On the Feasibility of Binary Authorship Characterization

Saed Alrabaee[a,b], Mourad Debbabi[c], Lingyu Wang[c]

[a] *United Arab Emirates University, UAE*
[b] *University of New Haven, USA*
[c] *Concordia University, Canada*

Abstract

This work aims to develop an automatic tool that can perform the laborious and error-prone reverse engineering task of binary authorship characterization, i.e., determining clues related to the author(s) of a piece of binary code. Software code written by human programmers reflects the author's educational background, level of expertise, and coding traits. Accordingly, these may be characterized by identifying meaningful features and examining them. Binary authorship characterization reveals information that can be extremely useful for security applications such as digital forensics, malware triage, and binary vulnerability tracking. This paper proposes a system, `BinChar`, that capture various aspects of author style, including code trait characteristics, code structure characteristics, and code behavior characteristics. For the purpose of detection, a Convolutional Neural Network (CNN) is used. The results generated by the CNN are evaluated more precisely using Bayesian calibration. We tested `BinChar` in identifying the characteristics of the authors of program binaries. Also, we applied it to almost 500 GB of malware samples provided by the Kaggle Microsoft Malware Classification Challenge, to demonstrate that `BinChar` is an appropriate tool for characterizing malware families. As an illustration, we report a case study in which we determine the author characteristics of the *Mirai* botnet and compare them with the author characteristics of 360,000 malware samples.

1. Introduction

When analyzing malware binaries, reverse engineers often pay special attention to their characterization for several reasons. First, reports from anti-malware companies indicate that finding the similarities between malware code characteristics can aid in developing profiles for malware families [1]. Second, recently released reports by Citizen team [2, 3] show that malware binaries written by authors having the same origin share similar characteristics. Third, many malware packages could have been written only by authors with a special level of expertise and special knowledge for dealing with specific resources; an example is SCADA system malware. This insight provides a critical clue for the extraction of information about the functionality of a malware binary. Fourth, although obfuscation techniques may be applied before the malware is released and may modify the code significantly, it is still desirable to determine which obfuscation techniques and tools have been used. Last, clustering binary functions based on a common origin may help reverse engineers identify the group of functions that belong to a particular malware family or decompose the binary based on the origin of its functions.

The ability to conduct these analyses at the binary level is especially important for security applications because the source code for malware is not always available. However, in automating binary authorship characterization, two main challenges are typically encountered: the binary code lacks many abstractions (i.e., function prototypes) 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 designed to develop automated systems for source code authorship characterization [4, 5, 6, 7], these often depend on features that will likely not be preserved in the strings of bytes representing executable file after the compilation process, such as variable and function naming, original control and data flow structures, comments, and space layout.

To the best of our knowledge, there have been no attempts to characterize the authors of program binaries. Nonetheless, a few approaches to binary authorship attribution have been proposed, but they typically use machine learning algorithms to extract unique patterns for each author and then compare a given target binary against such patterns to identify the author [8, 9, 10, 11, 12].

These approaches cannot be applied directly to binary authorship characterization because of the following limitations: the chosen features are generally not related to author style but rather to functionality; they are not applicable to real malware; and dealing with the binary authorship characterization problem requires that the chosen features have the power to detect, for example, author styles, the compilers used, the free packages reused, the functionality, the implementation frameworks, and the binary timestamps. More recently, the feasibility of authorship attribution for malware binaries was discussed at the

BlackHat conference [2]. It was concluded that a set of features can be employed to group malware binaries according to authorship characterizations. However, the process is not automated and requires considerable human intervention.

System overview. To address the aforementioned limitations, this paper presents an innovative system, `BinChar`, that describes the characteristics of programmers according to their educational background, level of expertise, and coding traits. To achieve this, we have defined a new set of features that extracts authorship attribution characteristics. These features are extracted from different levels: the level of the basic blocks, the level of the function bodies, the program level, and the file level. Based on them, our system is able to detect structural, optimization, knowledge, expertise, and code trait characteristics, and also overall characteristics that is resulted from the aforementioned characteristics together. All the extracted features are passed to the CNN. We observe note some attractive characteristics of neural networks such as they can learn end-to-end, where each stage is trained simultaneously to achieve the end goal [13]. Also, the nodes of CNN can act as filters over the input space and can discover the strong correlation in the binary code [14]. Third, CNN is considered a fast neural network in classification process when it is compared with other neural networks such as recurrent neural network [15]. The results obtained from CNN is passed to Bayesian calibration to precisely check the correctness of CNN.

Contributions. Our contributions are summarized below.

- We designed a new set of features that make *BinChar* accurate and efficient, enabling it to characterize the authors of program binaries with high speed while tolerating the noise injected by code transformations arising from the use of different compilers and optimization speed levels. The experimental results show that our system is able to cluster the samples according to similarities in authorship characteristics with a precision of over 95%.

- We investigated the effectiveness and the power of CNNs in the context of binary authorship characterization. To the best of our knowledge, this is the first work in which CNNs are used for binary authorship characterization. Further improvement is achieved by performing Bayesian calibration, which reduces the rate of false positives to 0.02%.

- We used `BinChar` to extract author characteristics from Mirai botnet binaries and compared the results with those for 360,000 malware samples collected from various sources. Finally, we report the authorship characteristics common to the Mirai botnet and other families.

2. Preliminaries

2.1. Threat Model

`BinChar` is designed to assist, instead of replacing, reverse engineers in various use cases of characterizing the author of program binaries, such as forensic analysis (e.g., linking a new malware to previously known malware or malware author(s), clustering a group of malware based on common characteristics, and finding co-authorship for the malware binary code (e.g., to determine "influencers" in the code community), and software copyright infringement analysis (e.g., detecting borrowed code fragments inside a program binary based on authorship characteristics). Therefore, the focus of `BinChar` is not on general reverse engineering tasks, such as unpacking and deobfuscating malware samples (although `BinChar` leverages some existing tools to de-obfuscate and unpacked such binaries), but rather on detecting and determine authorship characteristics clues such as using advanced resources, particular compiler, os, specific code traits, memory allocation habits, c&c commands, etc. We just pay attention to the author characteristics which can be common among different set of authors). Moreover, the authors of a specific malware family, they have to share common knowledge and expertise to deal with a particular environment [16, 17]. In designing the features and methodology of `BinChar`, we have taken into consideration some potential countermeasures. In particular, `BinChar` assumes the adversary may attempt to evade detection through the following.

- The adversaries may apply refactoring techniques, e.g., when a malware author aims to defeat forensic analysis by modifying his/her own code, or when an adversary attempts to modify borrowed code written by other authors in order to evade copyright infringement detection.

- The adversaries may apply obfuscation techniques on binary files to alter its syntax, e.g., when a malware author wants to defeat anti-virus signature-based detection.

- Since a program can be significantly altered by simply changing the compilers or their settings, the adversary may make such changes to evade detection.

We will show in later sections how `BinChar` may survive the aforementioned threats. Simply, the features of `BinChar` have been designed to determine binary authorship characteristics at multiple abstraction levels, which makes it harder for adversaries to evade system detection [2]. In addition, an operational solution is to customize and enrich the list of features employed by `BinChar` based on the actual use case and learning data, which will not only make it much more difficult for adversaries to hide all of their binary code characteristics such as code traits, but will also improve accuracy.

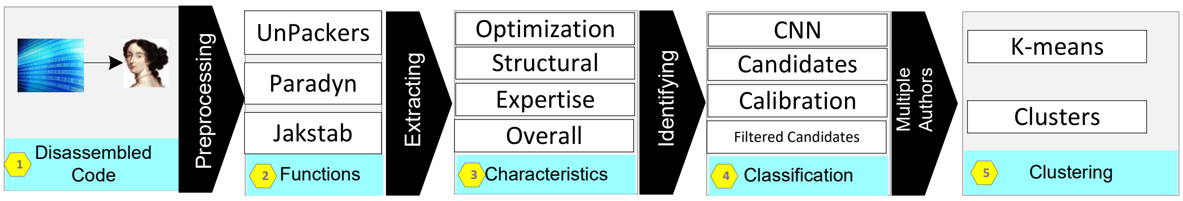

Figure 1: *BinChar* Architecture

2.2. Usage Scenarios

The interest of the `BinChar` framework is to determine the authorship characteristics of an anonymous piece of binary code. Additionally, given code that is written by multiple authors, it is required to determine which characteristics (e.g., coding habits) of the code are attributed to which author. It is assumed that a set of binary code samples is available, where the code is either labeled with an author(s) from a set of known candidate authors, or from other unknown authors. Given an anonymous piece of binary code, `BinChar` converts the code into a set of features that are consequently used to attribute the author(s) according to their characteristics. `BinChar` also labels the compiler-related functions in addition to discovering the reused functions and third-party libraries. This is useful for a number of applications, as listed in what follows.

Software Infringement. A set of candidate authors is clustered based on previously collected malware samples, online code repositories, etc. There are no guarantees that the anonymous author is one of the candidates, as the test sample may not belong to any known authors. However, it is not our intention to identify the author, but rather to characterize the author. Finally, it may be suspected that a piece of code is not written by the claimed author, but yet there are no leads as to who the actual author may be [4]. For this reason, we may compare the programs written by the claimed author and measure the degree of similarity in terms of binary code characteristics.

Forensic Investigation. FireEye [18] 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 vital assistance to forensic investigators. In addition, testing recent pieces of malware binary code using confidence metrics would verify if a specific author is the actual author.

2.3. System Overview

The architecture of `BinChar` is illustrated in Figure 1. As shown, the four main components are: (i) Preprocessing component where PEfile [19] is employed to check if the binary file is packed. If it is packed, the corresponding unpacker, such as UPX, is used to unpack the binary file and pass the unpacked binary file to the disassembler tools such as IDA Pro [20]. (ii) Feature extraction which deals with a different set of features that are related to the authorship characterization. This component is able to accept either an executable file or an assembly file as input. In the case of executable file, *BinChar* converts executable file to a gray-scale image. Then numerous features are extracted from this image. After that we rank these features by employing mutual information. These top ranked features are passed to detection component. Also, *BinChar* uses executable file as input to LLVM in order to optimize the code and compare it with the original executable. In the case of assembly file, *BinChar* extracts the following static features: (1) merging Annotated Control Flow Graph (ACFG) with Data Flow Graph and the new graph is decomposed then into graph truss. (2) We extract the longest path and then apply statistically analysis it. (4) Strings are extracted that can disclose the binary provenance such as the source compiler. The extracted featured are hashed by LSH algorithm. (iii) The fixed size of hashes resulted by LSH are passed to the third component. The CNN is trained by set of fixed size of hashes. Once we have a target binary, CNN will return a set of candidates according to the training hashes. These candidates are passed to Bayesian model calibration to seek to provide users with accurate probabilities that a given binary file. This component is used to identify the author of program binary if the characteristics are the same. (iv) The final component represents the outputs of `BinChar`. `BinChar` clusters function or programs together according to the similarity of the author's characteristics of the binary code. Standard clustering algorithm (k-means) [21] is used for this purpose. As shown in Figure 1, the author repository is connected to this component since our features can reveal a significant information such as the compilers, resources used, level of expertise, education background, API calls, and some code traits like the way of configuration.

3. `BinChar`: Design Overview

In this section, we describe our system in detail.

3.1. Feature Engineering

3.1.1. Optimization Characteristics

We designed a set of features based on existing tools [22, 23, 24, 25]. We leveraged these tools to construct a data flow graph incorporating data and control dependencies. The graph reveals the inline functions, making it easy to count them. Then we constructed a semantic graph and normalized it by applying the rules introduced in [23]; these are related to code optimization. The first rule is constant folding. For instance, the 3-node subgraph representing the expression `1+4` is rewritten into a single node to represent the value `5`. The second rule concerns side effects. The authors of [23] employ state variables to detect the dependencies between instructions. They define a state variable for memory states: every instruction contains of a memory register and its usual parameters. This additional register forces a dependency between, for example, a load instruction and the preceding store instructions [23], retaining all the relevant information about each register. The third rule concerns aliasing information that relaxes the strict ordering of instructions imposed by the transformation. For instance, the pointers in LLVM returned by `alloc` never alias with one another. Hence, if two load instructions from a memory involve a store to the same pointer, then the load can be simplified to a single instruction. The fourth rule deals with loops. The authors came up with a method for placing gates within looping control flow, including breaks, continues, and returns from within a loop. Once all the rules were applied, we conducted statistical analyses to compare the original code with the optimized code, including the mean, variance, standard deviation, and mean square error.

3.1.2. Structural Characteristics

In this subsection, we merge the Annotated Control Flow Graph (ACFG) and the Data Flow Graph (DFG) into one graph, named author style graph. The intuition behind merging them is that the ACFG may reveal the characteristics of the author's task implementation, while the DFG reveals how the author manipulates variables and also highlights the author's use of small functions, which are typically inlined by the compiler during optimization.

Author Style Graph

We build a semantic control flow graph called the *Annotated Control Flow Graph* (`ACFG`). It is an abstracted version of the CFG, generalizing specific types of CFG features according to multiple criteria. The steps for constructing an `ACFG` are summarized as follows. The system takes a `CFG` as an input and computes the frequency of opcodes across instruction groups. Then, it categorizes assembly instructions into six groups [26], as shown in Table 1.

The resulting `ACFG` describes structural characteristics. The next step is to complement it with the Data Flow Graph (DFG). Then, data flow dependencies can be incorporated to make it possible to use coarse reasoning about the program control flow and data dependencies to infer how the author manipulates program variables and how variable relations are built. Let R_k/W_k denote registers or memory that are read or written by instruction I_k. If i_1 and i_2 are instructions belonging to I, then we define the following possible dependencies: i_1 writes something which 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. Recognizing that the ACFG and DFG provide complementary views about the implementation of a specific task by emphasizing different aspects of the underlying authorship characteristics, we combine them into a joint data structure to facilitate more efficient graph matching between different binary codes to help identify similarities in authorship characteristics. The result is the Author Style Graph (*ASG*).

Table 1: Patterns for Annotation [26]

Feature	Description	Example
DTR & STK	Data Transfer and Stack	push, mov, etc
ATH & LGC	Arithmetic and Logical	add, xor, etc
CAL & TST	Call and Test	call, cmp, etc
REG & MEM	Register and Memory	esi, [esi+4]
REG & CONST	Register and Constant	esi, 30
MSC	Milestones	MEM and Const

Definition 1. The Author Style 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: $\zeta \longrightarrow \gamma$, where γ is a set of labels. Finally, ϑ is a data dependency function which labels each node $n \epsilon N$ based on the data dependency rules function λ.

3.1.3. Expertise Characteristics

Typically the path in the CFG has been shown as a robust feature [27]. Further, to find matched paths can be considered as an alignment problem where dynamic programming can be applied [27]. Longest path reflects the author traits in implementing tasks or using nested loops. Thus, we choose the longest path. We use depth first search to traverse the CFG, and then choose the path with the largest number of nodes. Once the longest path is constructed, we extract two categories of features: statistical features (cyclomatic complexity) and dynamic features (execution traces).

Cyclomatic complexity [28] computes the quantity of linear independent paths to represent the complexity of a code in the form of a graph metric. The complexity C of the longest path of CFG is as follows:$C = E + N + 2P$, where E is the edge quantity, N is node quantity, and P is the quantity of connected components in the longest path.

3.1.4. Overall Characteristics

We convert the binary file content to a set of bytes. This set is transferred into a matrix. This two-dimensional matrix has fixed width d. More specfically, the first d bytes move to the first row of the matrix, and the nth group of d bytes moves to the nth row of the matrix. This method is similar to the malware visualization techniques introduced in [29, 30, 31].

3.2. *Feature Hashing*

Since our tool makes use of various features extracted with different formats (numeric, graphs, instruction traces, etc.), LSH was selected to unify features before feeding them into CNN. Once employing *minhash* (with K unique hash functions) as a set of signatures to introduce represent the used features, the minhash values will be splitting by LSH into a signature matrix with l bands encompassing r rows each [32]. To compute the number of minhash values for a given band, we will divide the number of minhashes by the number of bands (K/l). The number of rows will be equal to the number of featured minhash signatures.

4. System Detection

4.1. *Convolutional Neural Network*

Convolutional Neural Networks (CNNs) are one type of Neural Networks that have shown efficiency in various areas such as image processing. Generally, each input node is connected to each output node in the next year. In CNN, it is not the case where convolution is used over the input layer to compute the output. This process leads in local connectivity, each input region is connected to a node in the output. Many filters are applied at each layer and combine their results. After convolution step, the subsequent step is called ReLU which stands for Rectified Linear Unit and is a non-linear operation. Its output is given by: Output = Max(zero, Input). This step is used to replace any unwanted features by zero. The purpose of ReLU is to introduce non-linearity in CNN. This operation is called activation function. Besides, there are other layers are called pooling layers. These layers are inserted periodically between convolution layers. The main goal of pooling layers is to reduce the spatial size of the representation, to minimize the number of parameters and computation in the network, and also to dominate overfitting. Finally, the last layer is then a classifier that uses these high-level features. It has been show that CNN can be more accurate and efficient to train if they contain shorter connections between layers close to the input and those close to the output [33]. Consequently, we have adopted their CNN algorithm for binary authorship characterization. Their algorithm follows a feed-forward fashion by connecting each layer to every other layer. Where the inputs used for each layer, are the feature-maps of all preceding layers. Besides, its own feature-maps are used as inputs into all subsequent layers [33]. This has one main advantage by alleviating the vanishing-gradient problem.

Consider a feature vector v_0, which is transferred through a convolutional network. The network consists of L layers, each of which implements a non-linear transformation T_l where l indexes the layer. H_l can be rectified linear units (ReLU) [34, 35]. We denote the output of i^{th} layer as v_i.

Identity function. In the traditional CNN forward networks, the output of l^i layer as input to $(l+1)^i$, which gives rise to the following layer transition: $v_i = H_l(v_{i-1})$. To bypass the non-linear transformation, a skip connection has been added with a transition layer, and form an identity function: $v_i = H_l(v_{i-1}) + v_{i-1}$. The advantage of such process is that the gradient flows directly through the identity function from later layers to the earlier layers.

Dense connectivity. The information flow between layers should be improved in order to enhance the accuracy and efficiency. Therefore, we use the proposed algorithm by [33]. They introduce direct connections from any layer to all subsequent layers. Hence, the i^{th} layer receives the feature-maps of all preceding layers, $v_0, \cdots, v_{i-1}$, as input: $v_i = H_i([v_0, v_1, \cdots, v_{i-1}])$, where $[v_0, v_1, \cdots, v_{i-1}]$ refers to the concatenation of the feature-maps produced in layer $0, \cdots, i-1$.

Activation function. For this purpose, we have employed rectified linear unit (ReLU) [35]. This unit is computed separately for each layer. Without this unit, the layer will be an affine function.

Pooling layer. One of the challenges is that when the feature-maps size is changed. Therefore, it is important component of convolutional networks is pooling layers that change the size of feature-maps.

To enable pooling in our network, we divide our network to different dense component followed the architecture in [33]. We illustrate the layers with their connectivity in Figure 2. As shown in Figure 2, the layers between blocks are called transition layers which do convolution and pooling. We follow in our model, the same settings are used by [33]. They use batch normalization layer and an $1x1$ convolutional layer followed by a $2x2$ average pooling layer [33].

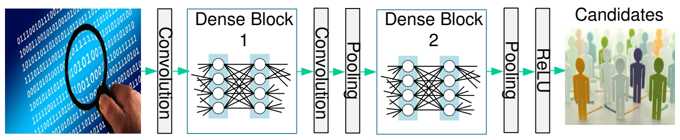

Figure 2: Our CNN architecture with deep DenseNet [33] with two dense blocks

4.2. *Calibration Model*

Motivated by [36], we are interested in using their Bayesian calibration model. The main goal of this model is to deliver users with accurate probabilities that a given file is how similar to another file. This calibration model combines our firs probability threshold value (represented as the ratio of author X to all other authors in our dataset) and the other information about our CNN's error profile against test data. In what follows we briefly introduce our adapted model for adjusting the CNN similarity score to reflect the true author label score, given this qualitatively assumed ratio of author to all other authors.

Considering $0 \leq c \leq 1$ be some score given by the CNN, which indicates the degree to which a CNN measures a binary how similar is to other author of program binaries, with 0 being completely not related to any author, and 1 being certainly related to that author. Our goal is to convert this number into a "calibration" score, which will provide the user a measure of how likely that the author of target binary is accurately related to the authors of candidates. To achieve this, we define the calibration score as the probability that the author of program binary will accurately related to the resulted candidates by CNN, $P(R = x_i|C = c)$, given the score c, and list of candidates $R = x_1, x_2, \cdots x_i, x_{i+1}, \cdots x_k$, where k is the threshold value that represents the number of candidates. By using Bayes' law we have:

$$P(x_i|c) = \frac{p(c|x_i)P(x_i)}{p(c)} =$$

$$\frac{p(c|x_i)P(x_i)}{p(c|x_i)P(x_i) + p(c|u)P(u)}$$

where p is the pdfs, we suppose that pdfs for the list of candidates and target scores for CNN, $p(C = c|R = u)$, where u is the set of all candidates except the target file, which means $u = \{x_1, x_2, \cdots x_i, x_{i+1}, \cdots x_k\} - x_i$.

To have the probabilities sum up to 1, we use the constraint that gives us the final value of the calibration score in terms of pdfs and probability of target author with the list of candidates. We define our problem by our CNN's expected pdf for target binary and list of candidates u. To derive the pdfs, we have used the non-parametric approach, like kernel density estimator (KDE) [37], where we approximation a value of pdf given C by taking a weighted average of the neighborhood []. Since the pdfs can only take values in [0,1], the samples are mirrored to the left of 0 and the right of 1, before computing the estimated pdf value at a specific point.

4.3. Clustering Similar Functions

For the clustering process, , we use a standard clustering algorithm (k-means) [21] 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. For more details, we refer the reader to [21]

5. Evaluation

In this section, we present the evaluation results for the possible use cases described earlier in this paper. Section 5.1 shows the setup of our experiments and provides an overview of the data we collected. The main results on authorship attribution characterization and identification are then presented. Also, we have studied the impact of different CNN parameters on the *BinChar* accuracy. Finally, *BinChar* is applied to real malware binaries and the results are discussed.

5.1. Implementation Environment

The described binary feature extractions are implemented using separate python scripts for modularity purposes, which altogether form our analytical system. For CNN setup, we first use a convolution with 16 output channels is performed on the input feature vectors before the first dense block. We use kernal size 3x3 for convolutional layers. We follow zero-padded for inputs to keep the feature-map size fixed [33]. We use 3x3 convolution followed by 4x4 average pooling as transition layers between two contiguous dense blocks. Further, the global average pooling is excuted at the end of the last dense block. Then a softmax classifier is attached. The feature-map sizes in the two dense blocks are 128x128, and 64x64, respectively.

5.2. Dataset

The used dataset is consisted of several files from different sources, as described below: i) GitHub [38]; ii) Google Code Jam [39]; and iii) a set of known malware files representing a mixture of 1500 different families including the nine families provided in Microsoft Malware Classification Challenge [2]. Statistics about the dataset are provided in Table 2.

Table 2: Statistics about the binaries used in the evaluation

Source	# of authors	# of programs	# of functions
GitHub	3550	12,910	4,900,0000
Google Jam	16,000	55,550	10,965,120
Malware	1500	360,000	45,650,214
Total	21,050	428,460	61,515,334

5.3. Dataset Compilation

The ultimate dataset is compiled with different compilers and compilation settings to measure the effects of such variations. We use g++, Clang, GNU Compiler Collection's gcc, ICC, as well as Microsoft Visual Studio (VS) 2010, with different optimization levels.

5.4. Accuracy

The purpose of this experiment is to evaluate the accuracy of characterizing the author of in program binaries.
Evaluation Settings. The methodology used in our evaluation uses both standard ten-fold cross-validation and random subset testing, based on the experiment: For classification of the entire data set, we use ten-fold cross-validation. The evaluation of *BinChar* system is conducted using the datasets described in Section 5.2. The data is randomly split into 10 sets, where one set is reserved as a testing set, and the remaining sets are used as training sets. The process is then repeated 15 times. To evaluate *BinChar* and to compare it with existing methods, precision P and recall R measures are applied. Furthermore, since the application domain targeted by *BinChar* is much more sensitive to false positives than false negatives, we employ an F-measure as follows:

$$F_1 = 2\,\frac{P\ .\ R}{P + R} \tag{1}$$

BinChar Accuracy. We first investigate the accuracy of our proposed system in identifying the author of program binaries based on author characteristics. The results are reported in Table 3. The highest accuracy obtained by our tool is 0.94 when all characteristics components are together. Further, we can observe that the expertise characteristics return the highes accuracy of 0.93. This is due to the fact that the author may use his expertise to implement a specific task. For instance, the author may use specific package or advance resources to reduce the execution time. As mentioned earlier, we use static and dynamic feature to detect the author expertise characteristics. However, the optimization characteristics return an accuracy of 0.81. Here, we optimize the original code and then we compare them in terms of statistical analysis. We observe through our experiments if the author tries to optimize the written code, then these characteristics may not help fully to identify the characteristics of the author.

Table 3: F-result

Characteristics	**Prec.**	**Rec.**	F_1
Structure	0.95	0.87	0.91
Optimization	0.84	0.79	0.81
Knowledge	0.92	0.88	0.90
Expertise	0.97	0.9	0.93
Overall	0.89	0.86	0.87
BinChar	0.98	0.91	0.94

5.5. *False Positive Rate*

We investigate the false positives in order to understand the situations where *BinChar* is likely to make incorrect attribution decisions. The average of false positive rate is 0.02%. It is very low and could be neglected. The reason behind the low false positive rate is that BinChar uses stratified detections system. We have observed that the false positives rate for google dataset is the highest rate and we believe the reason behind this is that each programmer should follow the standard coding instructions which restrict him/her to have their own code traits.

Table 4: Characteristics of malware datasets. (AF): Assembly Functions, (CF): Compiler Functions, (LF): Library Function.

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

5.6. *Authorship Clusters*

To verify the effectiveness of *BinChar*, we test it to cluster real-world malware dataset. Hence, we have applied *BinChar* to almost 500 GB of malware samples provided by the Kaggle Microsoft Malware Classification Challenge (BIG 2015) [2], whose goal is to predict classes of malware families. Due to the nature of malware, the identities of their authors are unknown. Attacker operations are known to be hard to attribute and rarely adjudicated, even if strong evidence can be found [2, 1]. This presents a major challenge to establishing a fully trustworthy ground truth. With this in mind, *BinChar* considers each family as an author or set of authors who has similar characteristics. The statistics are described in Table 4. We label functions by using existing tools: IDA pro and Paradyn for labeling library-related functions. While we use BinComp [26] tool for labeling compiler-related functions. These samples contain different variants of the same malware so we assume that these variants are have similar authorship characteristics written by the same set of authors. The number of compiler functions are obtained based on BinComp tool [26], while the fifth column shows the number of library functions acquired by F.L.I.R.T technology [20] and Paradyn. According to Table 4, we can observe that the percentage of compiler functions is quite high. For instance, the percentage of compiler functions in `Lollipop` family is *30%*.

Cluster Quality. One of the most challenging task is to assess the quality of the results that are produced by a clustering algorithm [40]. To tackle such challenge, we can quantify the number of clusters, the average number of samples per cluster, the relative sum of all pairwise distances for a cluster, or we choose a few clusters and manually verify that the samples in these clusters are similar. Since we have access to the used samples. We manually verify the correctness of our clusters.

The Correctness of Clusters. To assess the quality of our cluster algorithm results, we introduce two metrics, correct cluster (CC) and wrong clusters (WC). The goal of CC is to measure how well a clustering algorithm can distinguish between samples that are have different author characteristics. While WC measures the percentage of clustering functions from different classes to the same cluster.

We then apply our tool to cluster functions according to their similar authorship characteristics by using standard k-mean as we described earlier. Then we manually analysis the obtained clusters to classify them to correct/wrong clusters as shown in Table 5. Through our analysis for the obtained clusters, we have found different set of characteristics including the usage of variables, the usage of security rules, the way of configuration, memory allocation habits, etc.

6. Characterizing the *Mirai* Botnet

One challenge in applying *BinChar* to real world malware is the lack of ground truth concerning the attribution of authorship due to the nature of malware. Also, whether a malware package is created by an individual or an organization is generally unconfirmed. Those limitations partially explain the fact that few research efforts have been seen on this subject. We present in this section a case study by applying *BinChar* to Mirai botent and compare the extracted characteristics with 360,000

Table 5: Clustering results based on the features used in existing systems. (TC): the total number of clusters, (CC): the percentage of correct clusters, (WC): the percentage of wrong clusters.

Malware	Optimization			Structural			Knowledge			Expertise			Overall			BinChar		
	TC	CC	WC	TC	CC	WC	TC	CC	WC	TC	CC	WC	TC	CC	WC	TC	CC	WC
`Ramnit`	145	60%	13%	110	47%	25%	208	58%	17%	150	61%	12%	210	75%	22%	345	82%	10%
`Lollipop`	90	75%	18%	185	59%	28%	220	72%	21%	178	74%	25%	198	68%	20%	295	78%	18%
`Kelihos`	41	88%	8%	17	90%	4%	75	54%	25%	80	75%	19%	76	80%	17%	145	89%	14%
`Vundo`	200	62%	14%	89	48%	38%	384	79%	24%	215	82%	18%	289	72%	27%	544	90%	7%
`Simda`	52	64%	21%	41	92%	5%	109	82%	19%	77	81%	14%	100	67%	24%	124	84%	15%
`Tracur`	44	89%	9%	53	83%	12%	124	51%	40%	68	79%	14%	80	87%	11%	105	82%	10%
`Obfuscator.ACY`	30	78%	21%	45	74%	24%	89	89%	11%	114	80%	17%	97	75%	15%	109	84%	14%
`Gatak`	29	67%	20%	51	87%	12%	79	78%	16%	105	84%	12%	98	80%	17%	127	89%	10%

samples collected from various databases where 40% of these samples were packed and obfuscated and we unpacked them in our security laboratory. Some of our samples include `Bunny`, `Babar`, `Stuxnet`, `Flame`, `Duqu`, `Zeus`, `Citadel`, `zaccess`, `encpk`, `sality`, etc.

Table 7: Statistics about the Mirai binaries used in the case study

Sample	Size (KB)	No. of binary functions	Platform
File1.ARM.ELF	1.14	12	ARM
Mirai.arm	5	79	ARM
File2.MIPS.ELF	4	19	MIPS
mirai.sh4	3	11	SuperH
mirai.sparc	3	13	SPARC
mirai.x86	7	31	x86
mirai.ppc	3	14	PowerPC

An overview. Mirai is a DDoS botnet that has discovered by MalwareMustDie team in August 2016. It has been considered as one of the biggest DDoS attacks on Internet which causes to shut down a major parts of Internet. It has been created using ELF binaries. The statistics of Mirai binaries is introduced in Table 7. As shown in Table 7, the samples have different platform architectures such as MIPS and PowerPc. *BinChar* can handle multiple architectures since it uses some features (Optimization characteristics and Overall characteristics) that are cross-architecture independent.

Findings. Our tool is able to find some authorship characteristics links between Mirai botnet and very few sample from our dataset. *BinChar* extracts various characteristics that are related to the author(s) of Mirai botnet. It finds the following: the use of all capital letters for *config* in XML; the use of a common approach to managing functions; the characteristics of opening and terminating processes; the passing of primitive types by value, but the passing of objects by reference; the use of network resources rather than file resources; creating configurations using mostly config files; and the use of semaphores and locks. More specifically, we have found different set of characteristics.

Table 8: The number of extracted features from Mirari botnet. (●) means that cannot be extracted

Sample	**Characteristics**				
	Structure	Optimization	Knowledge	Expertise	Overall
File1.ARM.ELF	450	180	●	●	3800
Mirai.arm	721	225	●	●	3400
File2.MIPS.ELF	400	190	●	●	2900
mirai.x86	950	428	215	147	4500
mirai.ppc	145	175	●	●	3010
mirai.sparc	200	215	●	●	3700
mirai.sh4	315	160	●	●	3150

Table 6 presents examples of authorship characteristics. It can be seen that certain characteristics, such as expertise characterises including control command strings and dynamic memory allocation. Also, some examples on knowledges characteristics including the use of API calls,

Table 6: Subset of characteristics found by *BinChar* in certain malware families. (✓) means the characteristic is found in Mirai botnet and it is found in that malware family, whereas (✗) means the characteristic is found in Mirai botnet and it was not found in that malware family

Characteristics	`Ramnit`	`Encpk`	`Kelihos`	`Zaccess`	`Vobfus`	`Sality`	`Gatak`	`Asterope`	`Turla`
Error messages	✓	✗	✓	✓	✗	✗	✓	✗	✗
Control commands	✗	✗	✗	✗	✗	✗	✗	✓	✓
Preference in keywords	✗	✓	✗	✗	✗	✓	✓	✗	✗
Reusing free code	✗	✗	✓	✓	✗	✗	✓	✗	✗
Using specific API	✓	✓	✓	✓	✓	✓	✓	✓	✓
Using specific encryption	✓	✗	✗	✓	✗	✗	✓	✗	✓
Modifying constants	✗	✗	✗	✗	✗	✗	✗	✗	✓
Dynamic memory allocation	✗	✗	✗	✗	✗	✗	✓	✗	✗
Runtime protection strategies	✗	✗	✓	✓	✗	✗	✗	✓	✓
Compiler security features	✗	✗	✗	✓	✓	✓	✗	✓	✗
Files vs memory	✓	✓	✓	✓	✓	✓	✓	✓	✓
Global variables vs. local variables	✗	✗	✗	✓	✗	✗	✗	✗	✗
Accessing closed files	✗	✓	✓	✗	✗	✗	✗	✗	✗
Accessing freed memory	✗	✗	✗	✗	✓	✓	✗	✗	✗
Function overloading	✓	✓	✓	✗	✗	✗	✗	✓	✓
Parameter ordering	✗	✗	✓	✓	✗	✓	✗	✗	✗

runtime protection strategies, etc. We have found another characteristics that belong to code traits such as accessing closed files or modifying constants. We studied the number of features extracted from Mirai botnet samples (Table 8). For example, the number of features that are related to structure characteristics authors is 175 extracted from mirai.ppc. As shown, there are some features could not be extracted from files that are not x86-based architecture. We leave this issue for future work venue. We have studied the impact of each authorship characteristics on accuracy (Figure 3). For example, the F_1 score is 0.7 between Mirai.x86 and Ramnit when we compare them based on the knowledge characteristics. While the average F_1 score is 0.42 between Ramnit and Mirai.x86. As shown, the highest F_1 score is 0.83 between Mirai.x86 and Vobfus when we compare them based on optimization characteristics.

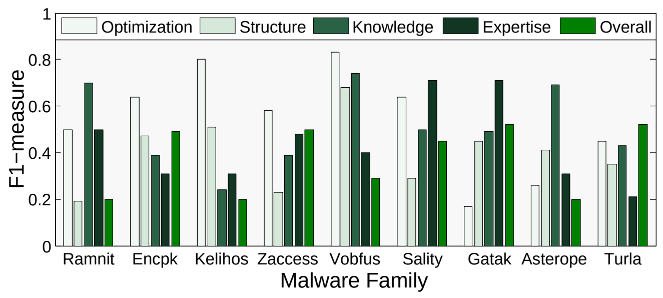

Figure 3: The F1 score between Mirai.x86 and the malware families in Table 6

Measuring the degree of similarity between ground truth datasets and malware binaries. As another verification of the correctness of the findings, we measured the degree of similarity between the Mirai Botnet here and other datasets for which we have the ground truth (e.g., Google code jam) to see how likely such a degree of similarity could come from shared authorship characteristics. The goal of computing the degree of similarity is to determine whether the authorship characteristics found in the Mirai botnet are present to the same degree in conventional binaries, which will reveal whether these characteristics are indeed specific to malware writers. To provide an even more convincing verification, we computed the similarity scores between related Mirai botnet samples and the rest of the available dataset. *BinChar* found a similarity of 7% with those characteristics in the Google code jam dataset and 17% with those characteristics in the Github dataset. We believe that one of the main reasons for the high similarity is that the programmers participating in the Github may have greater expertise, more extensive background knowledge, and better skills than the programmers who participate in Google code jam.

Measuring similarity between authorship characteristics in malware binaries. In this section, the goal is to assess the similarity between malware binaries by reporting the similarity in terms of authorship characteristics (Table 9).

Table 9: Similarity between authorship characteristics found in Mirai botnent and characteristics found in malware binaries

	Bunny	Babar	Stuxnet	Flame	Zeus	Citadel	Mirai
`Bunny`	-	61%	14%	8%	11%	13%	1.2%
`Babar`	61%	-	9%	10%	2%	8%	2.09%
`Stuxnet`	14%	9%	-	74%	19%	10%	0.96%
`Flame`	8%	10%	74%	-	14%	6%	0.71%
`Zeus`	11%	2%	19%	14%	-	80%	0
`Citadel`	13%	8%	10%	6%	80%	-	0.09%
`Mirai`	1.2%	2.09%	0.96%	0.71%	0	0.09%	-

We compare similarity between the author characteristics extracted from Mirai botnet 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 [41, 1, 2]. We observed that the similarity between the authorship characteristics found in `Mirai` botnent and characteristics found in malware binaries is very low. For instance, the similarity between `Mirai` and `Zeus` is 0. In the meantime, our tool is able to find common authorship characteristics among other malware families. For example, the similarity between `Bunny` and `Babar` is about 60%. This finding supports the claim that this pair originates from the same set of authors [2].

7. Limitations and Concluding Remarks

Our work has certain limitations. First, the system is unlikely to remain accurate if the authors used advanced obfuscation techniques to evade detection. Second, although we have tested our work on relatively large datasets, our dataset can still be enriched in terms of both scale and scope. Third, the features used by *BinChar* are static-based; as such, *BinChar* cannot detect characteristics that require dynamic features. Fourth, there is a room for performance improvement by including some techniques such as Map-reduced methods. Finally, since *BinChar* currently supports the x86 ISA, other ISAs such as ARM should also be considered. Our future work aims to extend *BinChar* to tackle these limitations.

To conclude, we have presented the first known effort on characterizing the author of binary code based on personnel characteristics. Previous existing works have only employed artificial datasets, whereas we included more realistic datasets. We also applied our system to known malware. In summary, our system demonstrates superior results on more realistic datasets and real malware and can detect the presence of multiple authors.

8. Acknowledgments.

The authors thank the anonymous reviewers for their valuable comments. Any opinions, findings, and conclusions or recommendations expressed in this material are

those of the authors and do not necessarily reflect the views of the sponsoring organizations.

References

[1] Techniqal report, Resource 207: Kaspersky Lab Research proves that Stuxnet and Flame developers are connected, http://www.kaspersky.com/about/news/virus/2012/.

[2] Big Game Hunting: Nation-state malware research, BlackHat, https://www.blackhat.com/docs/us-15/materials/us-15-MarquisBoire-Big-Game-Hunting-The-Peculiarities-Of-Nation-State-Malware-Research.pdf (2015).

[3] Citizen Lab, `https://citizenlab.org/`, university of Toronto, Canada (2015).

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

[5] G. Frantzeskou, Source code authorship analysis for supporting the cybercrime investigation process (2004) 470–495.

[6] Q. C. Taylor, J. E. Stevenson, D. P. Delorey, C. D. Knutson, Author entropy: A metric for characterization of software authorship patterns, in: Third International Workshop on Public Data about Software Development ((WoPDaSD08), 2008, p. 6.

[7] D. R. Woldring, P. V. Holec, B. J. Hackel, Scaffoldseq: Software for characterization of directed evolution populations, Proteins: Structure, Function, and Bioinformatics 84 (7) (2016) 869–874.

[8] S. Alrabaee, N. Saleem, S. Preda, L. Wang, M. Debbabi, Oba2: An onion approach to binary code authorship attribution, Digital Investigation 11 (2014) S94–S103.

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

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

[11] X. Meng, Fine-grained binary code authorship identification, in: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, 2016, pp. 1097–1099.

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

[13] E. C. R. Shin, D. Song, R. Moazzezi, Recognizing functions in binaries with neural networks., USENIX, 2015.

[14] L. lab, Deep Learning Tutorial Release 0.1, http://deeplearning.net/tutorial/deeplearning.pdf, university of Montreal (2014).

[15] Y. Wei, W. Xia, M. Lin, J. Huang, B. Ni, J. Dong, Y. Zhao, S. Yan, Hcp: A flexible cnn framework for multi-label image classification, IEEE transactions on pattern analysis and machine intelligence 38 (9) (2016) 1901–1907.

[16] Y. Nagano, R. Uda, Static analysis with paragraph vector for malware detection, in: Proceedings of the 11th International Conference on Ubiquitous Information Management and Communication, ACM, 2017, p. 80.

[17] M. Wagner, A. Rind, N. Thür, W. Aigner, A knowledge-assisted visual malware analysis system: Design, validation, and reflection of kamas, computers & security 67 (2017) 1–15.

[18] N. Moran, J. Bennett, Supply Chain Analysis: From Quartermaster to Sun-shop, Vol. 11, FireEye Labs, 2013.

[19] PEfile:, `http://code.google.com/p/pefile/`, accessed on Nov, 2016 (2012).

[20] HexRays: IDA Pro, https://www.hex-rays.com/products/ida/index.shtml (2011).

[21] F. Farnstrom, J. Lewis, C. Elkan, Scalability for clustering algorithms revisited, ACM SIGKDD Explorations Newsletter 2 (1) (2000) 51–57.

[22] R. Tate, M. Stepp, Z. Tatlock, S. Lerner, Equality saturation: a new approach to optimization, in: ACM SIGPLAN Notices, Vol. 44, ACM, 2009, pp. 264–276.

[23] J.-B. Tristan, P. Govereau, G. Morrisett, Evaluating value-graph translation validation for llvm, ACM Sigplan Notices 46 (6) (2011) 295–305.

[24] E. Schkufza, Stochastic program optimization for x86 64 binaries, Ph.D. thesis, STANFORD UNIVERSITY (2015).

[25] R. Sharma, Data-driven verification, Ph.D. thesis, Stanford University (2016).

[26] A. Rahimian, P. Shirani, S. Alrbaee, L. Wang, M. Debbabi, Bincomp: A stratified approach to compiler provenance attribution, Digital Investigation 14 (2015) S146–S155.

[27] H. Huang, A. M. Youssef, M. Debbabi, Binsequence: fast, accurate and scalable binary code reuse detection, in: Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, ACM, 2017, pp. 155–166.

[28] T. J. McCabe, A complexity measure, IEEE Transactions on software Engineering (4) (1976) 308–320.

[29] D. Kirat, L. Nataraj, G. Vigna, B. Manjunath, Sigmal: A static signal processing based malware triage, in: Proceedings of the 29th Annual Computer Security Applications Conference, ACM, 2013, pp. 89–98.

[30] L. Nataraj, S. Karthikeyan, G. Jacob, B. Manjunath, Malware images: visualization and automatic classification, in: Proceedings of the 8th international symposium on visualization for cyber security, ACM, 2011, p. 4.

[31] L. Nataraj, D. Kirat, B. Manjunath, G. Vigna, Sarvam: Search and retrieval of malware, in: Proceedings of the Annual Computer Security Conference (ACSAC) Worshop on Next Generation Malware Attacks and Defense (NGMAD), 2013.

[32] E. B. Karbab, M. Debbabi, S. Alrabaee, D. Mouheb, Dysign: dynamic fingerprinting for the automatic detection of android malware, in: Malicious and Unwanted Software (MALWARE), 2016 11th International Conference on, IEEE, 2016, pp. 1–8.

[33] G. Huang, Z. Liu, K. Q. Weinberger, L. van der Maaten, Densely connected convolutional networks, arXiv preprint arXiv:1608.06993.

[34] X. Glorot, A. Bordes, Y. Bengio, Deep sparse rectifier neural networks, in: Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, 2011, pp. 315–323.

[35] A. L. Maas, A. Y. Hannun, A. Y. Ng, Rectifier nonlinearities improve neural network acoustic models, ICML, 2013.

[36] B. Kolosnjaji, A. Zarras, G. Webster, C. Eckert, Deep learning for classification of malware system call sequences, in: Australasian Joint Conference on Artificial Intelligence, Springer, 2016, pp. 137–149.

[37] J. Saxe, K. Berlin, Deep neural network based malware detection using two dimensional binary program features, in: Malicious and Unwanted Software (MALWARE), 2015 10th International Conference on, IEEE, 2015, pp. 11–20.

[38] The GitHub repository, https://github.com/ (2016).

[39] The Google Code Jam., `http://code.google.com/codejam/` (2008-2015).

[40] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, E. Kirda, Scalable, behavior-based malware clustering., in: NDSS, Vol. 9, 2009, pp. 8–11.

[41] Techniqal report, Mcafee, www.mcafee.com/ca/resources/wp-citadel-trojan-summary.pdf (2011).