BinEye: Towards Efficient Binary Authorship Characterization Using Deep Learning

Saed Alrabaee[1,2], ElMouatez Billah Karbab[2], Lingyu Wang[2], and Mourad Debbabi[2]

[1] United Arab Emirates University, Al Ain, Abu Dhabi, UAE
salrabaee@uaeu.ac.ae
[2] CIISE, Concordia University, Montreal, QC, Canada
{s_alraba,e_karbab,wang,debbabi}@encs.concordia.ca

Abstract. In this paper, we present BinEye, an innovative tool which trains a system of three convolutional neural networks to characterize the authors of program binaries based on novel sets of features. The first set of features is obtained by converting an executable binary code into a gray image; the second by transforming each executable into a series of bytecode; and the third by representing each function in terms of its opcodes. By leveraging advances in deep learning, we are then able to characterize a large set of authors. This is accomplished even without the missing features and despite the complications arising from compilation. In fact, BinEye does not require any prior knowledge of the target binary. More important, an analysis of the model provides a satisfying explanation of the results obtained: BinEye is able to auto-learn each author's coding style and thus characterize the authors of program binaries. We evaluated BinEye on large datasets extracted from selected open-source C++ projects in GitHub, Google Code Jam events, and several programming projects, comparing it with previous approaches. Our experimental results demonstrate that BinEye characterizes a larger number of authors with a significantly higher accuracy (above 90%). We also employed it in the context of several case studies. When applied to Zeus and Citadel, BinEye found that this pair might be associated with common authors. For other packages, BinEye demonstrated its ability to identify the presence of multiple authors in binary code.

1 Introduction

When analyzing malware binaries, reverse engineers often pay particular attention to malware author style characteristics for several reasons [1]. First, reports from anti-malware companies indicate that finding the similarities among malware code styles can aid in developing profiles for malware families [2]. Second, recently released reports by Citizen Lab [3, 4] show that malware binaries written by authors having the same origin share similar styles. Third, many malware packages might have been written only by authors with the specialized knowledge required for dealing with particular resources, for example, the malware targeting the SCADA system; thus, this insight provides a critical clue for establishing the level of expertise of an author. Last, clustering binary functions

based on common style characteristics or author origin may help reverse engineers to identify the group of functions that belong to a particular malware family or to 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 such as malware analysis because the source code for malware is not always available. However, in automating binary authorship attribution, two challenges are typically encountered: the binary code lacks many abstractions (e.g., variable names) that are present in the source code; and the time and space complexities of analyzing binary code are greater than those for the corresponding source code. Although significant efforts have been made to develop automated approaches for source code authorship attribution [5, 6], these often rely on identifying features that will likely be lost in the strings of bytes representing binary code after the compilation process; these features include variable and function naming, comments, and space layout.

To this end, there have been a few attempts at binary authorship attribution. 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 [7, 8, 9, 10]. These approaches are affected by refactoring techniques, compilers, compilation settings, or code transformation methods. Further, these approaches are not applied to real malware binaries. Recently, the feasibility of authorship attribution for malware binaries was discussed at the BlackHat conference [3]. It was concluded that it is possible to group malware binaries according to authorship styles based on a set of features. However, the process is not automated and requires considerable human intervention.

Problem statement. We address the characterization of the author of a binary code based on its style and assume that the reverse engineer has access to a set of binary code samples each of which is attributed to one of a set of candidate authors. The reverse engineer may proceed by converting each labeled sample into a feature vector and training a classifier from these vectors using machine learning techniques. The classifier can then be used to assign the unknown binary code to the most likely author. Since we have a repository of known authors, we treat this part of the problem as a closed-world [9] supervised machine learning task. It can also be viewed as a multi-class problem in which the classifier calculates the most likely author for the unknown binary. A complication arises when the binary code is written by multiple authors, as is common for malware binaries [3] or projects in GitHub. In such instances, the reverse engineer is interested in clustering the functions written by the same author.

Solution overview. To address the aforementioned issues, this paper presents `BinEye`, an innovative framework that leverages recent advances in deep neural networks to build a framework robust to changes in the compiler or to code transformation. We used large collections of sample binary code from each author compiled with different compilers/compilation settings to train a deep learning network to attribute the authors in a robust way.

`BinEye` encompasses three deep learning networks, so its input comprises three sets of features. The first are the opcode sequences, which make it possible

to detect the styles of programmers according to their coding traits. The second are the function invocations extracted from assembly files, such as open-file and close-file. The underlying intuition is that one author may rely on a set of API functions, but another may use a different set of APIs. Specifically, we map each API call in the sequence invocation to a fixed length high-dimensional vector that semantically represents the method invocation and replace the sequence of the assembly API calls by a sequence of vectors. Afterward, we feed the sequence of vectors to a deep neural network (i.e., a convolutional neural network) with multiple layers. A similar approach is applied to the opcodes. The third set of features is obtained by converting the binary file into a gray scale image and then leveraging past efforts to identify transformed image features [11, 12]. The latter features are based on both the executable structure and the binary content of the executable. The intuition is that binary files produced by authors with similar style characteristics will generate similar image texture patterns. Compared to existing features such as N-grams, the image processing-based features are less vulnerable to the changes introduced by refactoring tools or code transformation [11]. We note some attractive characteristics of convolutional neural networks. For example, the nodes of CNNs can act as filters over the input space and can reveal strong correlations in the binary code. Also, in the classification process, a CNN is considered a fast neural network compared to other neural networks such as recurrent neural networks [13].

Having different votes from our three neural networks might be a sign of a binary code written by multiple authors. In this case, we perform authorship clustering comply with the following steps: First, the binaries are disassembled using IDA Pro [14]. However, since IDA fails to identify the boundaries of functions, we employ Nucleus [15] to perform compiler-agnostic function detection. Next, compiler-related functions and library-related functions are filtered out. After that, we extract opcode sequences from the user-related functions using a sliding window with a size of at most b opcodes, where b is a user-specified threshold. The resulting opcode sequences are then ranked based on mutual information, and finally, the ranked opcodes are used to cluster functions according to similarities in the author style characteristics. A standard clustering algorithm, k-means [16], is used for this purpose.

Contributions. Our contributions are summarized below.

- We design `BinEye`, a novel approach for authorship characterization. Based on deep learning, our approach discovers patterns that are related to author styles. `BinEye` is the first attempt to investigate the power of CNNs for binary authorship attribution.
- We evaluate `BinEye` on a set of binaries by more than 1,500 authors compiled with different compilers and compilation settings. The results show that `BinEye` achieves high precision: 98%. Moreover, it is also robust in its ability to tolerate the noise injected by refactoring tools, code transformation methods, and other code transformations arising from the use of different compilers and optimization speed levels.
- Finally, `BinEye` is among the first approaches toward applying automated authorship characterization to real malware binaries. The results provide

evidence and insights about malware authorship characterization that match the findings of domain experts obtained through manual analyses.

2 System Overview

The architecture of `BinEye` is illustrated in Figure 1.

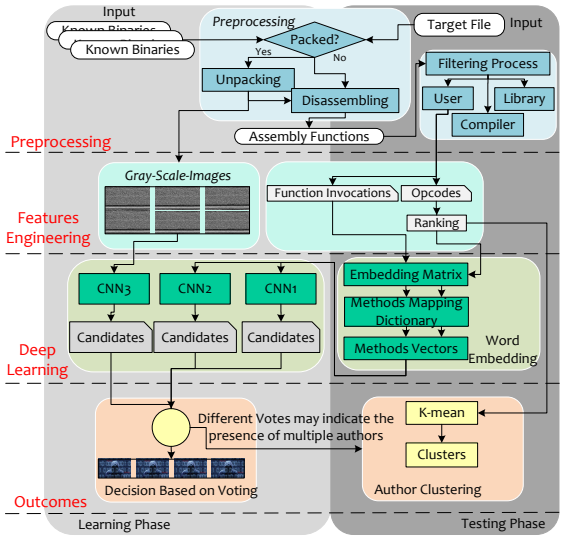


Fig. 1: `BinEye` Architecture

As shown, the four main components are: (i) Preprocessing component where PEfile [17] 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 [14]. (ii) Feature engineering which deals with a different set of features that are related to the authorship attribution. This component is able to accept either an executable file or an assembly file as input. In the case of executable file, *BinEye* converts executable file to a gray-scale images. These images are passed to CNN3 that does automatically features engineering. In the case of assembly file, BinEye extracts the following features: (1) opcode sequences that are ranked through mutual information; (2) function invocations. Subsequently, we map each opcode in the sequence to a fixed length high-dimensional vector that semantically represents the opcode and replace the sequence of the binary function by a sequence of vectors. Afterward, we feed the sequence of vectors to a neural network with multiple layers. Then, it is passed to CNN1 while we apply the same to function invocations and passed the engineered features to CNN2. The three CNNs are trained by these sets of features. Once we have a target binary, each CNN will return a set of candidates according to the training features. These candidates are passed to decision component that decides based on the majority voting

of three CNNs. In case of equal votes, we perform the following steps for clustering process: (1) Filtration task is performed on the assembly functions that filters out compiler-related functions and library-related function. (2) Opcode sequences are then extracted from the user-related functions using a sliding window with a size of at most b opcodes, where b is a user-specified threshold. (3) These opcode sequences are ranked through mutual information. Finally, (4) the ranked opcodes are used to cluster functions according to the similarity of the author's style characteristics of the binary code. Standard clustering algorithm (k-means) [16] is used for this purpose.

3 BinEye: Design Overview

3.1 Features Processing

Opcode Sequences. `BinEye` workflow starts by extracting the sequences of opcode. Our goal is to discover patterns that might discover the author style such as the preference in using keywords, compilers, resources, etc. It also captures the author's preference of using different keywords or resources. We only consider groups of such preferences with equivalent or similar functionality in order to avoid functionality-dependent features. For instance, keyword type preferences for inputs (e.g., using `cin`, `scanf`), preferences of using particular resources or a specific compiler, and the manner in which certain keywords are used can serve as further indications of an author's habits.

Function Invocations. `BinEye` starts by extracting the sequences of API calls from user-related functions. Our goal is to formalize the assembly to keep the maximum raw information with minimum noise. We require a manual categorization of APIs to correlate them to author style. For this reason, we treat assembly functions as a sequence of API calls. We consider all the API calls with no filtering, where the order is part of the information we use to characterize the author.

Binary Image Features Extraction. We take the binary file content as a vector of bytes. This vector is transferred into a two-dimensional matrix of fixed width d. In other words, the first d bytes go to the first row of the matrix, and the n^{th} group of d bytes goes to the n^{th} row of the matrix. This approach is similar to the malware visualization techniques proposed in [11, 18]. The two-dimensional matrix, after a necessary padding, is now considered as a digital gray-scale image, which is "resized" into a square image of width s for efficient computation. We use a fixed width transformation as proposed in [11] to maintain the consistency in the texture produced as a result of this transformation. As long as the same width is used for each binary file, the choice of the width does not affect the similarity of the texture produced among similar executables. The methodology for computing the texture-based features is described as follows: We compute the features based on GIST descriptors [12]. The descriptors are computed by first filtering the image in various frequency sub-bands and then computing local block statistics on the filtered images. In image processing terminology, the functions modeling the filters are often referred to as Gabor Wavelets [19]. The

image is then filtered using this filter bank to produce k filtered images. Each filtered image is further divided into BxB sub-blocks and the average value of a sub-block is computed and stored as a vector of length $L = B^2$. For more details we refer the reader to [11]. We test the intuition of using image features for binary authorship by modeling such images in a controlled experiment with source code (ground truth). We observe that different image regions usually represent the binary sections: `.text`, `.rdata`, `.data`, and `.rsrc`. Also, images could reveal `ASCII` text, initialized data, uninitialized data, etc. Moreover, we observe in our experiment that the image width varies according to the file size but the image texture remains same in case of different variants of the same binary file. Further, we test the image features against packed binaries. We observe through our experiment that when the binary is packed, there are two types of compression: low and high compression. The image-related features still do a good job if the binary compression is low, such as with Winupack packing tool; however, when binary compression is high, such as with UPX, the images texture will look different. It is worthy to mention that when unpacked author binaries with several similar variants are packed with a specific packer, then the images of the newly packed binaries (of the same author) are also similar. We will add such details in the final version of our research to further clarify this point.

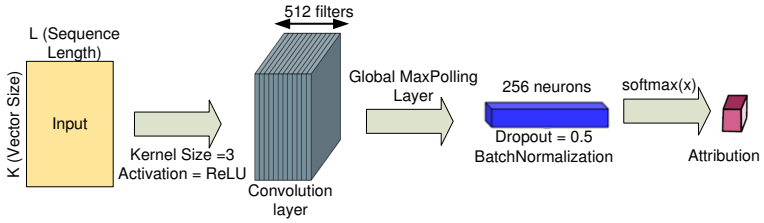


Fig. 2: `BinEye` Neural Network Architecture

3.2 Deep Learning Process

We follow the same process in [20].

1- Discretization of Opcode/function invocations Sequences In this step, we discretize the sequences of opcodes/function invocations (e.g., API) sequences that are in a binary code (Algorithm 1). More precisely, we replace each API keyword with an identifier, resulting in a sequence of numbers. We also build a dictionary that maps each API call to its identifier. Notice that in the current implementation, the mapping dictionary is deployed with the learning model to map the API calls of the analyzed binaries. In the deployment, we consider a big dataset that covers most of the API calls. Also, if we have unknown API calls, we replace them with fixed identifiers.

2- Unification of the Sequences' Size The length of the sequences varies from one binary to another. Hence, it is important to unify the length of the sequences. There are two cases depending on the length of the sequence and the hyper-parameter. We choose a uniform sequence size as follows: (i) If the length of a

Algorithm 1: Discretization

```
Input  : OSeq: Opcode Sequence
         MapDict: Mapping Dict
Output: DSeq: Discrete Sequence
1  begin
2      DSeq = EmptyList();
3      foreach P ∈ OSeq do
4          if m ∈ MapDict.Keys() then
5              Dvalue ← MapDict[P];
6              DSeq.Add(Dvalue);
7          else
8              length = size(MapDict);
9              Dvalue ← length + 1;
10             MapDict[m] = Dvalue;
11             DSeq.Add(Dvalue);
12     return DSeq, MapDict;
```

given sequence is greater than the uniform sequence size L, we take only the first L items to represent the binary. (ii) In case the length of the sequence is less than L, we pad the sequence with zeros. It is important to mention that the uniform sequence size hyper-parameter has an influence on the accuracy of `BinEye`. A simple rule is that the larger is the size, the better is, but this will require a lot of computation power and a long time to train the neural network.

3- Generation of the Semantic Vectors. The identifier in the sequences needs to be shaped to fit as input to our neural network. This could be solved by representing each identifier by a vector. The question that arises is *how are such vectors produced?* A straightforward solution is to use one-hot vectors, where a vector has one in the interface value row, and zero in the rest. Such a vector is very sparse because its size is equal to the number of API calls, which makes it impractically and computationally prohibitive for the training and the deployment. To address this issue, we resort to a dense vector that uses a continuous space. These vectors are semantically related, and we could express their relation by computing a distance. The smaller the distance is, the more related the vectors are (i.e., the API calls). We describe word embedding in later Section. The output of this step is sequences of vectors for each binary file that keeps the order of the original API calls; each vector has a fixed size M (hyper-parameter).

4- Prediction using a Neural Network The neural network is composed of several layers. The number of layers and the complexity of the model are hyper-parameters. However, we aim to keep the neural network model as simple as possible to gain in the execution time during its deployment. In our design, we rely on the convolution layers [21] to automatically discover the pattern in the assembly functions. The input to the neural network is a sequence of vectors,

i.e., a matrix of $L \times M$ shape. In the training phase, we train the neural network parameters (layers weight) based on the binary vector sequence and its labels. In the deployment phase, we extract the sequence of bytes and use the embedding model to produce the vector sequence. Finally, the neural network takes the vector sequence to decide about the given binary code.

3.3 Embedding Method

The neural network takes vectors as input. Therefore, we represent opcode/function invocation sequences as vectors. As a result, we formalize a binary file as a sequence of vectors with fixed size (L). We could use one-hot vector. However, its size is the number of unique patterns in our dataset. This makes such a solution not scalable to large-scale training. Also, the word embedding technique outperforms the results of the one-hot vector technique in our case [22], [23], [24]. Therefore, we seek a compact vector, which also has a semantic value. To fulfill these requirements, we choose the word embedding techniques, namely, word2vec [23] and GloVe [24]. Our primary goal is to have a dense vector for each sequence that keeps track of its contexts in a large dataset of binary code. Thus, in contrast with one-hot vectors, each word embedding vector contains a numerical summary of the opcode/function invocation sequences representation. Moreover, we could apply geometric techniques on the pattern vectors to measure the semantic relationship between their characteristics, i.e., developers tend to use certain API calls in the same context.

In our context, we learn these vectors from our dataset that contains different labels by using word2vec [23]. The latter is a computationally efficient predictive model from learning word embedding vectors, which are applied on the binary code. The output obtained from training the embedding word model is a matrix $K \times A$, where K is the size of the embedding vector, and A is the number of unique patterns. Both K and A are hyper-parameters; we use $K = 64$ in all our models. In contrast, the hyper-parameter A is a major factor in the accuracy of *BinEye*. The more unique patterns we consider, the more accurate and robust our model is. Notice that, our word embedding is trained along with the neural network, where we tune both of them for a given task such as detection. Despite that, it can be trained separately to generate the embedding word vector independently of the detection task. In the deployment phase, *BinEye* uses the word embedding model and looks up for each API call identifier to find the corresponding embedding vector.

4 BinEye Neural Network

Our proposed neural network is inspired by [22]. They employ a neural network for sentence classification task. The proposed architecture defenestrates high results and outperforms state-of-the-art works with a relatively simple neural network design. In this paper, we investigate the following questions: *Why could such a Natural Language Processing (NLP) model be efficient in Binary Authorship Characterization?* And *why do we choose to build it on top of this design [22]?* We formulate our answers as follows: i) Analyzing the text by employing NLP is a challenging field since there is an enormous number of vocabularies. We also have the same semantics with many combinations of words, which

we call the *natural language obfuscation*. In our context, we deal with sequences of opcode and API method calls (e.g., function convocation) and want to find the combination of patterns of opcode/method calls, which indicate the author style characteristics. We use the API method calls as they appear in the binary, i.e., there is a temporal relationship between API methods in basic blocks but we ignore the order among these blocks. By analogy to NLP, the basic blocks are the sentences and the API method calls are the words. It applies to the opcode sequences in assembly file. Further, the function (paragraph) is a list of basic blocks (unordered sentences). This task looks easier compared to the NLP one because of the huge difference in the vocabulary, i.e., the number of API method calls is significantly less than the number of words in natural language. Also, the combination in the NLP is much complex compared to API calls/opcode. ii) We choose to use this model due to its efficiency. Table 1 depicts the neural network architecture of BinEye attribution task.

Table 1: BinEye Neural Network

1 Layers	Options	Active
2 Convolution	Filter=512, FilterSize=3	ReLU
3 MaxPooling	-	-
4 FC	#Neurons=256, Dropout=0.5	ReLU
5 FC	#Neurons=1, # of authors in the training dataset	Softmax

There are multiple neurons, one for each author. As presented in Figure 2, the first layer is a convolution layer [22] with rectified linear unit (ReLU) activation function ($f(x) = max(0, x)$). Afterward, we use global max pool [22] and connect it to a fully-connected layer. Notice that in addition to Dropout [21] used to prevent over-fitting, we also utilize batch-normalization [21] to improve our results. Finally, we have an output layer, where the number of neurons depends on the attribution task.

4.1 Clustering Similar Functions

We use a standard clustering algorithm (k-means) [16] to group functions with similar author styles 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. In the following equation, μ_i denotes the mean of the feature values of each cluster.

$$\arg\min_{S} \sum_{i=1}^{k} \sum_{v_j \in S_i} \|v_j - \mu_i\|^2 \tag{1}$$

The parameter k may be estimated either by following standard practices in k-means clustering [16], or by beginning with one cluster and continually dividing the clusters until the points assigned to each cluster have a Gaussian distribution as described in [25].

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 are then presented. Subsequently, we evaluate the identification of the presence of multiple authors as well as the scalability of *BinEye*. Also, we have studied the impact of different CNN parameters on the *BinEye* accuracy. The impact of evading techniques is then studied. Finally, *BinEye* 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. A subset of the python scripts in our evaluation system is used in tandem with IDA Pro disassembler [14]. The Neo4j [26] graph database is utilized to perform complex graph operations such as k-graph (ACFG) extraction. Gephi [27] is used for all graph analysis functions (e.g., page rank) that are not provided by Neo4j. The MongoDB database is used to store extracted features according to its efficiency and scalability. For the sake of usability, a graphical user interface in which binaries can be uploaded and analyzed is implemented. 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 kernel size 3x3 for convolutional layers. We follow zero-padded for inputs to keep the feature-map size fixed [21]. We use 3x3 convolution followed by 4x4 average pooling as transition layers between two contiguous blocks. At the end of the last block, a global average pooling is performed and then a softmax classifier is attached. The feature-map sizes in the two blocks are 128x128, and 64x64, respectively. The initial convolution layer comprises 2k convolutions of size 7x7 with stride 2; the number of feature-maps in all other layers also follow from setting k. The GPU is TITAN X, RAM is 128 GB, and the CPU is Intel E5-2630.

5.2 Dataset

The utilized dataset is composed of several files from different sources, as described below: i) GitHub [28], where a considerable amount of real open-source projects are available; ii) Google Code Jam [29]; and iii) a set of known malware files representing a mixture of different families including the nine families provided in Microsoft Malware Classification Challenge [3]. Statistics about the dataset are provided in Table 2. To construct our experimental datasets, we compile the source code with different compilers and compilation settings to measure the effects of such variations. We use GNU Compiler Collection's gcc, g++, Clang, ICC, as well as Microsoft Visual Studio (VS) 2010, with different optimization levels.

Table 2: Statistics about the dataset used in the evaluation

Source	**# of authors**	**# of prog.**	**# of func.**
GitHub	600	9,650	1,900,000
Google Jam	1,300	21,500	2,165,120
Total	1,900	31,150	4,065,120

5.3 Evaluation Methodology

We have used the datasets introduced in Table 2 in our evaluation process. For each program in the datasets, we have author label. To construct a data representation suitable for learning and evaluation, we process the binaries in each corpus with the IDA Pro, ParseAPI in Paradyn, and Jakstab to obtain features that are related to characteristics. We eliminate statically linked library functions, compiler related functions, and other known binary code snippets or borrowed that are unrelated to the author styles. Our evaluation methodology involves both standard ten-fold cross-validation and random subset testing, depending on the experiment: For classification of the entire data set, we use ten-fold cross-validation. When evaluating how classification accuracy behaves as a function of the number of authors represented in the data, we randomly draw a subset of authors and use their programs in the test. The evaluation results are presented under the following metrics:

- *True positives* (TP): This metric measures the number of binaries that the system successfully able to identify their correct authors.
- *False negatives* (FN): This metric measures the number of binaries that the system incorrectly assigned to wrong authors.
- *False positives* (FP): This metric measures the number of binaries that the system incorrectly assigned to an author.
- *Precision* (P): It is the percentage of positive prediction, i.e., the percentage of the correct identified author out of all samples. $P = \frac{TP}{TP+FP}$
- *Recall* (R): It is the percentage of authors identified out of all samples. $R = \frac{TP}{TP+FN}$

5.4 Accuracy

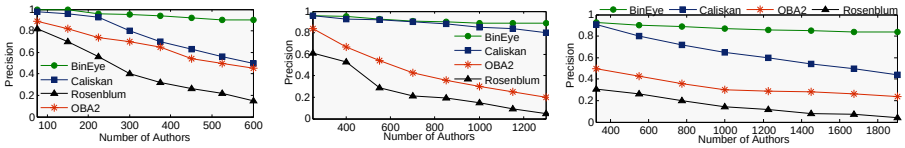
The purpose of this experiment is to evaluate the accuracy of characterizing the author of in program binaries.

Table 3: F-result

	Precision	**Recall**	F_1
CNN3	0.95	0.87	0.91
CNN2	0.89	0.86	0.87
CNN1	0.84	0.79	0.81
BinEye	0.98	0.91	0.94

Evaluation Settings. The evaluation of *BinEye* 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 *BinEye* and to compare it with existing methods, precision P and recall R measures are applied. Furthermore, since the application domain targeted by *BinEye* is much more sensitive to false positives than false negatives, we employ an F1-measure.
BinEye Accuracy. We first investigate the accuracy of our proposed system in attributing the author of program binaries based on author styles. The results

are reported in Table 3. The highest accuracy obtained by our tool is 0.94 when all features are together. Further, we can observe that the CNN3 (image features) returns the highest accuracy of 0.91. This is due to the fact that the author's knowledge, expertise, and styles may together form a unique image patterns. The second highest precision is 0.89 that obtained through CNN2. This is due to the fact that the author may use his expertise to implement a specific task by using specific API call and then leave a clue behind it. This clue might be captured through function invocations.



(a) (b) (c)

Fig. 3: Accuracy results of authorship attribution obtained by *BinEye*, Caliskan-Islam et al. [9], Rosenblum et al. [10], and OBA2 [7], on (a) Github, (b) Google Code Jam, and (c) Mixed.

Results Comparison. First, we compare *BinEye* with the existing authorship attribution methods [7, 9, 10]. We evaluate the authorship classification technique presented by Rosenblum et al. [10], whose source code is available at [30], although the dataset is not available. The source code of the proposed technique as well as the dataset by Caliskan-Islam et al. [9] are available at [31]. We have contacted the author of OBA2 [7] for getting the both dataset and the source code and we have them. Caliskan-Islam et al. present the largest scale evaluation of binary authorship attribution in the literature, which contains *600* authors with *8* training programs per author. Rosenblum et al. present a large-scale evaluation of *190* authors with at least *8* training programs, while Alrabaee et al. present a small scale evaluation of 5 authors with *10* programs for each. We compare our results with these methods by using the datasets mentioned in Table 2 except the malware dataset since the aforementioned methods are not designed for malware binaries. Figure 3 details the results of comparing the accuracy between *BinEye* and all other existing methods.

It shows the relationship between the accuracy (*precision*) and the number of authors present in all datasets, where the accuracy decreases as the size of author population increases. The results show that *BinEye* achieves better accuracy in determining the author of binaries based on authorship attribution in the case of GitHub dataset and mixed dataset. Taking all four approaches into consideration, the highest precision of authorship attribution is close to *100%* on the Github dataset with less than *200* authors, while the lowest accuracy is *2%* when *1900* authors are involved. *BinEye* also identifies the author of Google dataset with an average precision of *91%* while an average accuracy of *96%* when the dataset is GitHub. The main reason for this is due to the fact that the authors of projects in Github have no restrictions when developing projects. In

addition, the advanced programmers of such projects usually design their own class or template to be used in the projects. This makes the attribution process for such kind of authors are effective to identify them. The lower precision obtained by *BinEye* is approximately *84%* on a mixed dataset with *1900* authors. As shown in Figure 3 (b), Caliskan-Islam et al. [9] approach achieves highes accuracy among all. We believe that the reason behind Caliskan-Islam et al. approach superiority on Google Jam Code is that this dataset is simple and can be easily decompiled to source code.

We find that the accuracy of existing methods [9, 10] depends heavily on the application's domain. For example, in Figure 3, a good level of accuracy is observed for the Google data set, where the average accuracy is *90%* while the average accuracy is *70%* when the GitHub dataset is applied. Further, existing methods use disassembler and decompilers to extract features from binaries. Caliskan-Islam et al. [9] use a decompiler to translate the program into C-like pseudo code via Hex-Ray [32]. They pass the code to a fuzzy parser for C, thus obtaining an abstract syntax tree from which features can be extracted. In addition to Hex-Ray limitations [32], the C-like pseudo code is different from the original code to the extent that the variables, branches, and keywords are different.

For instance, we find that a function in the source code consists of the following keywords: (`1-do, 1-switch, 3-case, 3-break, 2-while, 1-if`) and the number of variables is two. Once we check the same function after decompiling its binary, we find that the function consists of the following keywords: (`1-do, 1-else/if, 2-goto, 2-while, 4-if`) and the number of variables is four. This will evidently lead to misleading features, thus increasing the rate of false positives.

5.5 False Positive Rate

We investigated the false positives to understand situations in which *BinEye* is likely to make incorrect attribution decisions. The average false positive rate of 0.02% is very low and could be neglected. The reason for this low false positive rate is that *BinEye* uses a multi CNNs networks that is based on majority votes. 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.

5.6 Run-Time Performance

In this section, we evaluate the efficiency of `BinEye`, i.e., the runtime during the deployment phase. We divide the runtime into two parts: i) *Preprocessing time*: the required time to extract and pre-process the features. ii) *Detection time:* time needed to make the prediction about a given feature vectors. We analyze the detection time on the model complexity of different hardware. Table 4 depicts the average preprocessing and detection time, related to each hardware. The server machines spend, on average, 7 seconds in the preprocessing time, which is very acceptable for production. It is worthy to mention that we do not optimize the current preprocessing workflow. Further, Table 4 presents the detection time on average that is related to each hardware.

Table 4: Run-Time vs. Hardware

	Preprocess			**Prediction**		
	Server GPU	Server CPU	Desktop	Server GPU	Server CPU	Desktop
Time (s)	7.25	9.4	18.6	1.75	2.35	3.85

As shown, the detection time is almost close to each other in all hardware. Also, the detection time is very low for all the hardware. As for the Desktop, the detection time is 5.15 seconds. Moreover, the pre-process time takes 25% more than the prediction time in case of server GPU. Here, we ask the following question: *Which part in the preprocessing needs optimization?* To answer this question, we measure the preprocessing time for the following tasks: (I) the disassembly time using IDA Pro; (II) the filtration time; (III) the features extraction; (IV) the embedding generation time from the extracted features. The average processing time for each task is shown in Table 5.

Table 5: Processing Time for Different Tasks

Task	**Preprocess**		
	Server GPU	Server CPU	Desktop
Disassembly	1.5	1.8	2.6
Filtration	1.75	1.9	3.0
Extraction	2.5	3.3	7.3
Embedding	1.5	2.4	6.7

As shown in Table 5, it is clear that the preprocessing time for disassembly and filtration tasks could be neglected. It is also the embedding task could be neglected. We observe that the feature extraction task takes the most time.

5.7 Impact of Compilers and Compilation Settings

We are further interested to study the impact of different compilers and compilation settings on the precision of our proposed system. We perform the following tasks: (i) testing the ability of *BinEye* when identifying the author from binaries compiled with the same compiler, but different compiler optimization levels. Specifically, we use binaries that were compiled with GCC/VS on x86 architecture using optimization levels O2 and O3. In this test, the average precision remains same (93%). (ii) We use a different configuration to identify the author of program compiled with both a different compiler and different compiler optimization levels. Specifically, we use programs compiled for x86 with VS -O2 and GCC -O3. In this test, the average precision is 91.9%. We also redo the test for the same binaries compiled with ICC and Clang compilers. The average precision remains almost the same 92.8%.

5.8 Impact of Evading Techniques

The adversary may use existing evading techniques to prevent authorship attribution systems. Hence, we consider a random set of *250* files from our dataset

that belong to 50 authors. Those files are used for the evading techniques experiments as described below.

Refactoring Techniques. First, we use the C++ refactoring tools [33, 34]. These tools may perform the following methods: (i) renaming a variable (RV); (ii) moving a method from a superclass to its subclasses (MM); and (iii) extracting a few statements and placing them into a new method (NM). We obtain a precision of *93%* in correctly characterizing authors, which is only a mild drop in comparison to the *92%* precision observed without applying refactoring techniques. More specifically, the accuracy remains the same when the RV technique is applied, whereas the accuracy drops slightly when MM and NM are applied. Since some of the features used in *BinEye* are based on semantic features, they are not significantly affected by these techniques.

Impact of Source Obfuscation. Second, we use some existing tools (e.g., Trigress [35]) to obfuscate the source code by applying the following methods: (i) Virtualization which transforms a function into an interpreter whose bytecode language is specialized for this function; (ii) Jitting which transforms a function into one that generates its machine code at runtime; and (iii) Dynamic which transforms a function into one that continuously modifies its machine code at runtime. Our system is able to deal with such binaries. In our system, we have dynamic features that can tackle such evading techniques. However, almost other features fail with the binaries resulted after such obfuscations. Additionally, we use LLVM framework to perform CFG flattening at source level. These methods affect the accuracy by about 1.5%. The main reason behind this small drop is that the compiler can reduce the effect of those methods when the executable is created.

Impact of Obfuscation. We are interested in determining how *BinEye* handles simple binary obfuscation techniques intended for evading detection, as implemented by tools such as Obfuscator-LLVM [36]. These obfuscators replace instructions by other semantically equivalent instructions, introduce spurious control flow, and can even completely flatten control flow graphs. Obfuscation techniques implemented by Obfuscator-LLVM are applied to the samples prior to classifying the authors. We apply *BinEye* directly on the obfuscated samples. We obtain an accuracy of *93%* in correctly classifying authors, which is only a slight drop in comparison to the *90%* accuracy observed without obfuscation. We combine the refactoring process, source obfuscation, with the binary obfuscation. More specifically, we first apply the refactoring techniques on the selected dataset (*250* files), then apply obfuscation methods at the source level, after which they are compiled using Visual Studio 2010. We apply the binary obfuscation techniques, the accuracy drops from *93%* to *89%*. For the advanced obfuscation, we choose: i) CFG Obfuscation (push/jmp), ii) Instruction Aliasing, ii) Hardware Breakpoint and iii) Instruction Counting., and iv) Encryption Packer. To test our system against the aforementioned techniques, we choose 10 representative obfuscated samples from [37]. The results show that our system precision drops from 93% to 81%.

5.9 Discussion

Through our experiments, we find the following observations:

Feature Pre-processing. With those existing methods, we have encountered top-ranked features related to the compiler (e.g., stack frame setup operation). It is thus necessary to filter irrelevant functions (e.g., compiler functions) in order to better identify author-related portions of code. To this end, we utilize a more elaborate method for filtration based on FLIRT technology for library identification as well as our proposed system for the filtration of compiler functions. Successful distinction between these two groups of functions leads to considerable time savings and helps shift the focus of analysis to more relevant functions.
Source of Features. Existing methods use disassembler and decompilers to extract features from binaries. Caliskan-Islam et al. [9] use a decompiler to translate the program into C-like pseudo code via Hex-Ray [32]. They pass the code to a fuzzy parser for C, thus obtaining an abstract syntax tree from which features can be extracted. In addition to Hex-Ray limitations [32], the C-like pseudo code is different from the original code to the extent that the variables, branches, and keywords are different.
Function Inlining: In practice, the compiler may sometimes inline a small function into its caller code as an optimization. This may introduce additional complexity. We notice that function inlining can drop the precision of our system by 5%. While the precision of OBA2 and Caliskan-Islam et al. approach are not affected.
Privacy Concerns: Our tool could be misused to violate privacy of the coders. Therefore, we have to consider the privacy implications of *BinEye* in the future work.

6 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 `BinEye` to discover the presence of multiple authors.
Synthetic Dataset. The first dataset we use for multi-authorship attribution originates from the Google Code Jam 2010-2017. It consists of single-authored programs and for each author there are multiple programs because it is a multi-round programming contest. We focus on participants from the final round (a total of 400 participants) who have at least 15 C++ programs. To evaluate multi-authorship attribution, we synthetically generate multi-author programs by merging randomly selected files from different authors. For example, consider two authors A and B. We merge two programs (F_A, F_B) written by these authors to form a multi-author program. Using the same approach, we generate multiple executables by merging files from two, three, four, five, and six different authors. We perform the following setup: first training each network in our model based on single-authorship dataset, then we test each of them based on the merged dataset (i.e., files that we create with merging multi-authors). We report the accuracy in Table 6.
GitHub Dataset. We manually collect a set of GitHub projects and check the contributors who have written the code of these projects. We limit our system to

Table 6: Evaluation result for identifying the presence of multiple authors in synthetic dataset

Network	Opt.	Metrics	# of Authors Per File						
			2	3	4	5	6	7	8
CNN1 (Opcode)	O0	Prec.	0.95	0.96	0.96	0.97	0.96	0.98	0.97
		Rec.	0.91	0.92	0.93	0.92	0.95	0.95	0.94
	O1	Prec.	0.95	0.96	0.96	0.96	0.96	0.97	0.97
		Rec.	0.90	0.90	0.91	0.91	0.90	0.92	0.91
	O2	Prec.	0.94	0.95	0.96	0.96	0.96	0.95	0.96
		Rec.	0.90	0.91	0.90	0.91	0.91	0.92	0.92
	O3	Prec.	0.93	0.94	0.93	0.93	0.94	0.95	0.95
		Rec.	0.91	0.91	0.90	0.90	0.92	0.92	0.92
CNN2 (Images)	O0	Prec.	0.98	0.99	0.98	0.98	0.98	0.99	0.99
		Rec.	0.96	0.96	0.95	0.95	0.97	0.96	0.95
	O1	Prec.	0.97	0.98	0.98	0.97	0.97	0.98	0.97
		Rec.	0.96	0.96	0.95	0.95	0.95	0.95	0.95
	O2	Prec.	0.97	0.96	0.98	0.97	0.97	0.97	0.97
		Rec.	0.96	0.96	0.95	0.96	0.96	0.95	0.95
	O3	Prec.	0.97	0.97	0.96	0.98	0.96	0.97	0.97
		Rec.	0.96	0.96	0.97	0.95	0.96	0.96	0.96
CNN3 (API calls)	O0	Prec.	0.91	0.92	0.92	0.92	0.91	0.93	0.93
		Rec.	0.89	0.90	0.90	0.89	0.89	0.89	0.89
	O1	Prec.	0.91	0.91	0.90	0.90	0.91	0.91	0.91
		Rec.	0.88	0.89	0.89	0.89	0.89	0.89	0.89
	O2	Prec.	0.90	0.91	0.90	0.89	0.89	0.89	0.89
		Rec.	0.89	0.89	0.88	0.89	0.89	0.89	0.90
	O3	Prec.	0.91	0.92	0.92	0.92	0.92	0.92	0.92
		Rec.	0.88	0.89	0.88	0.88	0.88	0.88	0.89

programs written in C/C++ and we ignore authors whose contribution is merely adding lines. For this purpose, we collect 50 projects (for the sake of privacy, we removed the details of these projects such as project names) to which 50-1500 authors contributed. We report the accuracy in Table 7. It is worthy to mention that the filtration process to isolate external library code from the code written by their users, it is very important and it increases the precision significantly. In this experiment, we find about 10%-23% of the functions are related to STL and Boost code. We learn our proposed method in Section 3.1 to determine such functions.

Table 7: 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
CNN1 (Opcode)	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
CNN2 (Images)	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
CNN3 (API calls)	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.	0.90	0.90	0.89	0.89	0.89	0.88	0.90
	O3	Prec.	0.91	0.91	0.91	0.90	0.90	0.90	0.90
		Rec.	0.89	0.89	0.88	0.87	0.87	0.87	0.87

7 Conclusion

To conclude, we have presented the first known effort on characterizing the author of binary code based on personnel characteristics. Previous research has applied machine learning techniques to extract stylometry styles and can distinguish between *5-800* authors, whereas we can handle up to *19500* authors. In addition, existing works have only employed artificial datasets, whereas we included more realistic datasets. We also applied our system to known malware (e.g., `Zeus` and `Citadel`). Our findings indicated that the accuracy of these techniques drops dramatically to approximately *45%* at a scale of more than *150* authors. It is easier to attribute authors with advanced expertise or authors of realistic datasets than it is to attribute authors of less expertise or authors of

artificial datasets. For example, in the GitHub dataset, the authors of a sample can be identified with greater than *90%* accuracy. In summary, our system demonstrates superior results on more realistic datasets and real malware and can detect the presence of multiple authors.

References

1. S. Alrabaee, P. Shirani, L. Wang, and M. Debbabi, "Fossil: a resilient and efficient system for identifying foss functions in malware binaries," *ACM Transactions on Privacy and Security (TOPS)*, vol. 21, no. 2, p. 8, 2018.
2. "Techniqal report, Resource 207: Kaspersky Lab Research proves that Stuxnet and Flame developers are connected," http://www.kaspersky.com/about/news/virus/2012/.
3. "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.
4. "Citizen Lab," https://citizenlab.org/, 2015, university of Toronto, Canada.
5. A. Caliskan-Islam, R. Harang, A. Liu, A. Narayanan, C. Voss, F. Yamaguchi, and R. Greenstadt, "De-anonymizing programmers via code stylometry." USENIX, 2015.
6. G. Frantzeskou, "Source code authorship analysis for supporting the cybercrime investigation process," pp. 470–495, 2004.
7. 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.
8. S. Alrabaee, L. Wang, and M. Debbabi, "On the feasibility of binary authorship characterization," *Digital Investigation*, vol. 28, pp. S3–S11, 2019.
9. 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.
10. N. Rosenblum, X. Zhu, and B. P. Miller, "Who wrote this code? identifying the authors of program binaries," in *Computer Security–ESORICS 2011*. Springer, 2011, pp. 172–189.
11. D. Kirat, L. Nataraj, G. Vigna, and 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.
12. A. Oliva and A. Torralba, "Modeling the shape of the scene: A holistic representation of the spatial envelope," *International journal of computer vision*, vol. 42, no. 3, pp. 145–175, 2001.
13. Y. Wei, W. Xia, M. Lin, J. Huang, B. Ni, J. Dong, Y. Zhao, and S. Yan, "Hcp: A flexible cnn framework for multi-label image classification," *IEEE transactions on pattern analysis and machine intelligence*, vol. 38, no. 9, pp. 1901–1907, 2016.
14. "HexRays: IDA Pro," https://www.hex-rays.com/products/ida/index.shtml, 2011, accessed on Feb, 2016.
15. D. Andriesse, A. Slowinska, and H. Bos, "Compiler-agnostic function detection in binaries," in *IEEE Euro S&P*, 2017.
16. F. Farnstrom, J. Lewis, and C. Elkan, "Scalability for clustering algorithms revisited," *ACM SIGKDD Explorations Newsletter*, vol. 2, no. 1, pp. 51–57, 2000.
17. "PEfile:," http://code.google.com/p/pefile/, 2012, accessed on Nov, 2016.

18. L. Nataraj, S. Karthikeyan, G. Jacob, and B. Manjunath, "Malware images: visualization and automatic classification," in *Proceedings of the 8th international symposium on visualization for cyber security*. ACM, 2011, p. 4.
19. J. G. Daugman, "Complete discrete 2-d gabor transforms by neural networks for image analysis and compression," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 36, no. 7, pp. 1169–1179, 1988.
20. E. B. Karbab, M. Debbabi, A. Derhab, and D. Mouheb, "Maldozer: Automatic framework for android malware detection using deep learning," *Digital Investigation*, vol. 24, pp. S48–S59, 2018.
21. G. Huang, Z. Liu, K. Q. Weinberger, and L. van der Maaten, "Densely connected convolutional networks," *arXiv preprint arXiv:1608.06993*, 2016.
22. Y. Kim, "Convolutional neural networks for sentence classification," *CoRR*, 2014.
23. T. Mikolov, I. Sutskever, and al, "Distributed Representations of Words and Phrases and their Compositionality," *NIPS Neural Inf. Process. Syst.*, 2013.
24. J. Pennington, R. Socher, and al., "GloVe: Global Vectors for Word Representation," in *Conf. Empir. Methods Nat. Lang. Process.*, 2014.
25. G. Hamerly, C. Elkan *et al.*, "Learning the k in k-means," in *NIPS*, vol. 3, 2003, pp. 281–288.
26. "The Scalable Native Graph Database. Available from:," http://neo4j.com/, 2015.
27. "The Gephi plugin for nneo4j. Avaiable from:," https://marketplace.gephi.org/plugin/neo4j-graph-database-support/, 2015.
28. "The GitHub repository," 2016, https://github.com/.
29. "The Google Code Jam." http://code.google.com/codejam/, 2008-2015.
30. "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.
31. "Programmer De-anonymization from Binary Executables," https://github.com/calaylin/bda, 2015.
32. "Hex-Ray decompiler," https://www.hex-rays.com/products/decompiler/, 2015.
33. "Refactoring tool," https://www.devexpress.com/Products/CodeRush/, 2016, accessed on Feb, 2017.
34. "C++ refactoring tools for visual studio," http://www.wholetomato.com/, 2016, accessed on Feb, 2016.
35. "Tigress is a diversifying virtualizer/obfuscator for the C language," 2016, http://tigress.cs.arizona.edu/.
36. 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.
37. 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.