CPA: Accurate *C*ross-*P*latform Binary *A*uthorship Characterization Using LDA

Saed Alrabaee, Mourad Debbabi, and Lingyu Wang,

***Abstract*—Binary authorship characterization refers to the process of identifying stylistic characteristics that are related to the author of an anonymous binary code. The aim is to automate the laborious and error-prone reverse engineering task of discovering information related to the author(s) of binary code. This paper presents `CPA`, a novel approach for characterizing the authors of program binaries. Instead of using generic features such as n-grams, *CPA* proposes a set of new features based on collections of various aspects of author style, including author code traits, code structure characteristics, and author expertise in solving coding tasks. It employs the Latent Dirichlet Allocation (LDA) algorithm to generate author style signatures to help identify similar author style characteristics in other binaries. We evaluated `CPA` on large datasets extracted from selected open-source C/C++ projects in GitHub and Google Code Jam events, and it successfully attributed a large number of authors with a significantly higher F_1 score: around 91% when the number of authors was 1,500. In addition, the false positive rate was low, around 1.5%. When the code was subjected to refactoring techniques or code transformation or was processed using different compilers/compilation settings, there was no significant drop in accuracy, demonstrating the robustness of our tool. Finally, in the case of code written by multiple authors, `CPA` was able to identify the authors with a high F_1 score, around 89%.**

***Index Terms*—Binary Code Analysis, Reverse Engineering, Authorship Characterization, LDA.**

I. Introduction

BINARY authorship characterization refers to the process of identifying stylistic characteristics that are related to the author of an anonymous binary code. The ability to conduct such analyses at the binary level is especially important for security applications because the source code for malware is seldom available. There are two primary difficulties that arise when computerizing binary authorship categorization. Firstly, there are numerous elements that are in the source code but are missing from the binary code, such as variable names. Secondly, analysis of binary codes requires greater time and resources than analysis of their related source codes. Although there have been great endeavors to implement automated source code authorship attribution [1], [2], the methods frequently entail dependence on aspects that have a high probability of being dropped in the strings of bytes that denote binary code following the process of compilation. These include the naming of variables and functions, original control and data flow structures, comments, and the layout of space.

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 author [3]–[5]. These approaches share the following limitations: the chosen features are generally not related to author style but rather to functionality; significantly lower accuracy is observed in the case of multiple authors [6]; the approaches are easily defeated by refactoring techniques or code transformation methods; and the approaches handle only one platform (e.g., x86). Dealing with the binary authorship characterization problem requires novel features that can capture author style characteristics, such as a preference for using specific compilers, keywords, reused packages, implementation frameworks, and binary timestamps.

Key idea. Security practitioners have encountered an increasing need to analyze binaries across multiple platforms, e.g., x86, ARM, or MIPS [7]. A more serious issue is that the rapidly evolving mobile and embedded market makes any binary authorship platform-specific solution ineffective [8]. Thus, choosing the right features is crucial for effective binary authorship characterization since what is unique in a particular binary may be attributed to either the author, the compiler, or functionality, which may result in a high false positive rate. Unfortunately, the aim of existing efforts devoted to binary authorship attribution is to identify the author of a single-platform binary code within a set of repositories of known authors, but such approaches are not applicable when the authors are unseen. An important advantage of `CPA` is that it classifies unseen authors according to their stylistic characteristics. This might help `CPA` create credible links between binaries originating from the same group of authors. To achieve these goals, `CPA` performs the following tasks. First, it employs Zipr [9], a static binary rewriter, to eliminate the effects caused by the choice of compiler and the compilation settings. We chose Zipr since, in the literature, it is widely regarded as a practical tool [10]–[12].

Moreover, it has been shown that Zipr can be applied to malware binaries. It has also been used in the DARPA Cyber Grand Challenge (CGC), the first fully automated cyber hacking contest [13]. In this work, it enables both space- and time-efficient analysis after the optimization effects introduced by the compiler are eliminated.

Second, it uses a set of features that labels compiler-related,

S. Alrabaee is an assistant professor at information systems and security department in United Arab Emirates University, Al Ain, Abu Dhabi, UAE. He is the corresponding author. e-mail: (salrabaee@uaeu.ac.ae).

M. Debbabi and L. Wang are with the Concordia Institute for Information Systems Engineering (CIISE), Concordia University, H3G 1M8, Montreal, QC, Canada. E-mail: debbabi,wang@ciise.concordia.ca.

Manuscript received July 02, 2019; revised November 19, 2019 and February 05, 2020. Accept February 28, 2020.

library-related, and user-related functions. Third, after labeling, CPA captures code traits from the user-related functions of different aspects, such as implementation details, infrastructure attributes, memory allocation habits, reused specific packages, the use of specific resources, and other traits. To this end, we identify a set of new features that defines authorship style characteristics; these are extracted from the static domain. Extracting features from various aspects helps to avoid excessive reliance on the binary representation of a sample [14], making it difficult for adversaries to fake all features on all levels.

System overview. The starting point for CPA is a set of known authors and samples of their binary code as ground truth files. The binaries are rewritten by Zipr [9] to enable both the space and time efficient transformation of binaries.

Then, the modified set of binaries is passed to a set of filters which label the user-related functions. These are passed to Angr [15], which lifts them to the intermediate language (IR), avoiding the effects of using different platforms. We chose Angr because it incorporates components that conduct data dependency analysis and value set analysis (VSA). These components were used to perform data flow analysis. In addition, it is known in the literature as a powerful, user-friendly framework [15].

After that, the set of features are extracted from IR. More precisely, we define the novel author attribution graph (AAG) based on the annotated control flow graph (ACFG) and the data flow graph (DFG). Merging them may provide clues about an author's style characteristics in implementing a task that reflects his or her expertise. Inspired by existing natural language processing efforts [16]–[19], we propose a new graph which we call the Graph-of-API keywords (GoA); it is a better representation of author preferences and skills in manipulating API calls. All the extracted features are passed to a decision tree, which selects the best set. Then, the features in this set are ranked based on mutual information. Finally, the top ranked features are passed to the Latent Dirichlet Allocation (LDA) algorithm to generate a set of signatures that represents the author style characteristics. The signatures generated are used for clustering/classification: the Jaccard distance is used for classification, as shown in Section IV-A and Section IV-B, and K-means are used for clustering, as shown in Section VIII-B. The rationale for applying LDA to binary authorship characterization signature generation is that it can extract key content from encoded strings in binary code. To minimize the false positive rate and improve efficiency, CPA contains a set of filters that discard compiler-related, library-related, and reused functions, allowing quick identification of user-related functions. This makes it possible to search efficiently for similarities in author style in large code bases.

Contributions. We make the following contributions.

1) To the best of our knowledge, CPA is the first approach that can identify the authors of program binaries across different platforms. Moreover, this work is the first to investigate the effectiveness of LDA in the context of binary authorship characterization.
2) The results produced by CPA have clear semantics and can be easily interpreted as forensic evidence or hints about the style characteristics of an author (e.g., memory allocation habits). In contrast, the generic features employed in existing work [3], [5] (e.g., n-grams) are difficult to interpret and may not always have a clear significance in terms of authorship. This capability of CPA is especially relevant in forensic applications, in which meaningful evidence is usually essential.
3) The experimental results show that CPA can resist different code transformations, such as refactoring methods, simple obfuscation techniques, and altered compilation settings. This demonstrates the breadth of its potential as a practical tool for assisting reverse engineers in a number of security-related tasks.
4) We compared CPA with other state-of-the-art tools by performing experiments in which all the above-mentioned transformation techniques were applied to the same real-world data, taken from the Google Code Jam programming competition. It was found that CPA achieves meaningful results with superior precision.

II. Design Overview

The architecture of CPA is illustrated in Figure 1. As can be seen, CPA encompasses two phases: offline and online search. As shown in the upper part of the figure, the offline phase contains (1) a disassembling component which employs two tools, Zipr [9] that eliminates the differences attributed to the choice of the compiler and the compilation settings. Then, the modified binary is passed to Angr [15], which lifts it to the intermediate language (IR), avoiding the effects of using different platforms. The offline phase also performs (2) feature extraction, encompassing static features, and (3) feature selection, ranking features to extract the relevant elements of the author style characteristics, as well as best feature selection. In addition, the offline phase carries out (4) signature generation, using LDA to generate the signatures and index them efficiently in a repository. The bottom half of Figure 1 illustrates the online phase, which comprises (A) Disassembling; (B) Filtration, which filters out compiler-related functions and library-related functions to reveal an author's preference for a specific compiler; (C) Feature Extraction; and (D) Detection Components.

A. Filtration Process

An important initial step in most reverse engineering tasks is to distinguish between user functions and library/compiler functions [3]; this saves considerable time and helps shift the focus to more relevant functions. As shown in Figure 1 (component B), the filtration process consists of three steps. First, FLIRT [20] (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) [20]; this set is added to the existing signatures of the IDA FLIRT list. CPA employs FLAIR to filter out FOSS functions. In the last step, compiler function filtration is performed. The original hypothesis is that an amalgamation of static signatures generated in the training phase, such as

opcode frequencies, can be utilized to identify compiler/helper functions. Several programmers with various operational aspects were analyzed. Their functionalities stretched from a basic "Hello World!" program to those capable of carrying out complicated tasks. A total of 120 functions were gathered as compiler/helper functions by merging the above mentioned functions with manual analysis [21].

The opcode frequencies are extracted from these functions, and the mean and variance of each opcode are calculated. After passing FLIRT, each disassembled program P consists of n functions $\{f_1, \cdots, f_n\}$. Each function f_k is represented as m pairs of opcodes, where m is the number of distinct opcodes in function f_k. Each opcode $o_i \in O$ has a pair of values (μ_i, ν_i), which represent the mean and variance values for that opcode.

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

TABLE I: An example of computing (μ_i, ν_i).

opcode	push	mov	lea	pop
Count	19	31	13	7
μ	(19/19+31+13+7) = 0.271	0.442	0.186	0.1
ν	0.563	45.563	5.063	27.563

Each opcode in the target function is compared with the same opcode in all compiler functions in the training set to obtain the measured distance $D_{i,j}$, where i represents the training function and j represents the target function. If $D_{i,j}$ is less than a predefined threshold value $\alpha = 0.005$, the opcode is considered to be a match. A function is labeled *compiler-related* if the proportion of matched opcodes is greater than a predefined threshold value, found experimentally to be $\gamma = 0.75$; otherwise, the target function is labeled *user-related*. Similarity and dissimilarity measurements are performed based on distance calculations according to the following equation [22]:

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

where $\overline{\mu}_j$ and $\overline{\nu}_j$ represent the opcode mean and variance of the target function, respectively. This similarity finds functions that have types of opcodes in common. For example, *compiler-related* functions do not contain logical opcodes. Lastly, α, a score is assigned to each distance that is under a predetermined limit.

III. Feature Engineering

This section discusses the extracted features in detail. As shown in Figure 1 (Component 2/C).

A. The Author Attribution Graph

In this subsection, we describe our novel Author Attribution Graph (AAG), which is formed by merging the Annotated Control Flow Graph (ACFG) and the Data Flow Graph (DFG). Control Flow Graph (CFG): It represents the order in which code instructions are executed and also the branch instructions that must be satisfied during a particular execution path. The nodes in the CFG represent statements and predicates, while the edges represent the transfer of control, which can be true, false, or unconditional. It is worthy to mention that constructing a CFG is a challenging process. When the static algorithm is used for this purpose, it should precisely determine the targets of an indirect control flow instruction, with an emphasis on jump tables. Also, it should identify all non-returning functions. A function call to such a function should not have a control flow edge to the next basic block. Failure to overcome any of these challenges will cause real control flow to be missed and bogus control flow to be reported. Inaccuracy in a CFG can confuse analysts and degrade precision. Data Flow Graph (DFG): It is a directed acyclic graph that represents

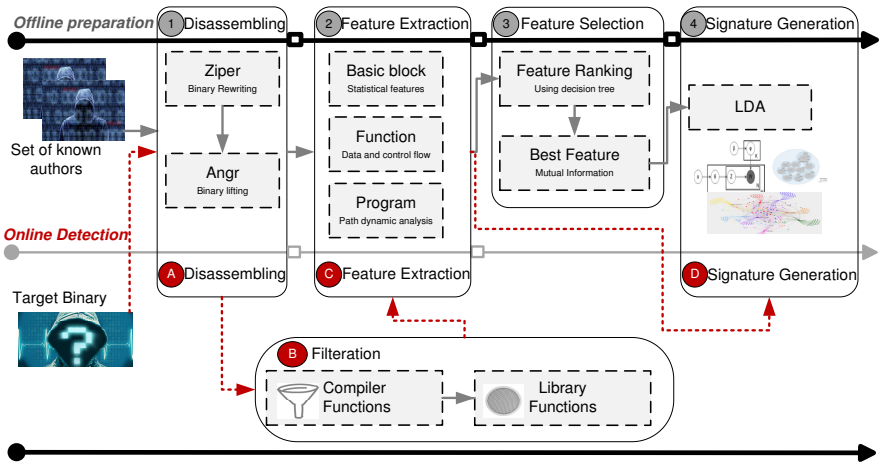

Fig. 1: CPA Architecture.

the data dependencies among a set of operations. A vertex represents an opcode operation. An edge from vertex v_1 to vertex v_2 means that v_1 (or the value produced by v_1, if v_1 is an operation) is an input operand for operation v_2. The intuition behind this construction is that the ACFG may reveal the characteristics of an author's task implementation, while the DFG reveals how an author manipulates variables and also highlights an author's use of small functions, which are typically inlined by the compiler during optimization.

AAG Construction. To construct the AAG, we first build a semantic control flow graph called the *Annotated Control Flow Graph* (ACFG). This is an abstracted version of the CFG that generalizes specific types of CFG features based on multiple criteria [23]. To construct the ACFG, the system takes the CFG as input and computes the frequencies of the opcodes across instruction groups. Then, it categorizes assembly instructions into different groups, as proposed in [24]. The ACFG explains structural traits, and in order to add data flow dependencies, it is supplemented with the Data Flow Graph (DFG). Doing so enables the use of broad reasoning about the control flow and data dependencies of the program, which subsequently provides an insight into how the author uses the program variables, and how correlations between them are created.

Let R_k/W_k denote registers/memory that instruction I_k reads or writes, and A_k denote the instruction address of I_k. We define the following possible dependencies: I_1 writes something that will be read by I_2; I_1 reads something before I_2 overwrites it; and I_1 and I_2 both write the same variable. It is worth mentioning that these dependencies hold true if and only if I_1 and I_2 are both in the same basic block; I_1 and I_2 are in basic blocks BB_1 and BB_2, respectively; I_1 can be executed last in block BB_1; I_2 can be executed first in block BB_2; and BB_2 is a successor of BB_1 in the control flow graph.

Through highlighting the various elements of the fundamental authorship traits, the ACFG and DFG provide corresponding perspectives of a particular tasks' operation. Both were merged in a joint data structure so as to enable greater efficiency in graph matching between diverse binary codes. This helps to detect similarities in authorship traits, and the outcome is the Author Attribution Graph (AAG) [25].

Definition 1. The Author Attribution Graph $G = (N,V,\zeta,\gamma,\vartheta,\lambda,\omega)$ is a directed attributed graph, where N is a set of nodes, $V \subseteq (N \times N)$ is a set of edges, and ζ is an edge labeling function which assigns a label to each edge: $\zeta : V \longrightarrow \gamma$, where γ is a set of labels. Finally, ϑ is a data dependency function which labels each node $n \epsilon N$ based on the function λ, which encapsulates the data dependency rules.

AAG Representation. As an illustration, we use one simple function to demonstrate the steps involved in constructing and representing the AAG. As explained in Section I, first we construct the ACFG, as shown in Figure 2 (1 and 2). The standard control flow graph is shown in (1). As illustrated in (2), we convert each basic block in accordance with our previous work [26]. Consider the last basic block in (1), which contains the two instructions `pop esi` and `retn`. We group these operators to STK and MSC, and the esi operand to REG.

To color the nodes, we perform the steps below:

1) The instructions in each basic block should be classified according to Table II.

TABLE II: Instruction categories.

Category	Description	Example
DTR	Data transfer instructions	mov
TST	Compare and test instructions	cmp
ATH	Arithmetic	add
LGC	Logic	or
CAL	Call	call
STK	Stack	push
MSC	Miscellaneous	float instructions, return
REG	Registers	esp
MEM	Memory	[esp+18h]
CONST	Constants (characters and integers)	5

2) For each basic block, we check the presence of the proposed groups in Table II. If the group is present, then the corresponding variable will be set to 1; otherwise it will be set to 0. For instance, the last basic block in Figure 2 contains the following instructions: 1. pop esi 2. retn. Here, we have $STK = 1$, $REG = 1$, and $MSC = 1$, while the other groups are set to 0.

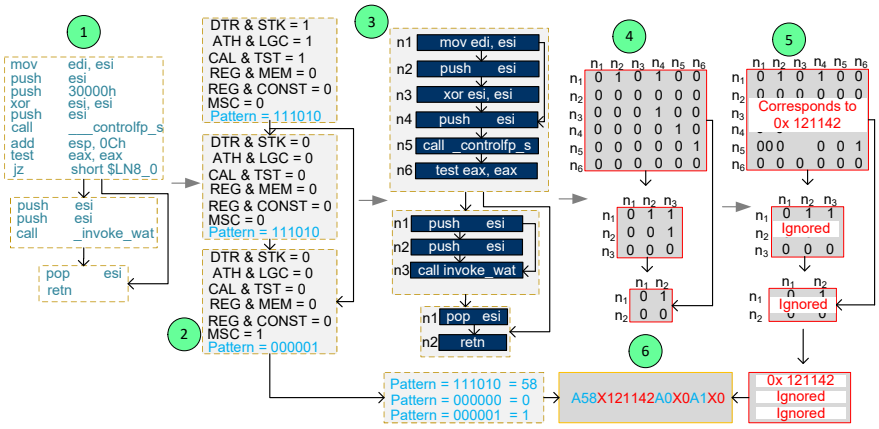

Fig. 2: AAG construction and representation.

3) Each basic block is assigned a six-bit value ($DTR \wedge STK$; $ATH \wedge LGC$; $CAL \wedge TST$; $REG \wedge MEM$; $REG \wedge CONST$; MSC), which is represented in the AAG by the equivalent decimal value, as shown in Figure 2 (6). For instance, for the last basic block, $DTR \wedge STK = 0 \wedge 1 = 0$, while $MSC = 1$. Therefore, the label is 000001.

In component (3) in Figure 2, we apply the aforementioned control/data dependency rules and then convert each basic block in the data flow graph into an ordered adjacency matrix (4). The matrix is then transformed into a unique and canonical representation using the Bliss open source toolkit [27]. A canonical graph labelling is assigned, and the adjacency matrix of the resulting graph is stored as a string; this string represents a feature of the author's style.

B. The Graph of API-Keywords

We adapt the algorithm proposed by [19], which constructs a graph of words and then decomposes it into graph trusses. Here, we list all 84 $C/C++$ keywords as well as the API calls. Using both categories, we construct a graph called the Graph of API-Keywords (GoA) to capture knowledge characteristics, since the author may prefer specific API calls or keywords.

GoA Construction. First, we index the set of $C/C++$ keywords, the list of Windows API [28] calls, and the list of Linux API system calls [29]. The C/C++ keywords can be tracked through the binary strings representing them, so they can be captured by tracking strings that are annotated as call or mov instructions. For instance, using fflush will cause the string "_imp_fflush" to appear often in the developer's output binary.

Intuitively, we try to construct clues that may disclose the characteristics that are related to author knowledge. For instance, the list of Windows API calls and Linux API system calls is large, which will discourage an author from using it, especially if the programmer has no knowledge of them. The GoA encodes assembly instructions as an undirected graph in which nodes are keywords and API calls in the program. If two keywords co-occur within a window of predefined size w, then an edge between two nodes is passed through the whole program. Furthermore, we assign integer weights matching co-occurrence counts for edges. In this case, we apply a statistical approach based on the Distributional Hypothesis [30], whose premise is that the relationship between words can be determined by the frequency with which they share local contexts of occurrence. Once the graph is constructed, we convert it into a k-degeneracy graph, an undirected graph in which every subgraph has a vertex of degree at most k. This concept was introduced by [18] along with the k-core decomposition technique. In this paper, we add the k-truss algorithm [16]. Specifically, the k-degeneracy graph is the maximal connected subgraph of graph H in which every node n has degree at least k [18]. The degree of n is equal to the number of its neighbors when the edges are not weighted. In the case of weighted edges, the degree of n is the sum of the weights of its incident edges [19]. We then extend the k-core to the k-truss, a triangle-based extension that was proposed by [16], and we follow this method as adapted by [19]. The k-truss subgraph is the largest subgraph in which every edge belongs to at least $k-2$ triangles, and every edge in the k-truss links two nodes that have at least $k-2$ common neighbors.

Example. First we extract the API keywords as listed in [31]. Then we connect them if they have a data flow relationship, as mentioned in Section III-A. Once we have constructed the GoA, we compute the graph classes by using the algorithm introduced in [32], in which the k-class of a graph G, denoted by Φ_k, is defined as $\{e : e \in E_G, \phi(e) = k\}$, where $\phi(e)$ is the truss number of e in H.

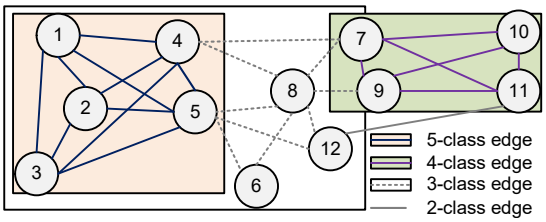

Fig. 3: A graph for the API keywords relationship in a program.

In the graph in Figure 3, the different types of edges indicate the different k-classes, where $2 \leq k \leq 5$. The 2-class Φ_2 has a single edge (i, k), i.e, Φ_2 = {(11, 12)} since (11, 12) is the only edge in the graph with 0 triangles. The 3-class Φ_3 consists of 9 edges, given by Φ_3 = {(5, 8), (5, 12), (5, 6), (4, 7), (4, 8), (7, 8), (8, 9), (8, 12), (8, 6)}. The 4-class Φ_4 consists of 6 edges, given by Φ_4 = {(7, 8), (7, 11), (7, 10), (9, 11), (9, 10), (11, 10)}. The 5-class Φ_5 consists of 10 edges, given by Φ_5 = {(1, 2), (1, 3), (1, 5), (1, 4), (2, 3), (2, 5), (2, 4), (3, 5), (3, 4), (5, 4)}. We have k_{max} = 5.

From the k-classes we obtain the k-trusses as follows. The 2-truss is simply the graph H itself. The 3-truss is the subgraph of H formed by the edge set $(\Phi_3 \cup \Phi_4 \cup \Phi_5)$. The 4-truss is the subgraph formed by $(\Phi_4 \cup \Phi_5)$, and the 5-truss is the subgraph formed by Φ_5. The k-trusses express the hierarchical structures of H at different levels of granularity, as depicted by the shading with different colors in Figure 3. For each API keyword, Table III shows the values of k for which that keyword is involved in the k-truss.

TABLE III: K-truss for API keywords.

API keyword ID	k-truss	API keyword ID	k-truss
1	{5}	7	{3,4}
2	{5}	8	{3,4}
3	{5}	9	{3,4}
4	{3,5}	10	{4}
5	{3,5}	11	{2,4}
6	{3,5}	12	{2,3}

C. Feature Processing

After extracting all the aforementioned features, we will end up with a number of features among which some might be the most relevant ones - those that appear more frequently and are most segregative in program. Therefore, a feature selection process (component 3 in Figure 1) is conducted to reduce the number of features as well as to find the best ones. Our feature

selection phase contains two major steps: best feature selection and feature ranking, which are described as follows.

Best Feature Selection. Our aim is to build a classification system which can efficiently segregate an author from another author. As a result, we employ the decision tree classifier on the extracted features. Each set of known author programs is passed through the classifier and accordingly the best features are chosen for that particular set. This automated task is performed on all training known author programs to create a list of best features for each.

Feature Ranking. Second, in the feature ranking phase, it was assumed that there exist a known authors and a set of programs written by each of those authors. The feature ranking algorithm was then designed to order the features by their correlation with authorship. The goal of the feature ranking algorithm is to rank such common features lower such that they will not be selected later on. The feature ranking algorithm was based on the concept of mutual information between the feature and known authorship.

$$Mutual\ Information = \Sigma\Sigma p(x,y) log_2 \frac{(p(x,y))}{p(x)p(y)} \quad (1)$$

where p(x,y) is the joint probability distribution function of a feature and an author, and p(x) and p(y) are the marginal probability distribution functions of a feature and an author respectively. For example, if feature `x` and author `y` are independent, then knowing `x` does not give any information about `y`; their mutual information is zero. To calculate the marginal probability of a specific feature, the algorithm calculates the number of its occurrences and the total number of features that exist.

IV. Detection Process

Figure 4 illustrates the proposed detection system. We added numerical labels to identify the components.

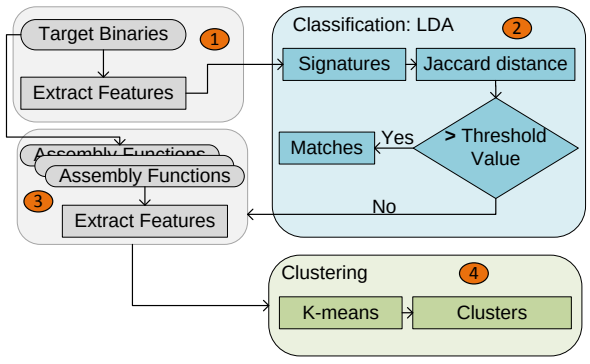

Fig. 4: Detection system.

Given a target binary, the features discussed in Section III are extracted (Component 1). Then, from these features, a set of signatures is generated using LDA (Component 2), which employs the Jaccard distance to measure the similarity between all signatures generated from the target binary and the candidate files according to a predefined threshold value. If the value is greater than the threshold, it counts as a hit; otherwise, it indicates that there is a possibility that the binary is written by a set of authors. In this case, the target binary is disassembled and decomposed into a set of assembly functions (Component 3). After decomposition, k-means are employed to cluster functions written by the same author (Component 4).

A. Latent Dirichlet Allocation (LDA)

The motivation behind applying LDA to binary authorship characteristics signature generation is that binary code and texts resemble each other in many respects. For instance, given that a binary code is composed of meaningful information, including strings, instructions, CFGs, keywords, and API calls, it can be considered to be a document and this information to be words. Similarly, a collection of binary code can be considered to be a corpus. Each binary code can also be viewed as a mixture of underlying topics since it conveys information about its application(s). Consequently, there is a correlation between word and topic in binary code. It is possible to apply LDA to generate semantic signatures of fixed size that can be used for accurate binary authorship characterization classification, and it correctly generates clusters of similar author characteristics; some words (i.e., features) contained in a cluster will represent the latent topics associated with the characterization. Accordingly, our approach relies on automatically extracting such words, which can be proper "signatures" for an author and can be used to detect such characteristics.

However, to date, the applicability of LDA in the context of binary authorship characterization has not been explored. Generally speaking, LDA is considered to be a probabilistic model of a corpus. It is supposed that each document is generated by weighted unseen latent topics [33], which typically refer to semantic themes present in the corpus [33]. The definition of LDA requires the following terminology. A feature f is the basic unit of code. A program i of length N_i (the number of basic blocks) is a collection of N_i features, $i = \{f_1, f_2, \cdots, f_{N_i}\}$, where f_n is the n^{th} feature in the document. A program corpus is a collection of K programs denoted by $P = \{1, 2, \cdots, K\}$.

Using this terminology, LDA assumes that each document is generated by the following process. Choose $\Theta_i \sim$ Dir(α), where $i\epsilon\{1, 2, \cdots, K\}$. Then choose $\vartheta_o \sim$ Dir(β), where $o\epsilon\{1, 2, \cdots, O\}$. For each of the features $f_{i,j}$, where $i\epsilon\{1, 2, \cdots, K\}$ and $j\epsilon\{1, 2, \cdots, N_i\}$, first choose a topic $z_{i,j} \sim$ multinomial(Θ_i). Then choose a feature $f_{i,j} \sim$ multinomial($\vartheta_{z_{i,j}}$). Here, Dir is the Dirichlet distribution, more specifically, the Dirichlet distribution $Dir(a)$ is a family of continuous multivariate probability distributions parameterized by a vector a of positive reals. It can be seen as a multivariate generalization of the Beta distribution. Dirichlet distributions are commonly used as prior distributions in Bayesian statistics and can be used to characterize the random variability of a multinomial distribution. Multinomial is the multinomial distribution with only one trial (also known as the categorical distribution). Suppose that each experiment results in one of m possible outcomes with probabilities $p_1, \cdots, p_m$. The

Multinomial distribution models the distribution of the histogram vector, which indicates how many times each outcome is observed over N trials. Θ_i is the Dirichlet prior on the per-document topic distributions with the parameter α; $\vartheta_{z_{i,j}}$ is the Dirichlet prior on the per-topic word distribution with the parameter β; and $z_{i,j}\epsilon\{1, 2, \cdots, O\}$ is the topic label of feature $f_{i,j}$.

The main goal of LDA is to automatically discover topics from a collection of documents. To achieve this, an LDA inferential problem needs to be solved based on the prior assumption that the corpus was constructed by the LDA generative process. Solving the LDA inferential problem yields two (inferred) probability distribution functions, Θ_i and ϑ_o, which are assumed to generate the (observable) sets of features $f_{i,j}$. Formally, given a corpus P, the key objective of the inferential problem is to compute the posterior distribution of the hidden variables $p(\Theta, \vartheta|P, \alpha, \beta) = {}^{p(\Theta, \vartheta, P|\alpha, \beta)}/_{p(P|\alpha, \beta)}$. This is component 4/D in Figure 1. For more details about LDA functions and definitions, we refer the reader to the original work concerning the LDA algorithm [34].

Example. For the sake of simplicity, we introduce an example (Figure 5) to show how LDA works and to illustrate the LDA configuration parameters. As explained, in LDA, each document in a collection is assumed to be generated from a set of latent topics by a Dirichlet prior distribution. As shown in Figure 5, to each topic is associated a set of probability values that follows a Dirichlet prior distribution. Each word is also associated with topics with a probability value that follows another Dirichlet prior distribution.

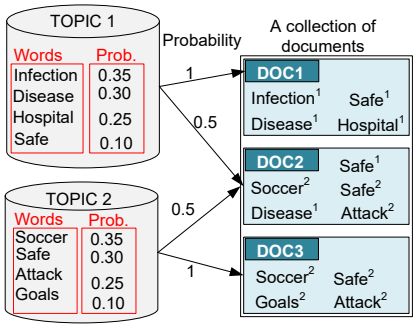

Fig. 5: LDA Example.

Figure 5 is a schematic of the probabilistic generative process for three documents: $DOC1$ draws from Topic 1 with probability 1; $DOC2$ draws from Topic 1 with probability 0.5 and from Topic 2 with probability 0.5; and $DOC3$ draws from Topic 2 with probability 1. As shown in the figure, the distributions for the two topics are represented by $(\{1, 0\}, \{0.5, 0.5\}, \{0, 1\})$. To form topics, LDA extracts clusters of co-occurring words.

For each binary file, we model the Author Attribution Graph and the Graph of API Keywords as multiple sequences, train a representation model, and produce a set of vectors, which are considered topics. Then, LDA checks the topics and generates the signatures. Analysis of the signatures showed that the following features rank high as unique topics: AAG with 3 nodes; AAG with 4 nodes; AAG with 7 nodes; AAG with 8 nodes; API with k-truss = 2; API with k-truss = 3; API with k-truss = 4; API with k-truss = 4; and API with k-truss = 5.

B. Jaccard Distance

To measure the similarity between the signatures generated from LDA, we employ the Jaccard distance used by [35].

C. Clustering Similar Functions

We use a standard clustering algorithm (k-means) [2] to group functions with similar author characteristics attributes $(v_{n_1}, \ldots, v_{n_z})$ into k clusters $S = S_1, \ldots, S_k$ and $(k \leq |F_{P_1}| + |F_{P_2}|)$ to minimize the intra-cluster sum of squares.

V. Implementation

In this section, we describe the implementation setup including the choice of dataset. We then provide our evaluation setup.

A. Implementation Environment

We follow the same environment by our previous work [36]. The feature extraction is implemented by individual python scripts for modularity purposes, which altogether form our analytical system. To perform complex graph operations such as k-graph (ACFG) extraction, we use the Neo4j [37] graph database. We use the MongoDB database to store extracted features according to its efficiency and scalability [36].

B. Datasets

Our dataset encompasses several files, as follows: i) GitHub and ii) Google Code Jam. We provide the statistics about our dataset in Table IV. We compile the presented dataset with different compilers and compilation settings. We use GNU Compiler Collection's gcc, g++, Clang, ICC, as well as Microsoft Visual Studio (VS) 2010 and 2012, with different optimization levels. It is worthy to mention that GitHub dataset might contain script projects (e.g., Python, PHP, etc.). Such projects excluded.

TABLE IV: Statistics about the binaries used.

Source	# of authors	# of programs	# of functions
GitHub	500	12,910	4,900,000
Google Jam	1,000	55,550	10,965,120
Total	1,500	68,460	15,865,120

From the GitHub dataset, we selected 300 authors with 28 programs each; 100 authors with 24 programs each; 99 authors with 21 programs each; and one author with 31 programs. From the Google dataset, we selected 400 authors with 63 programs each; 300 authors with 57 programs each; 290 authors with 44 programs each; and 10 authors with 49 programs each. None of these programs was written by multiple authors. Moreover, none of the authors has programs in both datasets.

C. Evaluation Settings

The evaluation of CPA system is performed on the datasets introduced in the aforementioned section. The dataset is randomly divided into 10 sets, where one set is kept as a testing set, and the remaining sets are used as training sets. The process is then repeated 100 times. To evaluate CPA and to compare it with existing methods, precision P and recall R measures are applied. Furthermore, since the application domain targeted by CPA is much more sensitive to false positives than false negatives, we employ an F1-measure as follows:

$$F_1 = 2\,\frac{P\,.\,R}{P+R} \tag{2}$$

The evaluation results are presented in terms of the following metrics:

- *True positives* (TP): This measures the number of binaries whose author was successfully identified by the system.
- *False negatives* (FN): This measures the number of binaries for which the system reported that it was unable to assign an author even though the correct author is in the database.
- *False positives* (FP): This measures the number of binaries that the system incorrectly assigned to an author.
- *Precision* (P): This is the percentage of positive predictions, i.e., the percentage of authors correctly identified: $P = \frac{TP}{TP+FP}$
- *Recall* (R): This is the percentage of the samples for which the author was correctly identified: $R = \frac{TP}{TP+FN}$

VI. Performance Comparisons

We conducted two experiments to evaluate the accuracy of author attribution for program binaries.

A. Non Cross-platform Attribution Accuracy

We compared CPA with the existing authorship attribution methods [3]–[5] as shown in Figure 6. We evaluated the authorship classification technique presented by Rosenblum et al. [5], whose source code is available at [38], although the dataset is not. The source code of the proposed technique as well as the dataset by Caliskan-Islam et al. [4] is available at [39]. The authors of [4] performed the largest-scale evaluation of binary authorship attribution reported in the literature. It consists of *600* authors, while [5] conducted a large-scale evaluation of *190* authors. Our previous work [3] performed a small-scale evaluation of 5 authors. Since all the existing efforts are x86-based, our comparison is also x86-based.

We compared our results with those obtained by applying the above approaches to the datasets mentioned in Table IV. We report the findings in Figure 6, which shows the relationship between the precision and the number of authors present in all the datasets: the precision decreases as the size of the author population increases. It can be seen that CPA achieves better accuracy in determining the author of binaries based on authorship style features in the case of the GitHub dataset and the mixed dataset. Taking all four approaches into consideration, the highest accuracy of authorship attribution is close to *100%* on the GitHub dataset with fewer than *200* authors, while the lowest accuracy is *2%* when *1500* authors are involved. On the Google dataset, CPA identifies the authors with an average accuracy of *92%,* and on the GitHub dataset, the average accuracy is *94%*. The main reason for the higher accuracy is that there are no restrictions on the authors of projects in GitHub. In addition, the advanced programmers of such projects usually design their own class or template, which makes it easier for the characterization process to identify the authors. The lower accuracy obtained by CPA is approximately *87%* on a mixed dataset with *1500* authors. As shown in Figure 6, the approach of Caliskan-Islam et al. [4] achieves the highest accuracy of all. We believe that the reason for its superior performance on Google Jam Code is that this dataset is simple and can be easily decompiled into source code.

We found that the accuracy of existing methods [4], [5] strongly depends on the application domain. For example, in Figure 6, a reasonable level of accuracy is observed for the Google data set: the average accuracy is *93%*, while, on the GitHub dataset, the average accuracy is *57%*. It should also be noted that existing methods use disassemblers and decompilers

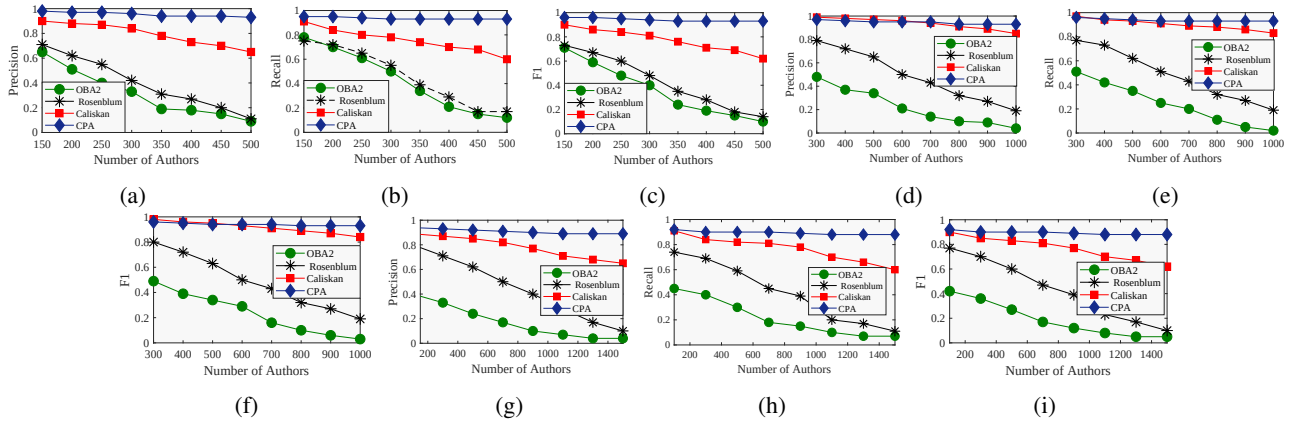

Fig. 6: Comparison of non cross-platform attribution using the precision, recall, and F_1 metrics for the following datasets: GitHub (a, b, c); Google jam (d, e, f); Mixed (g, h, i).

to extract features from binaries. According to our previous analysis [36], Caliskan-Islam et al. [4] used a decompiler to translate the program into C-like pseudo code via Hex-Ray [40]. They passed the code to a fuzzy parser for C, thus obtaining an abstract syntax tree from which features can be extracted. The C-like pseudo code is different from the original code because of the inherent limitations of Hex-Ray [40] and also because the variables, branches, and keywords are different. For instance, 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 two. Once the binary is decompiled, the function consists of the following keywords: (`1-do, 1-else/if, 2-goto, 2-while, 4-if`), and the number of variables is four. This will evidently generate misleading features, thus increasing the rate of false positives.

B. Cross-platform Attribution Accuracy

We evaluated the performance of CPA in the cross-platform setting using the datasets mentioned in Section V-B as shown in Figure 7. The statistics for the compiled dataset are shown in Table V.

TABLE V: Statistics about the binaries used.

Source	# of authors	# of programs
GitHub	500	7,500
Google Jam	1,000	15,000
Total	1,500	22,500

For each author, we selected 5 programs and compiled each on ARM, MIPS, and x86, producing 15 programs for each author. We trained CPA with ARM & MIPS binaries and tested it against x86; we trained CPA with ARM & x86 and tested it with MIPS; and we trained CPA with x86 & MIPS and tested it with ARM. The precision, recall, and F_1 results are reported in Figure 7.

Our experiments showed that CPA achieves similar results across multiple architectures with high F_1 scores (Figure 6); OBA2 produces the lowest F_1 scores.

Feature Extraction Level. Typically, existing methods extract features from only one code level: the instruction, function, or program level. A significant advantage of CPA is that it extracts features from all three levels, making it possible to discover authorship characteristics by considering different aspects. In addition, we leverage concepts from NLP to extract the semantics of structural features that are preserved to a great extent in intermediate representations.

Project Size. The precision of Caliskan is affected by the size of the function or of the project. For example, a comparison of the precision achieved by Calsikan for Google projects (0.89 and 0.83, respectively) with that for GitHub (0.71 and 0.67, respectively) demonstrates that feature extraction depends on the sizes of functions, and that the use of extracted features alone cannot reveal the semantics of a piece of code.

Semantic Features. When the instructions are left in the IR, the semantics, rather than the syntax, will be preserved. With this in mind, our new features are chosen to be semantics-based, so our system can greatly reduce the manual effort required to extract the IR semantics.

Complex Instruction Sets. We observed that, in ARM binaries, there are 120 assembly mnemonics having multiple possible IRs. This affected the work of Rosenblum et al. significantly since they extracted idioms (i.e., sequences of instructions). As shown in Figure 7, when the target is ARM, Rosenblum et al. achieve a low F_1 score.

Architecture Registers. OBA2 constructs RFG, which is based on a comparison of each instruction (e.g., cmp) and the associated registers. When the assembly instructions are lifted, the ARM and MIPS provide core registers ($R0-R3$ on ARM and $a0-$a3 on MIPS) for passing arguments to subroutines.

Function Inlining. We noted that the percentage of inlined functions in x86 is higher than for ARM and MIPS. This affects the syntax features as well as hindering the decompilation process.

VII. Impact of Evading Techniques

We describe the experimental setup in this Section more precisely as follows. We train CPA on the set of binary

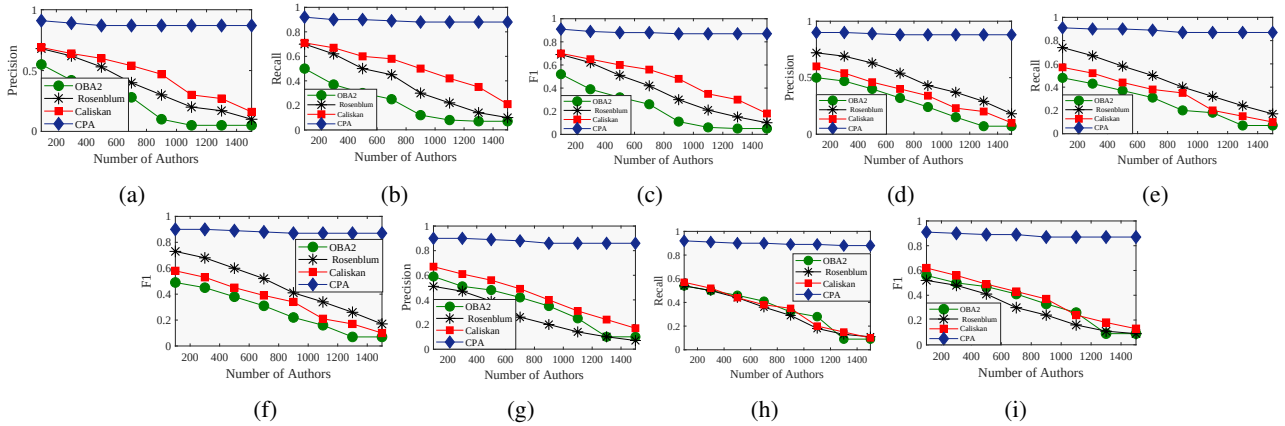

Fig. 7: Comparison of cross-platform attribution using the precision, recall, and F_1 metrics for the following scenarios: (1) Training: ARM & MIPS Target: x86 (a, b, c); (2) Training: ARM & x86 Target: MIPS (d, e, f); (3) Training: x86 & MIPS Target: ARM (g, h, i).

code samples that have been modified. Next, we apply code transformation mechanisms, such as changing the compiler and the compilation settings and also obfuscation, to the same samples and then use them for testing. It was found that different compilers generate binaries with significant differences. For instance, comparing GCC and VS, we observed that, on average, 42% of the bytes of a function are modified and that 9% of the functions are identical; moreover, 63% of the control flow graph is modified, and 40% of the function pairs share similarity. To see the effect of optimization, we compared O0 and O3. On average, 39% of the bytes of a function are modified, and none of the functions are identical, while 79% of the control flow graph is modified, and 21% of the function pairs have similar graph complexity. When investigating advanced code transformation methods such as a bogus control flow graph and flattening the control flow graph, we found that all CFGs are modified, and fewer than 5% of the function pairs have similar graph complexity. Finally, after simple advanced code transformations such as dead code elimination or instruction replacement, we found on average that 9% of the bytes of a function are modified. The datasets are used for experiments on evasion techniques, as described below.

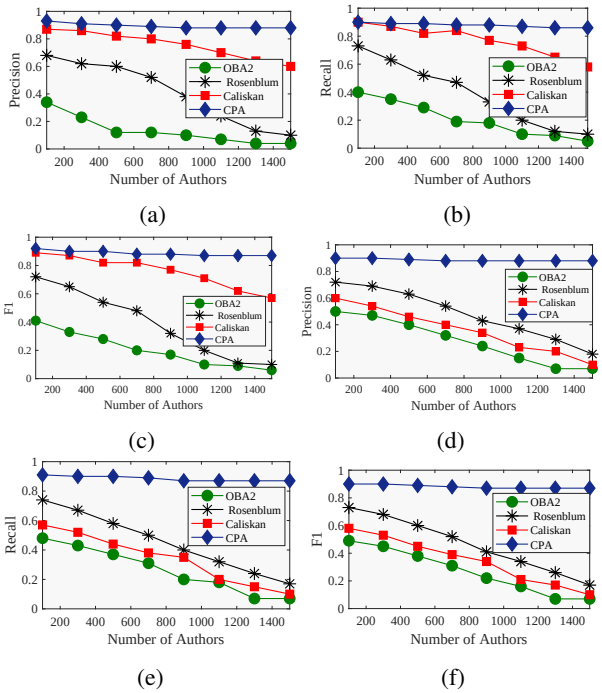

Fig. 8: Comparison of binary authorship attribution for different compilers using the precision, recall, and F_1 metrics for the following scenarios: (1) Testing: GCC, Training: VS, ICC, Clang (a, b, c); (2) Testing: ICC, Training: VS, GCC, Clang (d, e, f).

Authorship Attribution with different compilers. For this experimental setup, we tested the ability of CPA to identify the author from binaries compiled with the same optimization level, but with different compilers. Specifically, we used binaries that were compiled with O2 on the x86 architecture using the following compilers: VS, GCC, Clang, and ICC. This produced four sets of binaries from which we reserved one set for testing and kept the rest for training sets. We repeated this for all sets, and then we reported the best and the worst cases, as shown in Figure 8.

We tested the binary authorship attribution performance of *CPA* against different compilers. We evaluated it based on the datasets listed in Figure 8. We compiled a set of binaries using the GCC compiler 4.8.5 and three other compilers: Clang 5.0, ICC 16, and VS 2017; in this way, we obtained four different sets of binaries. It should be noted that we used a similar optimization setting for all the compilers. Then, we reserved one set as a testing set, while designating the remaining three as training sets. Given four binaries from the same set of binaries compiled with different compilers, we linked their assembly functions using the compiler output debug symbols and generated an authorship fingerprint mapping between functions; this was used as the ground truth data for evaluation only. We searched for a binary from the first binary set in each of the other three images in the database, and then we searched for a binary from the second binary set in each of the other three images in the database, and analogously for the third and fourth binary sets. Finally, we calculated the average precision, recall, and F_1 score. These compilers generate binaries with significant differences. For instance, in the comparison of GCC and VS, we observed that, on average, 32% of the bytes of a function are modified and that 9% of the functions are identical. Moreover, 63% of the control flow graph is modified, and 40% of the function pairs have similar complexity. The comparison showed that the binaries produced by the GCC and VS compilers are the closest. In the comparison of the GCC and ICC compilers, the modifications are much more significant: on average, 62% of the bytes of a function are modified, and none of the functions are identical, while 74% of the control flow graph is modified, and 28% of the function pairs share similar graph complexity. The GCC and ICC compilers produced the greatest differences. Figure 8 presents the results for two scenarios: training (ICC, VS, and Clang) and testing (GCC); training (GCC, VS, and Clang) and testing (ICC). Due to the large number of cases, we list the results for these two only to illustrate the best- and the worst-case scenario. The results for the other cases lie between the two and follow the same ranking.

The figure shows that *CPA* is robust against the heavy syntax modifications and intensive inlining introduced by compilers. Even in the worst-case scenario, the learned representation (LDA) can still correctly attribute the author more than 84% of the time. On average, it achieves more than 92% precision in attributing authors among various compilers. As can be seen in Figure 8, although the performance of *CPA* decreases in the worst-case scenario (Figures c, d, and e), it is much less sensitive than the other systems; this demonstrates its robustness.

Authorship Attribution with different optimization settings. For this experimental setup, we tested the ability of CPA to identify the author from binaries compiled with the same compiler, but with different compiler optimization levels. Specifically, we used binaries that were compiled with VS/GCC/Clang/ICC on the x86 architecture using optimization levels

O0, O2, and O3. We report the average accuracy metrics for the best and worst cases in Figure 9. More specifically, we compiled a set of binaries using the VS compiler with four optimization settings, which produced four different binaries (images) and then tested every pair of binaries. Given four binaries from the same library but with different optimization levels, we linked their assembly functions using the compiler output debug symbols and generated the ground truth. We attributed the first among the remaining three in the database. Finally, we performed analogous comparisons for each binary, calculating the average precision, recall, and F_1 score. The results are shown in Figure 9.

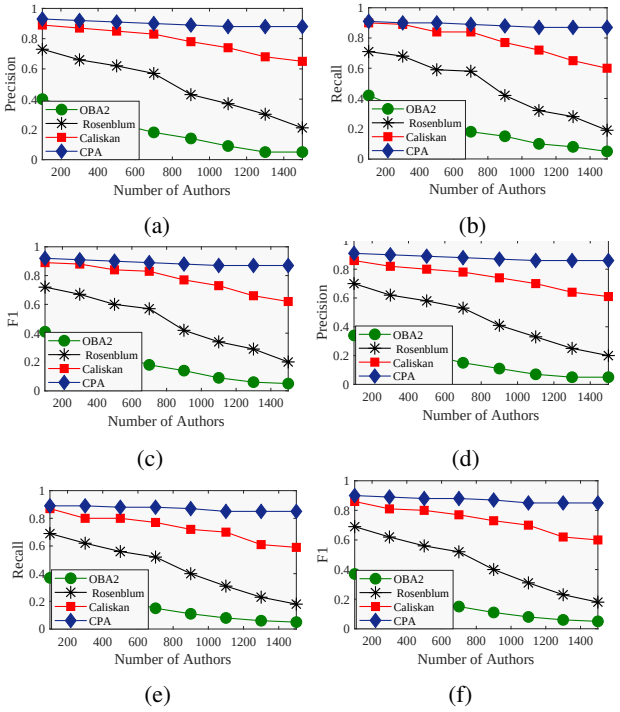

Fig. 9: Comparison of binary authorship attribution for different optimization levels using the precision, recall, and F_1 metrics for the following scenarios: (1) Testing: O2 Training: O3 (a, b, c); (2) Testing: O3 Training: O0 (d, e, f).

It was observed that a higher optimization level contains all the optimization strategies from the lower levels. In addition, we investigated the differences between the binaries produced by these optimizations. We defined the best case as that in which the difference is small, and the worst as that in which the difference is large. The best case is the comparison of O2 and O3. On average, 22% of the bytes of a function are modified, and none of the functions are identical. Moreover, 31% of the control flow graph is modified, and 75% of the function pairs have similar graph complexity. The best case is one in which the optimization strategies used in the two binary functions are similar. The worst case scenario is depicted by the comparison between O0 and O3 due to the significant disparity in the optimization strategies. An average of 39% of a functions' bytes are amended, and no two functions are the same. Furthermore, 79% of the control flow graph is amended, and there is comparable graph complexity in 21% of function pairs. The results of both scenarios are depicted in Figure 9. The decision was made to present the results for the best and worst case scenarios only, due to the overall volume of cases. The results for the other cases lie between the two and follow the same ranking. As shown in Figure 9, *CPA* significantly outperforms existing systems in both the best- and the worst-case scenario.

Authorship Attribution in the presence of code transformation methods. In this experiment, we investigated the impact of code transformation methods introduced by Obfuscator-LLVM (O-LLVM) [41], which is built on the LLVM framework. It functions at the intermediate language level and amends the program logic prior to the production of the binary file. The intricacy of the binary code is consequently increased. There are three diverse techniques, employed in their varying combinations, in the O-LLVM approach, which significantly amends the original assembly code. They are the Bogus Control Flow Graph (BCF), Control Flow Flattening (FLA), and Instruction Substitution (SUB) [42]. It breaks the CFG and the integrity of the basic blocks. By design, most static features are oblivious to the obfuscation. Using the CLANG compiler with O-LLVM, we successfully compiled the dataset for 800 authors and evaluated *CPA* on them.

We found a significant difference between the original version and those obfuscated with BCF and FLA: BCF doubles the number of vertices and of edges, and FLA almost doubles the number of vertices and of edges. With SUB, the number of assembly instructions increases significantly. We used the same set of systems and configurations as in the previous experiment. We compiled a binary without the application of obfuscation techniques, and then we compiled it again with an obfuscation technique to produce an original and an obfuscated binary. We linked their assembly functions using debug symbols and generated a binary mapping between them. After stripping the binaries, we searched the original against the obfuscated version, and then we performed the attribution of the obfuscated version in the original. The average results are reported in Figure 10.

The figure presents the results for O-LLVM. We found that instruction substitution can significantly reduce the performance of systems such as Rosenblum because they use n-grams, and the sequences are broken by SUB when it inserts instructions. However, features such as Graphlet can still determine the authorship of more than 60% of the matches since the graph structure is not heavily modified. Still, *CPA* can achieve more than 96% precision against assembly instruction substitution. Instructions are replaced with equivalent forms, which have semantics similar to those of the original. This information is well captured by *CPA*. When BCF obfuscation is applied, *CPA* can still achieve more than 88% precision, even though the control flow graph looks very different from that of the original. This shows that *CPA* is resilient to inserted junk code and faked basic blocks. The FLA technique destroys all the subgraph structures, which is reflected in the degraded performance of the tools that use subgraph structure features; most of them have a precision value close to zero. However, even in such cases, *CPA* can correctly attribute 84% of the binary matches, demonstrating that it is resilient

to substructure changes and linear layout changes. Moreover, when all the obfuscation techniques are applied, *CPA* can still attribute around 81% of the binary files. Since *CPA* employs LDA, it can correctly pinpoint and identify vital patterns in the presence of noise. Inserted junk basic blocks or noise instructions follow the general syntax of random assembly code, which can be easily predicted based on neighboring instructions.

Authorship Attribution in the presence of advanced obfuscation methods. For this experiment, we use the same binary with and without obfuscation to measure the performance of CPA. Obfuscated malware binaries are based on Windows binaries. In general, three types of binary analysis platforms [43] are used: disassembler, debugger, and virtual machine (VM). In our experiments, from the anti-disassembler category, we chose CFG Obfuscation (push/jmp), Instruction Aliasing, and Encryption Packer. From the Anti-Debug category, we chose Hardware Breakpoint and Instruction Counting. From the Anti-VM category, we chose Anti-VMware IN Instruction. To test our system against these techniques, we selected ten representative obfuscated samples from [44]. Note that one reason for selecting ten samples is that they allow us to verify the robustness of our system. The results showed that our approach made it possible to attribute a binary with 94% precision in the case of Anti-Debug, but the precision dropped to 10% in the case of Anti-Disassembler, and to 82% in the case of Anti-VM. It is worth mentioning that the precision of *CPA* drops significantly in Virtualization or in Just-In-Time (JIT) execution as introduced by Tigress [45]. Through a specialized byte code translation, the virtualization transformation changes a function into an interpreter. It is extremely difficult for matches protected by this approach to be detected with a static approach. In JIT operations, at run time the function is transformed to produce its code. A function call replaces the vast majority of instructions. A trait of the static approach is that it is only capable of recovering minimal variants.

Our results showed that *CPA* attributes binaries with a precision of 39% with Virtualization, and with a precision of 47% with JIT execution. However, if three obfuscation techniques are applied simultaneously, *CPA* can no longer recover any matches.

VIII. Additional Results

A. Impact of Distance Metrics and Clustering Algorithms

In this section, we studied the impact of various distance metrics on the precision of our system. In addition, we investigated two clustering algorithms and reported the false positive rate for the results obtained in the previous section.

Distance Metrics Figure 11 shows the impact of using various distance metrics on the precision of our system. The Jaccard distance outperforms the Cosine and Euclidean distances. For instance, when there are 250 authors in our repository, the precision of detecting a target author is 0.97. The Cosine and Euclidean distances achieve precisions of 0.91 and 0.88, respectively. Thus we chose the Jaccard distance.

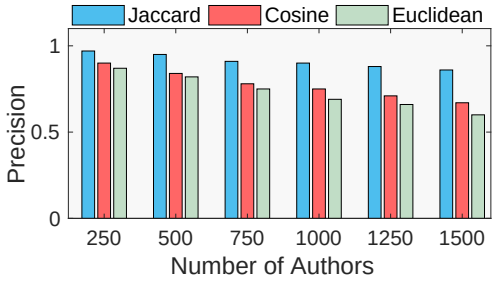

Fig. 11: Distance metrics comparison.

Clustering Algorithms Next, we studied the effect of changing the clustering algorithm, comparing the false positive rate for the clustering algorithm used (K-means) and that for the DBSCAN algorithm [46]. We investigated the false

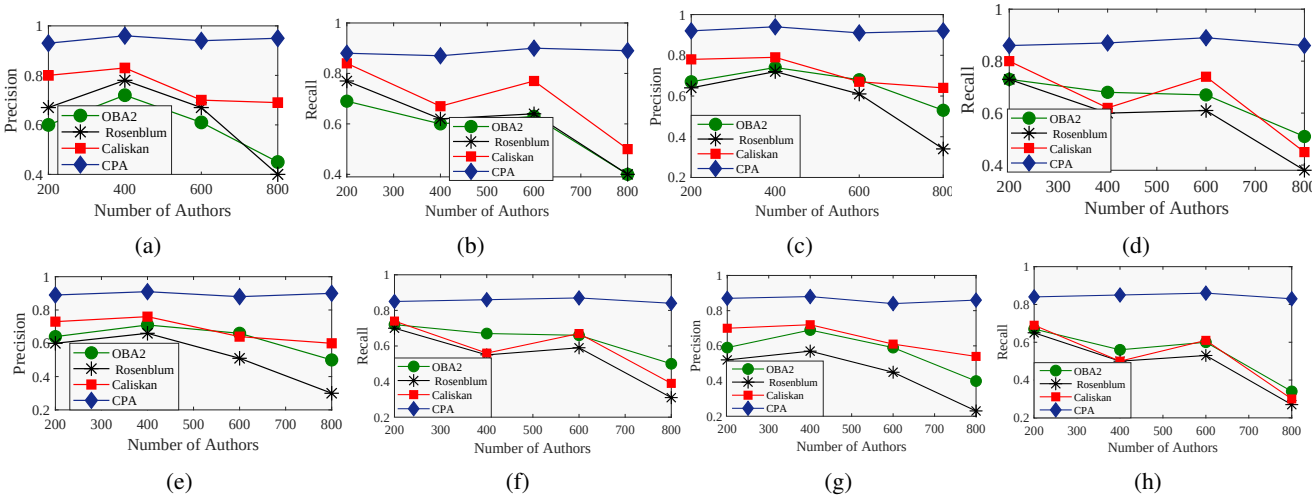

Fig. 10: Comparison of binary authorship attribution between different code transformation methods using the precision and recall metrics for the following scenarios: (1) Instruction Substitution (a, b); (2) Bogus Control Flow Graph (c, d); (3) Control Flow Flattening (e, f); (4) all methods together (g, h).

positive results in Table VII after we applied these clustering algorithms, and report the results in Table VI.

We tested two clustering algorithms: K-means and DBSCAN. We observed that K-means results in fewer false positives than DBSCAN. For instance, the lowest false positive rate achieved by K-means is 7% compared to 11% for DBSCAN, so we chose K-means.

TABLE VI: False positive ration of applying K-mean and DBSCAN clustering algorithms.

Malware	K-mean	DBSCAN	Malware	K-mean	DBSCAN
	False positive rate			False positive rate	
Ramnit	10%	16%	Simda	15%	23%
Lollipop	18%	19%	Tracur	10%	11%
Kelihos	14%	18%	Obfuscator.ACY	14%	17%
Vundo	7%	13%	Gatak	10%	20%

B. Authorship Clusters

To verify the effectiveness of CPA, we tested its ability to cluster a real-world malware dataset.

Clustering Process The Kaggle Microsoft Malware samples are decomposed to a set of functions. We then use CPA to cluster the functions that belong to the same malware family sample. As mentioned in Section I, we use for this purpose the k-mean algorithm. It computes the distance from the m points to c cluster centers, and repeat this computation for each of i iterations required to converge to a local optimum.

Reference Clustering The best option for demonstrating the correctness of a produced clustering, however, is to compare it with an existing reference clustering. To create a reference clustering, we took the following approach: First, we obtained a set of almost 500 GB of malware samples provided by the Kaggle Microsoft Malware Classification Challenge (BIG 2015) [14]. Then, we analyzed each sample with the proposed work [47] and the winner's work of the challenge [48], and the work proposed in [49]. For the reference clustering, we selected only those samples for which the majority of the aforementioned works that resulted the same malware family (this required us to define a mapping between the different labels that are used by different works). We then compare the resulted clusters by CPA with them.

Cluster Quality One of the most challenging tasks is to assess the quality of the results produced by a clustering algorithm [49]. To evaluate the quality of the clustering produced by our algorithm, we compared it to the reference clustering described above. To quantify the differences between the two clusterings, we introduce two metrics, precision and recall [49]. We use the same metrics and definitions proposed by [49]. The authors in [49] define the precision as follows: Assume we have a reference clustering $T = T_1, T_2, \cdots, T_t$ with t clusters and a clustering $C = C_1, C_2, \cdots, C_c$ with c clusters. For a sample set $M = m_1, m_2, \cdots, m_s$. For each $C_j \in C$, the cluster precision values as [49]: $P_j = max(|C_j \cap T_1|, |C_j \cap T_2|, \cdots, |C_j \cap T_t|)$. If P_j is greater than a predefined threshold value (0.81, this value is obtained through our experiments), we call it as a CC (correct cluster). Otherwise, we call it WC (wrong cluster). We report our results in Table VII.

TABLE VII: Clustering results. (TC): the total number of clusters, (CC): the percentage of correct clusters, (WC): the percentage of wrong clusters.

Malware	CPA			Malware	CPA		
	TC	CC	WC		TC	CC	WC
Ramnit	345	0.82	0.1	Simda	124	0.84	0.15
Lollipop	295	0.78	0.18	Tracur	105	0.82	0.1
Kelihos	145	0.89	0.14	Obfuscator.ACY	109	0.84	0.14
Vundo	544	0.9	0.07	Gatak	127	0.89	0.1

C. False Positive Rate

The false positives were examined in order to ascertain the circumstances under which CPA has a high probability of making errors in attribution decisions. The average false positive rate was determined to be 0.2%, which can be considered immaterial. This low rate can be attributed to the CPA utilizing numerous constituents within this detection system. We observed that the Google dataset has the highest false positive rate; we believe the reason is that each programmer follows standard coding instructions, which limits individual coding traits.

D. Multi-authorship attribution problem

In this section we carry out an experiment to demonstrate that it is possible to perform multi-author authorship attribution by applying CPA to discover the presence of multiple authors.

A set of GitHub projects were gathered, and the code contributors were checked. The parameters were set within the system to only include programmes written in C/C++, and disregard authors who simply add lines. To this end, 50 projects were collected, which included contributions for up to $1,500$ authors. To note: to ensure privacy, project details including project names were removed.

We report the accuracy metrics in Table VIII. In our experimental setup, we split the collected binary functions into ten sets, reserving one as a testing set and using the remaining nine sets as the training set.

IX. Related Work

Binary Authorship Attribution. In the context of authorship attribution, binary code has received much less focus, which can primarily be explained by the fact that during the compilation process, many pertinent elements that can help to determine an authors' style are lost. In [3]–[5], the authors illustrate how specific style elements can be retained during the compilation process and within binary code, thereby demonstrating that authorship attribution for binary code is possible. The strategy put forward by Rosenblum et al. [5] was the initial attempt to identify software binary authors automatically. This method mainly attempted to extract syntax-based elements using predefined templates including idioms, n-grams, and graphlets. Following this, Alrabaee et al. [3] proposed another method, which entailed extracting a series of instructions with particular semantics and using register manipulation to create a graph. More recently, Caliskan-Islam et al. [4] developed a

TABLE VIII: Evaluation result for identifying the presence of multiple authors in GitHub dataset.

Network	Opt.	Metrics	# of Authors Per Project						
			50	100	150	200	250	300	350
AAG	O0	Prec.	0.93	0.92	0.90	0.90	0.90	0.89	0.89
		Rec.	0.90	0.88	0.88	0.87	0.86	0.85	0.94
	O1	Prec.	0.92	0.91	0.90	0.89	0.88	0.88	0.97
		Rec.	0.91	0.90	0.90	0.88	0.88	0.87	0.91
	O2	Prec.	0.91	0.91	0.90	0.89	0.89	0.88	0.96
		Rec.	0.90	0.89	0.88	0.87	0.87	0.86	0.92
	O3	Prec.	0.91	0.90	0.90	0.89	0.89	0.88	0.95
		Rec.	0.92	0.91	0.91	0.89	0.89	0.89	0.92
CPA	O0	Prec.	0.99	0.99	0.99	0.99	0.99	0.99	0.99
		Rec.	0.98	0.97	0.97	0.95	0.95	0.96	0.96
	O1	Prec.	0.99	0.99	0.98	0.98	0.97	0.97	0.97
		Rec.	0.97	0.97	0.96	0.95	0.94	0.94	0.95
	O2	Prec.	0.97	0.97	0.96	0.96	0.96	0.96	0.97
		Rec.	0.96	0.96	0.96	0.95	0.95	0.94	0.95
	O3	Prec.	0.95	0.95	0.94	0.94	0.94	0.94	0.97
		Rec.	0.96	0.95	0.95	0.95	0.95	0.94	0.96
GoA	O0	Prec.	0.93	0.93	0.92	0.92	0.92	0.91	0.93
		Rec.	0.91	0.91	0.91	0.90	0.90	0.90	0.89
	O1	Prec.	0.92	0.92	0.91	0.91	0.91	0.91	0.91
		Rec.	0.91	0.91	0.90	0.89	0.89	0.89	0.89
	O2	Prec.	0.90	0.90	0.90	0.90	0.89	0.89	0.89
		Rec.	0.89	0.89	0.89	0.89	0.88	0.88	0.88
	O3	Prec.	0.90	0.90	0.90	0.89	0.89	0.89	0.89
		Rec.	0.89	0.88	0.87	0.87	0.86	0.86	0.86

new approach in which syntactical aspects of source code are extracted from decompiled executable binaries.

Although these approaches represent solid progress on authorship attribution, not one of these approaches is applied to real malware, mostly due to their dependency on training data with the same functionality (which is infeasible in the case of malware). Furthermore, these approaches suffer from certain limitations, including low accuracy in the case of multiple authors and being potentially thwarted by simple obfuscation. In [6], the authors implement novel fine-grained methods to determine the multiple authors of binary code which involves identifying the author of every basic block. At a base level, the authors extract syntactic and semantic elements including constant values in instructions, backward slices of variables, and width and depth of a function control flow graph (CFG).

We compare the existing authorship in terms of extracted features, implantation setup, and the availability of their code in Table IX. The extracted features belong to syntactic, structural, and semantic features. Most approaches are compatible with Linux binaries (e.g., ELF), and the binaries that they handle originally have C/C++ source code. Each approach requires a training dataset, with the exception of Meng's approach [6]. If the tool is available for researchers, we use private repositories, public repositories (indicates that the code is available but not included in general repositories), and general repositories.

Malware Authorship Attribution. Manual analysis is a primary component of the majority of the existing research on malware authorship attribution. In 2013, FireEye [50] published a technical report stating that malware binaries have common digital infrastructures and codes (for example, using certificates, executable resources, and development tools). Subsequently, Citizen Lab carried out malware author attribution based on the exploit form of the manual analysis located in binaries and the means of actions being performed; for instance, connecting to a command and control server. In [14], the authors displayed a sense of innovativeness in their strategy for creating viable connections between binaries that were generated by a particular group of authors. Their objective was to add transparency to the attribution process, and to develop a tool that could be used during analysis that underlines or refutes vendor statements. This approach is grounded in aspects of various domains, including implementation details, evasion strategies, classical malware characteristics, or infrastructure attributes, which enable the comparison of handwriting amongst binaries.

TABLE IX: A comparison between different systems that identify the author of program binaries. (SA) stands for single author. (MA) stands for multiple author. (L) means Linux, (W) means Windows, (A) means ARM, (M) means MIPS.

Approach	Features			Implementation			Applications
	Syntax	Semantics	Structure	Language	Platforms	Dataset	
[3]	✓	✓	✗	C++	Windows	✓	SA
[4]	✓	✗	✓	C	Windows/Linux	✓	SA
[5]	✓	✓	✓	C++	Linux	✓	SA
[6]	✗	✓	✓	C++	Linux	✗	MA
CPA	✓	✓	✓	C/C++	W,L, A, M, x86	✓	SA

X. Concluding Remarks

Our future work aims to extend CPA to tackle the aforementioned limitations. We will be studying other tools such as McSema [51] to transform binaries efficiently into LLVM IR. To conclude, we have presented the first known effort on identifying the author of binary code cross different CPU architecture based on style characteristics. When attempting to extract stylometry styles, the existing research has employed machine learning techniques and have proved capable of identifying up to 800 authors. In comparison, this research has been successful in distinguishing up to 1,500. Furthermore, existing research utilizes artificial datasets only, and conversely, this research includes datasets that are more reality based, and was also applied to the GitHub dataset. This research determined that once more than 250 authors are in play, the accuracy of these strategies falls significantly to approximately 45%. Advanced expertise and realistic dataset authors are much easier to attribute than those of lower expertise or artificial datasets. In the GitHub dataset for example, the sample authors can be distinguished to more than 90% accuracy level.

Acknowledgments

We are grateful to the anonymous TIFS reviewers for their comments and suggestions. The first author is partially supported by the United Arab Emirates University Start-up Grant *G00003261*.

References

[1] A. Caliskan-Islam, R. Harang, A. Liu, A. Narayanan, C. Voss, F. Yamaguchi, and R. Greenstadt, "De-anonymizing programmers via code stylometry." USENIX, 2015.

[2] F. Farnstrom, J. Lewis, and C. Elkan, "Scalability for clustering algorithms revisited," *ACM SIGKDD Explorations Newsletter*, vol. 2, no. 1, pp. 51–57, 2000.

[3] 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.

[4] 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.

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

[6] 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.

[7] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 363–376.

[8] T. Kim, C. H. Kim, H. Choi, Y. Kwon, B. Saltaformaggio, X. Zhang, and D. Xu, "Revarm: A platform-agnostic arm binary rewriter for security applications," in *Proceedings of the 33rd Annual Computer Security Applications Conference*. ACM, 2017, pp. 412–424.

[9] W. H. Hawkins, J. D. Hiser, M. Co, A. Nguyen-Tuong, and J. W. Davidson, "Zipr: Efficient static binary rewriting for security," in *Dependable Systems and Networks (DSN), 2017 47th Annual IEEE/IFIP International Conference on*. IEEE, 2017, pp. 559–566.

[10] Y. Hu, Y. Zhang, and D. Gu, "Automatically patching vulnerabilities of binary programs via code transfer from correct versions," *IEEE Access*, vol. 7, pp. 28 170–28 184, 2019.

[11] S. Nagy and M. Hicks, "Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 787–802.

[12] A. Nguyen-Tuong, D. Melski, J. W. Davidson, M. Co, W. Hawkins, J. D. Hiser, D. Morris, D. Nguyen, and E. Rizzi, "Xandra: An autonomous cyber battle system for the cyber grand challenge," *IEEE Security & Privacy*, vol. 16, no. 2, pp. 42–51, 2018.

[13] "Cyber Grand Challenge," https://www.darpa.mil/program/cyber-grand-challenge, November, 2019.

[14] "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.

[15] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 138–157.

[16] J. Cohen, "Trusses: Cohesive subgraphs for social network analysis," *National Security Agency Technical Report*, p. 16, 2008.

[17] F. Rousseau and M. Vazirgiannis, "Main core retention on graph-of-words for single-document keyword extraction," in *European Conference on Information Retrieval*. Springer, 2015, pp. 382–393.

[18] S. B. Seidman, "Network structure and minimum degree," *Social networks*, vol. 5, no. 3, pp. 269–287, 1983.

[19] A. Tixier, F. Malliaros, and M. Vazirgiannis, "A graph degeneracy-based approach to keyword extraction," in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, 2016, pp. 1860–1870.

[20] "IDA pro Fast Library Identification and Recognition Technology," https://www.hex-rays.com/products/ida/tech/, 2011.

[21] 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.

[22] 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.

[23] 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.

[24] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Polymorphic worm detection using structural information of executables," in *RAID*. Springer, 2005.

[25] S. Alrabaee, M. Debbabi, and L. Wang, "On the feasibility of binary authorship characterization," *Digital Investigation*, vol. 28, pp. S3–S11, 2019.

[26] S. Alrabaee, P. Shirani, L. Wang, and M. Debbabi, "Sigma: A semantic integrated graph matching approach for identifying reused functions in binary code," *Digital Investigation*, vol. 12, pp. S61–S71, 2015.

[27] T. Junttila and P. Kaski, "Engineering an efficient canonical labeling tool for large and sparse graphs," in *2007 Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 2007, pp. 135–149.

[28] "Windows API Sets," https://msdn.microsoft.com/en-us/library/windows/desktop/ff818516(v=vs.85).aspx/, 2017, accessed on December, 2016.

[29] "Linux API Sets," http://pubs.opengroup.org/onlinepubs/9699919799/functions/contents.html/, 2017, accessed on December, 2016.

[30] Z. S. Harris, "Distributional structure," *Word*, vol. 10, no. 2-3, pp. 146–162, 1954.

[31] "CRT Alphabetical Function Reference," https://msdn.microsoft.com/en-us/library/634ca0c2.aspx, July, 2018.

[32] J. Wang and J. Cheng, "Truss decomposition in massive networks," *Proceedings of the VLDB Endowment*, vol. 5, no. 9, pp. 812–823, 2012.

[33] S. Lee, S. Kim, S. Lee, H. Yoon, D. Lee, J. Choi, and J.-R. Lee, "Largen: Automatic signature generation for malwares using latent dirichlet allocation," *IEEE Transactions on Dependable and Secure Computing*, 2016.

[34] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *Journal of machine Learning research*, vol. 3, no. Jan, pp. 993–1022, 2003.

[35] 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.

[36] 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.

[37] "The Scalable Native Graph Database. Available from:," http://neo4j.com/, 2015.

[38] "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.

[39] "Programmer De-anonymization from Binary Executables," https://github.com/calaylin/bda, 2015.

[40] "Hex-Ray decompiler," https://www.hex-rays.com/products/decompiler/, 2015.

[41] 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.

[42] S. H. Ding, B. C. Fung, and P. Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 472–489.

[43] J. Zeng, Y. Fu, K. A. Miller, Z. Lin, X. Zhang, and D. Xu, "Obfuscation resilient binary code reuse through trace-oriented programming," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 487–498.

[44] R. R. Branco, G. N. Barbosa, and P. D. Neto, "Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies," 2012.

[45] "Tigress is a diversifying virtualizer/obfuscator for the C language," 2016, http://tigress.cs.arizona.edu/.

[46] M. Ester, H.-P. Kriegel, J. Sander, X. Xu *et al.*, "A density-based algorithm for discovering clusters in large spatial databases with noise." vol. 96, no. 34, 1996, pp. 226–231.

[47] J. Drew, M. Hahsler, and T. Moore, "Polymorphic malware detection using sequence classification methods and ensembles," *EURASIP Journal on Information Security*, vol. 2017, no. 1, p. 2, 2017.

[48] carson and X. Chen, "NO to overfitting," http://blog.kaggle.com/2015/05/26/microsoft-malware-winners-interview-1st-place-no-to-overfitting/, July, 2018.

[49] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, "Scalable, behavior-based malware clustering." in *NDSS*, vol. 9, 2009, pp. 8–11.

[50] N. Moran and J. Bennett, *Supply Chain Analysis: From Quartermaster to Sun-shop*. FireEye Labs, 2013, vol. 11.

[51] McSema, "McSema is an executable lifter," https://github.com/lifting-bits/mcsema, November, 2019.

PLACE PHOTO HERE

Saed Alrabaee is currently an Assistant Professor at the department of Information Systems and Security in UAEU. Prior to joining UAEU, Dr. Alrabaee was a visiting assistant professor at the department of Electrical and Computer Engineering and Computer Science at the University of New Haven (UNH), US. He is also a permanent research scientist at the Security Research Center, CIISE, Concordia University, Canada. Dr. Alrabaee holds a Ph.D. degree in information system engineering from Concordia University in Montreal, Canada, which was executed under the supervision of Prof. Mourad Debbabi and Prof. Lingyu Wang. His research and development activities and interests focus on the broad area of reverse engineering, including, binary authorship attribution and characterization, malware investigation, and function fingerprinting. Dr. Alrabaee has published in this domain a book, 10 Journals, and 5 conferences.

PLACE PHOTO HERE

Mourad Debbabi is a Full Professor at the Concordia Institute for Information Systems Engineering and Associate Dean Research and Graduate Studies at the Faculty of Engineering and Computer Science. He holds the NSERC/HydroQubec Thales Senior Industrial Research Chair in Smart Grid Security and the Concordia Research Chair Tier I in Information Systems Security. He is also the President of the National Cyber Forensics and Training Alliance (NCFTA) Canada. He is also a member of CATAAlliance's Cybercrime Advisory Council and a member of the Advisory Board of the Canadian Police College. He is the founder and one of the leaders of the Security Research Centre at Concordia University. In the past, he was the Specification Lead of four Standard JAIN (Java Intelligent Networks) Java Specification Requests dedicated to the elaboration of standard specifications for presence and instant messaging. Dr. Debbabi holds Ph.D. and M.Sc. degrees in computer science from Paris-XI Orsay, University, France. He published 3 books and more than 260 peer-reviewed research articles in international journals and conferences on cyber security, cyber forensics, privacy, cryptographic protocols, threat intelligence generation, malware analysis, reverse engineering specification and verification of safety-critical systems, smart grid, programming languages and type theory. He supervised to successful completion 26 Ph.D. students and more than 65 Master students. He served as a Senior Scientist at the Panasonic Information and Network Technologies Laboratory, Princeton, New Jersey, USA; Associate Professor at the Computer Science Department of Laval University, Canada; Senior Scientist at General Electric Research Center, New York, USA; Research Associate at the Computer Science Department of Stanford University, California, USA; and Permanent Researcher at the Bull Corporate Research Center, Paris, France.

PLACE PHOTO HERE

Lingyu Wang is a professor in the Concordia Institute for Information Systems Engineering (CIISE) at Concordia University, Montreal, Quebec, Canada. He is NSERC/Ericsson Industrial Research Chair (IRC) in SDN/NFV Security. He received his Ph.D. degree in Information Technology in 2006 from George Mason University. He holds a M.E. from Shanghai Jiao Tong University and a B.E. from Shenyang Aerospace University in China. His research interests include cloud computing security, network security metrics, software security, and privacy. He has co-authored five books, two patents, and over 100 refereed conference and journal articles at top journals/conferences, such as TOPS, TIFS, TDSC, TMC, JCS, S&P, CCS, NDSS, ESORICS, PETS, ICDT, etc. He is serving as an associate editor for IEEE Transactions on Dependable and Secure Computing (TDSC) and he has served as the program (co)-chair of seven international conferences and the technical program committee member of over 100 international conferences.