second phase to make sure that our tool attributes the binary according to its author. In this phase, we collect binaries for each author with different versions, and variant functionalities. 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.

4.3. Evaluation Metrics

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. To evaluate BINAUTHOR and to compare it with existing methods, precision (*P*) and recall (*R*) measures 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. We choose $F_{0.5}$ to preciously identify the author and this metric is more sensitive to false positives rather than false negatives. Therefore, precision is of higher priority than recall. We employ the F-measure as follows:

$$F_{0.5} = 1.25 \,.\, \frac{P\,.\,R}{0.25P + R}$$

4.4. Accuracy

We compare BINAUTHOR with the existing authorship attribution methods [2–4]. The source code and dataset of our previous work, OBA2 [2], 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. [3] and Rosenblum et al. [4] are available at [28] and [29], 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 datasets are not available, we compare *BinAuthor* with these methods by using the datasets mentioned in Table 8.

Figure 4 details the results of comparing the accuracy between BINAUTHOR and all other 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 BINAUTHOR achieves better accuracy in determining the author of binaries. Taking all four approaches into consideration, the highest accuracy of authorship attribution is close to 96% on the Google Code Jam with less than 50 authors, while the lowest accuracy is 22% when 150 authors are involved on all dataset together. We believe that the reason behind Caliskan-Islam et al. approach superiority on Google Jam Code is that this dataset is simple and can be easily decompiled to source code. BINAUTHOR also

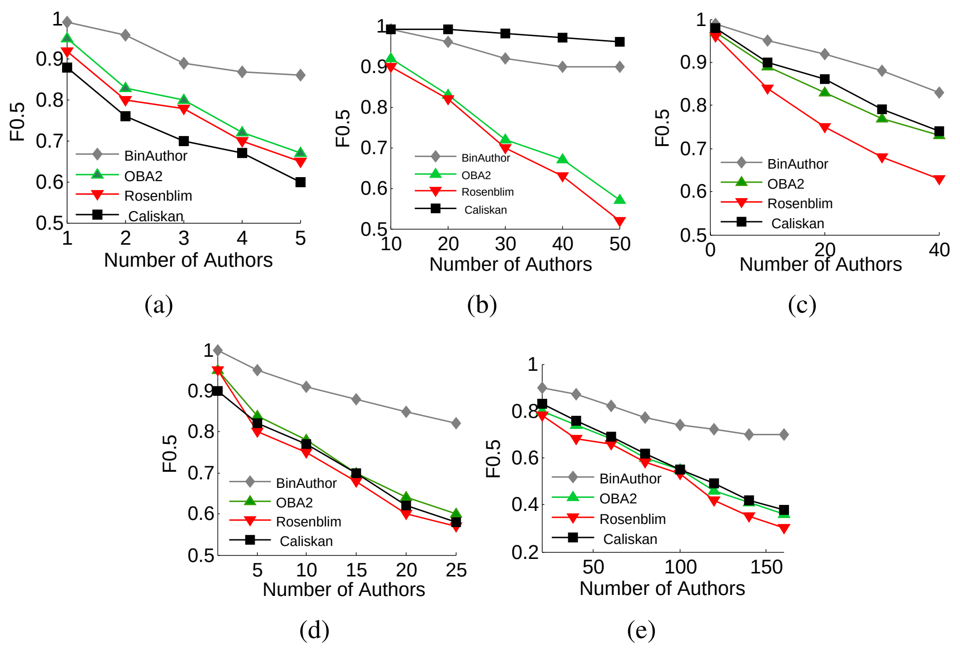


Fig. 4. Accuracy results of authorship attribution obtained by BINAUTHOR, Caliskan-Islam et al. [3], Rosenblum et al. [4], and OBA2 [2], on (a) Github, (b) Google Code Jam, (c) Planet Source Code, (d) Graduate Student Projects, and (e) All datasets

identifies the author of Github dataset with an accuracy of 90%. 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 lower accuracy obtained by BINAUTHOR is approximately 75% on a Graduate student projects with 25 authors. This is explained by the fact that programs in Graduate student projects have common choices among different students due to assignment rules, which force students to change/restrict their habits accordingly. When the number of authors is 140 on the mixed dataset, the accuracy of Rosenblum et al., Caliskan-Islam et al., and OBA2 approaches drop rapidly to 30% on all datasets, whereas our system's accuracy remains greater than 75%. This provides evidence for the stability of using coding habits in identifying authors. In total, the different categories of choices achieve an average accuracy of 84% for ten distinct authors and 75% when discriminating among 152 authors. These results show that author habits may survive the compilation process.

Accuracy Interpretation. In addition to superior accuracy, our results demonstrate the following advantages of BINAUTHOR:

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) *Application Type*: We find that the accuracy of existing methods [2, 4] depends heavily on the application's domain. For example, in Figure 4, a good level of accuracy is observed for the Graduate student projects dataset, where the average accuracy is 75%. One explanation involves the fact that Rosenblum et al.'s approach extracts LibCalls, which are more useful for code in academia than in other contexts. This can be explained by students' choice to systematically rely on external libraries and to implement MFC APIs, for example. The results also show that OBA2 [2] relies on the application, as their approach extracts the manner by which the author handles branches. For instance, the accuracy drops from 92% to 57% when Google Code Jam was used. After investigating the source code, we noticed that the number of branches is not high, which makes the attribution even more difficult. Moreover, the manner by which branches are handled becomes less distinct with larger numbers of authors.

3) *Source of Features*. Existing methods use disassembler and decompilers to extract features from binaries. Caliskan-Islam et al. [3] use a decompiler to translate the program into C-like pseudo code via Hex-Ray [30]. They pass the code to a fuzzy parser for C, thus obtaining an abstract syntax tree from which features can be extracted. In addition to Hex-Ray limitations [30], 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.

4.5. False Positives

We investigate the false positives in order to understand the situations where BinAuthor is likely to make incorrect attribution decisions. Figure 5 shows the false positives relationship with the number of authors in repository. For this experiment, we consider 4 programs for each author. For instance, when we have 50 authors (4*50 = 200 programs), BinAuthor misclassifies 16 programs. Also, when the number of authors is 1100 (1100*4 = 4400 programs), the number of false positives is 402. These false positives (402) are investigated in Figure 5 (a). Also, we show the false positives in each dataset as shown in Figure 5 (b). It is obviously shown that the false positives rate for student dataset is the highest rate and we believe the reason behind this is that each student should follow the standard coding instructions which restrict him/her to have their habits.

4.6. Identifying the Presence of Multiple Authors

In the previous section, we showed how our system identifies an author among a set of authors. In this section we are interested in determining whether our system is capable of identifying the presence of multiple authors. We therefore extend our system to an open world setting by clustering functions that are written by the same author.

We choose a common measure of cluster agreements known as Adjusted Mutual Information (AMI) [31] due to its stability across different numbers of clusters, which facilitates the comparison of different datasets [31]. For this measure, we use values within the range $[0, 1]$, where higher scores indicate better cluster agreement. We manually collect a set of GitHub projects and check the contributors who have written the code of these projects. We limit our system to programs written in C/C++ and

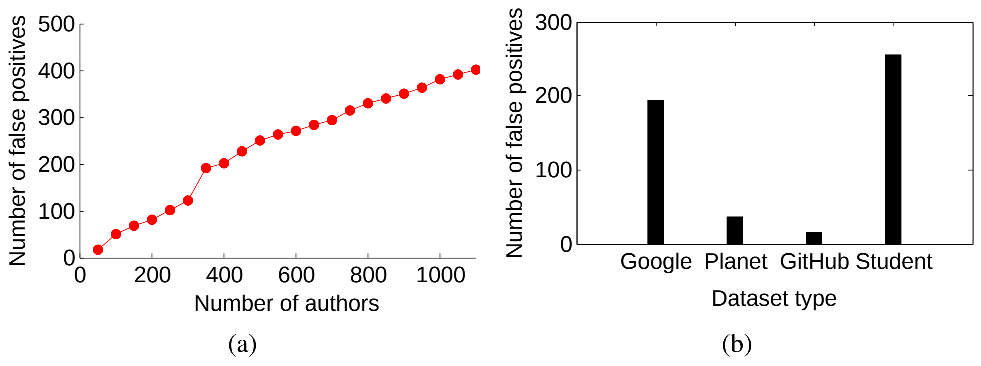


Fig. 5. False Positive Analysis

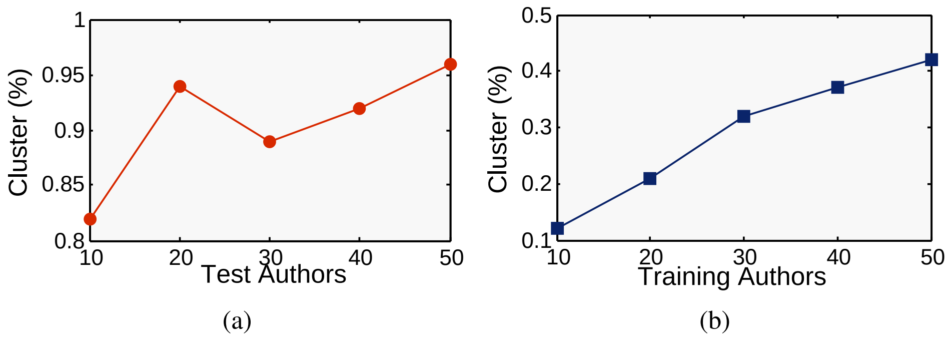


Fig. 6. Clustering with metric learning. (a) The correct cluster rate (b) The effect of training size on the correct cluster rate

we ignore authors whose contribution is merely adding lines. For this purpose, we collect 65 projects to which 50 authors contributed. In what follows, we briefly describe the authorship clustering algorithm. We randomly select *N* authors from GitHub projects and use large margin nearest neighbors to learn a distance metric over the feature space [32]. We then randomly select 10, 20, 30, 40, and 50 different authors, and cluster their programs using *k*-means with and without transforming the data with the learned metric. Due to the randomness in our experiment (dataset and *k*-means clustering algorithm), we repeat the experiment 10 times and compute the average AMI. Figure 6 depicts the clustering results.

In Figure 6 (a) we show the effect of the size of training authors on the improvement of clustering results. As shown in Figure 6 (b), using more training authors leads to greater improvement. We conclude that stylistic information derived from one set of authors can be transferred to improve the clustering of programs written by a different set of authors.

4.7. *Impact of Training and Testing Set Sizes*

Figure 7 shows the impact of training set size. The number of files per author is a different factor from the number of functions, although they are highly correlated. The former appears to be a better predictor of performance in our experiments. One possible reason involves splitting files into functions, which allows us to capture the variance in an author's habits. Two points are evident from the graph. First, accuracy is poor with less than 8 training files for the unusual choice. This is due to the fact that this choice appears very seldom in programs written by the same author. This choice serves as a unique

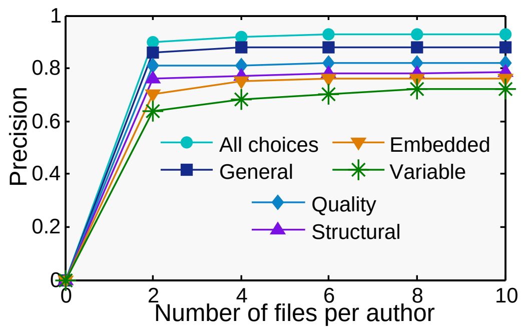


Fig. 7. Effect of training set size

signature for that author if it exists. Second, all choices together have high accuracy even with a low number of training files. The results suggest that authors who wish to be identified should consider 10 files in order to achieve an accuracy of 82%. This number is found through our experiments, and we notice this number of files allows our system to identify the consistent and common habits in author files. 10 files is considered as a minimum requirement to attribute an author with an accuracy of 80%. As the number of files increases, so does the accuracy our tool can achieve.

Figure 8 displays the results of identifying a target author when the number of his related functions varies. All choices together require 6 files of the target author in order to identify him/her with an accuracy of 88%, whereas the unusual choice identifies the author with an accuracy of 38% when the number of files is 10. These results are indicative of the ability of BinAuthor to match an author based on few functions that are related to him/her. Some functions do not have enough information to capture author related habits, and may thus be ignored.

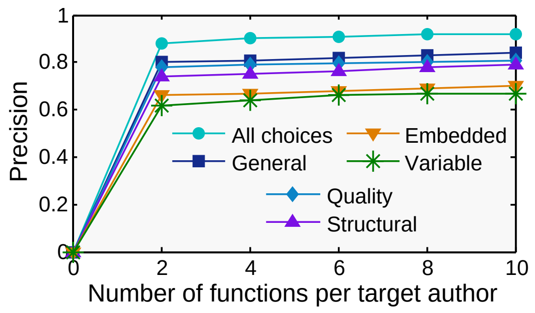


Fig. 8. Effect of test set size

Our results indicate that while the identifiability of small amounts of functions is slightly reduced, we still achieve 75% accuracy on average with only two functions. Furthermore, the curves appear to reach an asymptote at around 6 functions per target file with an accuracy of roughly 80%. We believe that this accuracy is a lower bound of the performance that could be achieved in practice when more functions are usually available.

4.8. Scalability

Security analysts or reverse engineers may be interested in performing large-scale author identification, and in the case of malware, an analyst may have to 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 a large dataset for the purpose of large-scale authorship attribution, we obtained programs from three sources: Google Code Jam, GitHub, and Planet Source Code. We eliminated 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 [3] to download all the code submitted to the Google Code Jam competition. The programs from the other two sources were downloaded manually. The programs were compiled with the Visual Studio and gcc compilers, using the same settings as those in our previous investigations. The experiment evaluated how well the top-weighted choices represent author habits. The results of the large-scale author identification are shown in Figure 9.

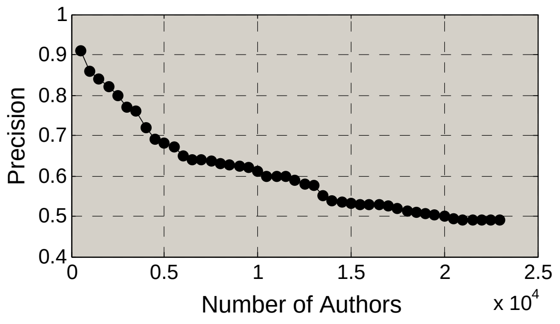


Fig. 9. Large-scale author attribution

The figure shows the precision with which BinAuthor identifies the author, and its scaling behavior as the number of authors increases is satisfactory . An author is identified among almost 4000 authors 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 tested BinAuthor on the programs from each of the sources. We found high precision (88%) for samples from the GitHub dataset, 82% precision for samples from the Planet dataset, and low precision (51%) for samples from Google code jam. After analyzing these results, we have found that the authors who wrote the code for difficult tasks is easier to attribute than easier tasks.

4.9. Impact of Choices

We study the impact of each choice on precision as depicted in Figure 10. For example, when the number of authors is $15,000$, BinAuthor achieves a precision of 70% based on the use of variables, while with structural considerations, it achieves a precision of 50%, the lowest for all the choices. When the number of authors reaches $21,000$, the precision for embedded, general, quality, variable, and structural

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

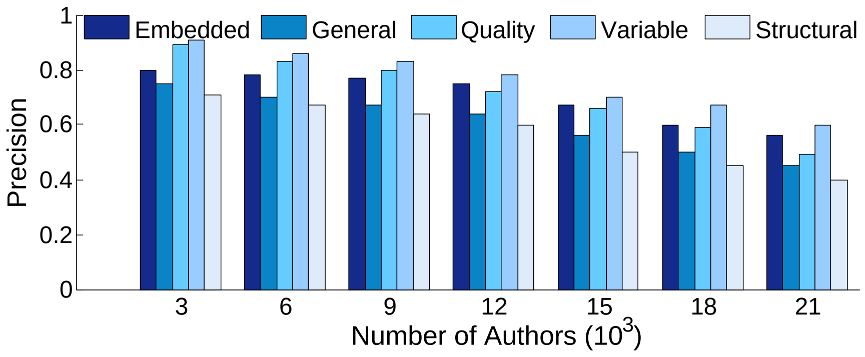


Fig. 10. The effect of choices on large-scale author identification

choices is 56%, 45%, 49%, 60%, and 40%, respectively. From Figure 10, it seems reasonable to expect that when the number of authors exceeds $20,000$, there will be little additional change in the precision. According to this experiment, the precision of the attributions made based on the choices extracted and stored in the BINAUTHOR repository will differ depending on the dataset as presented in Figure 11.

As mentioned earlier, BINAUTHOR weights a set of functionality-independent choices, and it can adjust the weights according to the dataset type. Hence, we show the percentage of choices that BINAUTHOR finds in each dataset as illustrated in Figure 11. For instance, the percentage of quality choices found in Google code dataset is 6%.

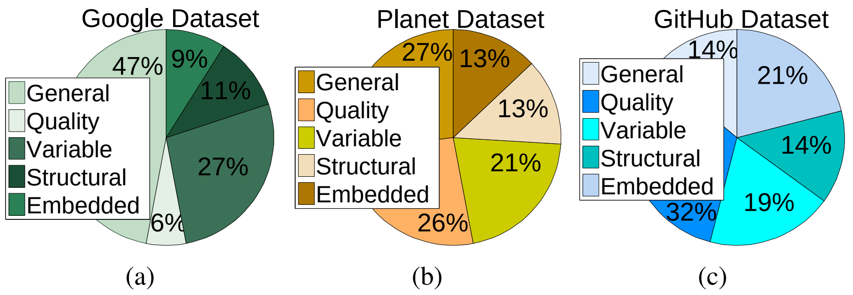


Fig. 11. The percentage of choices found in large-scale datasets

4.10. *Impact of Compilers*

To investigate the impact of different compilers, we studied three models: BINAUTHOR; BINAUTHOR$_1$, which combines BINAUTHOR with BINCOMP [12], a tool for compiler identification; and BINAUTHOR$_2$, which combines BINAUTHOR with the adopted proposed method in GITZ [13] for lifting assembly instructions to canonical form.

Model I: **BINAUTHOR**. Specifically, we are interested in studying the impact of different compilers and compilation settings on the features used by BINAUTHOR and consequently on its accuracy. To prepare the required datasets, we collected 5 programs from each of 50 authors to use as training sets. We then compiled the source code with the VS, GCC, ICC, and Clang compilers and various optimization settings (O0, O2, and Ox flags). To test the effects of different compilers, we kept the binaries compiled with VS as the training set and tried to identify the authors of binaries compiled with the other compilers. The results show that for most optimization levels, coding habits are largely preserved. However, the accuracy drops significantly (from 0.86 to 0.43) when the Clang or the ICC compiler is used since BINAUTHOR is not trained based on ICC and Clang binaries. While there is only a slight drop in accuracy (from 0.86 to 0.83) when the GCC compiler is used. We then classified the effect of each compiler as *full*, *partial*, and *no* effect, with a full effect corresponding to a significant difference between choices (here we consider the choices extracted from VS are the baseline).

As shown in Table 9, we used the choices extracted from binaries generated by the VS compiler as a reference for comparison with the same choices extracted from binaries generated by the other compilers (GCC, ICC, Clang) to judge the degree of the effect. Some of the choices are robust to compiler effects because compilers change the syntax of code but not its semantics. Since some of the choices used in BINAUTHOR (embedded and qualitative) are based on semantic features, they are not substantially affected. However, we observed that the choices involving variables are affected more significantly since they rely on the flow of instructions.

Table 9
Effects of Different Compilers on Choices: (●) full effect,(◐) partially effect, and (○) no effect.

	Compilers			
Choice	**VS**	**GCC**	**ICC**	**Clang**
General	○	○	◐	◐
Structural	○	◐	●	●
Qualitative	○	○	◐	◐
Embedded	○	○	○	○
Variable	○	◐	●	●

***Model II:* BINAUTHOR$_1$**. In light of the impact of compilers on the choices (Table 9), we made minor modifications to the proposed *Architecture* presented in Figure 1, which is illustrated in Figure 12).

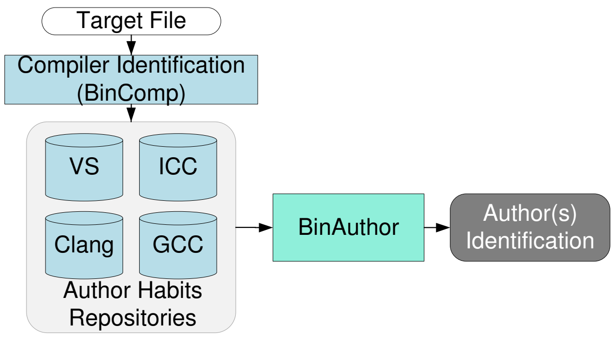


Fig. 12. The BINAUTHOR$_1$ architecture

We extract choices from binaries compiled by the four compilers (GCC, VS, Clang, and ICC) and then store them in four corresponding repositories. Given a target binary, we apply our existing tool BINCOMP to identify the source compiler so that BINAUTHOR can use the corresponding repository to identify the author of the binary. It should be pointed out that BinComp is designed to extract the elements comprising binary provenance, including the compiler, version, optimization speed levels, and compiler-related functions. The accuracy of this model is 0.86, 0.85, 0.89, and 0.81 for VS, GCC, ICC, and Clang, respectively. This shows that BinAuthor can achieve a high accuracy regardless the source of the compiler when it is leveraged with compiler identification tool.

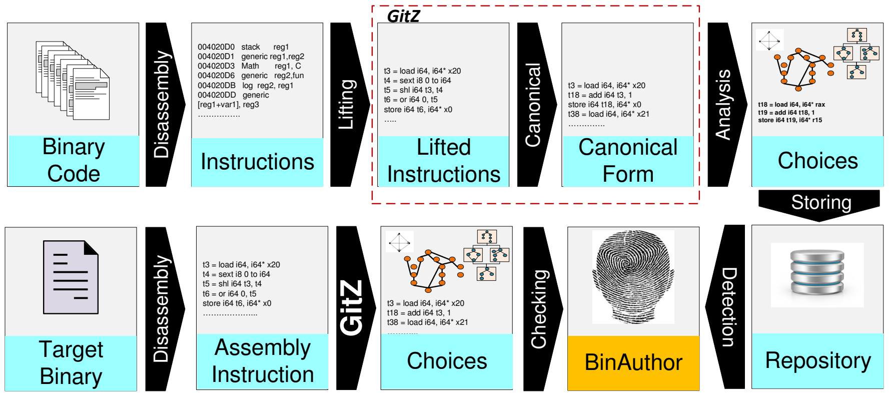


Fig. 13. BinAuthor$_2$ architecture

Model III: **BinAuthor$_2$** Because Model II deals only with a limited number of compilers, we designed a new model that can handle the effect of any compiler as presented in Figure 13. We adapted a recent technique used in GitZ [13], which converts assembly instructions into a semantic representation called the canonical form. Canonicalization is the process to translate binary code or assembly code into a unique representation. One of the main benefits for this representation is extremely robust to heavy forms of obfuscation. This process also can preserve the side effects of a function (e.g., flags). The following steps are performed. First, GitZ is used to lift binaries to the intermediate representation LLVM-IR and also to canonicalize the instructions; at the same time, this step reoptimizes code which might not have been optimized previously. Next, we apply BinAuthor to the canonical instructions to extract the choices. However, the structural choices cannot be extracted from the canonical instructions, which is a limitation of this model. Finally, we build a repository of the extracted choices, which is used for authorship detection. This model, which is a compiler-agnostic system, achieves an accuracy of 0.82.

Finally, we build a repository of the extracted choices, which is used for authorship detection. This model, which is a compiler-agnostic system, achieves an accuracy of 0.88 as shown in Figure 14.

However, efficiency is a problem, and the time complexity is high since all the target functions must be converted into canonical form. Model I achieves the highest accuracy since it does not incorporate leveraged tools such as BinComp and GitZ which may generate some false positives. Further, it is more efficient in terms of the time required for choice extraction and author detection. We compare the three models in terms of processing time (it includes preprocessing, extracting, and detecting) as shown in Figure 15.

4.11. *Impact of Evading Techniques*

Refactoring Techniques. We consider a random set of 50 files from our dataset which we use for the C++ refactoring process [33, 34]. We consider the techniques of i) renaming a variable, ii) moving a

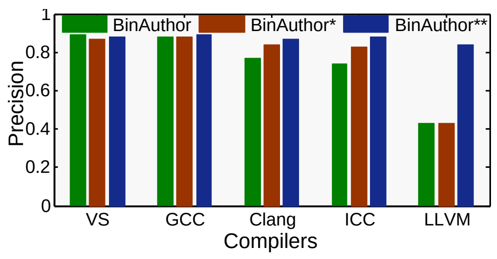


Fig. 14. The precision of the three different models

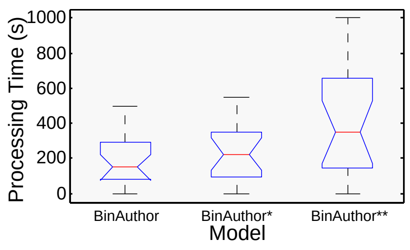


Fig. 15. The processing time for three models when the number of authors is 1000

Table 10

The evading techniques: methods used, tools used, and their affect on BinAuthor choices. (A) means the accuracy before applying obfuscation method while (A^*) means the accuracy after applying it. (○) means there is no effect while (●) means there is an effect. (NA) means is not applicable.

Tools	Method	Input	Output	A $\longrightarrow$ A^*	Choice				
					General	Variable	Quality	Embedded	Structural
Refactoring [33, 34]	RV	Binary	Binary	85 $\longrightarrow$ 84	○	●	○	○	○
	MM			86 $\longrightarrow$ 82	○	○	○	●	●
	NM			86 $\longrightarrow$ 83	○	○	○	○	○
Obfuscator-LLVM [35]	CFG flattening	Binary	Binary	86 $\longrightarrow$ 83	○	○	○	○	●
	Instruction substitution			86 $\longrightarrow$ 80	○	●	●	○	○
	CFG bogus			86 $\longrightarrow$ 81	○	○	○	○	●
DaLin [36]	Instruction reordering	Assembly	Assembly	86 $\longrightarrow$ 86	○	○	○	○	○
	Dead code insertion			86 $\longrightarrow$ 86	○	○	○	○	○
	Register renaming			86 $\longrightarrow$ 86	○	○	○	○	○
	Instruction replacement			86 $\longrightarrow$ 80	○	●	●	○	○
Trigress [37]	Virtulazation	Source	Source	86 $\longrightarrow$ 20	●	●	●	●	●
	Jitting			86 $\longrightarrow$ 0	●	●	●	●	●
	Dynamic			86 $\longrightarrow$ 10	●	●	●	●	●
PElock [38]	Hide procedure call	Assembly	Assembly	86 $\longrightarrow$ 85	○	○	○	○	●
	Insert fake instruction			86 $\longrightarrow$ 86	○	○	○	○	○
	Prefix junk opcode			86 $\longrightarrow$ 86	○	○	○	○	○
	Insert junk handlers			86 $\longrightarrow$ 86	○	○	○	○	○
Nynaeve [39]	Frame Pointer Omission	Source	Binary	86 $\longrightarrow$ 83	●	○	○	○	○
	Function inlining			86 $\longrightarrow$ 78	○	●	●	○	●
OREANS [40]	Encrypt binary	Source	Binary	NA	NA	NA	NA	NA	NA
Gas Obfuscator [41]	Junk byte	Assembly	Assembly	86 $\longrightarrow$ 86	○	○	○	○	○
Designed Script	Loop unrolling	Source	Source	86 $\longrightarrow$ 75	●	●	●	○	○

method from a superclass to its subclasses, and iii) extracting a few statements and placing them into a new method. We obtain an accuracy of 83.5% in correctly classifying authors, which is only a mild drop in comparison to the 85% accuracy 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 [42]. 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 functionality-independent choices from obfuscated samples. We obtain an accuracy of 82.9% in correctly classifying authors, which is only a slight drop in comparison to the 85% accuracy observed without obfuscation.

5. Applying BINAUTHOR to Real Malware Binaries

In this section, we apply BINAUTHOR to different sets of real malware: i) `Bunny` and `Babar`; ii) `Stuxnet` and `Flame`; and iii) `Zeus` and `Citadel`. These malware are selected based on researchers' claims that each of these pairs of malware originates from the same set of authors [6, 10, 11]. Due to the lack of ground truth, we compare BINAUTHOR outputs to the findings in those technical reports. Details about the malware dataset are shown in Table 11. Packed malware binaries are manually unpacked using OllyDbg [43]. For obfuscated binaries, we manually decrypt encrypted functions using a suitable decryption algorithm.

Table 11
Characteristics of malware datasets

Malware	Packed	Obfuscated	Source code	Binary code	Type	# of Binary functions	Source of malware sample
`Zeus`	✗	✗	✓	✓	PE	557	Our security laboratory
`Citadel`	✗	✗	✓	✓	PE	794	Our security laboratory
`Flame`	✗	✓	✗	✓	PE	1434	Contagio [44]
`Stuxnet`	✗	✓	✗	✓	PE	2154	Contagio [44]
`Bunny`	✓	✗	✗	✓	ELF	854	VirusSign [45]
`Babar`	✓	✗	✗	✓	ELF	1025	VirusSign [45]

5.1. *Applying* BINAUTHOR *to* `Bunny` *and* `Babar`

Findings. BINAUTHOR is able to find the following coding habits automatically: i) config all capital letters in XML, the preference in using Visual Studio 2008 (general choice); ii) using one variable over long chain (variable choice); iii) the methods of accessing freed memory, deallocating dynamically allocated resources, and opening the same process in the same function (quality choice); and iv) the common manner by which functions are managed (structural choice).

Statistics. BINAUTHOR finds common functions between the `Bunny` and `Babar` malware that share the aforementioned coding habits as follows: 494 functions share qualitative choices; *450* functions share embedded choices; *372* functions share general choices; *278* functions share variable choices; and *127* functions share structural choices. Furthermore, BINAUTHOR finds *340* functions which share *4* choices, *478* functions which share *3* choices, *50* functions which share *2* choices, and *230* functions which share *1* choice.

Summary. We consider the cluster based on *3* or more common choices written by one author, and *1* or *2* choices written by multiple authors. BINAUTHOR can cluster the functions written by one author, in which case the percentage is *44% ((340 + 478) / (854 + 1025))*, where *854* and *1025* are total binary functions for `Bunny` and `Babar` as shown in Table 11. On the other hand, the percentage of functions written by multiple authors is *23% ((150 + 290) / (854 + 1025))*. For the remaining *33%* of functions, no common choices are found. Based on our findings, we conclude that `Bunny` and `Babar` are written by a set of authors. Our results support the technical report published by Citizen Lab [6]. This report shows that both malware are written by a set of authors according to common implementation habits (general and qualitative choices) and infrastructure usage (general choice).

5.2. *Applying* BINAUTHOR *to* `Stuxnet` *and* `Flame`

Findings. BINAUTHOR automatically finds the following coding habits: i) using global variables, lua scripts, specific open source package SQlite, and choosing heap sort over other methods (general choice); ii) opening and terminating processes (qualitative choice); iii) recursion patterns and using POSIX socket API over BSD socket API (structural choice); and iv) closer functions in terms of Mahalanobis distance where the distance is closer to 0.1, and passing primitive types by value while passing objects by reference (embedded choice).

Statistics. BINAUTHOR finds common functions that share the aforementioned coding habits as follows: *725* functions share general choices; *528* functions share qualitative choices; *300* functions share embedded choices; and *189* functions share structural choices. Furthermore, BINAUTHOR finds *180* functions which share *4* common choices, *294* functions which share *3* choices, *515* functions which share *2* choices, and *689* functions which share *1* choice.

Summary. BINAUTHOR can cluster the functions written by one author where the percentage is *13% ((180 + 294) / (1434 + 2154))*, while *34% ((515 + 689) / (1434 + 2154))* of the functions are written by multiple authors. For the remaining *53%* of functions, no common choices are found. Based on our findings, we conclude that `Stuxnet` and `Flame` are written by an organization since they follow specific rules and targets. Our results support the technical report published by Kaspersky [10], which indicates that both malware use particular infrastructure (e.g., lua) and are related to an organization.

5.3. *Applying* BINAUTHOR *to* `Zeus` *and* `Citadel`

Findings. We apply BINAUTHOR to both binaries and cluster the functions based on functionality-independent choices. BinAuthor automatically finds the following coding habits: i) using network resources over file resources, creating configuration using mostly config files, and using specific packages such as webph and ultraVNC (general choice); ii) using switch statements over if statements (structural choice); iii) using semaphores and locks (qualitative choice); and iv) closer functions in terms of Mahalanobis distance with distance = 0.0004 (embedded choice).

Statistics. BINAUTHOR finds common functions that share the aforementioned coding habits as follows: 655 functions share general choices; 452 functions share qualitative choices; 370 functions share embedded choices; and 289 functions share structural choices. Furthermore, BINAUTHOR finds that 258 functions share 4 common choices, 194 functions share 3 choices, 588 functions share 2 choices, and 600 functions share 1 choice.

Summary. BINAUTHOR clusters 33% of functions as written by a single author, while *53%* of the functions are written by multiple authors. For the remaining 14% of functions, no common choices are found. Based on our findings, we conclude that `Zeus` and `Citadel` are written by a team of authors as results show that a large percentage of functions are written by more than one author. Our results match the findings of the technical report published by McAfee [11].

5.4. *Verifying the correctness of* BINAUTHOR *findings*

Due to the lack of ground truth, we verify the correctness of BINAUTHOR findings using following methods: Comparing BinAuthor outputs to the findings of human experts in available technical reports [6, 10, 11]; measuring the distance between the choices in one cluster and the choices in another to calculate the degree of similarity in order to provide a clear indication of whether the choices are closely related to specific malware packages.

Comparison with technical reports. We compare the BINAUTHOR 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 [6], 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 [10], which shows that both malware packages use similar infrastructure (e.g., Lua) and are associated with an organization. In addition, the BINAUTHOR 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, the BINAUTHOR findings provide much more information concerning the authorship of these malware packages. The correspondence between the BINAUTHOR findings and those in the technical report is as follows: all the choices found in the report [10] were found by BINAUTHOR, but they represent only 10% of our findings. The remaining 90% of the BINAUTHOR findings were not flagged by the report.
- For `Zeus` and `Citadel`, our results match the findings of the technical report published by McAfee [11], 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.

Table 12

Choices found in malware binaries

	Bunny	Babar	Stuxnet	Flame	Zeus	Citadel
Bunny	-	500	10	2	4	12
Babar	500	-	4	9	0	5
Stuxnet	10	4	-	750	14	3
Flame	2	9	750	-	17	6
Zeus	4	0	14	17	-	670
Citadel	12	5	3	6	670	-

Measuring similarity between choices in malware binaries. In this subsection, the goal is to assess the similarity between malware binaries by reporting the statistics about common choices (Table 12). We observed that there are only ten choices common to Bunny and Stuxnet, which clearly indicates that the malware packages were written by different authors. These choices are found in seven functions, which amounts to *(7/(854 + 2154))= 0.2%* shared author habits. In comparison, there are seventeen choices common to Flame and Zeus, found in thirty-eight functions, so the percentage of shared author habits is *(38 / (1434 +557)) = 2%*. The results in Table 12 may provide clues about the validity of the BINAUTHOR findings.

5.5. More examples of findings

In this subsection, we show some examples of the common choices that we find in our malware binaries. One of the good habit that we find is the termination of a process once it is no longer in use, as shown in Listing 1. Moreover, we find advanced feature (using mutex) as shown in Listing 2. We also notice that there is a bad habit in a set of functions where the author does not close the process. Listing 3 shows an example.

Listing 1: Qualitative choice: *opening process and terminating it*

```
        call    ds:OpenProcess
        mov     esi, eax
        test    esi, esi
        jz      short loc_41DB97
        push 104h
        lea     eax, [ebp+String1]
        push    eax;
        push    esi
        call    sub_42F93F
        cmp al, 1
        jnz     short loc_41DB94
        push    [ebp+lpExistingFileName]
        lea eax, [ebp+String1]
        call    sub_42DE63
        test    eax, eax
        jz      short loc_41DB94
        push    0FFFFFFFFh
        push    esi
        call    ds:TerminateProcess
```

Listing 2: Qualitative choice: *using mutex*

```
push    1
push    offset MutexAttributes ;
call    ds:CreateMutexW
test    eax, eax
jz      short loc_4229D
```

Listing 3: Qualitative choice: *opening process and not terminate it*

```
push    esi    ; hModule
mov     esi, ds:GetProcAddress
mov     ds:dword_436A84, eax
call    esi ; GetProcAddress
push    offsetaPr_seterror
push    ds:hModule
mov     ds:dword_436A80, eax
call    esi ;GetProcAddress
push    offset aPr_geterror
push    ds:hModule
mov     ds:dword_436A90, eax
call    esi ; GetProcAddress
movds:  dword_436A7C, eax
pop     esi
retn    10h
```

Moreover, we find a common sequence of instructions which is repeated among different functions. These sequences are neither because of functionally nor because of the compiler. Listing 4 shows an example of such common instructions. We classify this kind of sequences as general choices. Another examples of general choices is illustrated in Listings 5. We notice that in some functions there is a preference of using files over memory locations, and using command line over config file for configuration, as shown in Listings 6 and 7, respectively. We also observe in some functions, there is a preference in using specific file resources; for instance, using *Bitmap* as shown in listing 8.

Listing 4: General choice: *sequence of instructions*

```
inc     [ebp+var_2]
mov     al, [ebp+var_2]
cmp     al, [edi+1]
jb      loc_430509
```

Listing 5: General choice: *sequence of instructions*

```
        push    offset aChrome_dll
        push    eax
        call    ds:lstrcmpiW
        test    eax, eax
        jnz     short loc_41A766
        movzx   eax, byteptr[esp+14h+arg_0]
        push    eax
        mov     eax, esi
        call    CRT_sub_415C3B
        jmp     short loc_41A768
```

Listing 6: General choice: *preference of using files over memory*

```
        call    MTX_sub_431FEC
        mov     esi, ds:GetFileAttributesExW
        test    al, al
        jz      shortloc_422AC9
        xor     edi, edi
        jmp     short loc_422ABD
```

Listing 7: General choice: *preference of using command line over config file*

```
        mov      [ebp+var_1], bl
        call    ds:SetErrorMode
        lea     eax,[ebp+pNumArgs]
        push    eax             ; pNumArgs
        call    ds:GetCommandLineW
        push    eax             ; lpCmdLine
        call    ds:CommandLineToArgvW
        xor     esi, esi
        cmp     eax, esi
        jz      loc_422D33
        xor     edx, edx
        cmp    [ebp+pNumArgs], esi
        jle    shortloc_422CF2
```

Listing 8: General habit: *preference in specific file resources*

```
          call    edi ; GetDeviceCaps
          push    0Ah
          push    [esp+704h+hdc]  ; hdc
          mov     [esi+2Ch], eax
          call    edi ; GetDeviceCaps
          mov     ecx,[esi+2Ch]
          mov     edi, ds:CreateCompatibleBitmap
          push    eax
          push    ecx ;cx
          push    [esp+708h+hdc]  ; hdc
          mov     [esi+30h], eax
          call    edi ;CreateCompatibleBitmap
          cmp     eax, ebx
          jz      short loc_40F528
```

Regarding to the usage of API calls, we find MSDN-Windows shut-down has been used in different functions as shown in Listing 9. The `Win32 API ExitWindowsEx` makes an RPC call to `CSRSS.EXE`. `CSRSS` synchronously sends a message to all Windows applications.

Listing 9: Structural choice: *using specific API*

```
push    80000000h
push    14h
call    ds:ExitWindowsEx
```

6. Related Work

Source Code Authorship. Most previous studies of authorship analysis for general-purpose software have focused on source code [1, 46, 47]. These techniques are typically based on programming styles (e.g., naming variables and functions, comments, and code organization) and the development environment (e.g., OS, programming language, compiler, and text editor). The features selected by these techniques are highly dependent on the availability of the source code, which is seldom available when dealing with malware binaries.

Binary Code Authorship. 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 [2–4], 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. [4] 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 to automatically identify the authorship of software binaries is proposed by Alrabaee et al. [2]. 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. [3]. The authors extract syntactical features present in source code from decompiled executable binaries. Most recently, Meng et al. [5] 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 13 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".

Table 13

Comparing features, methods, applicability to other compiler binaries, resistance to evading techniques of different existing solutions with. (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. The symbol (✓) indicates that the proposal solution offers the corresponding feature. (○): Not affected by the code transformation method. (●): Affected by the code transformation method. (◐): Partially affected by the code transformation method. (SA) stands for single author. (MA) stands for multiple author

Effort	**Features**				**Compiler**				**CT**				**Binaries**		**Application**
	Syntax	Semantic	Structural	Statistical	VS	GCC	Clang	ICC	DCI	IR	IRO	RT	ELF	PE	
OBA2 [2]	✗	✓	✗	✗	✓	✗	✗	✗	◐	◐	○	●	✗	✓	SA
Caliskan [3]	✗	✓	✓	✓	✗	✓	✗	✗	○	○	○	●	✓	✗	SA
Rosenblum [4]	✓	✓	✓	✗	✗	✓	✗	✗	●	●	●	●	✓	✗	SA
Meng [5]	✗	✓	✓	✓	✗	✓	✗	✗	◐	◐	◐	●	✗	✓	MA
BINAUTHOR	✓	✓	✓	✓	✓	✓	✓	✓	○	○	○	○	✓	✓	SA/MA

Malware Authorship Attribution. Most existing work on malware authorship attribution relies on manual analysis. In 2013, a technical report published by FireEye [48] discovered that malware binaries share the same digital infrastructure and code; for instance, the use of certificates, executable resources, and development tools. More recently, the team at Citizen Lab[4] 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 [6] presented a novel approach to creating credible links between binaries originating from the same group of authors [6]. 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.

7. Limitations and Concluding remarks

Our work has a few important limitations. First, we assume that the binary code under analysis is unpacked. While this assumption may be reasonable for many general-purpose software, it implies the need for a pre-processing step involving unpacking before applying the method to malware. Second, our tool cannot handle most of the advanced obfuscation techniques such as virtualization and jitting

[4]https://citizenlab.org/

since our system does not deal with bytecode. Third, although we have tested our work on various datasets, our dataset is still limited in terms of scale and scope. Fourth, our work yields lower/ accuracy when dealing with code compiled by ICC or Clang. Fifth, the features used by BINAUTHOR are static analysis-based; as such, BINAUTHOR does not detect habits which require dynamic features. Finally, BINAUTHOR currently supports the x86 ISA; other ISAs, such as ARM, should also be considered.

Our future work aims to extend BINAUTHOR to tackle these limitations. We point out four main avenues for future research in terms of improving our system. First, we suggest investigating more principled and robust cognition models for program understanding and for extending binary choices. Second, we will seek to extend our system to include visualizations of functionality-independent choices rather than presenting numeric results. Third, while we have demonstrated the viability of our system in identifying the author or clustering the functions based on author habits, a more thorough investigation of different domains is necessary, along with an analysis of the impact of training and test set size. Finally, *Privacy threat:* Recent advances in binary authorship attribution can pose a serious threat to the privacy of anonymous coders. There are cases in which contributors to software projects may wish to hide their identities, for example, developers who do not want their employers to know about their independent activities [3]. While we believe that the ability to perform authorship attribution at the binary level has important security applications, we are also concerned about its implications for privacy. In response, we plan to design a tool that permits developers to anonymize themselves. In addition, we will investigate existing methods that address the privacy threat through the analysis of code writing style [49] and study their applicability to binary code. Another interesting future direction will involve developing rules which identify characteristics that make a binary code harder to attribute.

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. We also applied our system to known malware (e.g., `Zeus and Citadel`) as a case study. Our findings indicated that the accuracy of these techniques drops dramatically to approximately 45% at a scale of more than 50 authors. 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% accuracy. In summary, our system demonstrates superior results on more realistic datasets.

References

[1] A. Caliskan-Islam, R. Harang, A. Liu, A. Narayanan, C. Voss, F. Yamaguchi, and R. Greenstadt, "De-anonymizing programmers via code stylometry," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 255–270.

[2] S. Alrabaee, N. Saleem, S. Preda, L. Wang, and M. Debbabi, "Oba2: An onion approach to binary code authorship attribution," *Digital Investigation*, vol. 11, pp. S94–S103, 2014.

[3] A. Caliskan-Islam, F. Yamaguchi, E. Dauber, R. Harang, K. Rieck, R. Greenstadt, and A. Narayanan, "When coding style survives compilation: De-anonymizing programmers from executable binaries," *arXiv preprint arXiv:1512.08546*, 2015.

[4] N. Rosenblum, X. Zhu, and B. P. Miller, "Who wrote this code? identifying the authors of program binaries," in *Computer Security–ESORICS 2011*. Springer, 2011, pp. 172–189.

[5] X. Meng, B. P. Miller, and K.-S. Jun, "Identifying multiple authors in a binary program," in *European Symposium on Research in Computer Security*. Springer, 2017, pp. 286–304.

[6] "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.

[7] "GitHub-Build software better," https://github.com/trending?l=cpp, 2011.

[8] “The Google Code Jam.” http://code.google.com/codejam/, 2008-2015.
[9] “The planet source code. Available from:,” http://www.planet-source-code.com/vb/default.asp?lngWId=3 #ContentWinners, 2015.
[10] “Techniqal report, Resource 207: Kaspersky Lab Research proves that Stuxnet and Flame developers are connected,” http://www.kaspersky.com/about/news/virus/2012/.
[11] “Techniqal report, Mcafee,” 2011, www.mcafee.com/ca/resources/wp-citadel-trojan-summary.pdf.
[12] A. Rahimian, P. Shirani, S. Alrbaee, L. Wang, and M. Debbabi, “Bincomp: A stratified approach to compiler provenance attribution,” *Digital Investigation*, vol. 14, pp. S146–S155, 2015.
[13] Y. David, N. Partush, and E. Yahav, “Similarity of binaries through re-optimization,” in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2017, pp. 79–94.
[14] “The C Language Library, Cplusplus website,” http://www.cplusplus.com/reference/clibrary/, 2011.
[15] B. G. Hunting, “Nation-state malware research, blackhat (2015).”
[16] “IDA pro Fast Library Identification and Recognition Technology,” https://www.hex-rays.com/products/ida/tech/, 2011.
[17] J. T.-L. Wang, Q. Ma, D. Shasha, and C. H. Wu, “New techniques for extracting features from protein sequences,” *IBM Systems Journal*, vol. 40, no. 2, pp. 426–441, 2001.
[18] B. S. Elenbogen and N. Seliya, “Detecting outsourced student programming assignments,” *Journal of Computing Sciences in Colleges*, vol. 23, no. 3, pp. 50–57, 2008.
[19] R. Muth, “Register liveness analysis of executable code,” *Manuscript, Dept. of Computer Science, The University of Arizona, Dec*, 1998.
[20] V. Rajlich, “Software evolution and maintenance,” in *Proceedings of the Future of Software Engineering*. ACM, 2014, pp. 133–144.
[21] D. E. Knuth, “Backus normal form vs. backus naur form,” *Communications of the ACM*, vol. 7, no. 12, pp. 735–736, 1964.
[22] L. Yujian and L. Bo, “A normalized levenshtein distance metric,” *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 29, no. 6, pp. 1091–1095, 2007.
[23] P. C. Mahalanobis, “On the generalized distance in statistics,” *Proceedings of the National Institute of Sciences (Calcutta)*, vol. 2, pp. 49–55, 1936.
[24] T. A. Junttila and P. Kaski, “Engineering an efficient canonical labeling tool for large and sparse graphs.” in *ALENEX*, vol. 7. SIAM, 2007, pp. 135–149.
[25] R. Cilibrasi and P. Vitanyi, “Clustering by compression,” *Information Theory, IEEE Transactions on*, vol. 51, no. 4, pp. 1523–1545, 2005.
[26] “The Scalable Native Graph Database. Available from:,” http://neo4j.com/, 2015.
[27] “The Gephi plugin for nneo4j. Avaiable from:,” https://marketplace.gephi.org/plugin/neo4j-graph-database-support/, 2015.
[28] “Programmer De-anonymization from Binary Executables,” https://github.com/calaylin/bda, 2015.
[29] “The materials supplement for the paper "Who Wrote This Code? Identifying the Authors of Program Binaries",” http://pages.cs.wisc.edu/$\sim$nater/esorics-supp/, 2011.
[30] “Hex-Ray decompiler,” https://www.hex-rays.com/products/decompiler/, 2015.
[31] N. X. Vinh, J. Epps, and J. Bailey, “Information theoretic measures for clusterings comparison: Variants, properties, normalization and correction for chance,” *The Journal of Machine Learning Research*, vol. 11, pp. 2837–2854, 2010.
[32] K. Q. Weinberger and L. K. Saul, “Distance metric learning for large margin nearest neighbor classification,” *The Journal of Machine Learning Research*, vol. 10, pp. 207–244, 2009.
[33] “C++ refactoring tools for visual studio,” http://www.wholetomato.com/, 2016.
[34] “Refactoring tool,” https://www.devexpress.com/Products/CodeRush/.
[35] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, “Obfuscator-LLVM – software protection for the masses,” in *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO’15, Firenze, Italy, May 19th, 2015*, B. Wyseur, Ed. IEEE, 2015, pp. 3–9.
[36] D. Lin and M. Stamp, “Hunting for undetectable metamorphic viruses,” *Journal in computer virology*, vol. 7, no. 3, pp. 201–214, 2011.
[37] “Tigress is a diversifying virtualizer/obfuscator for the C language,” 2016, http://tigress.cs.arizona.edu/.
[38] “PELock is a software security solution designed for protection of any 32 bit Windows applications ,” 2016, https://www.pelock.com/.
[39] “Adventure in Windows debugging and reverse enigineering,” 2016, http://www.nynaeve.net/.
[40] “Advanced Windows software protection system. ,” 2016, http://www.oreans.com/themida.php.
[41] “script modifies GNU assembly files (.s) to confuse linear sweep disassemblers like objdump.” 2016, https://github.com/defuse/gas-obfuscation.
[42] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, “Obfuscator-llvm: software protection for the masses,” in *Proceedings of the 1st International Workshop on Software Protection*. IEEE Press, 2015, pp. 3–9.

[43] "OllyDbg, it is a assembler level analysing debugger," http://www.ollydbg.de/.
[44] "Contagio: malware dump," http://contagiodump.blogspot.ca, may, 2016.
[45] "VirusSign: Malware Research & Data Center, Virus Free," http://www.virussign.com/, may, 2016.
[46] I. Krsul and E. H. Spafford, "Authorship analysis: Identifying the author of a program," *Computers & Security*, vol. 16, no. 3, pp. 233–257, 1997.
[47] E. H. Spafford and S. A. Weeber, "Software forensics: Can we track code to its authors?" *Computers & Security*, vol. 12, no. 6, pp. 585–595, 1993.
[48] N. Moran and J. Bennett, *Supply Chain Analysis: From Quartermaster to Sun-shop*. FireEye Labs, 2013, vol. 11.
[49] M. Brennan, S. Afroz, and R. Greenstadt, "Adversarial stylometry: Circumventing authorship recognition to preserve privacy and anonymity," *ACM Transactions on Information and System Security (TISSEC)*, vol. 15, no. 3, p. 12, 2012.
[50] P. Shirani, L. Wang, and M. Debbabi, "Binshape: Scalable and robust binary library function identification using function shape," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2017, pp. 301–324.
[51] S. Alrabaee, P. Shirani, L. Wang, and M. Debbabi, "Fossil: a resilient and efficient system for identifying foss functions in malware binaries," *ACM Transactions on Privacy and Security (TOPS)*, vol. 21, no. 2, p. 8, 2018.
[52] A. Aiken *et al.*, "Moss: A system for detecting software plagiarism," *University of California–Berkeley. See www. cs. berkeley. edu/aiken/moss. html*, vol. 9, 2005.
[53] S. Alrabaee, P. Shirani, M. Debbabi, and L. Wang, "On the feasibility of malware authorship attribution," in *International Symposium on Foundations and Practice of Security*. Springer, 2016, pp. 256–272.
[54] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," in *Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on*. IEEE, 2006, pp. 459–468.
[55] G. Palmer *et al.*, "A road map for digital forensic research," in *First Digital Forensic Research Workshop, Utica, New York*, 2001, pp. 27–30.
[56] S. Schleimer, D. S. Wilkerson, and A. Aiken, "Winnowing: local algorithms for document fingerprinting," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM, 2003, pp. 76–85.
[57] S. Burrows, A. L. Uitdenbogerd, and A. Turpin, "Application of information retrieval techniques for source code authorship attribution," in *Database Systems for Advanced Applications*. Springer, 2009, pp. 699–713.
[58] R. Kilgour, A. Gray, P. Sallis, and S. MacDonell, "A fuzzy logic approach to computer software source code authorship analysis," 1998.
[59] P. Juola, "Authorship attribution," *Foundations and Trends in information Retrieval*, vol. 1, no. 3, pp. 233–334, 2006.
[60] A. Narayanan, H. Paskov, N. Z. Gong, J. Bethencourt, E. Stefanov, E. C. R. Shin, and D. Song, "On the feasibility of internet-scale author identification," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 300–314.
[61] Z. Zhao, J. Wang, and J. Bai, "Malware detection method based on the control-flow construct feature of software," *Information Security, IET*, vol. 8, no. 1, pp. 18–24, 2014.
[62] R. Chen, L. Hong, C. Lü, and W. Deng, "Author identification of software source code with program dependence graphs," in *Computer Software and Applications Conference Workshops (COMPSACW), 2010 IEEE 34th Annual*. IEEE, 2010, pp. 281–286.
[63] G. Balakrishnan and T. Reps, "Wysinwyx: What you see is not what you execute," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 32, no. 6, p. 23, 2010.
[64] Y. David and E. Yahav, "Tracelet-based code search in executables," in *ACM SIGPLAN Notices*, vol. 49, no. 6. ACM, 2014, pp. 349–360.
[65] "EXEINFO PE," http://exeinfo.atwebpages.com/.
[66] "SEI CERT C++ Coding Standard," https://www.securecoding.cert.org/, 2016.
[67] "The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler," 2011, http://www.amazon.ca/The-IDA-Pro-Book-Disassembler/dp/1593272898.
[68] S.-S. Choi, S.-H. Cha, and C. C. Tappert, "A survey of binary similarity and distance measures," *Journal of Systemics, Cybernetics and Informatics*, vol. 8, no. 1, pp. 43–48, 2010.
[69] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck, "Structural detection of android malware using embedded call graphs," in *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*. ACM, 2013, pp. 45–54.
[70] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Polymorphic worm detection using structural information of executables," in *Recent Advances in Intrusion Detection*. Springer, 2005, pp. 207–226.
[71] S. Cesare, Y. Xiang, and W. Zhou, "Control flow-based malware variantdetection," *Dependable and Secure Computing, IEEE Transactions on*, vol. 11, no. 4, pp. 307–317, 2014.
[72] A. Desnos and G. Gueguen, "Android: From reversing to decompilation," *Proc. of Black Hat Abu Dhabi*, pp. 77–101, 2011.
[73] A. Dasgupta, R. Kumar, and T. Sarlós, "Fast locality-sensitive hashing," in *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2011, pp. 1073–1081.

[74] H. Peng, F. Long, and C. Ding, "Feature selection based on mutual information criteria of max-dependency, max-relevance, and min-redundancy," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 27, no. 8, pp. 1226–1238, 2005.

[75] "Techniqal report," www.seifreed.es/docs/Citadel/Trojan/Report-eng.pdf, 2011.

[76] I. Santos, F. Brezo, J. Nieves, Y. K. Penya, B. Sanz, C. Laorden, and P. G. Bringas, "Idea: Opcode-sequence-based malware detection," in *Engineering Secure Software and Systems*. Springer, 2010, pp. 35–43.

[77] A. Shabtai, R. Moskovitch, C. Feher, S. Dolev, and Y. Elovici, "Detecting unknown malicious code by applying classification techniques on opcode patterns," *Security Informatics*, vol. 1, no. 1, pp. 1–22, 2012.

[78] X. Wang and R. Karri, "Detecting kernel control-flow modifying rootkits," in *Network Science and Cybersecurity*. Springer, 2014, pp. 177–187.

[79] K. Kim and B.-R. Moon, "Malware detection based on dependency graph using hybrid genetic algorithm," in *Proceedings of the 12th annual conference on Genetic and evolutionary computation*. ACM, 2010, pp. 1211–1218.

[80] "Citizen Lab," https://citizenlab.org/, 2015, university of Toronto, Canada.

[81] P. W. Oman and C. R. Cook, "A programming style taxonomy," *Journal of Systems and Software*, vol. 15, no. 3, pp. 287–301, 1991.

[82] S. D. Conte, H. E. Dunsmore, and V. Y. Shen, *Software engineering metrics and models*. Benjamin-Cummings Publishing Co., Inc., 1986.

[83] T. Hoff, "C++ coding standard," *Disponível: http://www. possibility. com/Cpp/CppCodingStandard. html [capturado em 15 fev. 2006]*, 2000.

[84] N. Krislock and H. Wolkowicz, *Euclidean distance matrices and applications*. Springer, 2012.

[85] M. Fowler, *Refactoring: improving the design of existing code*. Pearson Education India, 1999.

[86] "Malware channel: Channel for malware samples to download and analyze," https://twitter.com/MalwareChannel, july, 2016.

[87] "Microsoft Malware Classification Challenge (BIG 2015)," https://www.kaggle.com/c/malware-classification, july, 2016.

[88] "Blinded for submission."

[89] N. Rosenblum, B. P. Miller, and X. Zhu, "Recovering the toolchain provenance of binary code," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 2011, pp. 100–110.

[90] L. Martignoni, M. Christodorescu, and S. Jha, "Omniunpack: Fast, generic, and safe unpacking of malware," in *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*. IEEE, 2007, pp. 431–441.

[91] M. G. Kang, P. Poosankam, and H. Yin, "Renovo: A hidden code extractor for packed executables," in *Proceedings of the 2007 ACM workshop on Recurring malcode*. ACM, 2007, pp. 46–53.

[92] S. Alrabaee, P. Shirani, L. Wang, M. Debbabi, and A. Hanna, "On leveraging coding habits for effective binary authorship attribution," in *European Symposium on Research in Computer Security*. Springer, 2018, pp. 26–47.