Binary Code Multi-Author Identification in Multi-Toolchain Scenarios

Xiaozhu Meng and Barton P. Miller

Computer Sciences Department
University of Wisconsin - Madison
Madison, WI 53706
{xmeng, bart}@cs.wisc.edu

Abstract—Knowing the authors of a binary program has significant application to forensic analysis of malicious software (malware), software supply chain risk management, and software plagiarism detection. As different compilation toolchains may generate drastically different binary code for the same source code, it is essential to be able to reliably identify authors across multiple toolchains. However, existing identification studies have not thoroughly investigated the impact of compilation toolchains on code authorship. In this paper, we explore how toolchains impact programming style and present new techniques for authorship identification in multi-toolchain scenarios.

Since modern software is typically the result of team efforts, we investigate the impact of compilation toolchains in the context of multi-author identification. Existing multi-author identification studies used supervised machine learning and designed code features to capture programming style. We create a new multi-author, multi-toolchain data set, containing 700 authors and 15 toolchains, and evaluate existing techniques with our new data set. The accuracy can be as low as 5% when existing techniques are applied in multi-toolchain scenarios. We then present two new techniques: the two layer approach that first identifies the compilation toolchain that generated the binary and then applies the corresponding single-toolchain authorship model, and the unified training approach that trains multi-toolchain authorship models based on a multi-toolchain data set. Instead of manually designing code features to capture programming style, we apply deep learning to automatically extract features from the raw bytes of the binary. Our techniques can achieve 68% accuracy and 78% top-5 accuracy on our new data set, showing strong evidence that we can identify multiple authors in multi-toolchain scenarios. In addition, our results reveal interesting observations on how compilation toolchains impact code authorship, which inspired us to investigate the internals of our machine learning models. For example, our results show a counter-intuitive phenomena: unoptimized code is more difficult to attribute than optimized code. The investigation of our models shows strong evidence that compiler optimizations such as function inlining can actually help learning authorship style.

I. INTRODUCTION

The capability to identify authors of binary programs can significantly help analysts perform forensic analysis of malicious software (malware), detect malicious software implants in the software supply chain, and recognize software plagiarism. As modern software is typically the result of team efforts, the focus of binary code authorship identification research has evolved from single author identification [2], [7], [31], to multi-author identification [25]. However, none of this previous research has thoroughly investigated how the compilation toolchains, such as the compiler family, version, and optimization level, impact their techniques. Given the same source code, different compilation toolchains may generate drastically different binary code, which casts doubt on whether authorship identification can be reliably performed across multiple compilation toolchains. In this paper, we explore how compilation toolchains impact multi-author identification and show that existing techniques do not perform well in multi-toolchain scenarios. We then present new techniques for multi-toolchain, multi-author identification. Our experiments show that we can identify 700 authors with 68% accuracy in a data set containing real world binaries generated by 15 toolchains. We also investigate the internals of our learning models and present the lessons we learned.

Knowing the authors of a program can be of benefit in multiple application domains. Malware writers create, share, and adapt functional components such as command and control, encryption and decryption, beaconing, exfiltration, and domain flux, and assemble new malware with these functional components. It is convenient for malware writers to acquire and share functional components through online black markets [1], [6], [16] or by forming physically co-located teams [22]. This trend indicates that it is important for malware analysts to trace malware samples based on the connections between their authors. In the software supply chain domain, code that exhibits untrusted styles, such as the ones seen in malware, indicates potential malicious implants. Software plagiarism can be detected by matching programming styles against known code. As source code is typically not available for malware, commercial software, and legacy code, we focus on binary code authorship identification.

Early studies of binary code authorship identification focus on identifying the authors of single author programs [2], [7], [31], casting the problem of identifying authors as a supervised machine learning problem. The common workflow of these studies can be summarized in two steps. First, design binary code features to reflect low level code properties such as machine instruction details and high level code properties such as program control flow and data flow. Second, use supervised machine learning techniques, such as Support Vector Machine (SVM) [8] and Random Forests [15], to discover correlations
between code features and authorship. When applied to real world software, which typically is written by multiple authors, these single author identification techniques can identify at most one of the multiple authors, or report a merged group identity, which makes single author identification impractical for real world applications.

Similar to single author identification, recent multi-author identification studies [23], [25] are also based on supervised machine learning and designing code features. The key difference between single author identification and multi-author identification is that single author identification reports one author for a binary program, while multi-author identification reports one author for a finer unit of code, such as one author per basic block. These multi-author studies showed that attributing at the function level caused too much imprecision, as many functions are cooperated by multiple authors. Therefore, current multi-author identification operates at the basic block level. Attributing at the basic block level requires designing fine-grained code features, as the features used for single author identification may not be applicable at this finer granularity.

However, the impact of compilation toolchains on binary code authorship identification has not been thoroughly investigated. Hendrikse [13] investigated the impact of compilation toolchain on single author identification at a small scale; the experiments contained only 20 authors, 280 binaries, and 9 toolchains. Other single author identification and multi-author identification techniques were evaluated only in single toolchain scenarios, meaning that they performed training and testing based on binaries generated by the same toolchain [2], [7], [25], [31]. In particular, multi-author identification techniques were evaluated only with code generated by GCC at optimization level -O2 (GCC -O2) [23], [25]. As different compilation toolchains may generate significantly different binary code for the same source code, the code features that are extracted from a program may depend heavily on which compilation toolchain is used to generate the binary of the program. Therefore, the impact of compilation toolchains raises important questions: Can existing multi-author identification techniques work in multi-toolchain scenarios? Is a programmer’s programming style reflected on binary code similar across different toolchains? Do we need different machine learning models or different code features for different toolchains? Understanding these questions is vital for performing multi-author identification on real world software.

In this paper, we present the first thorough multi-toolchain, multi-author identification study. We explore how compilation toolchains impact multi-author identification techniques and present new techniques to identify multiple authors in multi-toolchain scenarios. We first conduct a study to evaluate how well our previous multi-author identification techniques [25] will function in a multi-toolchain scenario. We aim at answering two questions in this study: (1) Can these techniques achieve good accuracy when applied to binaries that are not generated with GCC -O2? And (2) can we achieve good accuracy when we apply models trained on binaries compiled by one toolchain to predict the authors of binaries generated by a different toolchain? To answer these questions, we created a multi-toolchain dataset by compiling three large, long-lived open source projects (Apache HTTPD Server [4], Dyninst binary analysis and instrumentation tool suite [28], and Git [11]), with three compilers (GCC, ICC, and LLVM), and five optimization levels (O0, O1, O2, O3, and Os). This yielded a data set consisting of 1,965 binaries generated by 15 toolchains, containing 50 million basic blocks and 700 authors.

We enumerate all toolchain pairs to generate training sets and testing sets, which yields $15 \times 15 = 225$ total evaluation combinations. Among these 225 combinations, 15 represent single toolchain scenarios and the other 210 represent multi-toolchain scenarios. Our results reveal two limitations of existing multi-author identification techniques. First, the accuracy for multi-toolchain scenarios (min: 5%, median: 13%, max: 64%) is significantly lower than the accuracy for single toolchain scenarios (min: 44%, median: 59%, max: 70%). In practice, as we typically do not know which compilation toolchain generated the target binary for authorship identification, applying a mismatched authorship model to perform prediction can lead to extremely low accuracy. Second, there is a significant difference between the accuracy for single toolchain scenarios; the optimization levels influence the accuracy. For example, GCC -O0 has the lowest accuracy (44%) and ICC -O3 has the highest accuracy (70%). This accuracy difference suggests that programmers can have a higher chance of evading current multi-author identification by carefully choosing a toolchain such as GCC -O0 to generate the binary programs. These two limitations confirm the need of new techniques for identifying multiple authors in multi-toolchain scenarios.

We then compare two approaches to address the issue that applying mismatched authorship model leads to low accuracy. The first one is a two layer approach. Since previous techniques work better in single toolchain scenarios, it is reasonable to first use toolchain identification techniques [29], [30] to determine which toolchain generated the binary and then apply the corresponding single-toolchain authorship model. Note that while previous authorship identification studies [7], [25] have discussed this approach, none of them have implemented or evaluated it. The second one is a unified training approach, where we construct a training set containing binaries from all known toolchains. With a multi-toolchain training set, we may be able to train an authorship model that can work in multi-toolchain scenarios, albeit at a higher training cost.

While previous studies focused on designing new code features, we instead use deep learning to automatically extract low level features from raw bytes. We apply feed-forward neural networks to multi-author identification. To the best our knowledge, we are the first project to apply deep learning to this problem and show that it can reliably identify low level code features. We focus on two areas when applying deep learning. First, we investigate what should be used as inputs to Deep Neural Networks (DNNs) and find that while using only raw bytes as inputs can achieve reasonable accuracy,
complementing raw bytes with higher-level structural features can further improve accuracy. Second, deep learning requires tuning several important learning hyper-parameters to achieve good results, such as the learning rate, the number of layers, and the number of hidden units per layer. We follow the general practices recommended by deep learning researchers [5], [12], and report our experiences of applying these practices so that future research on applying deep learning to authorship identification can benefit from our experiences.

We evaluated our techniques with our multi-author, multi-toolchain data set. We focused on three aspects of our techniques: whether the two layer approach and the unified training approach can address the issue of mismatched models, whether DNN can improve accuracy over traditional machine learning techniques such as SVM, and understanding and improving the accuracy differences between optimization levels. Our results showed that with toolchain identification, we achieved 59% accuracy for identifying 700 authors. Replacing SVM with DNN, we improved this accuracy to 68%. In addition, DNN achieved 78% top-5 accuracy and 81% top-10 accuracy. For the unified training approach, SVM training did not scale to this data set, while DNN achieved 66% accuracy. Our results also showed that DNN reduced the accuracy difference between optimization levels and improved accuracy for all 15 individual toolchains (min: 58%, median 67%, max: 77%). ICC -O0 had the lowest accuracy (58%) and ICC -O3 has the highest accuracy (77%).

To gain more insights on how compilation toolchain impacts multi-author identification and on how the machine learning models work, we investigated the structure of the learning models, including the feature weights of SVM and the activations of neurons in DNN. We learned three lessons: (1) different toolchains mainly have disjoint sets of features that are indicative of authorship; (2) DNNs can internally determine which toolchain generated the binary without explicitly training for toolchain identification; and (3) unoptimized code is more difficult to attribute at the basic block level. Previous studies have reported accuracy results, but have not analyzed their models.

II. BACKGROUND

We discuss four areas of existing authorship identification studies as background: the binary code features used to capture programming styles, commonly used workflow, relationship between accuracy and the complexity of the programs used in evaluation, and relationship between accuracy and the compilation toolchains.

A. Binary Code Features

Binary code features are extracted at block level or function level, and are then accumulated to the corresponding operation level. For example, single author identification accumulates features to the program level and multi-author identification accumulates features to the basic blocks level. As multi-author identification requires identifying authors at the basic block level, we discuss only basic block level features.

Existing block level features describe a wide variety of code properties, such as instruction details, control flow, data flow, and external dependencies. Low level code features include byte n-grams [18], [31], instruction idioms [7], [18], [19], [30], [31], [32], and individual instruction components such as instruction prefixes, instruction operands, and constant values [25]. Previous authorship identification studies explicitly designed these features. In our current work, we use deep learning to automatically learn low level features from the raw bytes of a basic block.

Higher-level structural features describe program control flow, data flow, function and loop context of a basic block, and external dependencies. Control flow features [25] describe the types of incoming and outgoing CFG edges (such as conditional taken, conditional not taken, direct jump, and fall through), and exception handling (such as whether a basic block throws exceptions and whether a basic block catches exceptions). Data flow features [25], [9] have three main types: (1) the number of input variables, output variables, and internal variables, (2) features that describe how a basic block uses a stack frame, and (3) features that describe data flow dependencies of variables. Context features [25] include the loop nesting level of a basic block and loop size of the enclosing loop. External dependency features included function names of external library call targets [31]. We will reuse these high level features as inputs to DNN.

B. Workflow

The most common workflow used in existing single author identification studies has four major steps: (1) design a large number of simple candidate features; (2) extract the defined features using binary code analysis tools such as IDA Pro [14] or Dyninst [28]; (3) select a small set of features that are indicative of authorship by using feature selection techniques such as ranking features based on mutual information between features and authors [31]; and (4) apply a supervised machine learning technique, such as Support Vector Machine (SVM) [8] or Random Forests [15], to learn the correlations between code features and authorship. Rosenblum et al. [31] used instruction, control flow, and library call target features, and used SVM for classification. Caliskan et al. [7] added data flow features, constant data strings, and function names derived from symbol information, and used Random Forests for classification.

Multi-author identification studies [25], [23] adapted this workflow with three modifications. First, they performed attribution at the basic block level. Second, they found that library code such as Standard Template Library (STL) and Boost C++ Library (Boost) is often inlined and needs to be distinguished from users’s code. Third, as a programmer typically writes more than one basic block at a time, they used a Conditional Random Field (CRF) model to capture potential author correlations between adjacent basic blocks.

C. Complexities of Programs

Single author identification studies performed evaluations of their techniques on single author programs such as Google
Code Jam, and multi-author programs that have a clear major author such as university course projects and certain programs extracted from Github. Rosenblum et al. reported 51% accuracy for classifying 191 authors on -O0 binaries from Google Code Jam and 38.4% accuracy for classifying 20 authors on -O0 binaries from university course projects. Caliskan et al. improved Google Code Jam accuracy to 92% for classifying 191 authors on -O0 binaries, and 89% accuracy for classifying 100 authors on -O2 binaries. They also extracted some programs from Github that have a major author who contributed more than 90% of the code, and got 65% accuracy for classifying 50 authors.

These studies reported significantly lower accuracy on multi-author programs than on Google Code Jam. There are at least two reasons for this accuracy difference. First, university course projects and programs extracted from Github contained code that was not written by the major author. This code can confuse machine learning algorithms. For example, course projects contained skeleton code from the professor and the extracted Github programs still had code from other authors and third party libraries. Second, these multi-author programs are typically more complex than the programs from Google Code Jam. Programs from Google Code Jam are written quickly, while course projects can take several days and programs from Github may take months to years. It is reasonable to perform evaluation on Google Code Jam if the techniques are used for plagiarism detection for programming contests. However, as programs from Google Code Jam are typically not written with common software engineering practices, they are less appropriate for malware forensics, intellectual properties violation detection, and supply chain risk management on commercial software.

Multi-author identification studies instead used large, long-lived, real world open source software that follows common software engineering practices such as Apache HTTP Server. They achieved 59% accuracy for identifying 284 authors using SVM and 65% accuracy with CRF. We will follow this practice and evaluate our techniques with real world software.

D. Impacts of Compilation Toolchains

Caliskan et al. repeated single toolchain evaluations with four toolchains (GCC -O0, -O1, -O2, -O3), with the programs from Google Code Jam. In other words, they trained and tested at the same optimization level. They reported that -O0 code has the highest accuracy (96%), and -O3 code has lowest accuracy (89%), but did not investigate why there is such an accuracy difference between optimization levels. They assumed that they could use previous toolchain identification techniques [29], [30] to identify the toolchain. Hendrikse [13] performed the only multi-toolchain study, though still based on single author identification, using the programs from Google Code Jam. He repeated single toolchain evaluations with 9 toolchains (GCC, MSVS, and ICC with -O0, -O2, and -Os) and reported no significant accuracy difference between optimization levels (-O0: 92%, -O2: 93%, and -Os: 94%). He created a multi-toolchain data set by randomly sampling a program from one of the 9 toolchains, and reported 92% accuracy. However, this study was at a small scale, as the experiments contained only 20 authors.

We are the first the project to investigate the impact of compilation toolchains on multi-author identification, develop new techniques for multi-toolchain, multi-author identification, and evaluate our techniques with real world software. Our results do not align with the results of these existing studies, as we achieved highest accuracy with -O3 code and lowest accuracy with -O0 code. In Section VII-C, we will present our analysis as to why we see such an accuracy difference.

III. MULTI-TOOLCHAIN STUDY

We evaluate the effectiveness of our previous techniques for multi-author identification [25] in a multi-toolchain scenario.

A. Data Set Generation

To evaluate their techniques, we created a multi-toolchain dataset by compiling three large, long-lived open source projects (Apache HTTPD Server [4], Dyninst binary analysis and instrumentation tool suite [28], and Git [11]), with three compilers (GCC 4.8.5, ICC 15.0.1, and LLVM 3.5.0), and five optimization levels (O0, O1, O2, O3, and Os) on a 64-bit Red Hat Linux 7 platform. For each of the 15 toolchains, we used the following steps to generate the author label as ground truth for each basic block.

1) Use git-author [26] to get a weight vector of author contribution percentages for all source lines in these projects. The source lines of STL and Boost code were attributed to author identity “STL” and “Boost”, respectively.

2) Compile these projects with debugging information using the given compiler and optimization level, and obtain a mapping between source lines and machine instructions. The compiler may generate binary code that does not correspond to any source line, such as the default constructor for a class when the programmer does not provide it. We exclude this code from our data set.

3) Derive weight vectors of author contribution percentages for all machine instructions and basic blocks in the compiled code. We first derived the weight vector for each instruction by averaging the contribution percentages of the corresponding source lines. We then derived the weight vector of a basic block by averaging the vectors of the instructions within the basic block.

4) Take the major author as the author label, based on the weight vector of contribution percentages of a basic block.

We generated a data set consisting of 1,965 binaries generated by 15 toolchains, containing 50 million basic blocks and 700 authors. The 1,965 binaries consisted of 131 programs, with each program having 15 different versions.

B. Evaluation Methodology

We enumerate all toolchain pairs to generate training sets and testing sets. For each evaluation pair, we used the following evaluate strategy.
We used Dynisnt [28] to extract code features, Liblinear [10] for linear SVM, and CRFSuite [27] for linear CRF. We performed the traditional leave-one-out cross validation, where each program was in turn used as the testing set and all other programs were used as the training set. So, the testing set contained the binary of the testing program generated by the testing toolchain, and the training set contained the binaries of the training programs generated by the training toolchain.

Each round of the cross validation had two steps. First, we selected the top 45,000 features that had the most mutual information with the major authors. We investigated the impact of the number of selected features and reported that selecting 45,000 features worked best [25]. Second, we trained a linear SVM and a linear CRF and predicted the author of each basic block in the testing set. We calculated accuracy as the percentage of correctly attributed basic blocks. We parallelized this cross validation with HTCondor [17], where each round of the cross validation is executed on a separate machine.

C. Results

Table I shows the accuracy for all the evaluation pairs using SVM. We make three observations from the result table. First, our previous techniques did not work well in multi-toolchain scenarios, as shown in the non-diagonal cells. The red-shaded cells show the minimal (5%), median (13%), and maximal (64%) accuracy achieved in multi-toolchain scenarios. Second, these techniques achieved much higher accuracy in single toolchain scenarios, as shown in the diagonal cells. The green-shaded cells show the minimal (44%), median (59%), and maximal (70%) accuracy achieved. However, there is a 26% accuracy difference between the minimal and maximal. Third, by comparing the accuracy numbers in each column, we find that we always get the highest accuracy for a testing toolchain when we use the model trained on binaries generated by the same toolchain. This observation supports the idea that as long as we can identify the toolchain that generated the testing binary, we know which authorship model is the most appropriate to use.

We were not able to get any accuracy results with CRF because CRF training took too long to finish. We commented in our previous work [25] that we spent 7 days to train a CRF model. The previous data set had 900,583 basic blocks and 284 authors, whereas in this study, the data used in each evaluation pair had average 1.5 million basic blocks and 700 authors. As the training of a linear CRF has a time complexity quadratic in the number of labels and linear in the total number of data instances [35], it is not surprising that we cannot finish CRF training in any reasonable amount of time.

The results of our study confirm that we need new approaches for authorship identification in multi-toolchain scenarios and investigate the accuracy differences between optimization levels.

IV. APPROACHES

We compare two approaches for multi-toolchain, multi-author identification. First, we attempt to identify the toolchain that generated the target binary and then apply the corresponding single toolchain authorship model. We call this the two layer approach. Second, we construct a training set that contains binaries from all known toolchains and then train a multi-toolchain model. We call this the unified training approach.

A. Two Layer Approach

Besides training multiple single-toolchain authorship models, the other key component of the two layer approach is to perform toolchain identification. Toolchain identification is typically performed at the function level [30]. The program level is not suitable for toolchain identification as a binary may contain code generated by different toolchains. For example, a programmer may compile their code with one toolchain and then statically link a library that was generated by a different toolchain. On the other hand, a function is typically generated by one toolchain, as the source code of a function is in a single source file and the source file is typically the compilation unit.

Toolchain identification can be summarized in three steps. First, we define and extract candidate binary code features. Rosenblum et al. [30] used instruction idioms, which represented consecutive machine instructions, and graphlets, which represented subgraphs extracted from the Control Flow Graph (CFG) of a function. Second, we perform feature selection by ranking features based on the mutual information with toolchain labels. Third, as a compilation unit typically contains more than one function, we perform joint classification by training a Conditional Random Field model to capture the correlations between code features and toolchain labels. Note that training a CRF model for toolchain identification is significantly faster compared to multi-author identification, as there are only 1.4 million functions and 15 toolchain labels in our data set.

The two layer approach needs to maintain multiple classifiers: one toolchain identification classifier, and one authorship identification classifier for each toolchain. On the other hand, maintaining multiple classifiers brings two advantages. First, each single-toolchain authorship model can capture the distinct characteristics of each toolchain. Second, the training effort of each model is modest.

B. Unified Training Approach

The unified training approach constructs a multi-toolchain training set. This idea is based on a general machine learning practice called data set augmentation [12]. Data set augmentation aims to improve the accuracy of a classifier by adding training examples that have been modified with transformations that do not change the label of the example. For example, in object recognition in computer vision, a cat image remains a cat image if it is shifted one pixel to the right. Similarly, the code written by an author remains code written by this author if it is compiled by a different toolchain.

Compared to the two layer approach, the unified training approach needs to maintain only one classifier. However, training this classifier requires significantly more computing
TABLE I: Evaluation results of our previous techniques [25]. The diagonal cells (in bold font) represent the single toolchain results and the green-shaded cells represent the minimal, median, maximal accuracy achieved in single toolchain scenarios. The non-diagonal cells represent the multi-toolchain results and the red-shaded cells represent the minimal, median, maximal accuracy achieved in multi-toolchain scenarios.

<table>
<thead>
<tr>
<th></th>
<th>GCC</th>
<th>ICC</th>
<th>LLVM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Train</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-O0</td>
<td>45%</td>
<td>47%</td>
<td>70%</td>
</tr>
<tr>
<td>-O1</td>
<td>8%</td>
<td>9%</td>
<td>10%</td>
</tr>
<tr>
<td>-O2</td>
<td>7%</td>
<td>13%</td>
<td>14%</td>
</tr>
<tr>
<td>-O3</td>
<td>6%</td>
<td>13%</td>
<td>13%</td>
</tr>
<tr>
<td>-Os</td>
<td>8%</td>
<td>11%</td>
<td>11%</td>
</tr>
</tbody>
</table>

A. Basics of Feed-forward Neural Networks

Figure 1 illustrates an example of a feed-forward neural network and breaks down its individual components. As shown in Figure 1a, nodes in the network are arranged into layers: one input layer, multiple hidden layers (two in this example), and one output layer. Typically, nodes between adjacent layers are fully connected, and there are no backward edges or cross layer edges.

As shown in Figure 1b, a node in the input layer takes an input value $f_i$ and outputs the same value $f_i$. The information used for constructing inputs is application specific. Two common choices are using manually designed features and using raw data as inputs. Examples of raw data include individual pixels in computer vision tasks and raw bytes in our case.

The internal computation of a hidden layer node is shown in Figure 1c. The number of hidden layers and the number of nodes in a hidden layer are two hyper-parameters that need tuning. A node in a hidden layer takes multiple inputs from the previous layer and generates new output value to the next layer. Denote $X_i = [x_{i1}, x_{i2}, \ldots, x_{in}]^T$ as the input vector, where $n$ represents the total number of nodes in the previous layer. The inputs first go through a linear transformation, defined as $z_i = W_i^T X_i + b_i$, where weights $W_i = [w_{i1}, w_{i2}, \ldots, w_{in}]^T$ and bias $b_i$ are two learning parameters, whose values are determined in the training process. Note that each node has separate weights and bias. The output is then defined as $y_i = \sigma(z_i)$, where $\sigma$ is called the activation function. $\sigma$ is a non-linear function so that the whole network can represent a non-linear prediction space. A commonly used activation function is Rectified Linear Unit (ReLU): $ReLU(x) = \max(0, x)$.

The goal of the output layer is to generate a probability vector $P = [p_1, p_2, \ldots, p_m]^T$, where $p_i$ represents the probability that the input data instance belongs to class $i$ and $m$ is the total number of classes. As shown in Figure 1d, similar to a node in a hidden layer, the inputs first go through a linear transformation $z_i = W_i^T X_i + b_i$, where $Z = [z_1, z_2, \ldots, z_m]^T$. The output is defined as $p_i = \text{softmax}(Z)_i$, where $\text{softmax}(Z)$ is a function that normalizes a vector of arbitrary values to a probability vector; $\text{softmax}(Z)_i$ is the $i$th element in the probability vector, defined as $\text{softmax}(Z)_i = (e^{z_i})/(\sum_{k=1}^{m} e^{z_k})$. We can then report the class with the highest probability as the prediction result, or report the top-k classes by choosing the $k$ highest probabilities.

The purpose of training is to determine the values for $W$ and $b$, which can be summarized in four steps [12].

1) Initialize the weights and biases. Typically, weights are randomly sampled from a Gaussian or uniform distribution; biases are set to heuristically chosen constants.

2) Calculate the loss for a training example. Suppose the training example belongs to the $L$th class. We calculate the output of each node along the layers in the network, and get the prediction probability vector $P$. A common loss function is cross-entropy. In this context, it is defined as $-\log(p_L)$. Intuitively, the larger $p_L$ is, the more likely we make the correct prediction and the smaller the loss.

3) Update weights and biases by gradient descent. This step aims to reduce the loss by updating the weights and biases. The direction of the update is specified by the negative of the gradient, as it represents the fastest direction along which the loss decreases. The magnitude of the update is specified by a user-defined hyper-parameter called the learning rate.

4) Repeat the second and third steps over the training set until converging. Since the training set for deep learning is typically too large to fit in memory, a common practice is to split the training set into multiple mini-batches, where one mini-batch contains dozens or hundreds of data.
instances. Only one mini-batch is loaded into memory at a time.

As a user of feed-forward neural networks, we need to construct appropriate inputs to the network and tune key hyper-parameters such as the learning rate, the number of layers, and the number of unit in each hidden layers.

**B. Extract Low Level Features**

A key advantage of deep learning is that it can automatically learn features from data. So, we use raw bytes of a basic block as inputs, as opposed to designing the features ourselves. As manual feature design is unlikely to cover all relevant code properties, using raw bytes can also potentially capture information that is not represented by existing features. To provide raw bytes as input, we have two issues to address. First, the length of a basic block is variable, but a feed-forward network takes a fixed number of inputs. For this issue, we empirically decided to use the first 70 bytes of the basic block as we found over 99% of the basic blocks are short than 70 bytes. For basic blocks shorter than 70 bytes, we add padding after the code bytes.

Second, how do we represent the value of a byte as the inputs to the network? Ideally, we would like a representation that will allow the network to capture instruction fields such as instruction prefixes, opcodes, and operands, and the encoding of machine instructions. For the x86-64 architecture, instruction fields do not necessarily align with byte boundaries. For example, the lower four bits of a REX prefix byte represents four different fields. Therefore, we use bit values as inputs. Specifically, we translate bit value 0 to input value -1, bit value 1 to input value 1, and padding bit to value 0. This representation allows the network to distinguish padding from real code bits.

**C. Use Structural Features**

Using only raw bytes of a basic block as input cannot capture higher-level structural features such as control flow, data flow, and the context. This is because structural properties are the results of interactions between multiple basic blocks and extracting structural code features is the result of extensive, semantic analysis of the binary code [3], [24], [33]. We reuse existing basic block level structural features [23], [25], [31] and feed them to the network as the inputs along with the raw bytes.

We improved two existing control flow features by considering the impact of compiler optimizations. These two features are whether a basic block has an incoming condition-taken edge and whether a basic block has an incoming condition-not-taken edge. Suppose a basic block has an incoming condition-taken edge when the code is compiled without optimization, such as with GCC -O0. When the same code is compiled with GCC -O2, which enables -freorder-blocks, this basic block may be reordered and instead have an incoming condition-not-taken edge. Therefore, whether these two features exist in a basic block is often decided by the compiler, rather than the style of the programmer.

**VI. Evaluation**

We evaluated our techniques with the same multi-toolchain data set discussed in Section III. Recall that this data consists of 1,965 binaries generated by 15 toolchains, containing 50 million basic blocks and 700 authors.

We focus on two aspects of our techniques: how well the two layer approach and the unified training approach can perform multi-toolchain, multi-author identification, and whether DNN can improve accuracy over traditional machine
learning techniques such as SVM. As either the two layer approach or the unified training approach can be paired with either SVM or DNN, we evaluated the following four techniques: \textit{two-layer-svm, two-layer-dnn, unified-svm, and unified-dnn.}

\subsection{Evaluation Methodology}

Our evaluations are based on leave-one-out cross validation, where all 15 versions of a program are considered as a fold. So, in each round of cross validation, the training set contains all 15 versions of 130 programs, and the testing set contains all 15 versions of the other program.

For \textit{two-layer-dnn}, we train a toolchain identification classifier with linear CRF using all binaries in the training set. We then train 15 single-toolchain authorship classifiers with feed-forward neural networks, where each classifier is trained with the corresponding version of the 130 programs. For testing, we first use the toolchain identification classifier to determine the which toolchain generated the functions in the testing binaries, and then we apply the corresponding single-toolchain authorship model to predict the authors of all basic blocks. The steps for \textit{two-layer-svm} is similar to the steps for \textit{two-layer-dnn}, but we replace feed-forward neural networks with linear SVM. For both \textit{unified-svm} and \textit{unified-dnn}, we train a multi-toolchain authorship model with all binaries in the training set and then predict the authors of binaries in the testing set. Accuracy is calculated as the number of correctly attributed basic blocks over the total number of basic blocks.

We used Dyninst [28] to extract code features, Liblinear [10] for linear SVM, CRFSuite [27] for linear CRF, and TensorFlow [36] for deep learning. It is straight-forward to parallelize the training and testing of neural networks in TensorFlow. In our experiments, we used four CPUs for training and testing each feed-forward neural networks. We parallelized our evaluations with HTCondor [17]. Note that the two layer approach has more parallelism than the unified training approach. For the unified training approach, we can parallelize different rounds of cross validation. For the two layer approach, we can parallelize different rounds of cross validation, as well as the training of toolchain identification and all single-toolchain authorship models within a round of cross validation.

\subsection{Evaluation Results}

A key question to answer in our evaluations is how well we can perform multi-toolchain, multi-author identification. Our results show that \textit{two-layer-svm, two-layer-dnn, and unified-dnn} achieved 59%, 68%, and 66% accuracy for classifying 700 authors on code generated by 15 toolchains, as opposed to as low as 5% accuracy when a mismatched authorship model is used. Our techniques can help prioritize investigation: \textit{two-layer-dnn} achieved 78% top-5 accuracy and 81% top-10 accuracy. Our results show that \textit{unified-svm} did not scale to the merged training set containing all 15 toolchains. These results show that we can perform practical multi-toolchain, multi-author identification with our deep learning models.

To better understand the accuracy achieved the two layer approach, we investigated how the accuracy of toolchain identification impact multi-author identification. Our toolchain identification classifiers achieved 93% accuracy for identifying 15 toolchains, where accuracy is calculated as the number of functions that we identified the correct toolchain over the total number of functions. To investigate the impact of 7% error rate on toolchain identification, we repeated experiments with \textit{two-layer-svm} and \textit{two-layer-dnn} using an oracle for toolchain identification. We observed less than 1% accuracy improvement by using an oracle for toolchain identification. Therefore, we believe current toolchain identification techniques are good enough for multi-toolchain, multi-author identification.

We then compared \textit{two-layer-dnn} and \textit{two-layer-svm} on two aspects to understand how \textit{two-layer-dnn} improved accuracy. First, we broke down the accuracy achieved by DNN and SVM in single toolchain scenarios. Table II shows that DNN improved not only the overall accuracy, but also the accuracy for all 15 toolchains. In addition, \textit{two-layer-dnn} had a slightly reduced accuracy difference between optimization levels, as ICC -O0 had the lowest accuracy (58%) and ICC -O3 had the highest accuracy (77%), compared to \textit{two-layer-svm}, where LLVM -O0 had the lowest accuracy (44%) and LLVM -O3 had the highest accuracy (70%).

Second, we compared the accuracy of SVM and DNN for different sizes of basic blocks. As shown in Figure 2a, we can see that SVM suffers when there are small basic blocks. This observation aligns with the results presented in our previous study [25] and can be explained simply as small basic blocks provide fewer code features for prediction. On the other hand, DNN achieved much higher accuracy on small basic blocks. We attribute this improvement to the advantage of deep learning: our DNN models extract features at bit level, which can capture information that is not represented by existing low level features. For larger basic blocks (byte size larger than 20), both DNN and SVM have varying accuracy, and DNN does not seem to outperform SVM. Note that we only use the first 70 bytes of a basic block, but this decision does not hurt accuracy for blocks larger than 70 bytes. Figure 2b shows the cumulative distribution of basic blocks in different sizes. Since about 80% of the total basic blocks are smaller than 20 bytes, the accuracy improvement on small basic blocks explains the overall improvement of DNN.

Since \textit{unified-svm} did not scale to the merged training set, it shows that \textit{unified-dnn} has better training scalability. We observed similar accuracy results from \textit{unified-dnn} compared to \textit{two-layer-dnn}: (1) \textit{unified-dnn} has similar accuracy differences between optimization levels, where O0 code has lowest accuracy, and O2 and O3 code have highest accuracy; (2) \textit{unified-dnn} has a similar accuracy trend in terms of block sizes.

Last, we discuss the needed training time. For
TABLE II: Comparison of single toolchain results between SVM and DNN. For convenience of comparison, we copy the results for SVM from the diagonal cells in Table I. DNN improved accuracy for all 15 toolchains.

<table>
<thead>
<tr>
<th></th>
<th>O0 SVM</th>
<th>O0 DNN</th>
<th>O1 SVM</th>
<th>O1 DNN</th>
<th>O2 SVM</th>
<th>O2 DNN</th>
<th>O3 SVM</th>
<th>O3 DNN</th>
<th>Os SVM</th>
<th>Os DNN</th>
<th>Avg SVM</th>
<th>Avg DNN</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC</td>
<td>45%</td>
<td>60%</td>
<td>59%</td>
<td>67%</td>
<td>63%</td>
<td>68%</td>
<td>60%</td>
<td>73%</td>
<td>55%</td>
<td>64%</td>
<td>58%</td>
<td>66%</td>
</tr>
<tr>
<td>ICC</td>
<td>47%</td>
<td>58%</td>
<td>58%</td>
<td>65%</td>
<td>66%</td>
<td>77%</td>
<td>68%</td>
<td>77%</td>
<td>58%</td>
<td>65%</td>
<td>60%</td>
<td>70%</td>
</tr>
<tr>
<td>LLVM</td>
<td>44%</td>
<td>60%</td>
<td>52%</td>
<td>62%</td>
<td>69%</td>
<td>72%</td>
<td>70%</td>
<td>73%</td>
<td>60%</td>
<td>68%</td>
<td>60%</td>
<td>67%</td>
</tr>
<tr>
<td>Avg</td>
<td>45%</td>
<td>59%</td>
<td>57%</td>
<td>64%</td>
<td>66%</td>
<td>73%</td>
<td>67%</td>
<td>75%</td>
<td>58%</td>
<td>66%</td>
<td>59%</td>
<td>68%</td>
</tr>
</tbody>
</table>

Fig. 2: Comparison between accuracy of SVM and DNN on basic blocks in different sizes. DNN significantly outperforms SVM for small basic blocks. As there are more small basic blocks, DNN achieved better overall accuracy.

two-layer-dnn, it took about 30 hours to train a DNN model and about 10 hours to train a linear CRF model for toolchain identification. For two-layer-svm, it took about 20 hours to train a linear SVM, and we use the same linear CRF models for toolchain identification. For unified-dnn, it took about 240 hours to train a DNN model. Note that we can significantly speed up the training of DNN models by deploying them on GPUs and more CPUs. We believe the training cost of unified-dnn is practical.

C. Experiences of Deep Learning

As we are the first project to apply deep learning to multi-author identification, we report our experiences of deep learning tuning. We used the Adam optimizer in our experiments, as it has been shown empirically to be more effective than other optimization methods [20]. We first discuss several factors that are fundamental for us to achieve any success on training DNN models:

1) Use a small initial learning rate. Typical values for the initial learning rate are between $10^{-6}$ and 1, with a default value 0.01 [5]. We followed this recommendation and found that we needed a small initial learning rate, around $10^{-4}$. Too large values such as 0.01 and 0.001 caused the training loss to increase with training and eventually diverge to infinity.

2) Choose an appropriate initialization strategy. We started with sampling weights from a truncated normal distribution and setting biases to 0, which is an initialization strategy borrowed from a tutorial example on TensorFlow’s website. This initialization strategy caused the training accuracy to stay low through training. We found that sampling weights from a uniform distribution and setting biases to 0.1 solved this issue [12].

3) Shuffle the training data before training. We started with dividing data into mini-batches by iterating every binary in the training set and putting consecutive basic blocks into one mini-batch. As adjacent basic blocks are likely written by the same author, this caused each mini-batch to have code from only a few authors and different mini-batches to have disjoint sets of authors, which in turn caused the training to repeatedly optimize the weights and biases for different authors and led to fluctuating training accuracy. Shuffling the training data so that each mini-batch contains code from more authors makes the training converge to a high training accuracy.

We did not observe significant improvement by tuning the number of hidden layers and the number of hidden units per layer. Our networks are set to have 5 hidden layers and each layer has 800 hidden units. We let all hidden layers have the same number of units as Larochelle et al. [21]...
empirically showed that an even-sized network architecture performs no worse than a decreasing-sized or an increasing-sized architecture. Our results align with this recommendation. The decision of 800 hidden units per layer is based on two considerations. First, a hidden layer wider than the input layer typically performs better than a hidden layer narrower than the input layer [5]. Our input layer has 704 units (560 units for 70 bytes and 144 units for structural features), so our hidden layer should be wider than 704 units. Second, on the other hand, too wide hidden layers will significantly increase training time as the number of learning parameters is quadratic in the number of units per layer. We tried more units and fewer units per layer and observed about the same or lower accuracy. We also varied the number of hidden layers to 8 and 10 and observed less than 1% accuracy improvement.

We investigated the impact of training iteration, shown in Figure 3. One epoch is defined as one iteration over all the data in the training set. unified-dnn requires fewer number of epochs to achieve good accuracy, compared to two-layer-dnn. However, note that the training set for unified-dnn averages about 15 times larger than the training set for two-layer-dnn, explaining why we need to spend much longer training on unified-dnn than two-layer-dnn.

VII. UNDERSTANDING THE MODELS

We investigate the internals of our machine learning models to better understand how the models work, including the selected features and their weights in two-layer-svm and the activation outputs of the hidden units in unified-dnn. We focused on three toolchains (GCC -O2, ICC -O2 and LLVM -O2). We stress that our observations are anecdotal, meaning that we did not find any conflicting facts, but we cannot prove them either due to the complexity of the data sets and the machine learning models.

A. Different Features Are Selected for Different Toolchains

Two key questions for understanding the impact of compilation toolchains on authorship are (1) do we need different features for different toolchains? And (2) if there are features that are indicative of authorship for all toolchains, do these features contribute similarly to the models for the different toolchains? We try to answer these questions by examining the selected features and their weights in two-layer-svm. Note that while two-layer-svm has lower accuracy than two-layer-dnn and unified-dnn, we choose to analyze two-layer-svm as SVM models are easier to interpret than deep learning models.

For the first question, Figure 4 shows a Venn diagram of the number of uniquely selected features and the number of commonly selected features. We selected 45,000 features each for GCC, ICC, and LLVM. Since some features were selected by more than one toolchain, there are only 88,000 selected features in total. 65% of the selected features are uniquely selected by a single toolchain, with 22%, 23%, and 20% of features uniquely selected by GCC, ICC, LLVM, respectively. On the other hand, only 18% of the selected features are selected by all three toolchains. Among the 88,000 features, there are 591 structural features. 40% of the structural features are uniquely selected by a single toolchain, while 49% of the selected structural features are selected by all three toolchains. Our results show that overall, we need different features for different toolchains, but a much higher percentage of structural features are shared.

For the second question, we analyze the weights of features that are selected by all three toolchains. An SVM model defines a weight for each pair of author $a$ and feature $f$. When $f$ is observed in a basic block, the larger the weight,
the more likely that the basic block is written by $a$. For a given feature, we can generate a ranking of all authors to summarize its contribution to a model. If a feature has similar rankings for different toolchains, we can conclude that this feature contributes similarly to all models for the different toolchains. Note that we analyze the rankings instead of the actual values of weights because the values of weights of a feature also depend on the other features in the model. As GCC, ICC, and LLVM each have uniquely selected features, the actual values of weights in different models are not directly comparable.

We calculated the Spearman correlation [34] to measure similarities between author rankings. The value of the Spearman correlation of two rankings is between -1 and 1, where -1 means opposite rankings, 0 means that there is no association between two rankings, and 1 means the same ranking. Figure 5 shows the histogram of Spearman correlations for the GCC model and the ICC model. We can see that most features have Spearman correlation values between -0.1 and 0.3, indicating that the rankings for GCC has little or no association with the rankings for ICC. We repeated the same analysis for GCC and LLVM, and ICC and LLVM, with results similar to Figure 5. Therefore, our results suggest that commonly selected features typically contribute differently to predictions in different toolchains.

**B. DNNs Can Internally Recognize the Toolchain**

In Section VI-B, our evaluation results show that two-layer-dnn and unified-dnn have similar results. This suggests that unified-dnn can internally recognize the toolchain that generated the input data, even without explicitly training for toolchain identification. Note that a similar phenomena was observed in computer vision research, where Zhou et al. [37] found that a deep learning model trained for scene classification (whether an image describes a office or a bedroom) can internally recognize objects (whether an image contains a desk or a bed), even without explicitly training for object classification.

We adapted the techniques presented by Zhou et al. to investigate whether unified-dnn can internally identity toolchains. The basic idea is that the larger the absolute value of the activation of a hidden unit, the more active the unit. By examining the top $K$ activations generated by using a data set as input, we can determine whether a unit is most active for GCC, ICC, or LLVM code.

We used 500,000 basic blocks as input and made sure that GCC, ICC and LLVM each accounted for about one third of the total input set. We recorded the top $K = 1000$ activations for each unit and summarized each unit with a tuple $(G, I, L)$, where $G$ is the number of GCC basic blocks in the unit’s top $K$ activation list, $I$ is the number for ICC and $L$ is the number for LLVM. We call a unit GCC dominant if GCC basic blocks are the most in the unit’s top 1000 activation list and GCC exclusive if GCC basic blocks constitute more than 90%. Similarly, we have ICC and LLVM dominant and exclusive units.

Table III breaks down unit distributions in different layers. In the first hidden layer, there are many LLVM exclusive units, but almost no ICC or GCC exclusive units. Layer 2 - 5 have similar unit distributions to each other: there are more ICC dominant units than LLVM dominant units, and there is a modest number of GCC dominant units.

We make three observations from our results. First, the large number of LLVM exclusive units suggests that LLVM code has many unique patterns and the network dedicates a significant number of units to represent them. The first hidden layer has a large number of LLVM exclusive units to learn these patterns, indicating that these patterns can be easily derived from the input. Second, the large number of ICC dominant units suggests that ICC code also exhibits distinguishable patterns. Learning these patterns happened mostly in the last four layers. However, the small number of ICC exclusive units suggests that most of these patterns are not unique to ICC code. Third, GCC code has the fewest distinct patterns to learn, where learning the GCC patterns happened mostly in the last four layers. Our observations align with our knowledge about these compilers. Both ICC and LLVM aim to be compatible with and extend GCC, so it is reasonable that GCC code has the fewest unique patterns. On the other hand, as LLVM is developed by a large community of compiler researchers, it is not surprising that LLVM code shows the most unique patterns.

In summary, the distinct activation patterns of our network provide strong evidence that unified-dnn can internally determine which toolchain generated the input basic block.

**C. Unoptimized Code Is More Difficult to Attribute**

Traditional wisdom is that compiler optimization can drastically change the structure of binary and distort the styles of the
TABLE III: Unit distributions in hidden layers.

<table>
<thead>
<tr>
<th></th>
<th>GCC</th>
<th>ICC</th>
<th>LLVM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layer 1</td>
<td>0.6%</td>
<td>24.0%</td>
<td>75.4%</td>
</tr>
<tr>
<td>Layer 2</td>
<td>10.4%</td>
<td>48.6%</td>
<td>6.4%</td>
</tr>
<tr>
<td>Layer 3</td>
<td>7.6%</td>
<td>61.4%</td>
<td>10.5%</td>
</tr>
<tr>
<td>Layer 4</td>
<td>8.0%</td>
<td>55.6%</td>
<td>7.3%</td>
</tr>
<tr>
<td>Average</td>
<td>7.3%</td>
<td>51.4%</td>
<td>5.8%</td>
</tr>
</tbody>
</table>

TABLE IV: Total number of basic blocks (in millions).

<table>
<thead>
<tr>
<th></th>
<th>GCC</th>
<th>ICC</th>
<th>LLVM</th>
</tr>
</thead>
<tbody>
<tr>
<td>O0</td>
<td>1.46</td>
<td>1.55</td>
<td>1.62</td>
</tr>
<tr>
<td>O1</td>
<td>1.30</td>
<td>1.18</td>
<td>1.21</td>
</tr>
<tr>
<td>O2</td>
<td>1.29</td>
<td>2.36</td>
<td>2.36</td>
</tr>
<tr>
<td>O3</td>
<td>1.62</td>
<td>2.36</td>
<td>1.60</td>
</tr>
<tr>
<td>Os</td>
<td>1.11</td>
<td>1.16</td>
<td>1.14</td>
</tr>
</tbody>
</table>

TABLE V: Percentages of basic blocks that contains boilerplate code patterns.

<table>
<thead>
<tr>
<th></th>
<th>GCC</th>
<th>ICC</th>
<th>LLVM</th>
</tr>
</thead>
<tbody>
<tr>
<td>O0</td>
<td>14.0%</td>
<td>17.0%</td>
<td>12.8%</td>
</tr>
<tr>
<td>O1</td>
<td>0.7%</td>
<td>0.2%</td>
<td>2.1%</td>
</tr>
<tr>
<td>O2</td>
<td>0.7%</td>
<td>0.1%</td>
<td>1.2%</td>
</tr>
<tr>
<td>O3</td>
<td>0.5%</td>
<td>0.1%</td>
<td>1.1%</td>
</tr>
<tr>
<td>Os</td>
<td>0.8%</td>
<td>0.2%</td>
<td>1.7%</td>
</tr>
</tbody>
</table>

original authors. So, optimized code should be more difficult to attribute than unoptimized code. However, our results in Table II show the exact opposite: we achieved higher accuracy on optimized code than unoptimized code. Our investigation suggests that compiler optimizations actually work in our favor for improving accuracy. We find two pieces of supporting evidence.

First, when compiling without optimizations, the compiler tends to generate boilerplate code regardless of the authors; in practice such code provide little useful information for learning author style. In optimized code, the boilerplate code is replaced by optimized code, which reflects the structure and style of the surrounding code, so is more useful for learning. In the code that we studied, we identified two prevalent examples of boilerplate code when not using optimizations, function preamble and function epilogue. To estimate the effects of this boilerplate code, we counted the total number of basic blocks and calculated the percentage of basic blocks containing these examples. Table IV shows the total number of basic blocks for each toolchain and Table V shows that -O0 code has a modest number of basic blocks containing boilerplate code (above 12%), while code compiled at other optimization levels have significantly fewer such basic blocks (below 3%). These results suggest that boilerplate code negatively impacts learning.

Second, function inlining can improve learning. When an author’s code is inlined at multiple call sites, the compiler creates more data instances in the author’s style as compared to no inlining. We used Dyninst to understand the debugging information to determine whether a basic block is from an inlined function. Table VI shows that -O0 code has no inlined basic blocks and -O3 code has the most inlined basic blocks. We notice that the percentages of inlined basic blocks are positively correlated to the single-toolchain accuracy achieved by two-layer-dnn. In Figure 6, we show linear regression analysis between the percentages of inlined basic blocks and the single-toolchain accuracy. This linear regression model has an R-squared coefficient of 0.8683 and a p-value of about $10^{-5}$, indicating that there is indeed a strong positive correlation between function inlining and accuracy.

In summary, our investigation suggests that compiler optimizations improve our learning, making unoptimized code more difficult to attribute.

VIII. Conclusion

We have presented new techniques to perform multi-toolchain, multi-author identification. We started with an
extensive evaluation of existing multi-author identification techniques, using our large multi-toolchain data set, which contained 15 toolchains and 700 authors. Our results showed that existing techniques did not work well in multi-toolchain scenarios: the accuracy was as low as 5% when mismatched authorship models were applied. We designed two new techniques to overcome the weaknesses of existing techniques: the two layer approach that first identifies the compilation toolchain that generated the binary and then applied the corresponding single-toolchain authorship model, and the unified training approach that trains multi-toolchain authorship models based on a multi-toolchain data set. We also applied deep learning to multi-author identification, using raw bytes of basic blocks as input to automatically extract low level features, and complementing the input with higher level structural features.

We evaluated our techniques with our multi-toolchain data set, achieving 68% accuracy using deep learning models with the two layer approach. Our results also showed that our deep learning models consistently outperformed traditional learning methods such as SVM; both the two layer approach and the unified training approach can be used for multi-toolchain, multi-author identification. We then tried to understand the internals of our machine learning models by investigating the feature weights of SVM and the activations of neurons in DNN. We learned three lessons from our investigation: different toolchains typically require different features, deep learning models can internally recognize the compilation toolchains without explicitly training for toolchain identification, and unoptimized code is more difficult to attribute.

In summary, we showed that we can perform practical multi-author identification in multi-toolchain scenarios. Our work lays the foundation for future research topics such as whether we can suppress programming styles to evade identification, whether we can impersonate someone else’s coding style to mislead identification, and investigating the impact of code obfuscation techniques on code authorship.

IX. ACKNOWLEDGMENTS

This work is supported in part by National Science Foundation Cyber Infrastructure grants ACI-1547272 and ACI-1449918, the Department of Homeland Security under AFRL Contract FA8750-12-2-0289, and grants from Cray Inc. and Intel Corp.

REFERENCES


