


Identifying Multiple Authors in a Binary Program

Xiaozhu Meng¹, Barton P. Miller¹, and Kwang-Sung Jun²

¹ Computer Sciences Department, University of Wisconsin - Madison,
Madison, WI, USA

{xmeng,bart}@cs.wisc.edu

² Wisconsin Institutes for Discovery, University of Wisconsin - Madison,
Madison, WI, USA

kjun@discovery.wisc.edu

Abstract. Knowing the authors of a binary program has significant application to forensics of malicious software (malware), software supply chain risk management, and software plagiarism detection. Existing techniques assume that a binary is written by a single author, which does not hold true in real world because most modern software, including malware, often contains code from multiple authors. In this paper, we make the first step toward identifying multiple authors in a binary. We present new fine-grained techniques to address the tougher problem of determining the author of each basic block. The decision of attributing authors at the basic block level is based on an empirical study of three large open source software, in which we find that a large fraction of basic blocks can be well attributed to a single author. We present new code features that capture programming style at the basic block level, our approach for identifying external template library code, and a new approach to capture correlations between the authors of basic blocks in a binary. Our experiments show strong evidence that programming styles can be recovered at the basic block level and it is practical to identify multiple authors in a binary.

Keywords: Binary code authorship · Code features · Software forensics

1 Introduction

Binary code authorship identification is the task of determining the authors of a binary program from a set of known authors. This task has significant application to forensic of malicious software (malware) [29], identifying software components from untrusted sources in the software supply chain [11], and detecting software plagiarism [18]. Previous studies [3, 8, 38] have shown that programming styles survive through the compilation process and can be recovered from binary code. However, these studies operated at the program level and assumed that each binary program is written by a single author, which is not true for real world software and significantly limits their uses in practice. In this paper, we present

the first study on *fine-grained* binary code authorship identification, with concrete evidence showing that programming styles can be recovered at the *basic block* level and it is practical to perform authorship identification on binaries from multi-author software.

Knowing the authors of a computer program is a key capability for many analysts. Malware analysts want to know who wrote a new malware sample and whether the authors have connections to previous malware samples. This information can be useful to determine the operations and intentions of the new sample. For example, if several cyber attacks use different malware written by closely connected authors, these attacks can be related and can be a part of a bigger offensive plot. In the domain of software supply chain risk management, analysts can identify untrusted code in the supply chain by matching programming styles against known untrusted software such as malware. The idea of matching programming styles can also be used for detecting software plagiarism, where analysts can match programming styles against known code.

Identifying program authors can be performed on either source code or binary code. Source level techniques [6, 7, 9] are not applicable when the source code is not available, which is typically the case when handling malware, proprietary software, and legacy code. Binary level techniques [3, 8, 38] do not have this limitation and can be used under broader scenarios such as when only a code byte stream is discovered in network packets or memory images.

However, existing binary level techniques assume that a binary is written by a single author, which is not true in real world for two reasons. First, modern software, including malware, is often the result of a team effort. Malware development has become similar to normal software development, evolving from an individual hacking to cooperation between multiple programmers [12, 29, 36]. Malware writers share functional components and adapt them [27, 39]. Studies have shown that malware writers share information and code by forming physically co-located teams [28] or virtually through the Internet [1, 5, 21]. These exchanges of information and code gradually develop into underground black markets [2]. Malware writers can then acquire malware functional components such as command and control, key logging, encryption and decryption, beaconing, exfiltration, and domain flux, to facilitate their development of new malware [42]. This trend indicates that current software, including malware, often contains code from multiple authors.

Second, even if the source code is written by a single author, the corresponding binary may contain code that is not written by this author. External library code, such as the Standard Template Library (STL) and Boost C++ Library (Boost), is often inlined in the binary. In addition, compilers may generate binary code that does not correspond to any source code written by this author, such as the default constructor and destructor of a class.

When applied to multi-author binaries, existing single-author techniques have two significant limitations. First, they can identify at most one of the multiple authors or report a merged group identity. Second, these techniques do not distinguish external library code or code solely generated by the compiler from

the code written by the authors, which may confuse the styles of these libraries or the compiler with the styles of the authors. Therefore, authorship identification techniques must be fine-grained, identify multiple authors, and recognize library code and compiler introduced code in a binary.

Fine-grained techniques can better help analysts in real world applications. Malware analysts can link the code written by the same author from a set of malware samples and link the authors who cooperated in writing a particular piece of malware. The same-author links can help build profiles of individual malware writers and infer their distinct attacking skills and their roles in cyber attacks. The cooperating-author links can help identify connections among malware writers and infer the structure of physically co-located teams or black markets. Malware analysts can determine who or which organization may be responsible for a new malware sample and relate the new sample to existing samples written by the same authors or even samples written by authors who are linked to the identified authors. In the area of software supply chain risk management and detecting software plagiarism, the untrusted code and plagiarized code may consist of only a small fraction of the whole binary, making fine-grained authorship identification essential in these applications.

Previous single-author techniques provide a foundation for us to develop fine-grained techniques. These techniques have cast binary code authorship identification as a supervised machine learning problem. Given a set of binaries with their author labels as the training set, these techniques extract stylistic code features such as byte n-grams [24, 38] and machine instruction sequences [8, 24, 37, 38, 40], and use supervised machine learning algorithms such as Support Vector Machine (SVM) [10] or Random Forests [20] to train models to capture the correlations between code features and author labels. The generated machine learning models are then used to predict the author of a new binary.

To develop fine-grained techniques, three additional challenges must be addressed. First, what is the appropriate unit of code for attributing authorship? Fine-grained candidates include the function or the basic block. However, as programmers could change any line of code or even parts of a line, there is no guarantee that a function or a basic block is written by a single author. Therefore, this challenge requires us to balance how likely a unit of code is written by a single author and how much information it carries.

Second, what stylistic code features do we need for a fine-grained unit of code? Code features used in previous program level studies may not be applicable, or may not be effective at a finer code granularity. As we will discuss in Sect. 7, reusing code features in these program level studies does not yield good results in our case. Therefore, it is crucial to design new fine-grained code features.

Third, how do we identify external template library code? Failing to address this challenge may cause authorship identification algorithms to confuse library code with their users' code. However, existing library code identification techniques are designed for only non-template library code. These techniques either create a function fingerprint [16, 23] or build a function graph pattern based on the program execution dependencies [33]. When a template function is para-

meterized with different data types, the final binary code can be significantly different, making function fingerprints or graph patterns ineffective. Therefore, we need new techniques for identifying template library code.

In this paper, we present the first practical fine-grained techniques to identify multiple authors of a binary. We summarize how we address the three fine-grained challenges and how we capture author correlations between code within the same binary to improve accuracy:

- To determine what granularity of authorship attribution is the most appropriate, we conducted an empirical study on three large and long lived open source projects, for which we have authorship ground truth: the Apache HTTP server [4], the Dyninst binary analysis and instrumentation tool suite [32], and GCC [15]. Our results show that 85% of the basic blocks are written by a single author and 88% of the basic blocks have a major author who contributes more than 90% of the basic block. On the other hand, only 50% of the functions are written by a single author and 60% of the functions have a major author who contributes more than 90% of the function. Therefore, the function as a unit of code brings too much imprecision, so we use the basic block as the unit for attribution. See Sect. 3.
- We designed new code features to capture programming styles at the basic block level. These features describe common code properties such as control flow and data flow. We also designed a new type of code features, *context features*, to summarize the context of the function or the loop to which the basic block belongs. See Sect. 4.
- We made an initial step towards more effective library code identification. This step focuses on identifying inlined code from the two most commonly used C++ template libraries: STL and Boost. We add group identities “STL” and “Boost” to represent the code from these libraries and let the machine learning algorithms to distinguish them from their users. See Sect. 5.
- As a programmer typically writes more than one basic block at a time, we hypothesize that adjacent basic blocks are likely written by the same author. To test our hypothesis, we compared *independent classification models*, which make predictions based only on the code features extracted from a basic block, and *joint classification models*, which make predictions based on both code features and the authors of adjacent basic blocks. See Sect. 6.

We evaluated our new techniques on a data set derived from the open source projects used in the empirical study. Our data set consisted of 147 C binaries and 22 C++ binaries, which contained 284 authors and 900,583 basic blocks. The binaries were compiled with GCC 4.8.5 and -O2 optimization. Overall, our new techniques achieved 65% accuracy on classifying 284 authors, as opposed to 0.4% accuracy by random guess. Our techniques can also prioritize investigation: we can rank the correct author among the top five with 77% accuracy and among the top ten with 82% accuracy. These results show that it is practical to attribute program authors at the basic block level. We also conducted experiments to show the effectiveness of our new code features, handling of inlined STL and Boost code, and joint classification models. See Sect. 7.

In summary, our study provides concrete evidence that it is practical to identify multiple program authors at basic block level. Our ultimate goal is to provide automated fine-grained authorship analysis and many research challenges remain. For example, we are currently investigating the impact of compilers or optimization flags on coding styles, and plan to handle the cases when the target author is not in the training data, and apply our techniques to malware samples. In this paper, we make the first step towards practical fine-grained binary code authorship identification and significantly advance the frontier of this field.

2 Background

We discuss two areas of related work as background: designing binary code features that reflect programming style and using machine learning techniques to discover correlations between code features and authors.

2.1 Binary Code Features

Existing binary code features used in authorship identification describe a wide variety of code properties, such as instruction details, control flow, data flow, and external dependencies. These features were extracted at the function and block level and then accumulated to the whole program level. Basic block level features usually included byte n -grams, instruction idioms, and function names of external library call targets [38]. Function level features mainly include graphlets [8, 38], which represent subgraphs of a function’s CFG, and register flow graphs [3], which captures program data flow. Here, we discuss only basic block level features as the function level features are not applicable at the basic block level.

Byte n -grams represent consecutive n bytes extracted from binary code [24, 38]. Authorship identification techniques typically use small values for n . For example, Rosenblum et al. [38] used $n = 3$ for authorship identification to capture styles reflected on instruction details. While byte n -grams have been shown to be effective, byte n -grams are sensitive to the specific values of immediate operands, and do not capture the structure of programs.

Instruction idioms are consecutive machine instruction sequences [8, 24, 25, 37, 38, 40]. Besides the length of instruction idioms, many other variations have been defined, including allowing wild cards [38], ignoring the order of instructions [25], normalizing operands with generic symbols such as representing all immediate operands with a generic symbol [24, 38], and classifying opcodes into operation categories such as arithmetic operations, data move operations, and control flow transfer operations [40]. Existing techniques for authorship identification typically used short instruction idioms, ranging from 1 to 3 instructions.

2.2 Workflow of Authorship Identification

Instead of designing complicated features to represent specific aspects of programming styles, existing techniques [3, 8, 38] define a large number of simple

candidate code features and use training data to automatically discover which features are indicative of authorship. This feature design philosophy is a general machine learning practice [17].

Based on this feature design philosophy, a common workflow used in existing studies has four major steps: (1) designing a large number of simple candidate features; (2) extracting the defined features using binary code analysis tools such as Dyninst [32] or IDA Pro [19]; (3) determining 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 [38]; and (4) applying a supervised machine learning technique, such as Support Vector Machine (SVM) [10] or Random Forests [20], to learn the correlations between code features and authorship. Rosenblum et al. [38] used instruction, control flow, and library call target features, and used SVM for classification. Caliskan-Islam et al. [8] added data flow features, constant data strings, and function names derived from symbol information, and used Random Forests for classification.

Previous projects performed evaluations of their techniques on single author programs, including Google Code Jam, university course projects, and single author 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. They commented that the university course project data set was significantly more difficult than the Google Code Jam data set, mainly because programs in this data set contained not only code written by individual students, but also skeleton code provided by the course professor. We believe this actually reveals the fundamental limitation of existing program level approaches and motivates our fine-grained techniques.

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 evaluated their technique on single author programs from Github and got 65% accuracy for classifying 50 authors. However, two concerns need further investigations. First, stripping the binaries led to a decrease in accuracy by 24%. In this work, we do not use or depend on symbol information to derive code features. Second, the accuracy of the Github data set is significantly lower than the accuracy of the Google Code Jam data set. They commented two reasons: (1) the binaries from the Github data set were the results of a collaborative effort and each binary had a major author who contributed more than 90% of the code, and (2) the Github data set might contain third party library code that was not written by the major author. Again, we believe our fine-grained techniques can overcome these limitations.

3 Determining Unit of Code

Our fine-grained techniques start with determining whether the function or the basic block is a more appropriate attribution unit. We investigated the authorship characteristics in open source projects to make this granularity decision.

Our study included code from three large and long lived open source projects: the Apache HTTP server [4], the Dyninst binary analysis and instrumentation tool suite [32], and GCC [15]. Intuitively, the more the major author contributes to a function or a basic block, the more appropriate it is to attribute authorship to a single author. We quantify this intuition by first determining how much authorship contribution is from its major author for all basic blocks and functions, and then summarizing these contribution data to compare the basic block with the function.

3.1 Determining Contributions from Major Authors

Our approach to determine the major authors and their contributions can be summarized in three steps:

1. Use *git-author* [30] to get a weight vector of author contribution percentages for all source lines in these projects. Note that well-known tools such as *git-blame* [35], *svn-annotate* [43], and *CVS-annotate* [41] attribute a line of code to the last author who changed it, regardless of the magnitude of the change. We believe *git-author* provides ground truth of higher quality. The source lines of STL and Boost were attributed to author “STL” and “Boost”, respectively.
2. Compile these projects with debugging information using GCC 4.8.5 and `-O2` optimization, and obtain a mapping between source lines and machine instructions. Note that the compiler may generate binary code that does not correspond to any source line. For example, the compiler may generate a default constructor for a class when the programmer does not provide it. We exclude this code from our study.
3. Derive weight vectors of author contribution percentages for all machine instructions, basic blocks, and functions 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. Similarly, we derived the weight vector of a function by averaging the vectors of the basic blocks within the function.

3.2 Study Results

To compare the function with the basic block, we plot the tail distributions of contribution percentages from the major authors. As shown in Fig. 1, 85% of the basic blocks are written by a single author and 88% of the basic blocks have a major author who contributes more than 90% of the basic block. On the other hand, only 50% of the functions are written by a single author and 60% of the functions have a major author who contributes more than 90% of the function. Therefore, the function as a unit of code brings too much imprecision, so we use the basic block as the unit for attribution.

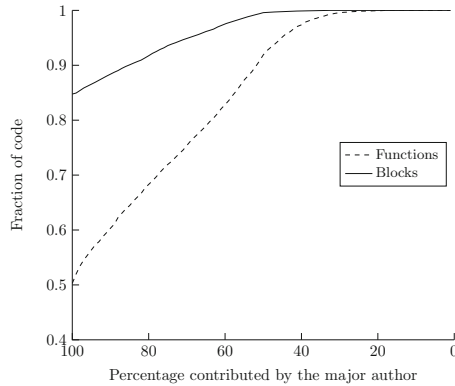


Fig. 1. Tail distributions of major author contribution. The x-axis represents the contribution percentage from the major author. The y-axis represents the fraction of the number of blocks or functions that have a major author who contributed more than a given percentage.

4 New Code Features

We followed an exploratory process for designing new features: designing new features to cover code properties that are not covered, testing new features to see whether they improve accuracy, and keeping those features that turn out to be useful. We have four types of new features: (1) instruction features that describe instruction prefixes, operand sizes, and operand addressing modes, (2) control flow features that describe incoming and outgoing CFG edges and exception-based control flow, (3) data flow features that describe input and output variables of a basic block, stack usages, and data flow dependencies, and (4) *context features* that capture the context of a basic block such as the loops or the functions to which the block belongs. Our new features are summarized in Table 1.

4.1 Instruction Features

There are three new features to describe instruction details.

1. Prefix features: x86 and x86-64 instruction sets contain instruction prefixes that reflect valuable code properties. For example, `REP` (repeat) prefixes are often used for string operations and `REX` prefixes are often used to address 64-bit registers. We count how many times each instruction prefix is used.
2. Operand features: Instruction operands represent the data manipulated by programmers, so we designed instruction operand features to capture operand sizes, types, and addressing modes. First, operand sizes capture the granularity of data and may correlate to the data types operated by a programmer. For example, a one-byte operand often represents a `char` data type in C. We count the number of operands in each operand size. Second, we count the numbers of memory, register, and immediate operands. Third, operand

Table 1. An overview of new basic block level features.

Code property	New block level features
Instruction	(1) Instruction prefixes, (2) instruction operands, (3) constant value in instructions
Control flow	(1) CFG edge types, (2) whether a block throws or catches exceptions
Data flow	(1) # of live registers at block entry and exit, (2) # of used and defined registers, (3) # of input, output, and internal registers of a block, (4) stack height delta of a block, (5) stack memory accesses, (6) backward slices of variables
Context	(1) Loop nesting levels, (2) loop sizes, (3) width and depth of a function CFG, (4) positions of a block in a function CFG

addressing modes can reflect data access patterns used by programmers. For example, PC-relative addressing often represents accessing a global variable, while scaled indexing often represents accessing an array. We count the number of operands in each addressing mode.

3. Constant value features: We count the number of constant values used in a basic block, such as immediate operands and offsets in relative addressing.

4.2 Control Flow Features

We designed control flow features that describe the incoming and outgoing CFG edges on three dimensions: (1) the control flow transfer type (such as conditional taken, conditional not taken, direct jump, and fall through), (2) whether the edge is interprocedural or intraprocedural, and (3) whether the edge goes to a unknown control flow target such as unresolved indirect jumps or indirect calls.

In addition, for languages that support exception-based control flow such as C++, we distinguish whether a basic block throws exceptions and whether it catches exceptions.

4.3 Data Flow Features

Our new data flow features can be classified into three categories.

1. Features to describe input variables, output variables, and internal variables of a basic block: We count the number of input, output, and internal registers. To calculate these features, we need to know what registers are live at the block entry and exit, and what registers are used and defined.
2. Features to describe how a basic block uses a stack frame: Features in this category include distinguishing whether a basic block increases, decreases, or does not change the stack frame, and counting the number of stack memory accesses in the basic block. These stack frame features capture uses of local

variables. Note that stack memory accesses are often performed by first creating an alias of the stack pointer, and then performing a relative addressing of the alias to access the stack. So, data flow analysis is necessary to identify aliases of the stack pointer.

3. Features to describe data flow dependencies of variables: Features in this category are based on backward slices of variables within the basic block. A basic block potentially can be decomposed into several disjoint slices [13]. We count the total number of slices, the average and maximum number of nodes in a slice, and the average and maximum length of slices. We also extract slice n -grams of length three to capture common data flow dependency patterns.

4.4 Context Features

Context features capture the loops and the functions to which a basic block belongs. We count loop nesting levels and loop sizes to represent loop contexts. When a basic block is in a nested loop, we extract loop features from the innermost loop that contains this basic block. For function context, we calculated the width and depth of a function’s CFG with a breadth first search (BFS), in which we assigned a BFS level to each basic block. We also included the BFS level of a basic block and the number of basic blocks at the same BFS level.

5 External Template Library Code

We must distinguish external library code from the code written by their users. In the study discussed in Sect. 3, about 15% of the total basic blocks are STL and Boost code. If we are not able to identify STL or Boost code, our techniques would wrongly attribute this large amount of code to other authors.

Our experience with STL and Boost is that their source code looks significantly different from other C++ code. So, our initial attempt is to add group identities “STL” to and “Boost” to represent each of these libraries. Our results discussed in Sect. 7 show that both STL and Boost have distinct styles and we are able to identify the inlined code.

6 Classification Models

Our next step is to apply supervised machine learning techniques to learn the correlations between code features and authorship. Commonly used machine learning techniques such as SVM [10] and Random Forest [20] perform prediction solely based on individual features. While this is a reasonable approach when operating at the program level, it may not be the case for the basic block level. Based on the intuition that a programmer typically writes more than one basic block at a time, we hypothesize that adjacent basic blocks are likely written by the same author. To test our hypothesis, we built joint classification models, which perform prediction based on code features and who might be the authors

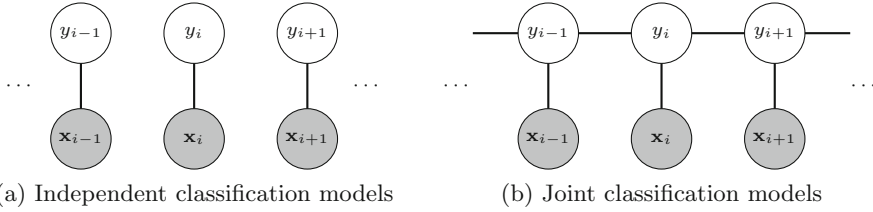


Fig. 2. Comparison between independent classification models and joint classification models. Each basic block has an author label y_i and a feature vector \mathbf{x}_i . An edge connects two inter-dependent quantities. In both models, the author label and feature vector are dependent. In joint classification models, the author labels of adjacent basic blocks are also inter-dependent.

of adjacent basic blocks, and compare them to independent classification models, which perform prediction solely based on code features. The key difference between the two types of models are illustrated in Fig. 2.

For the independent classification models illustrated in Fig. 2a, we divide a binary into a set of basic blocks $B = \{b_1, b_2, \dots, b_m\}$. A basic block b_i is abstracted with a tuple $b_i = (y_i, \mathbf{x}_i)$, where y_i is the author of this basic block and \mathbf{x}_i is a feature vector extracted from the code of this basic block. The training data consists of a set of binaries and we convert them to a collection of training basic blocks B_{train} . Similarly, the testing data consists of a different set of binaries, producing a collection of testing basic blocks B_{test} . The author labels in B_{train} are used for training, while the author labels in B_{test} are not used during prediction and are only used for evaluation. With this modeling, it is straight-forward to use SVM to train independent classification models.

For the joint classification models illustrated in Fig. 2b, we use the same notation $b_i = (y_i, \mathbf{x}_i)$ to represent a basic block, where y_i is the author label and \mathbf{x}_i is the feature vector. The key difference here is that we convert a binary to a sequence of basic blocks $B = \langle b_1, b_2, \dots, b_m \rangle$, where b_i and b_{i+1} are adjacent in the binary. The training data and testing data contain two disjoint collections of basic block sequences. We can then use linear Conditional Random Field (CRF) [26] to train joint classification models.

7 Evaluation

We investigated five aspects of our new techniques: (1) whether we can recover authorship signals at the basic block level, (2) whether our new basic block level features are effective, (3) the impact of the number of selected features, (4) whether the joint classification models based on CRF can achieve better accuracy than the independent classification models based on SVM, and (5) whether we can identify inlined STL and Boost library code. Our evaluations show that:

1. We can effectively capture programming styles at the basic block level. Our new techniques achieved 65% accuracy for classifying 284 authors, compared to 0.4% accuracy by random guess. If a few false positives can be tolerated, we can rank the correct author among the top five with 77% accuracy and among the top ten with 82% accuracy. Our results show that it is practical to perform authorship identification at the basic block level.
2. Our new basic block level features are crucial for practical basic block level authorship identification. F_e represents the existing basic block level features discussed in Sect. 2.1, such as byte n-grams and instruction idioms; F_n represents our new feature set, which is a union of F_e and the new features discussed in Sect. 4. The results of the first row and the sixth row in Table 2 show that adding our new features leads to significant accuracy improvement, adding 26% to 43%, when compared to using only F_e .
3. When operating at the basic block level, we need to select many more features to achieve good accuracy than operating at the program level. As we will discuss in Sect. 7.2, previous program level techniques have selected less than 2,000 features. Our results show that 2,000 features significantly limit the prediction power of our models and we can improve accuracy from 43% to 58% by selecting about 45,000 features at the basic block level, as shown in the sixth and seventh row in Table 2.
4. The CRF models out-perform the SVM models, but CRF requires more training time. As shown in the last two rows in Table 2, the accuracy improved from 58% to 65% when we used CRF for training and prediction. In our experiments, SVM needs about one day for training and CRF needs about 7 times more training time than SVM. Both CRF and SVM can finish predic-

Table 2. Summary of our experiment results. We investigated the impact of three key components of our techniques on the accuracy: our new features, the number of selected features, and the joint classification models. F_e represents the existing basic block level features discussed in Sect. 2.1. We have four types of new features discussed in Sect. 4: instruction, control flow, data flow, and context features, denoted as F_I , F_{CF} , F_{DF} , F_C , respectively. $F_n = F_e \cup F_I \cup F_{CF} \cup F_{DF} \cup F_C$, represents our new feature set.

Classification model	Feature set	Number of selected features	Accuracy
SVM	F_e	2,000	26%
SVM	$F_e \cup F_I$	2,000	31%
SVM	$F_e \cup F_{CF}$	2,000	33%
SVM	$F_e \cup F_{DF}$	2,000	34%
SVM	$F_e \cup F_C$	2,000	38%
SVM	F_n	2,000	43%
SVM	F_n	45,000	58%
CRF	F_n	45,000	65%

tion in a few minutes on binaries of several hundred megabytes. As training new models is an infrequent operation and prediction on new binaries is the major operation in real world applications, it is reasonable to spend more training time on CRF for higher accuracy.

5. We can effectively distinguish STL and Boost code from other code. We calculated the precision, recall, and F1-measure for each author in our data set. Our results show that “STL” has 0.81 F-measure and “Boost” has 0.84 F-measure. For comparison, the average F-measure over all authors is 0.65.

7.1 Methodology

Our evaluations are based on a data set derived from the open source projects used in our empirical study discussed in Sect. 3. Our data set consists of 147 C binaries and 22 C++ binaries, which contains 284 authors and 900,583 basic blocks. C and C++ binaries are compiled on x86-64 Linux with GCC 4.8.5 and -O2 optimization. In practice, newly collected binaries may be compiled with different compilers and optimization levels. We can train separate models for different compilers or optimization levels and then apply compiler provenance techniques [34,37] to determine which model to use. The handling of these cases is the subject of ongoing research.

We used Dynisnt [32] to extract code features, Liblinear [14] for linear SVM, and CRFSuite [31] for linear CRF. We performed the traditional leave-one-out cross validation, where each binary was in turn used as the testing set and all other binaries were used as the training set. Each round of the cross validation had three steps. First, each basic block in the training set was labeled with its major author according to the weight vector of author contribution percentages derived in Sect. 3. Second, we selected the top K features that had the most mutual information with the major authors, where we varied K from 1000 to 50,000 to investigate the how it impacted accuracy. Third, 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 [22], where each round of the cross validation is executed on a separate machine.

An important characteristic of our leave-one-out cross validation is that the author distribution of the training sets is often significantly different from the author distribution of the testing sets. We believe this characteristic represents a real world scenario. For example, an author who wrote a small number of basic blocks in our training data may take a major role and contribute a large number of basic blocks in the new binary. For this reason, we do not stratify our data set, which is to evenly distribute the basic blocks of each author to each testing folds, as stratifying the data set does not represent real world scenarios.

Our data set is also imbalanced in terms of the number of basic blocks written by each author. The most prolific author wrote about 9% of the total basic blocks and the top 58 authors contribute about 90% of the total basic blocks.

We stress that to the best of our knowledge, we are the first project to perform fine-grained binary code authorship identification, so there are no previous

basic block or function level techniques with which to compare. We do not compare with any previous program level techniques either, as these techniques can only attribute a multi-author binary to a single author. We experimented with applying program level techniques to our data set to estimate the upper bound accuracy that can be achieved by any program level technique. As a program level technique reports one author per binary, the best scenario is to always attribute all basic blocks in a binary to the major author of the binary. For each binary in our data set, we counted how many basic blocks were written by its major author and got an average accuracy of 31.6%, which is significantly lower than our reported numbers.

7.2 Impact of Features

As we mentioned before, adding our new features can significantly increase the accuracy. We now break down the contribution from each of our new feature type and investigate the impact of number of selected top features.

Our new features can be classified into four types: instruction, control flow, data flow, and context features. We denote these four types of features as F_I , F_{CF} , F_{DF} , and F_C , respectively. To determine the impact of each feature type, we calculated how much accuracy can be gained by adding only this type of new feature to the existing feature set F_e . In this experiment, we used SVM classification and selected top 2,000 features. As shown in the first row of Table 2, the baseline accuracy of using only feature set F_e is 26%. The second to the fifth row of Table 2 show that adding F_I , F_{CF} , F_{DF} , and F_C leads to 31%, 33%, 34%, 38% accuracy, respectively. Therefore, all of our new features provide additional useful information for identifying authors at the basic block level, with the context features providing the most gain.

In terms of the number of selected top features K , previous program level techniques have shown that a small number of features are sufficient to achieve good accuracy. Rosenblum et al [38] selected 1,900 features. Similarly, Caliskan et al. [7] selected 426 features.

We investigate the impact of K and find that the basic block level needs more features. Figure 3 shows how K affects accuracy. We can see that we need over 20,000 features to achieve good accuracy, which is significantly larger than the number used in previous studies. While the number of selected features is large, we have not observed the issue of overfitting: we repeated this experiment with selecting 100,000 features and got the same accuracy as selecting 50,000 features.

7.3 Classification Model Comparison

Our results show that CRF can achieve higher accuracy than SVM, but requires more training time. Specifically, CRF can significantly improve the accuracy for small basic blocks and modestly improve accuracy for large basic blocks. Figure 4a shows how our techniques work with basic blocks of different sizes. We can see that SVM suffers when the sizes of basic blocks are small, which is

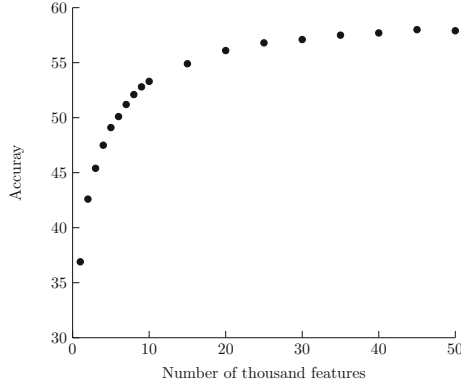
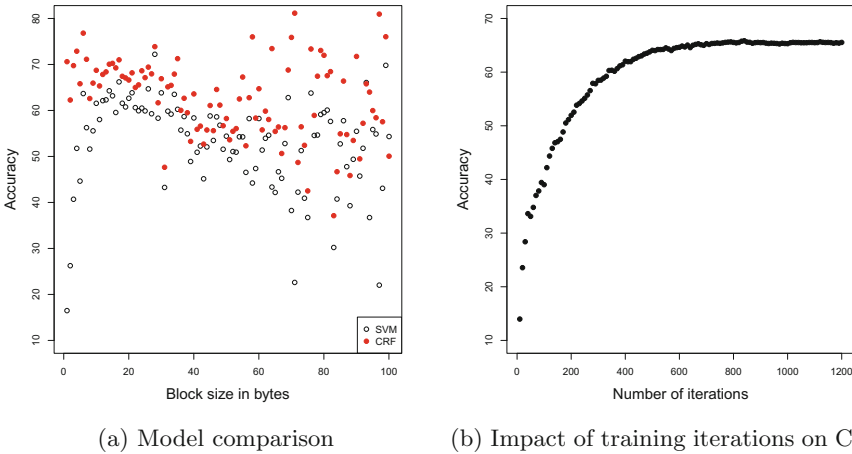


Fig. 3. Experiment results on the impact of the number features. The accuracy results are based on using the new feature set F_n and SVM classification.

not surprising as small basic blocks contain little code, thus few code features and little information. On the other hand, CRF performs better than SVM for all sizes of basic blocks. CRF provides the most benefits when the sizes of basic blocks are small, where we do not have enough information from the code features to make good prediction and have to rely on the adjacency. As the sizes of basic blocks grow, we have more information about these basic blocks and adjacency plays a smaller role in prediction and provides smaller accuracy improvement. We also observe that the accuracy starts to have a large variance when the sizes of basic blocks are larger than 30 bytes. We find that large basic blocks are few in our data set, so their results are unstable.



(a) Model comparison

(b) Impact of training iterations on CRF

Fig. 4. Comparison between CRF and SVM. The results of both figures are based on the new feature set F_n and selecting 45 K features.

The accuracy improvement of CRF comes at the cost of more training time. In our experiments, training SVM needs about one day, while training CRF needs about a week. As CRF training consists of iterations of updating feature weights, there is a trade-off between training time and accuracy. Figure 4b shows how the number of iteration of CRF training impacts the accuracy. In our experiments, we can finish about 150 iterations per day. We need about two days for CRF to reach the accuracy achieved by SVM in one day and need about three days for CRF accuracy to converge.

8 Conclusion

We have presented new fine-grained techniques for identifying the author of a basic block, which enables analysts to perform authorship identification on multi-author software. We performed an empirical study of three open source software to determine the most appropriate attribution unit and our study supported using the basic block as the attribution unit. We designed new instruction, control flow, data flow, and context features, handled inlined library code from STL and Boost by representing them with group identities “STL” and “Boost”, and captured local authorship consistency between adjacent basic blocks with CRF. We evaluated our new techniques on a data set derived from the open source software used in our empirical study. Our techniques can discriminate 284 authors with 65% accuracy. We showed our new features and new classification models based on CRF can significantly improve accuracy. In summary, we make a concrete step towards practical fine-grained binary code authorship identification.

Acknowledgments. This work is supported in part by Department of Energy grant DE-SC0010474, National Science Foundation Cyber Infrastructure grants ACI-1547272, ACI-1449918, Department of Homeland Security under AFRL Contract FA8750-12-2-0289, and a grant from Intel Corporation. This research was performed using the compute resources and assistance of the UW-Madison Center For High Throughput Computing (CHTC) in the Department of Computer Sciences.

References

1. Abbasi, A., Li, W., Benjamin, V., Hu, S., Chen, H.: Descriptive analytics: examining expert hackers in web forums. In: 2014 IEEE Joint Intelligence and Security Informatics Conference (JISIC), Hague, Netherlands, September 2014
2. Allodi, L., Corradin, M., Massacci, F.: Then and now: on the maturity of the cybercrime markets (the lesson that black-hat marketeers learned). *IEEE Trans. Emerg. Top. Comput.* **4** (2015)
3. Alrabaee, S., Saleem, N., Preda, S., Wang, L., Debbabi, M.: Oba2: an onion approach to binary code authorship attribution. *Digit. Investig.* **11**(Suppl. 1), S94–S103 (2014)
4. Apache Software Foundation: Apache http server. <http://httpd.apache.org>

5. Benjamin, V., Chen, H.: Securing cyberspace: identifying key actors in hacker communities. In: 2012 IEEE International Conference on Intelligence and Security Informatics (ISI), Arlington, VA, USA, June 2012
6. Burrows, S.: Source code authorship attribution. Ph.D. thesis, RMIT University, Melbourne, Victoria, Australia (2010)
7. Caliskan-Islam, A., Harang, R., Liu, A., Narayanan, A., Voss, C., Yamaguchi, F., Greenstadt, R.: De-anonymizing programmers via code stylometry. In: 24th USENIX Security Symposium (SEC), Austin, TX, USA, August 2015
8. Caliskan-Islam, A., Yamaguchi, F., Dauber, E., Harang, R., Rieck, K., Greenstadt, R., Narayanan, A.: When coding style survives compilation: de-anonymizing programmers from executable binaries. Technical report. arxiv <http://arxiv.org/pdf/1512.08546.pdf>
9. Chatzicharalampous, E., Frantzeskou, G., Stamatatos, E.: Author identification in imbalanced sets of source code samples. In: 2012 IEEE 24th International Conference on Tools with Artificial Intelligence (ICTAI), Athens, Greece, November 2012
10. Cortes, C., Vapnik, V.: Support-vector networks. *Mach. Learn.* **20**(3), 273–297 (1995)
11. Croll, P.R.: Supply chain risk management—understanding vulnerabilities in code you buy, build, or integrate. In: 2011 IEEE International System Conference (SysCon), Montreal, QC, Canada, April 2011
12. de la Cuadra, F.: The geneology of malware. *Netw. Secur.* **4**, 17–20 (2007)
13. David, Y., Partush, N., Yahav, E.: Statistical similarity of binaries. In: 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Santa Barbara, California, USA, June 2016
14. Fan, R.E., Chang, K.W., Hsieh, C.J., Wang, X.R., Lin, C.J.: Liblinear: a library for large linear classification. *J. Mach. Learn. Res.* **9**, 1871–1874 (2008)
15. GNU Project: GCC: the GNU compiler collection. <http://gcc.gnu.org>
16. Guilfanova, I., DataRescue: fast library identification and recognition technology (1997). <https://www.hex-rays.com/products/ida/tech/flirt/index.shtml>
17. Guyon, I., Elisseeff, A.: An introduction to variable and feature selection. *J. Mach. Learn. Res.* **3**, 1157–1182 (2003)
18. Hemel, A., Kalleberg, K.T., Vermaas, R., Dolstra, E.: Finding software license violations through binary code clone detection. In: 8th Working Conference on Mining Software Repositories (MSR), Waikiki, Honolulu, HI, USA, May 2011
19. Hex-Rays: IDA. <https://www.hex-rays.com/products/ida/>
20. Ho, T.K.: Random decision forests. In: 3rd International Conference on Document Analysis and Recognition (ICDAR), Montreal, Canada, August 1995
21. Holt, T.J., Strumsky, D., Smirnova, O., Kilger, M.: Examining the social networks of malware writers and hackers. *Int. J. Cyber Criminol.* **6**(1), 891–903 (2012)
22. HTCondor: High Throughput Computing (1988). <https://research.cs.wisc.edu/htcondor/>
23. Jacobson, E.R., Rosenblum, N., Miller, B.P.: Labeling library functions in stripped binaries. In: 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools (PASTE), Szeged, Hungary, September 2011
24. Jang, J., Woo, M., Brumley, D.: Towards automatic software lineage inference. In: 22nd USENIX Conference on Security (SEC), Washington, D.C. (2013)
25. Khoo, W.M., Mycroft, A., Anderson, R.: Rendezvous: a search engine for binary code. In: 10th Working Conference on Mining Software Repositories (MSR), San Francisco, CA, USA, May 2013

26. Lafferty, J.D., McCallum, A., Pereira, F.C.N.: Conditional random fields: probabilistic models for segmenting and labeling sequence data. In: 8th International Conference on Machine Learning (ICML), Bellevue, Washington, USA, June 2001
27. Lindorfer, M., Di Federico, A., Maggi, F., Comparetti, P.M., Zanero, S.: Lines of malicious code: insights into the malicious software industry. In: 28th Annual Computer Security Applications Conference (ACSAC), Orlando, Florida, USA, December 2012
28. Mandiant: Mandiant 2013 threat report. Mandiant White paper (2013). https://www2.fireeye.com/WEB-2013-MNDDT-RPT-M-Trends-2013_LP.html
29. Marquis-Boire, M., Marschalek, M., Guarnieri, C.: Big game hunting: the peculiarities in nation-state malware research. In: Black Hat, Las Vegas, NV, USA, August 2015
30. Meng, X., Miller, B.P., Williams, W.R., Bernat, A.R.: Mining software repositories for accurate authorship. In: 2013 IEEE International Conference on Software Maintenance (ICSM), Eindhoven, Netherlands, September 2013
31. Okazaki, N.: Crfsuite: a fast implementation of conditional random fields (CRFs) (2007). <http://www.chokkan.org/software/crfsuite/>
32. Paradyn Project: Dyninst: Putting the Performance in High Performance Computing. <http://www.dyninst.org>
33. Qiu, J., Su, X., Ma, P.: Library functions identification in binary code by using graph isomorphism testings. In: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Montreal, Quebec, Canada, March 2015
34. Rahimian, A., Shirani, P., Alrbaee, S., Wang, L., Debbabi, M.: Bincomp: a stratified approach to compiler provenance attribution. *Digit. Investig.* **14**(Suppl. 1), S146–S155 (2015)
35. Rahman, F., Devanbu, P.: Ownership, experience and defects: a fine-grained study of authorship. In: Proceedings of 33rd International Conference on Software Engineering (ICSE), Waikiki, Honolulu, HI, USA, May 2011
36. Roberts, R.: Malware development life cycle. In: Virus Bulletin Conference (VB), October 2008
37. Rosenblum, N., Miller, B.P., Zhu, X.: Recovering the toolchain provenance of binary code. In: 2011 International Symposium on Software Testing and Analysis (ISSTA), Toronto, Ontario, Canada, July 2011
38. Rosenblum, N., Zhu, X., Miller, B.P.: Who wrote this code? Identifying the authors of program binaries. In: 16th European Conference on Research in Computer Security (ESORICS), Leuven, Belgium, September 2011
39. Ruttenberg, B., Miles, C., Kellogg, L., Notani, V., Howard, M., LeDoux, C., Lakhota, A., Pfeffer, A.: Identifying shared software components to support malware forensics. In: 11th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA), Egham, London, UK, July 2014
40. Sæbjørnsen, A., Willcock, J., Panas, T., Quinlan, D., Su, Z.: Detecting code clones in binary executables. In: 18th International Symposium on Software Testing and Analysis (ISSTA), Chicago, IL, USA, July 2009
41. Śliwerski, J., Zimmermann, T., Zeller, A.: When do changes induce fixes? In: Proceedings of 2005 International Workshop on Mining Software Repositories (MSR), St. Louis, Missouri, USA, May 2005
42. Yavvari, C., Tokhtabayev, A., Rangwala, H., Stavrou, A.: Malware characterization using behavioral components. In: 6th International Conference on Mathematical Methods, Models and Architectures for Computer Network Security (MMM-ACNS), St. Petersburg, Russia, October 2012

43. Yin, Z., Yuan, D., Zhou, Y., Pasupathy, S., Bairavasundaram, L.: How do fixes become bugs? In: Proceedings of 19th ACM SIGSOFT Symposium and 13th European Conference on Foundations of Software Engineering (ESEC/FSE), Szeged, Hungary, September 2011