Fine-Grained Binary Code Authorship Analysis: Identification and Evasion

by

Xiaozhu Meng

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2018

Date of final oral examination: August 17th, 2018

To Xiaoying.

ACKNOWLEDGMENTS

Pursuing a Ph.D. is a long journey. Without the help and support from many people, there is no chance that I can be where I am now.

My advisor, Bart Miller, patiently taught me how to be an independent researcher. I needed to be able to identify the key issues in a problems, strategically plan multiple threads of work, and effectively communicate with others. It took me several years to have a better understanding of how to do this, and understand that I can always improve myself. As a non-native English speaker, Bart also spent enormous amount of time editing my writing, correcting my grammar and wording. I appreciate all the lessons from Bart throughout the years.

I would like to thank the other members of my Ph.D. committee. Somesh Jha pointed me in a new research direction that ended up as my last technical chapter. Without his advice, I cannot imagine how much longer it would take me to complete my Ph.D. Jerry Zhu shared his encyclopedic knowledge of machine learning and gave me valuable feedback on machine learning throughout my years in Madison. Kevin Roundy agreed to be join my defense committee. As he was not on my Preliminary Exam committee, he provided a fresh perspective on my research. In addition, my first year at Madison overlapped with Kevin's last year at Madison. I appreciate his help for getting me on board with Dyninst. Paul Wilson rescued my defense, agreeing to be in my defense committee with a short notice in advance.

I am fortunate to work under the Paradyn/Dyninst research project. Bill Williams shared the office with me for the past six years. I learned a lot from his programming expertise; his work on debugging Dyninst saved my time to focus on my research. Benjamin Welton joined the group in the same year as I did. We supported each other to survive on our long journeys of pursuing a Ph.D. Wenbin Fang was the first person I contacted before joining group. He answered my questions and was the first person to go to for help. Andrew Bernat graduated in my first year and continued to work as a staff member in my second year. His knowledge of Dyninst was comprehensive and he helped to explain what Bart means from a different perspective.

During my years in Madison, I am fortunate to have met many new friends. Linhai Song, Jun He, Zuyu Zhang, and Peng Liu, to name a few, are precious friends to have.

My family give me enormous support. My parents support me emotionally and financially. When the research was not going well, when I was feeling pressures from life and work, they always encouraged me and supported me. My cousins took care of my parents while I was far away from them. I must thank Wei Zhou, who unconditionally came to my parents when they needed help; I wish him a peaceful rest in heaven.

Finally, my wife has been by my side for many years, always supporting me. I thank her for her unending patience and the happiness she brings to me.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Binary code authorship identification is the task of determining the authors of a binary program. It has significant application to forensic analysis of malicious software (malware), software supply chain risk management, and software plagiarism detection. The key message of this dissertation is that authorship identification techniques *must be fine-grained and be able to identify multiple authors in a binary*. This is because modern software, including malware, is typically developed by multiple programmers.

The opposite side of authorship identification is to evade authorship identification. Authorship evasion has the goal of modifying a binary so that authorship identification would generate misleading results. Assessing authorship identification from an attacker's perspective is a key step for improving the robustness of authorship identification techniques.

In this dissertation, we develop new fine-grained techniques for identifying authors of binary programs and new techniques for evading identification. We first describe our new techniques for deriving accurate source code authorship from software repositories, which serves as the foundation for deriving high quality ground truth for evaluating our new fine-grained techniques for binary code authorship identification. We then describe our new techniques for identifying multiple authors in a binary and our new techniques for reliably identifying multiple authors when the binaries are compiled by different compiler families, versions, and optimization levels. Last, from an attacker's perspective, we describe our new techniques for authorship evasion.

1 INTRODUCTION

Recovering code authorship is the task of determining the authors of a computer program. This task can have many variations, such as determining how many authors there are of a program, whether there are unknown authors, and whether two pieces of code are written by the same author. The ability to perform these tasks has significant application to forensics of malicious software (malware), detecting software intellectual property violations, and identifying software components from untrusted sources in the software supply chain. Malware analysts are eager 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 supported by the same organization and can be a part of a bigger offensive plot. In the domain of software intellectual property, software plagiarism can be detected by matching programming styles against known code. The idea of matching programming styles can also be used for software supply chain risk management, where untrusted code in the supply chain can be identified by matching programming styles against known untrusted software such as malware.

Existing techniques for recovering code authorship work with either source code or binary code. Most techniques have focused on source code authorship [19, 21, 24, 36, 71, 123]. These techniques are not applicable when the source code is not available, which is often the case when handling malware, proprietary software, and legacy code. Techniques that work with binary code 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 instead of a complete program.

However, recovering binary code authorship has unique challenges.

First, compilers remove many source level stylistic features, such as code comments, space indentation, and identifier names. Second, compilers perform optimizations that can drastically change the structure of the original program and may distort programming styles. Third, even if certain stylistic features survive compilation, it may not be easy to extract them from binaries as the compiler generates challenging code constructs that make various tasks of binary code analysis difficult [83]. For example, it is difficult to build accurate Control Flow Graphs (CFGs) of program binaries due to challenging code constructs such as non-code bytes embedded in code sections, indirect control flow, functions laid out non-contiguously in memory, functions sharing code, and tail calls. Despite these challenges, recent studies have shown that it is possible to recover programming styles from binaries [4, 21, 108].

While encouraging, these studies have assumed that each binary is written by a single author. This single-author assumption does not hold for modern software, as modern software is often the result of a team effort. Even malware development has become similar to normal software development, evolving from an individual hacking to cooperation between multiple programmers [31, 77, 106]. Malware writers share functional components and adapt them [73, 90, 94, 112]. Studies have shown that malware writers share information and code by forming physically co-located teams [76] or virtually through the Internet [2, 10, 54]. Through black markets, malware can be shared, sold or purchased [3, 115]. 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 [27, 90, 94, 131]. This trend indicates that current malware often contains code from multiple authors and the connections between the authors could be key to trace cyber attacks.

In addition, 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, *an approach of binary code authorship analysis must be fine-grained and be able to identify multiple authors in a binary.* This is the key area of contribution of this dissertation.

1.1 Advantages of Fine-Grained Analysis

Fine-grained binary code authorship analysis enables new ways of performing real world analysis tasks including malware forensics, detecting software plagiarism, and software supply chain risk management.

Existing techniques for malware forensics have inferred various types of relations between malware samples, including identifying common functional components [112], determining malware evolution lineage [29, 60, 88], recognizing well-known malware variants [56, 59, 96], and classifying behaviors of malware [105, 131]. Even though they capture valuable connections between malware, these techniques would miss connections between malware samples that are written by common authors, but exhibit different malware behaviors, attacking techniques, or code structure. These connections between authors can be crucial for tracing modern attacks.

The current trend of malware development and the limitations of existing techniques for malware forensics inspire us to build social networks of malware writers to capture their social connections and establish connections between malware written by them. The basic idea is to link the code written by the same author 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 close to the identified authors in the social networks. Fine-grained binary code authorship analysis is the foundation for building these networks.

The need for detecting software intellectual property violations is also increasing. Either driven by commercial interests or being unintentional, individuals or companies may bring in and adapt external software without respecting the corresponding software license terms [18, 38, 49, 79]. For example, Compuware filed various intellectual property claims against IBM and the two companies reached $400 million settlement agreement [1]. Discovering these violations in the wild is challenging as commercial software is usually distributed in binary form and the plagiarized code may have been changed for adaptation or for evading detection [61, 75, 125].

Existing techniques for detecting software plagiarism are based on the following observation. If the suspicious software contains plagiarized code, the plagiarized code should be similar to its original version. If we can measure the similarity between two pieces of binary code, we can then match the suspicious software against a large set of known software to detect plagiarism. This observation has inspired a large amount of research on measuring binary code similarity [30, 49, 61, 64, 70, 75, 119, 125]. These studies have measured binary code similarity based on the four dimensions: constant data such as string literals and integers; code structure such as how similar are the CFGs; code semantics such as whether two pieces of code provide similar functionalities; and program runtime behaviors such as the traces of executed instructions. However, the offender may slightly change the the plagiarized code for adaptation or avoiding detection and these changes can lead to significant differences on the four similarity dimensions. On the other hand, the plagiarized code in the suspicious software would still exhibit the programming styles of its original authors despite of these small changes, making fine-grained binary code authorship analysis essential for detecting software plagiarism.

Fine-grained binary code authorship analysis can also be used to detect untrusted software components in a software supply chain. Large software systems often integrate software components from software suppliers without verifying their trustworthiness and some software components may even come from unknown sources [46]. These untrusted supply chains can create significant security and reliability risks to the whole software system [26, 35]. By matching programming styles against a large set of untrusted code such as malware, we can determine whether the software supply chains contain code from dangerous origins and whether we need thorough investigation before integrating certain software components.

1.2 Challenges of Fine-Grained Analysis

To develop fine-grained techniques, several challenges must be addressed. First, having high quality ground truth is essential for evaluating our fine-grained techniques. As we no longer make the assumption that a program binary is written by a single author, the ground truth should be able to tell us who are the authors of any piece of code within a program binary, such as the authors of a function, a basic block, and even a machine instruction. To the best of our knowledge, there is no such data set available and we need to construct new datasets satisfying this requirement.

Second, what is the appropriate unit of code for attributing authorship? Previous single-author techniques report one author per binary, so they used the program as the attribution unit. 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.

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 [44, 58] or build a function graph pattern based on the program execution dependencies [102]. When a template function is parameterized 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.

Fourth, authorship identification algorithms must be reliable across different compilation toolchains. Here, we define the compilation toolchain as the combination of the compiler family, version, and optimization level. Different compilation toolchains may generate significantly different binary code for the same source code. This makes it difficult to reliably recognize authors as the same code written by the same author may exhibit drastically different structures when compiled by different compilation toolchains.

Fifth, is it possible to evade authorship identification? When programmers are aware of authorship identification techniques, they may want to evade identification for various reasons, including avoiding criminal charges for writing malware or avoiding plagiarism detection. It is important to be able to recognize authors under such adversarial setting. To answer this question, we assess authorship identification from an attacker's point of view, which is to perform authorship evasion. Performing authorship evasion can help us better understand the strengths and limits of identification algorithms, and identify opportunities for improving them.

1.3 Guiding Principles

We have identified three principles to guide our research of developing fine-grained binary code authorship analysis techniques.

Machine learning driven Previous techniques for single-author identification have provided 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 [60, 108] and machine instruction sequences [21, 60, 107, 108, 113], and use supervised machine learning algorithms such as Support Vector Machines (SVMs) [25] or Random Forests [53] 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. We base our research on this principle and push it to the next level. For example, we will show that deep learning can automatically extract a broad class of style features from binary code.

Deep understanding of learning models While the application of machine learning has been successful in various domains, an unsatisfying phenomenon is that it is typically difficult to interpret the results generated by the machine learning systems. It is important for forensic analysts to know why the learning systems generate such results so that they can acquire concrete evidence. Throughout our dissertation, we not only aim to develop effective fine-grained techniques, but also attempt to investigate the learning models and explain the results.

Wearing the black hat The field of adversarial machine learning has shown real threats to machine learning systems used in real world applications [11]. When developing new fine-grained identification techniques, we attempt to put on our black hat and critically assess our designs.

1.4 Techniques and Contributions

This dissertation describes four key techniques to address the challenges presented in Chapter 1.2. Our new techniques constitute the main contributions of this dissertation.

1.4.1 Source Code Authorship

Having high quality ground truth is the foundation for developing fine-grained binary code authorship identification techniques. We derive ground truth for each machine instruction in a binary program and get ground truth for larger pieces of code by accumulating machine instruction level authorship. Our approach is to derive accurate source line authorship from open source project repositories, compile the source with debugging information, and map each machine instruction back to source lines.

However, commonly used version control systems only provide limited support for determining source code authorship. Tools such as git-blame [33, 104], svn-annotate [132], and CVS-annotate [117] lose information as they only report the last commit that changed a line of code. Even when the last commit changed only a small fraction of a line of code, the author of the last commit still is credited for the entire line. We implement a new tool, *git-author*, to overcome these limitations [84]. *git-author* traverses the development history of a repository and calculates a vector of author contribution weights for each line of code. We use *git-author* to derive fine granularity ground truth for binary code.

We describe our techniques for deriving accurate source code authorship in Chapter 3.

1.4.2 Identify Multiple Authors

To determine whether to use the function or the basic block as the attribution unit, we conducted an empirical study of three large and long-lived open source projects, for which we have authorship ground truth: the Apache HTTP server [7], the Dyninst binary analysis and instrumentation tool suite [100], and GCC [40]. The results of our study support the use of the basic block as the attribution unit.

Our new techniques for identifying multiple authors contain three components:

- We design new basic block level features to capture programming style at the basic block level. Our new features describe common code properties such as control flow and data flow. We also design a new type of code features, *context features*, to summarize the context of the function or the loop to which the basic block belongs.

- We make 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 use the machine learning algorithms to distinguish them from their users.

- 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 compare *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. We found that joint classification models significantly out-performed independent classification models.

We describe our techniques for identifying multiple authors in Chapter 4.

1.4.3 Multi-Toolchain, Multi-Author Identification

To investigate the impact of compilation toolchains, we first conduct a study to evaluate how well our multi-author identification techniques described in Chapter 4 will function in a multi-toolchain scenario. Our results reveal two limitations. First, when models trained with binaries compiled by one toolchain are applied to binaries compiled by another toolchain, the identification accuracy is low. Second, there is a significant difference between the accuracy of different single toolchain scenarios; the optimization levels influence the accuracy. This accuracy difference suggests that programmers can have a higher chance of evading current multi-author identification by carefully choosing a toolchain. These two limitations confirm the need for new techniques for identifying multiple authors in multi-toolchain scenarios.

Naively, applying models trained for one toolchain to binaries compiled by another toolchain leads to low accuracy. To address this situation, we investigated two new approaches. The first approach is a two layer one. Since previous techniques work better in single toolchain scenarios, it is reasonable to first use toolchain identification techniques [103, 107] to determine which toolchain generated the binary and then apply the corresponding single-toolchain authorship model. Note that while previous authorship identification studies [4, 21] have discussed this approach, none of them have implemented or evaluated it. The second approach is to use unified training, 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 of 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 describe our techniques for multi-toolchain, multi-author identification in Chapter 5.

1.4.4 Authorship Evasion

We perform authorship evasion to have a better understanding of authorship identification techniques. We chose the techniques presented by Caliskan-Islam et al. [21] for single-author identification as our evasion target. We make this choice for two reasons. First, evading multi-author identification techniques present much greater challenges compared to evading single-author identification. Thus, evading single-author identification is the natural first step. Second, techniques presented by Caliskan-Islam et al. [21] represent an external view of performing authorship identification. Evading their techniques helps avoid a parochial view of authorship identification, if we only attempted to evade our own techniques.

We make two main assumptions about the threat model. First, the attackers plan to perform a test time attack, so they can affect the prediction results only by providing a crafted input binary. Other possible attacks against learning systems such as training set poisoning [12, 80] are not in the scope of this study. Second, the attackers have perfect knowledge of the target identification classifier. This assumption allows performing a worst-case evaluation of the security of the authorship identification techniques, common when performing test time attacks [13, 23, 116, 121].

Authorship identification techniques have a training stage and a testing stage. While we do not directly attack the training stage, three choices made in this stage impact our attacks. First, the design of the binary code features determines the program properties of the binary to modify during attacks. Features are typically defined to describe program properties including machine instructions, program control flow, constant strings, and program meta-data such as function symbols. Second, identification techniques use binary code analysis tools such as Dyninst [100], NDISASM [122] or Radare2 [97] for feature extraction. A key part of our attack is to modify the binary and trick the binary code analysis tools into extracting modified features to cause misprediction. Third, based on the machine learning algorithm used by the identification technique, the attacker may need to use different attack algorithms to determine which features should be modified to cause misprediction. There are existing attack algorithms for a variety of learning models, including Deep Neural Networks (DNNs) [23], Random Forests (RFs) [63], and Support Vector Machines (SVMs)[13, 43].

The testing stage has two key steps: extracting code features from the input binary to construct a feature vector and applying the pre-trained model on the feature vector to generate the prediction results.

Targeting the two steps in the testing stage, our evasion attack focuses on developing two interacting attacking abilities: *feature vector modification*, changing the feature vector to cause mis-prediction of the target classifier, and *input binary modification*, modifying the input binary to match the adversarial feature vector while *maintaining the functionality of the input binary*. Feature vector modification guides what input binary modification should be performed to cause mis-prediction, while input binary modification gives feedback to feature vector modification as to which features are difficult to modify, guiding feature vector modification to avoid modifying difficult features. These two attacking abilities form an attack-verify loop.

We describe our evasion techniques in Chapter 6.

2 RELATED WORK

This dissertation explores code authorship identification and evasion. Three essential topics of this research are: designing binary code features that reflect programming styles, using machine learning techniques to discover correlations between code features and authors, and abusing the discovered correlations to evade identification. We survey related work in each of these three topics.

We first examine binary code features used in existing single-author identification techniques and other related research areas, such as measuring binary code similarity (Section 2.1). We then discuss existing techniques for single-author identification, including their workflow and evaluation results (Section 2.2). Next, we discuss related work in evading authorship identification (Section 2.3). We conclude this chapter with a summary of the related work (Section 2.4).

2.1 Binary Code Features

Instead of designing complicated features to represent specific aspects of programming styles, we 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 [45] and has been used in previous binary code authorship identification techniques [21, 108]. One key aspect of applying this feature design philosophy is to design code features that can cover a wide variety of code properties, such as instruction details, control flow, data flow, program semantics, and external dependencies. We review binary code features that have been used in existing binary code authorship techniques [4, 21, 108] and also features used in other related research areas such as binary code similarity [30, 49, 61, 64, 70, 75, 119, 125]. Even though code features in other research areas are used to capture different information, these features can complement and inspire our feature design.

2.1.1 Features for Binary Code Authorship

In previous code authorship studies, binary code features were extracted at the function and block level and then accumulated to the whole program level. Block level features usually included byte n-grams [21, 108], instruction idioms [4, 108], and function names of external library call targets [108]. Our current work continues to include these features when working at the basic block level. Function level feature types mainly include graphlets [21, 108], which represent subgraphs of a function CFG, and register flow graphs [4], which captures program data flow. We cannot directly reuse these function level features, so need to design new basic block level features to capture code properties such as control flow and data flow.

Byte n-grams represent consecutive n bytes extracted from binary code [59, 60, 108]. Commonly used values of n range from 3 to 16. When n is small, byte n-grams can capture information within individual instructions, such as specific instruction opcodes, immediate operands, and memory operands. For example, Rosenblum et al. [108] used $n = 3$ for authorship analysis to capture styles reflected on instruction details. On the other hand, longer byte n-grams can be used to measure similarity between binaries. For example, Jang et al. [59] used $n = 16$ to locate similar malware. While byte n-grams have been shown to be effective in many studies, byte n-grams are sensitive to the specific values of immediate operands in instructions, and do not capture the structure of programs.

Instruction idioms are consecutive machine instruction sequences [21, 60, 64, 107, 108, 113]. Besides the length of instruction idioms, many other variations of instruction idioms have been defined, including allowing wild cards [108], ignoring the order of instructions [64], normalizing operands with generic symbols such as representing all immediate operands with a generic symbol [60, 108], and classifying opcodes into a small number of operation categories such as arithmetic operations, data move operations, and control flow transfer operations [113]. Existing binary code authorship techniques typically have used short instruction idioms ranging from 1 to 3 instructions.

Graphlets represents subgraphs of a function CFG to capture program control flow [5, 64, 103, 108]. A node in a graphlet represents a basic block. In some variations, colors are assigned to nodes, where the color is derived from the mix of opcodes of the instructions in the basic block [5, 103, 108]. An edge in a graphlet means that the two basic blocks are adjacent in the function CFG. Rosenblum et al. [108] assigned labels to edges to distinguish the control flow types, such as conditional taken, conditional not taken, direct jump, and fall through.

Register flow graphs were designed to capture program data flow and are extracted at the function level [4, 5]. The basic idea was to perform perform backward slicing on the registers in all `cmp` and `test` instructions and convert each slice to a hash value. While existing techniques only extract register flow graphs for `cmp` and `test` instructions, we believe the data flow of registers used in other instructions, such as memory and arithmetic instructions, could also reflect programming style.

2.1.2 Features for Binary Code Similarity

Techniques in the area of binary code similarity are based on code features that measure structural and semantic differences between two pieces of code. Besides the features described in Section 2.1.1, other features used in binary code similarity include "strands" capturing program data flow [30] and "code juice" capturing program semantics [70, 112].

David et al. [30] define a strand to represent a backward slice of an instruction within the basic block. A basic block can then be decomposed into several potentially disjoint strands. David et al. also defined a similarity measure between two strands, where the similarity of two pieces of binary code can be calculated based on the similarity measures between the two sets of strands extracted from the binary code.

Code juice [70, 112] represents the semantics of a basic block with a set of symbolic expressions. The specific register names and constants in symbolic expressions are replaced with generic symbols for generalization. Ruttenberg et al. [112] converted the symbolic expressions of each basic block to a hash value, represented a binary function with a set of hash values, and calculated the similarity between two binary functions by measuring the similarity between the two sets of hash values.

Even though strands and code juice are both extracted at the basic block level, our experiment results presented in Section 4.3 do not support directly reusing them for authorship analysis. Strands and code juice are not suitable for authorship because they represent complete data flow dependencies and program semantics of basic blocks and highly depend on the functionality of the code rather than the style. Nevertheless, as we will discuss in Section 4.2, they provide a foundation for us to design new basic block level code features to capture program data flow and semantics.

In summary, only a few types of existing features can be directly reused in our research when operating at basic block level, including byte n-grams, instruction idioms, and library call targets. These features only cover a small subset of code properties, so we need to design new basic block level features to cover other code properties such as control flow, data flow, and program semantics.

2.2 Binary Code Authorship Identification

Previous code authorship identification techniques have used machine learning techniques to discover the relationship between code features and authors. We discuss the commonly used workflow, relationship between accuracy and the complexity of the programs used in evaluation, and relationship between accuracy and the compilation toolchains.

2.2.1 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 [52] or Dyninst [100];
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 [108]; and
4. Apply a supervised machine learning technique, such as SVM [25] or Random Forests [53], to learn the correlations between code features and authorship.

Rosenblum et al. [108] used instruction, control flow, and library call target features, and used SVM for classification. Caliskan et al. [21] added data flow features, constant data strings, and function names derived from symbol information, and used Random Forests for classification.

2.2.2 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. Our fine-grained techniques do not suffer the same limitation.

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 takes 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. For this reason, we will use large and long-lived open source software to evaluate our techniques.

2.2.3 Impact 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 [103, 107] to identify the toolchain.

Hendrikse [50] 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 believe that previous studies on the impact of compilation toolchains are inconclusive due to the small scale of experiments and the focus on single author identification. Our research found that the impact of compilation toolchain on code authorship to be greater at the basic block level compared to the program level, as compiler optimizations may create, modify, and delete basic blocks. Therefore, it was crucial to investigate the impact of compilation toolchains on fine-grained authorship identification.

2.3 Evasion of Authorship Identification

To the best of our knowledge, there is no existing work on evading binary code authorship identification. The closest was a study conducted by Simko et al. [116], with the goal of evading source code single-author identification.

Their evasion target is a classifier that has 98% accuracy on classifier 250 authors, evaluated with the Google Code Jam [20]. The classifier used lexical features such as variable names and language keywords, layout features such as code indentation, and syntactic features derived the abstract syntax trees parsed from the source code.

28 programmers participated their study, including undergrad students, former or current software developers. Each programmer was given code from author X and Y and then was asked to modify source code written by X to look like code written by Y. This manual attack achieved 80% success rate for causing misprediction.

Simko et al. then inspected the modified source code and summarized the most common modifications:

1. Copy entire lines of code written by Y into code written by X;
2. Make typographical changes such as brackets, newlines, space between operators;
3. Modify variable names and the location of variable declarations, typically either from or to a global variable;
4. Add or swap library calls; and
5. Change the source code structure such as adding or removing macros, changing loop types, or breaking up an if-statement

These changes are mostly local, involving a few lines of code. The study participants did not need to understand the structure or the functionality of the code to make such modifications. The modification strategies presented in this study are unlikely to achieve equal success for evading binary code authorship identification, as many of the modifications are irrelevant at the binary code level, such as typographic changes, variables renaming, and modifying macros.

2.4 Summary

While prior work has designed many types of binary code features for tasks such as single author identification and measuring binary code similarity, only a few types of existing features can be directly reused in our research when operating at basic block level. How to effectively capture programming styles at the basic block level is a theme throughout this dissertation. We describe our new block level features in Chapter 4, and describe how we use deep learning to automatically extract style features in Chapter 5.

Prior work leaves unanswered questions on the impact of compilation toolchains on authorship. 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. We explore them in Chapter 5.

The evasion of authorship identification has been performed only at the source code level and done manually. Many of the source code authorship evasion techniques do not apply at the binary code level. Developing an automated way for performing evasion of binary code authorship identification is the focus of Chapter 6.

3 MINING SOFTWARE REPOSITORIES FOR ACCURATE AUTHORSHIP

High quality ground truth is essential for evaluating fine-grained techniques for binary code authorship identification. To perform evaluation on large, long-lived open source software, we need to derive accurate source line authorship from software repositories. However, current tools in common version control systems approximate line level authorship by assuming that the last person to change a line is its author, while ignoring all earlier changes. Such example tools include git-blame [33, 104], svn-annotate [132], and CVS-annotate [117].

In this chapter, we describe our approach for deriving accurate authorship for each source line by mining software repositories. We describe our abstraction for software repositories in Section 3.1. In Section 3.2, we define *structural authorship* to capture the complete development history of a source line and present an algorithm for computing structural authorship. In Section 3.3, we define *weighted authorship* to summarize the development history of a source line with a vector of author contribution weights and present an algorithm for computing the contribution vectors.

We implemented our structural authorship model and weighted authorship model in a new git built-in tool: *git-author*. We describe the two experiments for evaluating the effectiveness of *git-author* in Section 3.4. In the first experiment, we measured the quantity of the multi-author lines in five open source repositories. Our results show that 10% of source lines contain development contributions from multiple authors. *git-author* provides more information than *git-blame* on these lines. In the second experiment, we show that we can use the additional authorship information to improve source code bug prediction to prioritize software testing [47, 48, 62, 92, 93]. This experiment shows that our new authorship models have a boarder impact than deriving ground truth for evaluating fine-grained binary code authorship identification.

3.1 Repository Abstraction

We define the repository graph to capture the fundamental capability of a version control system (VCS). With the repository graph, we can focus on the contents of development history without considering which specific VCS is used. A VCS records the development history of a project by storing all the revisions of source code and the dependent relationship between these revisions. Our graph abstraction models revisions as nodes and the relationship between revisions as edges. We are able to implement the graph structure based on any current mainstream VCS.

A VCS allows programmers to checkpoint their changes. A new revision is created when a programmer commits their modifications to the VCS. The dependencies between revisions also is recorded to maintain the relative order of commits. Current VCS's support concurrent development. Programmers can work on different branches without affecting other people's work and later combine their work by merging branches. Therefore, it is also necessary to record on which existing revision the new revision is based. In addition to these basic capabilities, a VCS often supports reverting previous changes, browsing development history, and other complementary capabilities to facilitate daily development work.

The repository graph is a directed graph $G = (V, E, \Delta)$ used to describe the basic capability of a VCS. A node in V represents a revision or a snapshot of the project and is annotated with information about the snapshot including the author of the snapshot. The snapshot of node i is denoted as s_i, $s_i \in V$, and the author is denoted as a_i. Node s_0 is a virtual node representing the empty repository before any changes are committed. E is the set of edges, representing development dependencies between revisions. Δ is a labeling of E that represents code changes and there is a

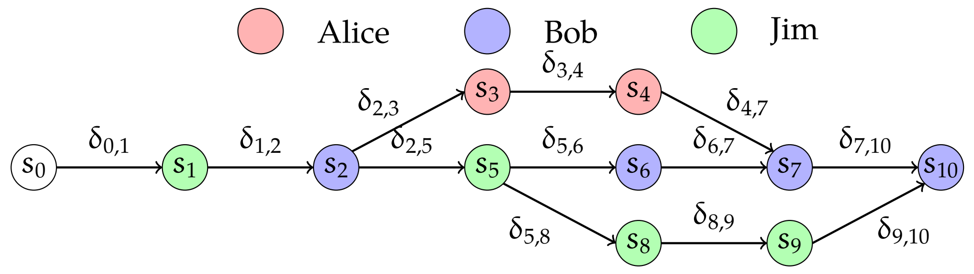


Figure 3.1: **An example of the repository graph.** Nodes are source code revisions, denoted from s_0 to s_{10}. The color of a node shows the author creating the revision. The virtual node s_0 has no author information. Edges represent development dependencies between revisions. $\delta_{i,j}$ on edge $e_{i,j}$ is the code change from s_i to s_j.

one-to-one mapping between the elements in E and Δ. We adapt our definition of code changes from Zeller and Hildebrandt [133], where a change δ is a mapping from old code to new code. An edge $e_{i,j}(\delta_{i,j})$, $e_{i,j} \in E$ and $\delta_{i,j} \in \Delta$, means that by applying the change $\delta_{i,j}$ to s_i, we can get code snapshot s_j; so $\delta_{i,j}(s_i) = s_j$. We define $\delta_{i,j}$ to be a tuple of $(\mathcal{D}_{i,j}, \mathcal{A}_{i,j}, \mathcal{C}_{i,j})$ where $\mathcal{D}_{i,j}$ is the set of lines deleted from s_i, $\mathcal{A}_{i,j}$ is the set of lines added to s_i, and $\mathcal{C}_{i,j}$ is the set of pairs of lines changed from s_i to s_j. For node s_i, s_j, and s_k such that $e_{i,j} \in E$ and $e_{j,k} \in E$, we define the composition of change sets as $\delta_{i,k} = \delta_{j,k} \circ \delta_{i,j}$ meaning applying $\delta_{i,j}$ first, and then $\delta_{j,k}$. Our definition implies that the operator $\circ$ is right associative. One key property of the composition operation is that the result of composition of change sets is path independent. The result only depends on the two end nodes.

We illustrate our definition in Figure 3.1. The repository consists of ten revisions (ten nodes) and three developers: Alice, Bob, and Jim. The author information for a node is represented by its color. The virtual node s_0 has no author information, so we leave it blank. Alice created a branch for her work and committed s_3 and s_4. Later Bob merged Alice's work back to the master branch and created s_7. For the path independent property, we have $(\delta_{4,7} \circ \delta_{3,4} \circ \delta_{2,3})(s_2) = (\delta_{6,7} \circ \delta_{5,6} \circ \delta_{2,5})(s_2) = s_7$. The first part in the equation is the composition along Alice's branch. The second one is the composition along the master branch. The two paths yield the same overall effects, which is the third part in the equation.

Our definition of the repository graph is applicable to any current mainstream VCS. To demonstrate that, we consider how to derive the nodes, edges, and the change sets on edges in three popular version control systems: git, svn, and CVS. Git and svn store each commit as a snapshot of the repository, so the commits correspond to the nodes in the repository graph. CVS on the other hand stores commits as the change set containing added lines and deleted lines. We can derive the contents of nodes by composing consecutive change sets. All the three version control systems record branching and merging, so edges are easy to find. Git and svn provide built-in differencing tools to calculate change sets, but they are not sufficient for our definition of δ because they report changed lines separately as added lines and deleted lines. *ldiff* [22] calculates source code similarity metrics (such as best edit distance and cosine similarity) to match added lines and deleted lines and derive pairs of changed lines. We use *ldiff* to implement our definition of δ. Since we can implement the repository graph on any mainstream VCS, we assume a code repository is represented as a repository graph in this chapter.

3.2 Structural Authorship

Structural authorship represents the development history of a line of code. We define structural authorship as a subgraph G_l of the repository graph G that includes only the revisions that change line l of code, and the development dependences between these revisions. We present a backward flow analysis algorithm on the repository graph G that extracts the structural authorship. Our analysis processes all lines in a file to provide sufficient context for programmers to view code history. After extracting structural authorship, analysis tools have access to all historical information of a line so that they are not limited to the last change of that line.

Our structural authorship model can be seen as a generalization of the current method that only reports the last change. Both our model and the current method stop searching the history of a line when the line is found to be added. The distinction is that our model can make use of the information in the set of changed lines $\mathcal{C}$, while the current method cannot.

3.2.1 Model definition

For a given line of code l appearing in a revision s_v, the *structural authorship* of the pair of (s_v, l) is defined to be a directed graph $G_l = (V_l, E_l, \Delta_l)$. V_l is the set of nodes that changed or added line l. E_l is the set of edges that represent development dependences between nodes in V_l. Δ_l is a labeling of E_l that represents code changes. Before giving the formal definitions of V_l, E_l, and Δ_l, we first introduce notation to describe the relationships between nodes and then extend our definition of $\delta_{i,j}$.

We define $s_i \to s_j$ if and only if there is a directed path in G from s_i to s_j. For the starting revision s_v, its ancestor set contains the potential revisions that could be in V_l. We define the ancestor set of a node s_i as

$$ance(s_i) = \{s_k \in V | s_k \to s_i\}$$

To determine what lines a node s_i has changed or added, we define the total effect of s_i as:

$$\delta_i = \bigcup_{s_k \in pred(s_i)} \delta_{k,i}$$

$$\mathcal{D}_i = \bigcup_{s_k \in pred(s_i)} \mathcal{D}_{k,i}$$
$$\mathcal{A}_i = \bigcup_{s_k \in pred(s_i)} \mathcal{A}_{k,i}$$
$$\mathcal{C}_i = \bigcup_{s_k \in pred(s_i)} \mathcal{C}_{k,i}$$

Now we can define V_l as the set of revisions of s_v and its ancestors that add or change the line l:

$$V_l = \{s_j \in (ance(s_v) \cup \{s_v\}) | l \in (\mathcal{A}_j \cup \mathcal{C}_j)\}$$

An edge in E_l represents a path that does not go through nodes in V_l. For s_i and s_j such that $s_i \to s_j$, we define $s_i \xrightarrow{\overline{V_l}} s_j$ if and only if there exists one or more directed paths from s_i to s_j and none of the intermediate nodes on the path are in V_l. This relationship is used to describe the development dependency between two nodes in V_l. We can define E_l as:

$$E_l = \{e_{i,j} | (s_i, s_j \in V_l) \wedge (s_i \xrightarrow{\overline{V_l}} s_j)\}$$

Note that a single $e_{i,j}$ in E_l can result from multiple directed paths in the original G.

We now extend our definition of $\delta_{i,j}$ to the case where $s_i \to s_j$ so that δ can be used to describe Δ_l. If $<s_i, s_{k_1}, \ldots, s_{k_m}, s_j>$ is a directed path from s_i to s_j, then

$$\delta_{i,j} = \delta_{k_m,j} \circ \delta_{k_{m-1},k_m} \circ \cdots \circ \delta_{i,k_1}$$

Note that the specific choice of the path is not important because the result of composition of change sets is path independent. Δ_l then can be defined as

$$\Delta_l = \{\delta_{i,j} | e_{i,j} \in E_l\}$$

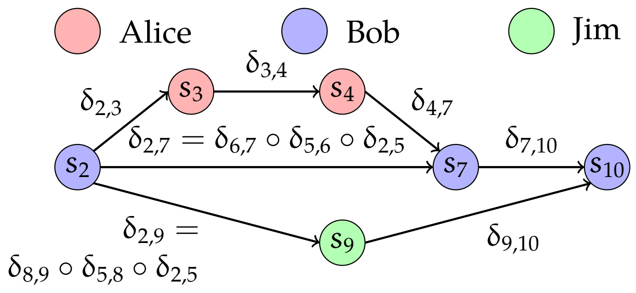


Figure 3.2: **An example of the structural authorship graph.** Nodes in $V_l = \{s_2, s_3, s_4, s_7, s_9, s_{10}\}$ changed or added line l. Edges represent extended development dependencies between revisions. $\delta_{i,j}$ on edge $e_{i,j}$ is the extended code change from s_i to s_j.

We illustrate our subgraph definition with an example. In the repository graph G shown in Figure 3.1, suppose we have the following scenario: Line l was first introduced into the project by Bob in revision 2 (s_2). Alice changed l in revisions 3 and 4 in her branch. Jim changed l in revision 9 in his branch. Bob merged Alice's branch in revision 7. Since Alice and Jim made independent changes to l, when Bob finally tried to merge Jim's branch, Bob had to solve the conflict by taking either Alice's change or Jim's change; we assume that Bob took Jim's change. The structural authorship G_l is shown in Figure 3.2.

3.2.2 Backward flow analysis

We calculate the structural authorship graphs in two steps. In the first step, we use a backward flow analysis to calculate V_l. In the second step, a depth first search is used to calculate E_l and Δ_l. In our repository graph abstraction, V and E can be directly accessed through API of the underlying VCS, but we have to use *ldiff* to calculate Δ in our analysis.

In the first step, we use the backward flow analysis shown in Figure 3.3 to extract V_l from the repository graph G. We perform our analysis on

```
   input  : $V$, $E$, $F$, and $s_v$
   output: $\{V_l | l \in F\}$
   // The live lines that can reach $s_v$
 1 $liveLines[s_v] \leftarrow F$;
 2 for $s_i \in (ance(s_v) \cup \{s_v\})$ in reverse topological order in $G$ do
       // Phase 1:  calculate $\delta$ for $s_i$
 3     for $s_k \in pred(s_i)$ do
 4         $\delta_{k,i} \leftarrow ldiff(s_k, s_i, F)$;
 5         $\delta_i \leftarrow \delta_i \cup \delta_{k,i}$;
       // Phase 2:  update $V_l$
 6     for $l \in liveLines[s_i]$ do
 7         if $l \in \mathcal{A}_i \cup \mathcal{C}_i$ then
 8             $V_l \leftarrow V_l \cup \{s_i\}$;
       // Phase 3:  pass live lines to preds
 9     for $s_k \in pred(s_i)$ do
10         for $l \in liveLines[s_i]$ do
11             if $l \notin \mathcal{A}_{k,i}$ then
12                 $liveLines[s_k] \leftarrow liveLines[s_k] \cup \{l\}$;
13     $liveLines[s_i] \leftarrow \emptyset$;
```

Figure 3.3: *S-Author*: An algorithm that extracts V_l for all lines of code in file F starting at revision s_v

all of the lines in a file F rather than an individual line l for two reasons. First, by processing all lines in F together, we can order the computation so that we neither make redundant calls to *ldiff* nor store the results of *ldiff*. Second, programmers usually want to view code history in a context, so presenting histories of several lines together is more useful.

Our algorithm calculates dataflow information for each node and adds nodes to V_l. For node s_i, its dataflow information records the live lines that can reach the starting node s_v from s_i before being deleted. We use a map *liveLines* that associates a node to a set of live lines to efficiently update the dataflow information. At the beginning, all lines in F are live (line 1).

Because G is acyclic, the traditional work list algorithm for dataflow analysis is not necessary in our case. It is sufficient to visit each node from s_v in the reverse topological order of G (line 2). For each node s_i, there are three major phases: calculating change sets (lines 3-5), updating V_l (lines 6-8) and passing live lines to the predecessors of s_i (lines 9-12).

In phase 1, we call *ldiff* to calculate a subset of Δ that are sufficient and necessary for the next two phases. In phase 2, for each live line l, we determine whether s_i is in V_l or not (line 7). In phase 3, we check whether the current live lines will still be live in each predecessor s_k of s_i (line 11). It is possible that l will be dead along one branch, but still be live along another branch.

The analysis finishes after it visits the virtual node s_0. As a special case, we can add s_0 to V_l to represent the state where l has not yet been introduced into the repository. For any $l \in F$, V_l are the nodes in the structural authorship graph.

The memory used for the results of *ldiff* in phase 1 can be freed after the phase 3 in this iteration. *ldiff* produces the δ between two files and has a relative high time complexity, quadratic in terms of the size of the files [22]. Caching the results of *ldiff* can avoid redundant calls to *ldiff*. But we estimate that caching the results of *ldiff* on a large code repository could take a few gigabytes of memory, which is too much for a built-in tool for a VCS.

In the second step, for each node that we have determined is in V_l, we can do a depth first search in G to calculate E_l and Δ_l according to our definitions.

The running efficiency of our algorithms both depends on the actual sizes of structural authorship graphs. G_l could be as large as G in theory. However, G_l is usually small in practice (Section 3.4) and our algorithms show good performance.

3.3 Weighted Authorship

The structural authorship graph G_l represents the complete development history of a line of code l. However, existing analysis tools typically operate on numerical or ordinal features rather than a graph, so we wish to provide summaries of this information in a form such tools can consume. We define the *weighted authorship* of l to be a vector of author contribution weights. For each author, we can then use the weighted authorship to determine their contribution, model their familiarity of the line, or estimate their efforts spent on the line. This type of summary information is often used to analyze software quality [15, 104], help familiarize new developers [37], and estimate software development cost [87].

3.3.1 Model description

For a line of code l, we define the *weighted authorship* W_l as a vector $(c_1, c_2, \ldots, c_m)$. Each element c_i is the percentage of contribution made by developer i; elements in W_l sum to 1. m is the total number of developers that changed l. By examining G_l, we can determine the value of m. We define each c_i to be the number of characters attributed to developer i divided by the total number of characters in l. For example, if Alice, Bob and Jim are developers 1, 2 and 3, $W_l = (30\%, 20\%, 50\%)$ means that Alice, Bob, and Jim contribute 30%, 20%, 50% of the line respectively. We use characters as the unit of contribution because it is simple and avoids being dependent on the programming language used. While we do not consider the semantics of the code, we do collapse white space to minimize the effects of simple formatting changes. We do not isolate the affect of each of these choices, however the experiments in the following section show that these choices produce satisfactory results.

```
input  : $l$, $V_l$, $E_l$, and $\Delta_l$
output: $\mathtt{attr}$: maps a character in $l$ to its attributed node
1 Let $s_v$ be the last node that changed $l$;
  // The live characters that can reach $s_v$
2 $liveC[s_v] \leftarrow l$;
3 for $s_i \in V_l$ in reverse topological order in $G_l$ do
4     if $|\mathtt{pred}(s_i)| == 1$ then
          // $s_i$ is created by a normal commit
5         Let $s_k$ be the element in $\mathtt{pred}(s_i)$;
6         $chars \leftarrow AC\text{-}BestEdit(l, \delta_{k,i})$;
7         for $c \in liveC[s_i] \cap chars$ do
8             if $(c \notin attr.keys())$ or $(tstamp(s_i) < tstamp(attr[c]))$ then
9                 $attr[c] \leftarrow s_i$;
10        $liveC[s_k] \leftarrow liveC[s_k] \cup (liveC[s_i] - chars)$;
11    else
          // $s_i$ is created by a merge commit
12        for $s_k \in \mathtt{pred}(s_i)$ do
13            $chars \leftarrow AC\text{-}BestEdit(l, \delta_{k,i})$;
14            $liveC[s_k] \leftarrow liveC[s_k] \cup (liveC[s_i] - chars)$;
```

Figure 3.4: *W-Author*: An algorithm calculating the attribution map for l

3.3.2 Algorithm

We calculate W_l based on G_l. We first attribute each character in l to the node that introduced that character and then attribute each node to the appropriate developer. We define the attribution map *attr* to maintain this character-to-node attribution. The node-to-developer attribution can be done by checking the author label of each node.

We use the algorithm shown in Figure 3.4 to compute the attribution map *attr*. The idea is to attribute a character to the node in which the character is added or changed. The algorithm first finds the last revision s_v that changed l; this s_v is the starting point of our algorithm (line 1). For each node in G_l, we maintain the live characters that can reach s_v before being deleted. All characters in l at s_v are live (line 2). We visit each node in G_l in reverse topological order. For each node s_i, we distinguish whether s_i is created by a normal commit or a merge commit by checking the number of its predecessors (line 4). In both case, we define *AC-BestEdit* to calculate the set of characters added or changed in this node (line 6 and 13). These characters are not passed to the predecessors of s_i. For a normal commit, we update the attribution map and pass the live characters (lines 5-10). For a merge commit, we only pass the live characters (lines 12-14).

AC-BestEdit adapts the Wagner-Fischer algorithm [128] for computing the best edit distance to calculate the set of characters in l added or changed by s_i. In the Wagner-Fischer algorithm, the best distance is defined as the minimum number of steps needed to change a source string to a target string. Each step can be adding, deleting, or substituting a character. The algorithm computes a shortest path and returns the minimal number of steps. For an edge $e_{k,i} \in E_l$, the string in s_k is the source string and the string in s_i is the target string. *AC-BestEdit* calculates the shortest path to change the source string to the target string using Wagner-Fischer, and it returns the set of characters added or changed by s_i.

A normal commit has a single predecessor s_k. A character that is added or changed in this node may be also added or changed independently in other nodes (in other branches). Since characters are the unit of contribution, we do not divide the contribution of a character among the multiple commits. In this case, we attribute the character to the node with the earlier commit timestamp (line 8).

For a merge commit, we assume that the commit is either produced during an automatic merge by the VCS or manual selections from one of the multiple branches; therefore a merge commit does not introduce new characters. Since the added or changed characters in one branch actually come from other branches, we just ignore these characters in this merge commit and attribute them to other branches.

The performance of the algorithm depends on the size of G_l. As we will discuss in the next section, the size of G_l is usually small. In our experience, running this algorithm on all lines in a file finishes in around second.

3.4 Evaluation

We implemented our structural authorship model and weighted authorship model in a new git built-in tool: *git-author*. *git-author* uses a syntax similar to that of *git-blame* so has a familiar feel to current users of git. We designed two experiments to compare our new authorship models to the current model that only reports the last change to a line. In the first experiment, we ran *git-author* on five open source code repositories to study the number of lines that were changed in multiple commits and the number of lines that were changed by multiple authors. This experiment shows that *git-author* can recover more information than *git-blame* on about 10% of lines. The results show that most lines are touched only by one author in one commit and the cooperation between developers is restricted to small regions of code. We hypothesized that these small regions of code contain rich information about the software development process and that analysis tools can benefit from this extra information. We conducted our second experiment to verify this hypothesis. Our second experiment evaluated whether the additional information would be useful to build a better analysis tool. We built a new line-level model for source code bug prediction and compared it with the best previously report work on a file-level model [62]. We found that our line-level model consistently performed better than the file-level model. This demonstrates that our new authorship models can help build better analysis tools.

3.4.1 Multi-author study

In this experiment, we ran *git-author* on the following five open source projects: Dyninst [100], the Apache HTTP server [7], GCC [40], the Linux

Table 3.1: **Number of lines changed by multiple commits and multple authors.** The second column shows the number of lines changed in multiple commits and the percentage they account for in the repository. The third column shows the same information for lines that changed by multiple authors.

Repository	Multi. Commits	Multi. Authors	# of lines
Dyninst	53K (12.11%)	40K (9.12%)	434K
Httpd	27K (10.90%)	20K (8.15%)	247K
GCC	279K (8.08%)	217K (6.27%)	3454K
Linux	1440K (9.69%)	1072K (7.22%)	14857K
GIMP	122K (12.82%)	78K (8.12%)	955K

Kernel [74], and Gimp [41], extracting the structural authorship for each line of the code. We then counted the number of nodes and the number of authors in each structural authorship graph. Note that we did not run *git-blame* on the five projects because *git-blame* would output only one commit and one author for each line of code.

The results are shown in Table 3.1. About 10% of lines are changed by multiple commits and about 8% of lines are changed by multiple authors. *git-author* produces more information than *git-blame* on these lines.

3.4.2 Line-level bug prediction

Our second experiment evaluated whether the information provided by *git-author* would be helpful to build a better bug prediction model. We show that we can build a line-level bug prediction model that is more effective than the best previously reported work on a file-level model by Kamei, Matsumoto at el. [62]. To the best of our knowledge, we are the first project to try to predict bugs at the line level.

We first give an overview of bug prediction and our experiment. We then introduce our new line-level model and the file-level model we compared it to. We discuss our data sets and the metrics used to evaluate the models. Finally, we present our results.

Overview

Many research efforts have been dedicated to source code bug prediction to prioritize software testing [47, 48, 62, 92, 93]. Two comprehensive surveys are from Arisholm, Briand, al el. [8] and D'Ambros, Lanza al et. [28].

Three decisions affect the performance of a bug prediction model: the granularity of prediction, a set of bug predictors, and a machine learning technique that trains the model and predicts bugs. Using *git-author* changes the granularity of prediction to the line level and introduces new bug predictors. We do not explore the influence of different machine learning techniques as it is beyond the scope of this paper.

Most of the existing source code bug prediction models predict at the granularity of a source file [47, 91] or even a module [92, 93]. The disadvantage of coarse-grained prediction models is that, even if the prediction results are accurate, developers still have to spend effort to locate the bugs within a module or file. Predicting at a finer granularity, such as at the method level can help to reduce the problem [48, 65]. Our line-level model can locate the suspected lines and help focus testing efforts. It uses the development history of lines of code provided by *git-author* to make predictions. Note that since the development history of a line of code produced by *git-blame* is incomplete, it is impractical to do line-level prediction with *git-blame*. We compared our line-level model to the file-level model because predicting at a file level is well understood.

Two types of bug predictors are commonly used: product predictors that summarize code in the predicting snapshot [136] and process predictors that summarize the history of the predicting snapshot [91]. The process predictors have been shown to be more effective than the product predictors [62, 91]. In our experiment, most of our predictors are process

Table 3.2: **Bug predictors used in the study.**

Level	Predictor name	Definition
Line	WA	Weighted authorship defined in Section 4
	NOA	# of authors
	NOC	# of commits
	LEN	Length of the line
	VAR	Variance of the length of the line across all commits in G_l
	FIX	# of times a line involved in a bug-fix commit
	REF	# of times a line involved in a refactoring commit
	COM	Whether a line is a comment
	AGE	The age of the line
File [62]	Codechurn	Sum of (added lines of code - deleted lines of code)
	LOCAdd	Sum of added lines of code over all revisions
	LOCDel	Sum of deleted lines of code over all revisions
	Revisions	# of revisions
	Age	The age of the file
	BugFixes	# of times a file involved in a bug-fix commit
	Refactor	# of times a file involved in a refactoring commit

predictors.

Many machine learning techniques have been adopted for bug prediction. However, previous studies have shown that the influence of bug predictors on the final prediction results is much larger than the chosen machine learning technique [8, 62]. Therefore, we selected linear learning techniques for both our line-level model and the file-level model. We do not believe this choice will have a noticeable effect on our results.

Models

The goal of our line-level model is, given a line of code, to output the probability that the line is buggy. Based on these outputs, a developer could prioritize testing of the software to the lines with higher probabilities of being buggy. We used a linear SVM as the learning technique in our line-level model [34]. The predictors in our new model are shown in Table 3.2. We introduce new predictors including the weighted authorship, the length of the line, the variance of the length of the line across all commits in G_l, and whether the line is a comment. The other predictors were adapted from existing file-level predictors. We compute the values of these line-level predictors from the outputs of *git-author*.

We compared our line-level model to the file-level model from Kamei, Matsumoto et al. [62]. Their model outputs the predicted fault density when given a file. They compared the prediction results of using process predictors and product predictors with three learning techniques: linear regression [32], regression tree [16], and random forest [17]. Their results showed that using process predictors produced consistently better results than using product predictors and combining them together did not provide further advantages. Therefore, we implemented the file-level process predictors listed in Table 3.2. We chose the logistic regression [34], one type of linear regression, as the learning technique of the file-level model to match the linear SVM used in our line-level model.

Note that when evaluating the effects of *git-author*, it would have been preferable to use the same machine learning technique in the line-level model and the file-level model. However, because the outputs of the line-level model and the file-level model are different, we cannot use the exact same learning technique. Therefore, we can only try to minimize the effects on performance from the factors rather than *git-author*.

Data collection

We are unaware of existing bug prediction data sets with line-level predictors; instead we generated new such data sets. Producing a bug prediction data set takes two steps. We first create a *bug map* from a bug record in the bug database to the pair of commits that caused the bug and fixed the bug. We then choose a time point, typically a release, and use the bug map to produce data instances for this snapshot. The second step is repeated at several different release time points so that we can do cross release prediction.

For the first step, we used the SZZ algorithm [117] to find buggy commits and the corresponding *fixing commits* that fixed the bugs in the Apache HTTP server repository. The quality of the results in this step is improved by *Relink* [129], which addresses the problem of missing valid mappings in the original SZZ algorithm [14].

In the second step, we projected the *bug map* onto the chosen snapshot. A bug is relevant to the snapshot if and only if the snapshot is inside the time interval between the buggy commit and the fixing commit. For each relevant bug, we first produced line-level data, and then summarized the data into file-level data. We assume the lines that are deleted or changed in the fixing commit are the buggy lines. Two methods can be used to summarize the line-level data. We can either count all buggy lines as a single bug or count the lines separately. The first method assumes that it takes the same effort to fix every bug, while the second method takes this factor into consideration. We adopted both methods and produced two data sets.

We collected data for seven releases in the Apache HTTP server project and produced two data sets described above. The first data set is denoted as "Bug count" and the other one is denoted as "Line count". Table 3.3 summarizes our data sets.

Evaluation metrics

Many metrics are used to evaluate bug prediction models. The most commonly used metrics include precision and recall [92, 93], the area under the curve (AUC) of ROC curves [86, 91], and effort-aware metrics

Table 3.3: **Summary of the data sets.** Each row in the table summarizes the number of files, bugs, lines of code, and buggy lines in a release snapshot of Apache.

Release	# of files	# of bugs	SLOC	# of buggy lines
2.1.1	305	129	177K	670
2.2.0	319	171	202K	746
2.2.6	320	167	205K	708
2.2.10	321	172	207K	664
2.3.0	383	179	207K	680
2.3.10	372	195	218K	747
2.4.0	362	181	223K	555

[62, 81]. Comparison studies have shown that the choices of metrics can significantly affect the performance of prediction models [8, 28]. The difference of performance on metrics does not mean inconsistent results because different metrics are designed to answer different questions. We use the effort-aware metrics because they are domain specific metrics for bug prediction. They measure not only the accuracy of the predicting results but also the efforts needed to fix the bugs.

In our study, we use two effort-aware metrics: P_{opt}, which measures the closeness of a model to the optimal file level model [81] and cost-effectiveness (CE), which measures the advantages of that model over a random prediction model [8]. The idea of effort-aware metrics is that a developer can first test or inspect the most suspicious lines or the files with largest fault densities and see what percentage of bugs can be found. The assumption is that the effort needed to test a piece of code is roughly proportional to the size of the code [8]. Using the percent of lines tested as the x-axis and the percent of bugs covered as the y-axis, we can draw a curve to visualize the performance of a model. We denote the area under the curve of a model m as $AUC(m)$. P_{opt} and CE can be defined as:

$$P_{opt}(m) = 1 - (AUC(FileOptimal) - AUC(m))$$

$$CE(m) = \frac{AUC(m) - AUC(Random)}{AUC(FileOptimal) - AUC(Random)}$$

In the above formulas, the file optimal model tests files in decreasing order of the fault densities. It represents the upper bound of a file level model. The random model orders the files randomly. We use the average performance of the random model in the CE formula, which is a straight line from $(0, 0)$ to $(1, 1)$. For both P_{opt} and CE, larger values mean better performance. When the values are larger than 1, the model m performs better than the optimal file-level model.

Results

We performed cross release prediction on our data set. We chose cross release prediction instead of cross-validation inside a release because the cross-release prediction represents the real practice of how a bug prediction model is used. We used *Liblinear* to do training and prediction on our two data sets [34]. We denote our line-level model as lm, the file-level model as fm, and the optimal file-level model as fm_o.

In the "Bug count" data set, we need to aggregate line-level prediction results into the bug count. We provide three interpretations for our line level models. The first one is that we can identify a bug as long as any line comprising bug is identified. This is the optimistic interpretation and represents the maximal benefits that can be acquired by using our line-level model. The second one is that we take partial credit when we identify a buggy line. For example, if we identify one buggy line for a five-line bug, we say we find 20% of a bug. This is the average interpretation and assumes that the more information about a bug is provided, the more likely the bug can be identified. The third one is that only after we identify all buggy lines of a bug, we cover the bug. This is the pessimistic interpretation. We denote the three views as lm_{opti}, lm_{avg}, and lm_{pes}.

The results for the "Bug count" data set are shown in Table 3.4. Our results of $P_{opt}(fm)$ are consistent with the results shown by Kamei, Mat-

Table 3.4: **Results of "Bug count" data set.** The two bold numbers in row "2.3.10 → 2.4.0" are larger than one indicating that the performance of our line level model can exceed the upper bound of any file level model.

Train → Predict	P_{opt}				CE			
	lm_{opti}	lm_{avg}	lm_{pes}	fm	lm_{opti}	lm_{avg}	lm_{pes}	fm
2.1.1 → 2.2.0	0.9695	0.9392	0.9023	0.8321	0.9132	0.8243	0.7220	0.5221
2.2.0 → 2.2.6	0.9884	0.9632	0.9297	0.8166	0.9664	0.8935	0.7965	0.4693
2.2.6 → 2.2.10	0.9997	0.9706	0.9339	0.8453	0.9990	0.9148	0.8082	0.5509
2.2.10 → 2.3.0	0.9647	0.9325	0.8965	0.8716	0.8956	0.8007	0.6943	0.6208
2.3.0 → 2.3.10	0.9664	0.9275	0.8848	0.8870	0.8961	0.7756	0.6433	0.6504
2.3.10 → 2.4.0	**1.0013**	0.9665	0.9245	0.9267	**1.0040**	0.8979	0.7700	0.7769
Mean	0.9817	0.9499	0.9120	0.8632	0.9457	0.8511	0.7391	0.5984
Std. Dev.	0.0154	0.0173	0.0184	0.0368	0.0460	0.0532	0.0585	0.0998

sumoto at el. [62]. The results of $CE(fm)$ are slightly better but still consistent with the results shown by Arisholm, Briand et al. [8]. Therefore, we believe that our implementation of fm is comparable to other implementations and that we can compare our lm to this implementation of fm.

The optimistic interpretation and the average interpretation are consistently much better than the file model in both P_{opt} and CE. The pessimistic interpretation loses to the file model slightly in two rounds of prediction but has a much higher mean value. All the three interpretations have much smaller standard deviation than the file model, so prediction results are more stable on line level. Notice that the value of $P_{opt}(lm_{opti})$ and $CE(lm_{opti})$ in row "2.3.10 → 2.4.0" are larger than 1, which shows that the performance of the line level model can even exceed the upper bound of file level models.

Figure 3.5 shows the prediction results of training on release 2.2.10 and predicting on release 2.3.0. If we only test a small amount of code, the lm_{opti} is actually better than the fm_o, but the lm_{pes} is a little bit worse than the fm. As we test more code, the three interpretations of the line-level model are consistently better than the fm.

The "Bug count" data set assumes that every bug involves the same amount of work to fix. We use the "Line count" data set to measure how

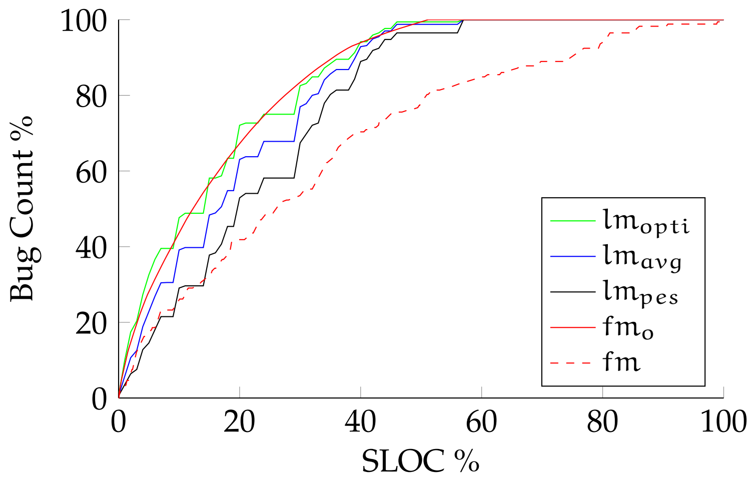


Figure 3.5: **Cross release prediction from 2.2.10 to 2.3.0 on "Bug count" data set.** The x-axis is the percentage of source line of code to test. The y-axis is the percentage of bugs that can be identified.

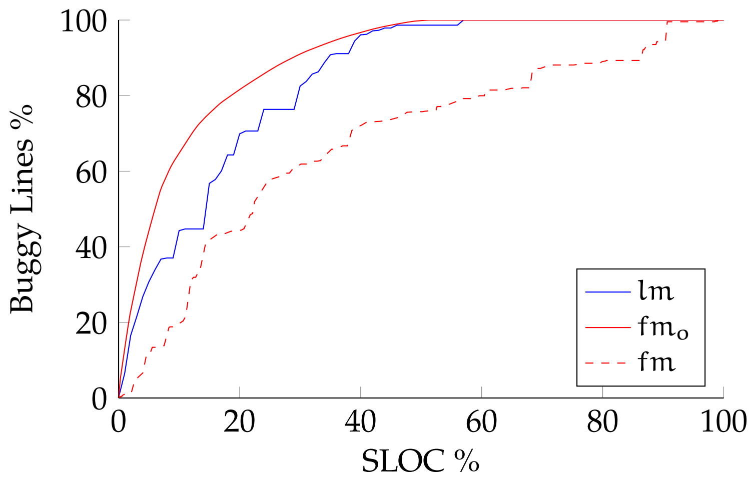


Figure 3.6: **Cross release prediction from 2.2.10 to 2.3.0 on "Line count" data set.** The x-axis is the percentage of source line of code to test. The y-axis is the percentage of buggy lines that can be identified.

Table 3.5: **Results of "Line count" data set.**

Train → Predict	P_{opt}		CE	
	lm	fm	lm	fm
2.1.1 → 2.2.0	0.9148	0.8113	0.7925	0.5404
2.2.0 → 2.2.6	0.9425	0.7704	0.8578	0.4321
2.2.6 → 2.2.10	0.9470	0.7860	0.8658	0.4579
2.2.10 → 2.3.0	0.9153	0.8288	0.7834	0.5624
2.3.0 → 2.3.10	0.8660	0.7711	0.6590	0.4173
2.3.10 → 2.4.0	0.9343	0.8860	0.8299	0.7050
Mean	0.9200	0.8089	0.7981	0.5192
Standard Deviation	0.0271	0.0404	0.0692	0.0988

many buggy lines can be covered during testing. The overall results are shown in Table 3.5 and confirm that the line level model consistently performs better in both P_{opt} and CE. Figure 3.6 shows the results of training on release 2.2.10 and predicting on release 2.3.0 in the "Line count" data set. The line level model performs better than the file level model over all ranges of the curve.

In summary, our two experiments confirm the effectiveness of our new authorship models. The first experiment shows that *git-author* provides more information than *git-blame* by the structured authorship model. The second experiment shows that the information is useful to build better analysis tools.

3.5 Summary

We have presented two line-level source code authorship models: the structural authorship, which represents the complete development of a line of code, and the weighted authorship, which summarizes the structural authorship to produce author contribution weights. Our two authorship models overcome the limitations of the current methods that only report the last change to a line of code. We define the repository graph as a graph abstraction for a code repository and define a backward flow analysis on the repository graph that derives the structural authorship. Another backward flow analysis is used on the structural authorship to compute the weighted authorship.

We have implemented our two authorship models in a new git built-in tool *git-author*. We have evaluated *git-author* in two experiments. In the first experiment, we ran *git-author* on five open source projects and find that *git-author* can recover more information than *git-blame* on about 10% of the lines. In the second experiment, we built a line-level model for bug prediction based on the output of *git-author*. We compared our line-level model with a representative file-level model and found that our line-level model is consistently better than the file-level model on our data sets. These results show that our new authorship models can produce more information than the existing methods and that information is useful to build a better analysis tool.

4 IDENTIFYING MULTIPLE AUTHORS

The foundation of fine-grained binary code authorship identification is the capability of identifying multiple authors in a binary program. In this chapter, we describe our new techniques for identifying multiple authors, which consist of four components.

- 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 [7], the Dyninst binary analysis and instrumentation tool suite [100], and GCC [40]. 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 Section 4.1.

- 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 feature, the *context feature*, to summarize the context of the function or the loop to which the basic block belongs. See Section 4.2.

- 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 Section 4.3.

- 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 Section 4.4.

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 Section 4.5.

4.1 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 [7], the Dyninst binary analysis and instrumentation tool suite [100], and GCC [40]. 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.

4.1.1 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* described in Chapter 3 to get a weight-vector of author contribution percentages for all source lines in these projects. 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.

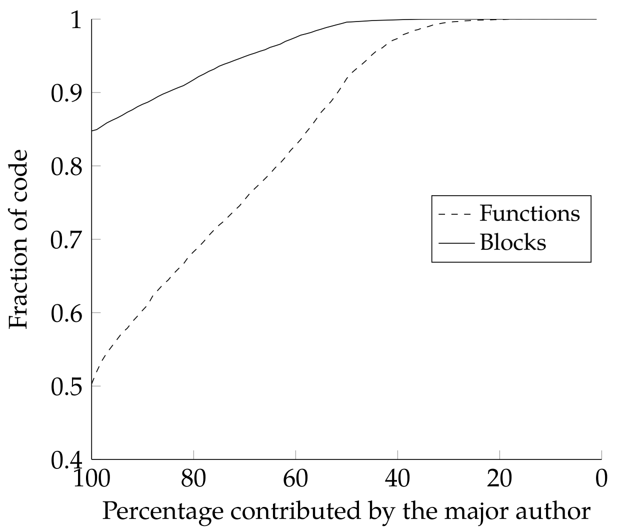


Figure 4.1: **Tail distributions of major author contribution**. The x-axis represents the contribution percentage from the major author. The y-axis represents the fraction of the number of blocks or functions that have a major author who contributed more than a given percentage.

4.1.2 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 Figure 4.1, 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.

Table 4.1: **An overview of new basic block level features.**

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

4.2 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 4.1.

4.2.1 Instruction Features

There are three new features to describe instruction details.

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.

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 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.

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

4.2.2 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.

4.2.3 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 size, 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 [30]. 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.

4.2.4 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 inner-most 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.

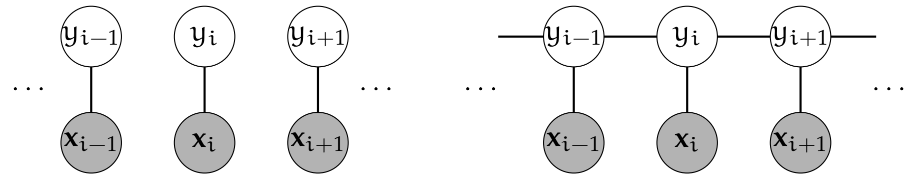
4.3 External Template Library Code

We must distinguish external library code from the code written by their users. In the study discussed in Section 4.1, 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" and "Boost" to represent each of these libraries. Our results discussed in Section 4.5 show that both STL and Boost have distinct styles and we are able to identify the inlined code.

4.4 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 [25] and Random Forest [53] perform prediction solely based on individual features. While this is a reasonable approach when operating at the program level, it may not be



(a) Independent classification models

(b) Joint classification models

Figure 4.2: **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.

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 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 Figure 4.2.

For the independent classification models illustrated in Figure 4.2a, 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 Figure 4.2b, 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 $B = [b_1, b_2, \ldots, b_m]$, 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 then use linear Conditional Random Field (CRF) [69] to train joint classification models.

4.5 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 basic block level features used in previous work, as discussed in Section 2.1,

Table 4.2: **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 Section 2.1. We have four types of new features discussed in Section 4.2: 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%

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 Section 4.2. The results of the first row and the sixth row in Table 4.2 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 later in this section, 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 4.2.

4. The CRF models out-perform the SVM models, but CRF requires

more training time. As shown in the last two rows in Table 4.2, 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 prediction 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.

4.5.1 Methodology

Our evaluations are based on a data set derived from the open source projects used in our empirical study discussed in Section 4.1. 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 [103, 107] to determine which model to use. The handling of these cases is the subject of ongoing research.

We used Dyninst [100] to extract code features, Liblinear [34] for linear SVM, and CRFSuite [95] 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 Section 4.1. 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 [55], 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

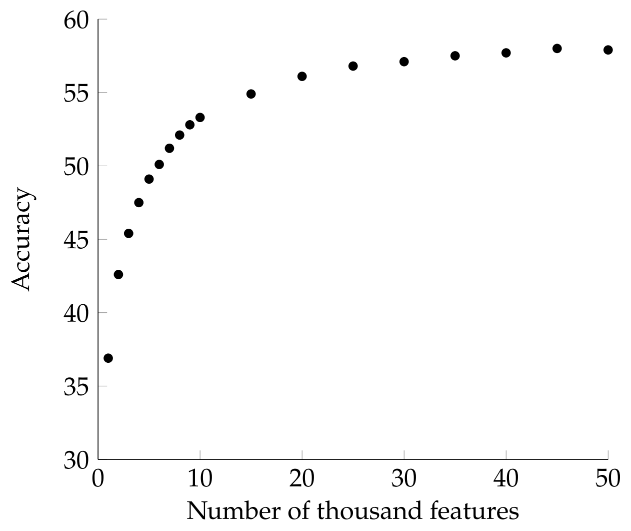


Figure 4.3: **Experiment results on the impact of the number features.** The accuracy results are based on using the new feature set F_n and SVM classification.

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.

4.5.2 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.

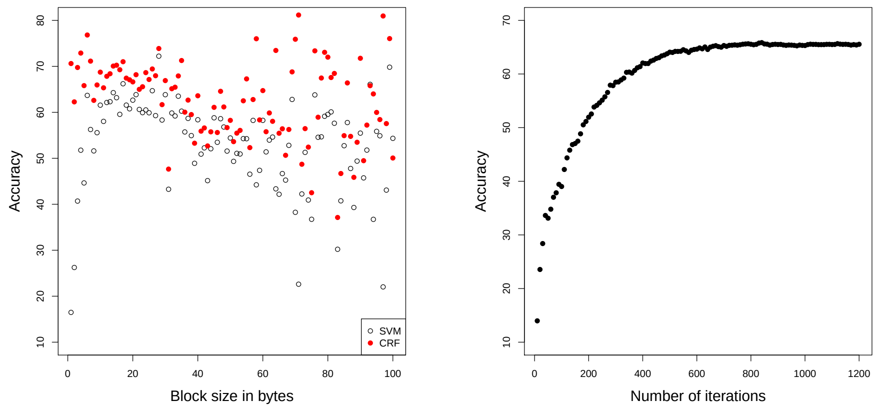
Our new features can be classified into four types: instruction, control flow, data flow, and context features. We denote these four types of features as F_I, F_{CF}, F_{DF}, and F_C, respectively. To determine the impact of each feature type, we calculated how much accuracy can be gained by adding only 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 4.2, the baseline accuracy of using only feature set F_e is 26%. The second to the fifth row of Table 4.2 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 [108] selected 1,900 features. Similarly, Caliskan et al. [20] selected 426 features.

We investigate the impact of K and find that the basic block level needs more features. Figure 4.3 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.

4.5.3 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 4.4a 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 not surprising as small basic blocks contain little code, thus few code features and little information. On the other hand,



(a) Model comparison

(b) Impact of training iterations on CRF

Figure 4.4: **Comparison between CRF and SVM.** The results of both figures are based on the new feature set F_n and selecting 45K features.

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.

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 4.4b shows how the number of iterations 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.

4.6 Summary

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.

5 MULTI-TOOLCHAIN MULTI-AUTHOR IDENTIFICATION

In this chapter, we build upon the multi-author identification techniques described in Chapter 4 to investigate the impact of compilation toolchains on multi-author identification. We present the first thorough study of multi-toolchain, multi-author identification and present new techniques to identify multiple authors in multi-toolchain scenarios.

We first describe a study in Section 5.1, where we evaluate how well our multi-author identification techniques presented in Chapter 4 will function in a multi-toolchain scenario. We created a multi-toolchain dataset, containing binaries generated by 15 toolchains. The results of our study revealed two limitations. First, when models trained with binaries compiled by one toolchain are applied to binaries compiled by another toolchain, the identification accuracy is low. Second, there is a significant difference between the accuracy of different single toolchain scenarios; the optimization levels influence the accuracy. The results of the study confirm the need for new techniques for identifying multiple authors in multi-toolchain scenarios.

We then describe two approaches for multi-toolchain multi-author identification in Section 5.2. The first approach is a two layer one, incorporating toolchain identification [103, 107] to determine which toolchain generated the binary and then applying the corresponding single-toolchain authorship model. The second approach is to use unified training, where we construct a training set containing binaries from all known toolchains.

Next, we describe how to apply deep learning to automatically extract style features in Section 5.3. 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 [9, 42], and report our experiences of applying these practices.

In Section 5.4, we describe the evaluations of our new techniques. Our results showed that with toolchain identification, we achieved 59% accuracy for identifying 700 authors. Replacing SVM with DNN, we improved this accuracy to 71%. In addition, DNN achieved 82% top-5 accuracy and 86% top-10 accuracy, showing that our techniques can effectively prioritize investigation. For the unified training approach, SVM training did not scale to this data set, while DNN achieved 68% accuracy.

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 describe the lessons we learned in Section 5.5.

5.1 Multi-toolchain Study

We evaluate the effectiveness of our previous techniques for multi-author identification [85] in a multi-toolchain scenario.

5.1.1 Data Set Generation

We created a multi-toolchain dataset by compiling three large, long-lived open source projects (Apache HTTPD Server [7], Dyninst binary analysis and instrumentation tool suite [100], and Git [39]), 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 [84] 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.

5.1.2 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 Dyninst [100] to extract code features, Liblinear [34] for linear SVM , and CRFSuite [95] 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 [85]. 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 [55], where each round of the cross validation is executed on a separate machine.

5.1.3 Results

Table 5.1 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

Table 5.1: **Evaluation results of our previous techniques [85]**. 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.

Predict / Train		GCC					ICC					LLVM				
		-O0	-O1	-O2	-O3	-Os	-O0	-O1	-O2	-O3	-Os	-O0	-O1	-O2	-O3	-Os
GCC	-O0	**45%**	8%	7%	7%	8%	9%	7%	7%	7%	7%	8%	8%	7%	7%	7%
	-O1	8%	**59%**	25%	22%	16%	8%	14%	16%	16%	14%	7%	10%	24%	23%	16%
	-O2	7%	27%	**63%**	38%	18%	6%	14%	17%	17%	14%	6%	10%	28%	29%	18%
	-O3	7%	25%	41%	**66%**	16%	6%	14%	19%	18%	14%	6%	9%	25%	25%	16%
	-Os	8%	19%	20%	17%	**55%**	7%	13%	13%	13%	13%	7%	11%	19%	19%	14%
ICC	-O0	9%	6%	6%	6%	6%	**47%**	6%	7%	7%	6%	6%	6%	8%	7%	6%
	-O1	7%	13%	13%	12%	11%	7%	**58%**	19%	19%	45%	7%	9%	16%	16%	12%
	-O2	6%	14%	15%	15%	10%	7%	20%	**66%**	64%	21%	6%	8%	19%	19%	12%
	-O3	6%	13%	15%	15%	10%	8%	20%	64%	**66%**	21%	5%	8%	19%	19%	12%
	-Os	7%	13%	13%	12%	11%	7%	45%	20%	20%	**58%**	6%	9%	16%	16%	12%
LLVM	-O0	9%	9%	9%	8%	9%	9%	9%	8%	8%	9%	**44%**	9%	8%	8%	8%
	-O1	9%	10%	9%	8%	10%	7%	9%	8%	8%	9%	7%	**52%**	16%	15%	21%
	-O2	8%	19%	19%	18%	13%	8%	14%	17%	17%	14%	7%	18%	**69%**	63%	40%
	-O3	8%	19%	19%	18%	13%	7%	14%	17%	17%	14%	6%	17%	63%	**70%**	37%
	-Os	8%	19%	18%	16%	14%	7%	13%	14%	14%	13%	6%	21%	40%	38%	**61%**

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 [85] 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 [120], 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.

5.2 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*.

5.2.1 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 [107]. 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. [107] 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 contain 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.

5.2.2 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 [42]. 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 power, as the size of the training set is multiplied by the total number of available toolchains.

5.3 Applying Deep Learning

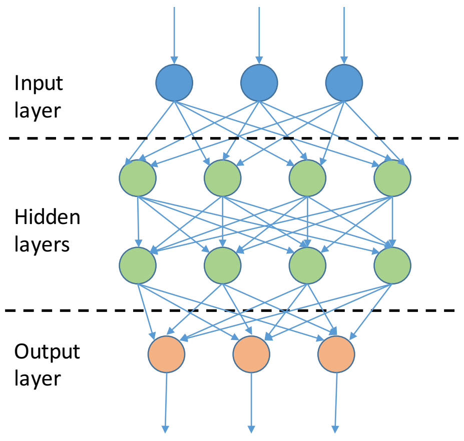
We apply deep learning to automatically learn features from raw bytes of basic blocks. We first introduce the basics of feed-forward neural networks and then focus on how to construct inputs to the network.

5.3.1 Basics of Feed-forward Neural Networks

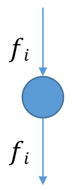
Figure 5.1 illustrates an example of a feed-forward neural network and breaks down its individual components. As shown in Figure 5.1a, nodes in the network are arranged into layers: one input layer, multiple hidden layers (two in this example), and one output layer. Typically, nodes between adjacent layers are fully connected, and there are no backward edges or cross layer edges.

As shown in Figure 5.1b, 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.

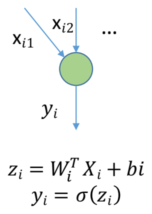
The internal computation of a hidden layer node is shown in Figure 5.1c. The number of hidden layers and the number of nodes in a hidden layer are two hyper-parameters that need tuning. A node in a hidden layer takes multiple inputs from the previous layer and generates new output value to the next layer. Denote $X_i = [x_{i1}, x_{i2}, \dots, x_{in}]^T$ 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 $W_i = [w_{i1}, w_{i2}, \dots, w_{in}]^T$ 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 σ is called the activation function. σ is a non-linear function so that the whole network can represent a non-linear prediction space. While the Rectified Linear Unit (ReLU) is a commonly used activation function, we found that the Scaled Exponential Linear Unit (SELU) [67] achieved better results.



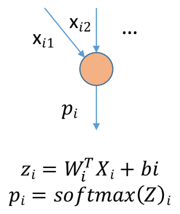
(a) A network example



(b) An input layer node



(c) A hidden layer node



(d) An output layer node

Figure 5.1: **Overview of a feed-forward neural network.** Nodes are arranged into an input layer, hidden layers, and an output layer. Nodes in adjacent layers are fully connected. From an user point of view, the input layer takes user provided inputs and the output layer generates prediction probabilities. Internally, an input layer node outputs the input value f_i without modification. A hidden layer node defines learning parameters weights W_i and bias b_i, and performs a linear transformation with the input X_i. The result of the linear transformation z_i then goes through a non-linear activation function σ to generate the output to the next layer. An output layer node also defines W_i and b_i and generates the prediction probability for a class.

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 5.1d, 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 $p_i = \mathtt{softmax}(Z)_i$, where $\mathtt{softmax}(Z)$ is a function that normalizes a vector of arbitrary values to a probability vector; $\mathtt{softmax}(Z)_i$ is the ith element in the probability vector, defined as $\mathtt{softmax}(Z)_i = (e^{z_i})/(\sum_{k=1}^{m} e^{z_k})$. 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 [42].

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 Lth 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.

5.3.2 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.

However, 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 [6, 83, 114]. We reuse existing basic block level structural features [82, 85, 108] as additional inputs.

5.4 Evaluation

We evaluated our techniques with the same multi-toolchain data set discussed in Section 5.1. 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: `two-layer-svm`, `two-layer-dnn`, `unified-svm`, and `unified-dnn`.

5.4.1 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 `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 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 `two-layer-svm` are similar to the steps for `two-layer-dnn`, but we replace feed-forward neural networks with linear SVM. For both `unified-svm` and `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 [100] to extract code features, Liblinear [34] for linear SVM, CRFSuite [95] for linear CRF, and TensorFlow [124] 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 [55]. 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.

5.4.2 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 `two-layer-svm`, `two-layer-dnn`, and `unified-dnn` achieved 59%, 71%, and

Table 5.2: **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 5.1. DNN improved accuracy for all 15 toolchains.

	O0		O1		O2		O3		Os		Avg	
	SVM	DNN	SVM	DNN	SVM	DNN	SVM	DNN	SVM	DNN	SVM	DNN
GCC	45%	64%	59%	69%	63%	70%	66%	75%	55%	66%	58%	69%
ICC	47%	62%	58%	68%	66%	79%	66%	79%	58%	68%	60%	73%
LLVM	44%	63%	52%	64%	69%	77%	70%	77%	60%	69%	60%	71%
Avg	45%	63%	57%	67%	66%	76%	67%	77%	58%	68%	59%	71%

68% 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: `two-layer-dnn` achieved 82% top-5 accuracy and 86% top-10 accuracy. Our results show that `unified-svm` did not scale to the merged training set containing all 15 toolchains.

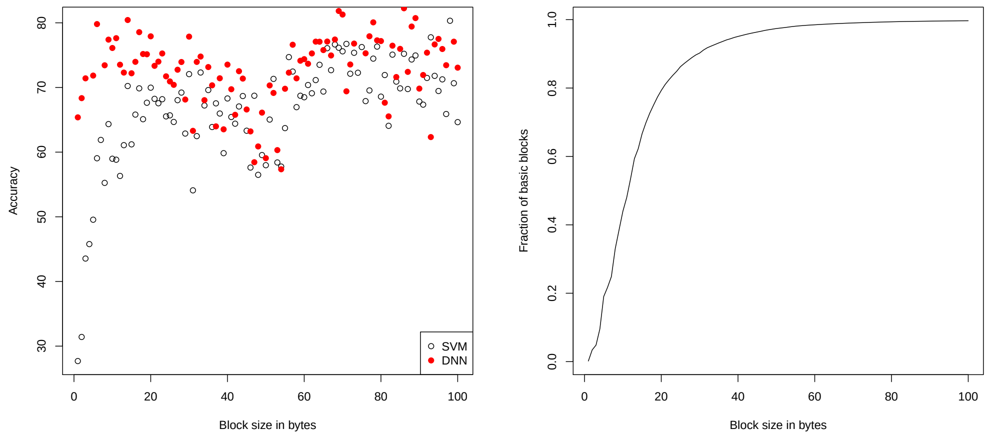
To better understand the accuracy achieved the two layer approach, we investigated how the accuracy of toolchain identification impact multi-author identification. Our toolchain identification classifiers achieved 93% accuracy for identifying 15 toolchains, where accuracy is calculated as the number 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 `two-layer-svm` and `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 `two-layer-dnn` and `two-layer-svm` on two aspects to understand how `two-layer-dnn` improved accuracy. First, we broke down the accuracy achieved by DNN and SVM in single toolchain scenarios. Table 5.2 shows that DNN improved not only the overall accuracy, but also the accuracy for all 15 toolchains. In addition, `two-layer-dnn` had a slightly reduced accuracy difference between optimization levels, as ICC -O0 had the lowest accuracy (62%) and ICC -O3 had the highest accuracy (79%), compared to `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 5.2a, we can see that SVM suffers when there are small basic blocks. This observation aligns with the results presented in our previous study [85] 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 5.2b 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 `unified-svm` did not scale to the merged training set, it shows that `unified-dnn` has better training scalability. We observed similar accuracy results from `unified-dnn` compared to `two-layer-dnn`: (1) `unified-dnn` has similar accuracy differences between optimization levels, where O0 code has lowest accuracy, and O2 and O3 code have highest accuracy; (2) `unified-dnn` has a similar accuracy trend in terms of block sizes.

Last, we discuss the needed training time. For `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



(a) Model Comparison (b) Cumulative Distribution

Figure 5.2: **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.

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.

5.4.3 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 [66]. 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^{-5} and 1, with a default value 0.01 [9]. 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. Using SELU as the activation function allows training deeper networks compared to using ReLU, and we achieved better results when using SELU. The best SELU network had 10 hidden layers, while the best ReLU network had 5 hidden layers.

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-bathes to have disjoint sets of authors, which in turned 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 units per layer. Our networks are set to have 800 hidden units per layer. We let all hidden layers have the same number of units, as Larochelle et al. [72] 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 [9]. 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.

5.5 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.

5.5.1 Different Features Are Selected for Different Toolchains

Two key questions for understanding the impact of compilation toolchains on authorship are do we need different features for different toolchains? And 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 5.3 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

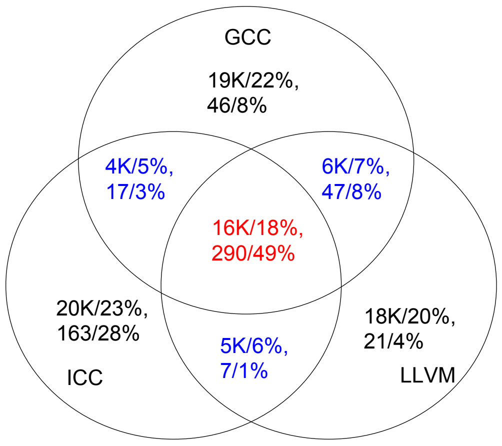


Figure 5.3: **A Venn diagram of selected features in models from** `two-layer-svm`. Each model selected 45,000 features. There are 88,000 features selected and 591 features are structural features. The results are represented in a form "A/B,C/D", where A represents the total number of features, B represents the percentage of total features, C represents the number of structural features, and D represents the percentage of structural features.

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 [118] 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 5.4 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 5.4. Therefore, our results suggest that commonly selected features typically contribute differently to predictions in different toolchains.

5.5.2 DNNs Can Internally Recognize the Toolchain

In Section 5.4.2, our 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. [135] 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 im-

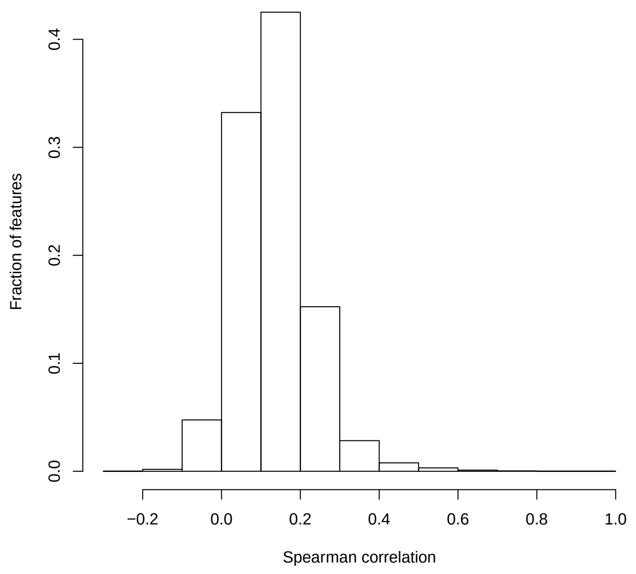


Figure 5.4: The histogram of Spearman correlations for commonly selected features by all three toolchains in GCC model and ICC model.

Table 5.3: Unit distributions in hidden layers.

	GCC		ICC		LLVM	
	Dominant	Exclusive	Dominant	Exclusive	Dominant	Exclusive
Layer 1	0.6%	0.0%	24.0%	1.0%	75.4%	71.4%
Layer 2	10.4%	0.1%	48.6%	6.4%	41.0%	22.6%
Layer 3	7.6%	0.4%	61.4%	10.5%	31.0%	17.5%
Layer 4	8.0%	0.5%	55.6%	7.3%	36.4%	19.5%
Layer 5	9.7%	0.1%	67.4%	3.9%	22.9%	8.4%
Average	7.3%	0.2%	51.4%	5.8%	41.3%	27.9%

age 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 5.3 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.

Table 5.4: **Total number of basic blocks (in millions)**.

	O0	O1	O2	O3	Os
GCC	1.46	1.30	1.29	1.62	1.11
ICC	1.55	1.18	2.36	2.36	1.16
LLVM	1.62	1.21	1.60	1.70	1.14

Table 5.5: **Percentages of basic blocks that contains boilerplate code patterns.**

	O0	O1	O2	O3	Os
GCC	14.0%	0.7%	0.7%	0.5%	0.8%
ICC	17.0%	0.2%	0.1%	0.1%	0.2%
LLVM	12.8%	2.1%	1.2%	1.1%	1.7%

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.

5.5.3 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 original authors. So, optimized code should be more difficult to attribute than unoptimized code. However, our results in Table 5.2 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

Table 5.6: **Percentages of basic blocks caused by function inlining.**

	O0	O1	O2	O3	Os
GCC	0.0%	40.6%	41.9%	57.0%	29.1%
ICC	0.0%	0.0%	66.7%	66.7%	0.0%
LLVM	0.0%	0.0%	57.2%	59.8%	37.0%

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 5.4 shows the total number of basic blocks for each toolchain and Table 5.5 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 5.6 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 5.5, 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 0.85 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.

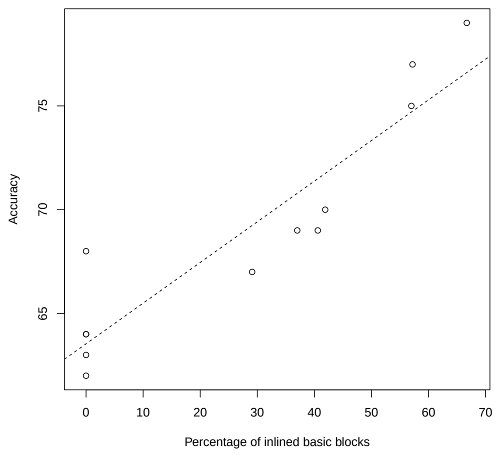


Figure 5.5: **Correlation between inlining and accuracy.** Each data point represents a toolchain. The x-axis represents the percentage of inlined basic blocks. The y-axis represents the single-toolchain accuracy achieved by `two-layer-dnn`. The dashed line represents regression line for all points.

5.6 Summary

We have presented new techniques to perform multi-toolchain, multi-author identification. We started with an extensive evaluation of existing multi-author identification techniques, showing that existing techniques did not work well in multi-toolchain scenarios. We designed two new techniques to overcome the weaknesses of existing techniques: the two layer approach that combines toolchain identification and single-toolchain authorship identification, and the unified training approach that train 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. Our techniques achieved 71% accuracy, 82% top-5 accuracy, and 86% top-10 accuracy for classifying 700 authors, showing that analysts can effectively prioritize investigation based on our prediction. Our results also showed that our deep learning models consistently outperformed traditional learning methods such as SVM. 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 interesting lessons from our investigation, such as that unoptimized code is more difficult to attribute. These lessons learned from investigating our models provide valuable insights on how compilation toolchains impact authorship styles.

In summary, we showed that we can perform practical multi-author identificaiton in multi-toolchain scenarios. Our work lays the foundation for research topics such as whether we can suppress programming styles to evade identification and whether we can impersonate someone else's coding style to mislead identification.

6 EVASION OF AUTHORSHIP IDENTIFICATION

In the previous two chapters, we described our new techniques for fine-grained binary code authorship identification. In this chapter, we look at the problem of authorship identification from a different angle and attempt to perform authorship evasion, whose goal is to trick machine learning classifiers for authorship identification into make wrong predictions.

Authorship evasion is the application of adversarial machine learning to authorship identification. The field of adversarial machine learning has focused on attacking and defending machine learning systems used in real world applications [11]. A specific threat is called a *test time attack*, where attackers change a test example to cause misprediction. Researchers have performed successful test time attacks for a wide range of domains, including computer vision [13, 23, 121], audio processing [134], and program analysis tasks [43, 116]. Such test time attacks can have serious security implications. For example, Simko et al. [116] showed that when given source code from other people, a programmer can change the source code to avoid authorship attribution.

However, currently there are no such attacks against binary code authorship identification. The key challenge for developing such attacks is to modify the binary to not only cause misprediction, but also maintain the structural validity and functionality of the binary. Even flipping one bit of a binary may cause the binary to either be invalid, such as not loadable by the loader, or lose functionality that the attackers care about. Therefore, attacks against binary code are intrinsically more difficult than attacks targeted at domains such as computer vision, where attackers can change each pixel of the input image independently and still maintain a valid image.

In this chapter, we describe our new automated attacks against binary code authorship identification. The contributions of our new attacks are three-fold. First, we show that it is realistic to automatically attack binary code authorship identification in an end-to-end fashion: we take a binary program as input and generate a new, valid binary that has the same functionality as the input binary and will cause misprediction. Second, our techniques can be used for adversarial re-training to mitigate the threats from the wild, incorporating the generated adversarial examples into the training set to re-train a model with a modified loss function [89]. Third, based on our experiences, we summarize the lessons we learned for designing more secure machine learning systems for binary code authorship identification.

Operationally, there are three different goals for attacking authorship identification: (1) the *losing-confidence attack*, which is to remove any fingerprints and anonymize the program, such that the target classifier rejects to make a prediction; (2) the *untargeted attack*, which is to cause misprediction to any of the incorrect authors; and (3) the *targeted attack*, which is to cause misprediction to a specific incorrect author. To perform a losing-confidence attack, the target classifier must have the capability to reject an input binary based on the lack of confidence in the prediction, which is not the case for most existing techniques for binary code authorship identification [4, 108]. Therefore, we focus on untargeted and targeted attacks.

We assume that we have perfect knowledge of the target classifier. This assumption allows performing a worst-case evaluation of the security of the target classifier, common when performing test time attacks [13, 23, 116, 121].

We developed two interacting attack abilities for evading binary code authorship identification: *feature vector modification*, changing the feature vector to cause mis-prediction of the target classifier, and *input binary modification*, modifying the input binary to match the adversarial feature vector while maintaining the functionality of the input binary.

Our approach to feature vector modification started with existing attacks for computer vision tasks [23, 99, 134], and then extended these attacks in two ways to address the structural validity requirement of binary programs. First, existing attacks did not consider the difficulty of modifying a feature; changing any pixel of an image is equally easy for maintaining the validity of the image. However, for binary analysis, some features are easier to modify than others. For example, local features that describe machine instructions are typically easier to modify than global features that describe program control flow, because modifying global features can require changing more code, making it more difficult to maintain structural validity. Second, existing attacks modified each feature independently; changing one pixel of an image does not impact other pixels. However, features in our domain can be correlated. Without considering feature correlation, we may generate feature vectors for which there do not exist corresponding valid binaries.

Our input binary modification removes or injects features according to the results of feature vector modification, with the additional goals of maintaining structural validity and preserving functionality. To remove features, we need to ensure that the program properties that correspond to the removed feature are replaced with semantically equivalent ones. In many cases, we cannot simply remove them because such modification would break the functionality of the binary. On the other hand, the main challenge of injecting features is to ensure that the binary code analysis tools used for feature extraction indeed recognize the injected code, data or meta-data.

We describe an attack example in Section 6.1, using untargeted attacks on a classifier trained with the techniques presented by Caliskan-Islam et al. [21] for single author identification. This example represents a complete overview of our attack.

In Section 6.2, we describe our attack framework, which has three key components:

1. *Feature grouping*: We categorize binary code features into a small number of feature groups such that features in a group can be modified with the same strategy. Two key factors for grouping features are the program property that the features describe and the underlying binary analysis tool used for feature extraction.

2. *Feature correlation*: We derive feature correlation from a data set. Note that this data set can be, but does not have to be the training set used for training the target classifier. We can derive useful feature correlation information, as long as this data set is drawn from the same application domain as the training set. We then use the correlation information to ensure that correlated features are modified in a consistent way.

3. *Binary modification strategies*: We design injection and removal strategies for each feature group. These modification strategies consist of a sequence of binary modification primitives, including inserting, deleting, and replacing code, data, and meta-data. Binary instrumentation and rewriting tools such as Dyninst provide such binary modification primitives to support our modification strategies.

In Section 6.3, we evaluate our evasion attacks using five classifiers trained with the techniques presented by Caliskan-Islam et al. [21]. We achieved 96% success rate for untargeted attacks and 46% success rate for targeted attacks. Our results show that we can effectively suppress authorship signal for authorship evasion, but it is significantly more difficult to impersonate the style of another author.

6.1 An Attack Example

We present an example showing how to perform untargeted attacks to a classifier for binary code authorship attribution. The goal of this section is to give an overview of our attack process. In the subsequence sections, we describe the steps in more detail.

We first describe the procedures for setting up the target classifier, which is trained with the techniques presented by Caliskan-Islam et al. [21]. We then describe our feature correlation analysis and how to use the correlation information to generate feature vectors that correspond to real binaries and cause misprediction. Finally, we give examples on how to modify the binary to match the generated feature vectors.

6.1.1 Binary Code Authorship Attribution

Caliskan-Islam et al. [21] assume that a binary is written by a single author, so, they predict one author for a binary. Their workflow can be summarized in four steps.

1. *Define candidate features*: They used binary code features that describe machine instructions and program control flow. They also included source code features derived from decompiled source code. The source code features include character n-grams and tree n-grams. The tree n-grams are extracted from abstract syntax trees (ASTs) built by parsing the source code. These source code features have been shown to be effective for source code authorship attribution [20].

2. *Extract features*: They used two disassemblers, NDISASM [122] and radare2 [97], to extract binary code features. To derive source code features, they first used the Hex-Ray decompiler [51], and then used Joern [130] to parse the source code into ASTs. They represent each feature as a string. To derive feature strings, they first split the results

of disassembly, decompiling, and source code parsing into tokens and then normalize hex tokens to the generic symbol "hexdecimal" and decimal digit tokens to the generic symbol "number". They use string matching to count the frequency of a feature string and use the frequencies of feature strings to construct feature vectors.

3. *Select Features*: Typically, hundreds of thousands of features are extracted from a data set. So, feature selection is necessary to avoid overfitting. They selected features that have information gain with respect to the author labels.

4. *Train a classifier*: They compared Random Forests (RFs) with Support Vector Machines (SVMs) and reported that RFs outperformed SVMs.

They used a data set derived from Google Code Jam (GCJ) and evaluated their techniques with binaries compiled by GCC on a 32-bit platform. For binaries compiled with GCC and `-O0`, they achieved 96% accuracy for classifying 100 authors. For binaries compiled with higher optimization levels, they reported slightly lower accuracy.

We obtained the GCJ source files used by Caliskan-Islam et al. [21] and their source code for extracting features. Due to the predominance of 64-bit platforms, we perform attacks on 64-bit platforms. Note that while Caliskan-Islam et al. only evaluated their techniques on 32-bit platforms, their techniques can be directly applied to 64-bit platforms. We compiled the GCJ sources with GCC 5.4.0, using `-O0` optimization on a 64-bit platform, and achieved 90% accuracy for classifying 30 authors.

6.1.2 Attack Procedure

Given the target classifier to attack, the goal of our attack is to modify an input binary to cause misprediction. The key steps for attacking this authorship attribution classifier are performing feature correlation analysis, generating feature vectors that can cause misprediction, and modifying the input binary to match the feature vector.

Feature Correlation Analysis

We derive correlations between features to guide feature vector modification to generate feature vectors corresponding to real binaries.

We identify two types of feature correlation for this classifier. First, a feature can contain other features. For example, if feature "`push rax; push rbx`" is present in a binary, features "`push rax`" and "`push rbx`" are also present. So, the frequencies of "`push rax`" and "`push rbx`" should be no fewer than the frequency of "`push rax; push rbx`". Second, the same properties extracted by different binary analysis tools are treated as different features. For example, the instruction "`call fprintf`" corresponds to three different features: "`call fprintf`" extracted by NDISASM, "`call fprintf`" extracted by radare2, and "`fprintf`" extracted from decompiled source code. These features should all have the same frequency.

We derive linear correlation between features based on the training set. For each pair of features, we perform linear regression and calculate the correlation coefficient. If the coefficient is larger than a threshold value, such as 0.9, we merge the pair into one feature. While this simple strategy will miss non-linear feature correlation, our experiments showed that capturing linear correlation is sufficient for launching successful attacks against authorship attribution.

Generating Adversarial Feature Vectors

We extend the attacks presented by Carlini and Wagner [23] to generate adversarial feature vectors. Their attacks are designed for DNNs trained for images, and can be readily applied to other gradient based learning algorithms. However, Caliskan-Islam et al. used RFs, which is a non-gradient based learning algorithm. Fortunately, researchers have shown that adversarial examples created for classifiers trained with one type of learning algorithms (such as DNN) are likely to cause misprediction for classifier trained with a different type of learning algorithms (such as RF) [98, 126]. Therefore, we first trained a substitute DNN using the same training data and then applied the adversarial vectors to the RF classifier. The substitute DNN is a simple feed-forward neural network, containing 7 hidden layers with each layer having 50 hidden units. The substitute DNN has 80% accuracy. While the substitute DNN has modestly lower accuracy than the target classifier, as we will show in Section 6.3, this accuracy gap does not impact the success rate of our attack.

To ensure that the generated adversarial feature vector confuses not only the substitute classifier but also the target classifier, we keep generating new feature vectors until the resulting vectors can mislead the target classifier. Our new attack strategy can generate effective adversarial feature vectors, reducing the accuracy of both the substitute DNN and the RF classifier to 0%.

However, it is difficult to modify the input binary to completely match the feature vectors generated in this way, as they contain hundreds of modified features.

Categorizing Features

We have observed that while the attacks presented by Carlini and Wagner can make effective changes to the feature vector to cause misprediction, not all changes are necessary for causing misprediction. Therefore, we attempt to modify fewer features to cause misprediction, making it easier to perform binary modification to match the generated feature vector. We categorize features into feature groups, so that features in the same feature group can be modified with the same strategy. And then we modify one feature group at a time until misprediction occurs.

Two important factors for categorizing the features are the program properties that the features describe and the strength of the binary analysis tools. For the first factor, features describing low level code properties such as machine instructions are easier to modify compared to features describing higher level structural properties such as program control flow and data flow. Therefore, we started by attacking instruction features.

For the second factor, recall that Caliskan-Islam et al. used two disassemblers: NDISASM, which disassembles the binary linearly from the first byte of the binary file, and radare2, which understands the layout of the binary, performs binary analysis to identify code bytes, and attempts to disassemble only code bytes. It is easier to modify features extracted by NDISASM, because NDISASM also disassembles non-loadable sections and editing or adding non-loadable sections has no impact on the functionality of the program. On the other hand, instruction features extracted by radare2 typically represent real code. So, we need to ensure that we do not change the functionality when removing a radare2 feature, and ensure that radare2 disassembles the inserted code when injecting a radare2 feature.

After grouping features, we first modify instruction features extracted by NDISASM, reducing the accuracy from 90% to 45%. We then modify instruction features extracted by radare2, further reducing the accuracy from 45% to 7%. Note that only features in the these two feature groups are modified and we can generate new binaries to complete the attack.

6.1.3 Binary Modification Strategies

Finally, we describe our binary modification strategies for injecting and removing NDISASM and radare2 features, using four typical examples. These examples are extracted from our successful attacks. In each example, we describe the modification primitives that constitute the modification strategy and explain why our modifications do not change the functionality of the input binary.

Feature string	`or [rax],ebp`
Raw bytes	`09 28`
Modification	Insert bytes `09 28` into a new non-loadable section

Figure 6.1: **An example of injecting a NDISASM feature**. We can insert the bytes into a non-loadable section.

Binary name	1835486_1481492_paladin8
Feature string	`imul ebp,[fs:rsi+hexadecimal],dword hexadecimal`
Offset in the binary	0x3e09
Raw bytes	`64 69 6E 38 2E 63 70 70`
Modification	Overwrite bytes to other values

Figure 6.2: **An example of removing a NDISASM feature.** This feature seems to represent an instruction, but actually represents a string in the `.strtab` section.

Modifying NDISASM Features

We show two examples of modifying NDISASM features. The first example shows the case where we can inject a feature by inserting bytes into the binary. As shown in Figure 6.1, we need to inject instruction feature "`or [rax],ebp`" into the target binary. Since NDISASM disassembles every bytes in the binary, we can add a new non-loadable section to store the bytes of the corresponding instruction. This simple injecting strategy causes NDISASM to extract this feature and does not change the functionality of the program.

The second example shows the case where we can simply remove a feature, without replacing the removed program property with a semantically equivalent one. As shown in Figure 6.2, this feature seems to represent an `imul` instruction. However, offset 0x3e09 of the binary is in the `.strtab` section, which stores symbol names for the compile-time symbol table. Therefore, instead of representing an instruction, the feature represents string "in8.cpp". To remove this feature, We can change the string "in8.cpp" to any another string. `.strtab` is used at debug-time, and not used at the link-time or run-time (it disappears if the binary is stripped), so changing its content does not impact the functionality of the original program.

In addition, we tried to understand why the string "in8.cpp" is a useful feature. We found that the string is extracted from source file name "1835486_1481492_paladin8.cpp" and "paladin8" is the author's name. So, this feature turns out to contain three characters of the author's name. While a string containing three characters of the author's name is useful for identifying the author, such author name feature is not available in any realistic context. This example teaches us a lesson that machine learning practitioners need to ensure that the feature definition and the extracted features actually match. In this case, instruction features should only be extracted from real code bytes. So, the use of NDISASM is not robust for real world identification because it disassembles all bytes in the binary.

Modifying radare2 Features

We now show two examples of modifying radare2 features. The first example shows the case where we need to insert new code and data. As shown in Figure 6.3, feature "number.in" represents a string. Note that this feature is not present in the target binary, and we need to inject it into the target binary to cause misprediction. We found feature "number.in" in another binary, based on the instruction "`mov $0x400c57,%edi`". Here, address 0x400c57 points to a string "number.in"; radare2 recognizes the string and prints it in the disassembly results.

To inject this feature, we need to (1) insert string "number.in" into the target binary, and (2) insert a `mov` instruction that loads the address of the inserted string. However, to trick radare2 to disassemble the inserted instruction, there are two additional steps. First, we create a function symbol pointing to the inserted code. Second, we append a return instruction after the inserted code. Since most binary analysis tools treat function symbols as ground truth for specifying the locations of code bytes, our

Feature string	`number.in`
Machine instruction	`mov $0x400c57,%edi`
Modifications	Insert string “number.in” into a new data section Insert new instructions to load the inserted string

Figure 6.3: **An example of injecting a radare2 feature**. This feature represents a string. We need to insert the string and insert an instruction to load the address of the string.

injection strategies can be also applied to other binary analysis tools.

The second example shows the case where we need to replace existing code with semantically equivalent code to remove a feature. As shown in Figure 6.4, we need to remove a feature describing an object symbol. The feature is extracted from instruction “`mov 0x20157d(%rip),%rax`”. Here radare2 recognizes that the result of the PC-relative calculation points to an object symbol, so it annotates the instruction with the name of the object symbol in the disassembly results.

To remove this feature, we need to transform the calculation of the symbol address to a semantically equivalent calculation done by one or more instructions, so that radare2 cannot recognize the loading of the symbol address. To do this, we can split the address loading into two instructions: loading the address minus one into the target register and incrementing the target register by one. We cannot just overwrite the symbol name with a different string because this symbol is in the `.dynsym` section and it is used for dynamic linking (Overwriting the name of a dynamic symbol will cause the program to not be loadable).

6.2 Attack Framework

We describe our attack framework in this section, based on the attack algorithm in Figure 6.5. The inputs to our algorithm includes an input binary `b`, a target classifier `m`, feature groups `fg`, and a misprediction

Feature string	`obj.stdin`
Machine instruction	`mov 0x20157d(%rip),%rax`
Modifications	Load `0x20157d(%rip)-1` into `%rax` Increment `%rax`

Figure 6.4: **An example of removing a radare2 feature.** This feature represents an object symbol "`stdin`". We can split the address loading instruction into two instructions to remove the feature.

```
input  : an input binary $b$; a pre-trained model $m$; feature groups $fg$;
         and a misprediction target $tar$ ($tar = -1$ represents
         untargeted attacks)
output : an adversarial binary $b'$ that causes misprediction
1 $P \leftarrow$ FeatureCorrelationAnalysis($m$);
2 $x \leftarrow$ FeatureExtraction($b$);
3 $y \leftarrow$ Prediction($m, x$);
  // Keep looping until causing misprediction
4 for $g$ in $fg$ do
5     $x' \leftarrow$ FeatureVectorModification($x, g, P, tar$);
6     $b' \leftarrow$ InputBinaryModification($b, x', g, P$);
7     $y \leftarrow$ Prediction($m$, FeatureExtraction($b'$));
      // Non-targeted attacks succeed
8     if $tar = -1$ and $y \neq y'$ then break;
      // Targeted attacks succeed
9     if $tar \neq -1$ and $tar = y'$ then break;
```

Figure 6.5: **The attack algorithm.** The main structure of the algorithm is to iterate over feature groups until we generate a new binary that causes misprediction.

target label tar. The output of the algorithm is an adversarial binary b' that causes the required misprediction. The main component of our algorithm is an attack-verify loop, where we iterate over feature groups until we generate a new binary that causes misprediction.

Our algorithm relies on two routines from the application we are attacking: `FeatureExtraction` to extract features and `Prediction` to generate a prediction label from a set of known labels. The meaning of these labels depend on the target application. For example, a label can describe an author for authorship attribution or a compiler for compiler identification. We now describe the other routines in our algorithm.

6.2.1 Feature Correlation Analysis

Given a set of features $F = \{f_1, f_2, \ldots, f_k\}$ used in the target classifier m, our feature correlation analysis generates a partitioning of the features, $P = \{p_1, p_2, \ldots, p_k\}$, where each partition consists of all correlated features. So, $\forall f_x \in p_i$ and $f_y \in p_i$, f_x and f_y are correlated; and $\forall i \neq j$, $f_x \in p_i$, and $f_y \in p_j$, f_x and f_y are not correlated. In addition, feature partitions are disjoint. So, $\forall i \neq j$, $p_i \cap p_j = \emptyset$.

We build a undirected graph to generate the feature partitioning. Let $G = (V, E)$, where each node in the graph represents a feature (so $V = F$), and each edge in the graph represents the correlation between two features. We only capture linear correlation between features, creating an edge between two nodes if the linear correlation coefficient between two features is larger than a pre-specified threshold. In another words, $E = \{(f_i, f_j) : coe_{ij} \geqslant T\}$, where coe_{ij} is the linear correlation coefficient between f_i and f_j and T is the pre-specified threshold. Finally, each connected component in the graph represents a partition of the correlated features.

An important observation is that we do not have to capture the exact correlation between features to launch successful attacks. For example, suppose we have three features: "f_1: push rax; push rbx", "f_2: push rax", and "f_3: push rbx". The precise correlation is

$$(\mathtt{freq}(f_1) \leqslant \mathtt{freq}(f_2)) \wedge (\mathtt{freq}(f_1) \leqslant \mathtt{freq}(f_3)) \qquad (6.1)$$

where $\mathtt{freq}(f)$ represents the frequency of feature f. Our algorithm will put all three features in the same partition and derive the following correlation:

$$\mathtt{freq}(f_2) = A_1\mathtt{freq}(f_1) + B_1, \mathtt{freq}(f_3) = A_2\mathtt{freq}(f_1) + B_2 \qquad (6.2)$$

As we will discuss in the next section, it is straightforward to incorporate correlation (6.2) into our feature vector modification. In addition, as the linear correlation is derived from a data set drawn from the same domain as the training set for the target classifier, feature vectors satisfying correlation (6.2) typically also satisfy correlation (6.1).

6.2.2 Feature Vector Modification

Given an input feature vector $\mathbf{x} = [x_1, x_2, \ldots, x_k]$, where x_i represents the feature value of feature f_i, our feature vector modification outputs a modified feature vector $\mathbf{x}'$, such that the prediction results for $\mathbf{x}'$ are different from the prediction results of $\mathbf{x}$ (for untargeted attacks) or are the specified results (for targeted attacks).

We use the approach of training a substitute DNN and transferring the adversarial example to the target classifier [98]. We extend the attack presented by Carlini and Wagner [23], denoted as the CW attack, to generate adversarial feature vectors. We first summarize the CW attack and then explain how we extend it to our domain. The CW attack is regarded as a powerful targeted attack. The CW attack can also be used for untargeted attacks, but the *projected gradient descent* is regarded as a stronger untargeted attack [57]. As we will show in Section 6.3, the untargeted version of the CW attack works well for us.

CW Attack

The CW attack was designed for a DNN. We describe only the prediction process of the DNN as we are attacking a pre-trained model. Given a feature vector $\mathbf{x}$, a pre-trained DNN model can seen as a function $\mathtt{pred}(\mathbf{x})$, which generates a prediction label y and is defined as $y = \mathtt{pred}(\mathbf{x}) = \mathtt{argmax}(\mathtt{softmax}(Z(\mathbf{x})))$, where:

- $\mathbf{z} = Z(\mathbf{x})$, where $\mathbf{z} = [z_1, z_2, \ldots, z_l]$ is a vector of raw (non-normalized) predictions that the DNN generates; l is the total number of labels. $\mathbf{z}$ is also known as the *logits*. The calculation of $Z(\mathbf{x})$ is specified by various hyper-parameters, including the depth and the width of the neural network, the choice of the activation function, and the training parameters for each hidden unit. For a pre-trained model, all these parameters are constant.

- $\mathbf{pr} = \mathtt{softmax}(\mathbf{z})$, where $\mathbf{pr} = [pr_1, pr_2, \ldots, pr_l]$ is a probability vector and pr_i is the probability that the input belongs to label i. $\mathtt{softmax}$ normalizes the raw prediction $\mathbf{z}$ to probability distribution.

- $y = \mathtt{argmax}(\mathbf{pr})$, meaning that the predicted label is the one that has the highest probability.

The CW attack has two variations: one for untargeted attacks, and one for targeted attacks. We first describe the untargeted version. Denote y as the original prediction label for $\mathbf{x}$. The output of the untargeted CW attack is a new vector $\mathbf{x'}$ such that $\mathtt{pred}(\mathbf{x'}) \neq y$. $\mathbf{x'}$ is defined as $\mathbf{x} + \boldsymbol{\delta}$, so once we have calculated $\boldsymbol{\delta}$, we know $\mathbf{x'}$.

Carlini and Wagner formulated an optimization problem to calculate $\boldsymbol{\delta}$, balancing two factors for minimization. First, to cause misprediction, the new logits vector $\mathbf{z'} = Z(\mathbf{x} + \boldsymbol{\delta})$ should satisfy the condition that z'_y is no longer the maximum element in $\mathbf{z'}$, which in turn means that y is not the predicted label for $\mathbf{x'}$. Carlini and Wagner defined function $g(\mathbf{x'})$ to measure the difference between z'_y and $\mathtt{max}_{i \neq y}(z'_i)$:

$$g(\mathbf{x'}) = \mathtt{max}(z'_y - \mathtt{max}_{i \neq y}(z'_i) + s, 0) \tag{6.3}$$

Intuitively, the smaller the $g(\mathbf{x'})$, the more likely there will be a misprediction. Here, s is a hyper-parameter to control the separation between z'_y and $max_{i \neq y}(z'_i)$. s is a positive value, typically ranging from 1 to 1000.

Second, the modification to the original feature vector should be minimized to avoid detection. So, the number of non-zero elements in $\boldsymbol{\delta}$ and the magnitude of individual $\boldsymbol{\delta}_i$ should also be part the optimization function. For this purpose, Carlini and Wagner used the L_q norm, defined as

$$L_q(\boldsymbol{\delta}) = \|\boldsymbol{\delta}\|_q = (\sum_{i=1}^{n} \delta_i^q)^{1/q} \tag{6.4}$$

The attacker chooses a value for q based on the target domain. Common choices for q are 0, 2, and ∞. L_0 measures only the number of modified features and ignores the magnitude of changes. On the other hand, L_∞ measures only the maximal magnitude of changes and ignores all other changes. L_2 balances the number of changed features and the magnitude of changes.

In general, minimizing (6.3) and (6.4) are conflicting; the more changes are made to $\mathbf{x}$ (larger value for L_q), the more likely the attack can cause misprediction (smaller value for $g(\mathbf{x'})$). Carlini and Wagner introduced a hyper-parameter c to balance these two conflicting optimization targets and defined the final optimization function as

$$L_q(\boldsymbol{\delta}) + c \times g(\mathbf{x} + \boldsymbol{\delta}) \tag{6.5}$$

When $q = 2$, the optimization function (6.5) is differentiable. A general purpose optimizer such as Adam [66] can be used to minimize (6.5) and calculate $\boldsymbol{\delta}$. For $q = 0$, (6.5) is not differentiable. Carlini and Wagner designed an iterative algorithm to calculate $\boldsymbol{\delta}$. In each iteration, the algorithm uses their L_2 attack to identify some features that do not have much effect on causing misprediction and then fixes those features. The values of the fixed features will not change in later iterations. By iteratively eliminating unimportant features, the algorithm identifies a small (but possibly not minimal) subset of features that can be modified to generate an adversarial example. They designed another iterative algorithm for $q = \infty$.

Finally, after calculating δ, we derive $x' = x + \delta$.

For targeted attacks, denote tar as our misprediction target. The only difference between the targeted CW attack and the untargeted version is the definition of g. For targeted attacks, we want to make z'_{tar} the maximal element in the new z', so that tar will be the new prediction label. So, Carlini and Wagner defined $g(x')$ as

$$g(x') = max(max_{i \neq tar}(z'_i) - z'_{tar} + s, 0) \tag{6.6}$$

Extension to CW Attack

To apply the CW attack to our domain, we need to make two modifications. First, the CW attack may generate feature vectors with non-integer values. However, as discussed in Section 6.1.1, Caliskan-Islam et al. [21] used feature counts to construct feature vectors. So, x' should only have integer values. A simple strategy that works well for us is to round values generated by the CW attack to the nearest integer.

Second, we must incorporate the feature correlation information derived in Section 6.2.1 into the attack. To do this, we normalize each individual feature to a Gaussian with zero mean and unit variance, merge all correlated features into one feature, and let the CW attacks work with only the merged features. Recall that we track linear correlation between features; for two correlated features f_1 and f_2, $freq(f_1) = Afreq(f_2) + B$. After the normalization step, A is normalized to 1 and B is normalized to 0. Therefore, we can merge them into a single feature.

We then need to determine the values of the hyper-parameters used in CW attacks. For c and s, we perform a grid search to find a successful value-pair. For L_q, we use the L_0 norm because we would like to minimize the number of modified features rather than the magnitude of the modifications.

We found that CW's L_0 attack often did not generate adversarial feature vectors with the minimal number of modified features. So, we design a two-step post-processing to further reduce the number of modified features and the magnitude of changes. First, for each modified feature, we undo the modification and set its value to its unmodified value. If we can still cause misprediction, we finalize the undo of the modification. Second, for each modified feature, we enumerate every integer between the unmodified value and the new value. We set the value of this feature to the one that is closest to the unmodified value and causes misprediction. As we will show in Section 6.3, this simple post-processing strategy can effectively reduce the number of modified features.

6.2.3 Binary modification strategies

Given a new feature vector x' that causes misprediction, we describe how to modify the input binary to match x', grouping features based on the program properties that the feature describe and the binary analysis tool used to extract the feature. We also describe feature injection and removal strategies for feature groups. Our modification strategies consist of binary modification primitives supported by tools such as Dyninst [100]. Last, we discuss how to determine which modification strategy to use for a modified feature.

Feature injection strategies

Table 6.1 summarizes our feature injection strategies. The first column lists the program properties we are going to inject, including machine instructions and loading the address of a symbol or data. The second and

Table 6.1: **Summary of feature injection strategies**. The first column lists the program properties to inject. The second and third columns list the binary modification primitives used for the injection.

Program Property	NDISASM	radare2
Instructions `I`	`InsertNonCodeBytes(I)`	`InsertFunction(I)`
Loading symbol `S`	NA	`addr = InsertSymbol(S)` `InsertFunction(loading addr)`
Loading data `D`	NA	`addr = InsertData(D)` `InsertFunction(loading addr)`
Calling function `F`	NA	`addr = InsertCall(F)` `InsertFunction(calling addr)`

third columns list the modification primitives needed to inject features that can be extracted by NDISASM and radare2. A cell with "NA" means that the binary analysis tool cannot extract the program property. We discuss the non-NA cells in more details:

- Instructions extracted by NDISASM: The modification primitive `InsertNonCodeBytes(I)` creates a new non-loadable section in the binary to store the bytes representing new instructions. As NDISASM disassembles all bytes in the target binary, `InsertNonCodeBytes(I)` ensures that the features are injected and the functionality is unchanged.

- Machine instructions extracted by radare2: The modification primitive `InsertFunction(I)` creates a new function in which we store the inserted instructions. To ensure that radare2 disassembles the inserted code, `InsertFunction(I)` creates a new code section to store the inserted instructions, appends a return instruction at the end, and create a new function symbol to point to the inserted instructions.

- Loading symbol `S`: The modification primitive `InsertSymbol(S)` inserts the symbol `S` into the target binary and returns the address pointing to the symbol. It is important to properly fill in all fields of the symbol in the symbol table, including symbol type, symbol

Table 6.2: **Summary of feature removal strategies**. The first column lists the program properties to remove or replace. The second and third columns list the binary modification primitives for feature removal.

Program Property	**NDISASM**	**radare2**
Instructions `I` from debug-time sections	`Overwrite(I)`	NA
Instructions `I` from code sections	`Swap(I)` or `InsertNop(I)`	
Addressing loading of symbol `S`	NA	`SplitAddrLoad(S)`
Addressing loading of data `D`	NA	`SplitAddrLoad(D)`
Function call to function symbol `S`	NA	`ConvToIndCall(S)`

visibility, and symbol section index. Binary analysis tools may ignore incomplete symbols, causing the injection to fail. We then use `InsertFunction(I)` to insert code that loads the address of the new symbol.

- Loading data `D`: The modification primitive `InsertData(D)` inserts the specified data into the target binary. We typically need to create a new data section to hold the injected data. Then, we use `InsertFunction(I)` to insert code that loads the data.

- Calling function `F`: The modification primitive `InsertCall(F)` inserts the specified function `F` into the target binary, where `F` can be a function from an external library. In such case, we also need to add information for dynamic linking into the target binary, including a dynamic function symbol, a relocation entry, and a procedural linkage stub (PLT) for performing the external call. Then, we use `InsertFunction(I)` to insert code that calls `F`.

Feature removal strategies

Table 6.2 summarizes our feature removal strategies. The first column lists the program properties we are going to remove or replace. The second and third columns list the binary modification primitives needed for removing a feature group:

- Instructions `I` from debug-time sections: The modification primitive `Overwrite(I)` overwrites the target instruction bytes to other bytes. This strategy does not change the program functionality as debug-time sections are not used at link-time or run-time.
- Instructions `I` from code sections: We design two strategies for this feature group. The modification primitive `Swap(I)` checks the operand dependencies and reorders the instructions if there is no dependency. The modification primitive `InsertNop(I)` inserts a nop instruction between the original instructions. Note that to insert a nop instruction, we may need to relocate the original instructions to a different location to create extra space for the nop. Therefore, we prefer `Swap` over `InsertNop` if possible.
- Address loading of `S`: The modification primitive `SplitAddrLoad(S)` splits the address loading instruction into two instructions so that radare2 will not recognize the address loading. We use two instructions: loading the address minus 1 into the target register and incrementing the target register.
- Function call to `S`: The modification primitive `ConvToIndCall(S)` converts a function call to `S` to an indirect (pointer-based) function call, so that radare2 will not recognize the call target. `ConvToIndCall(S)` uses `SplitAddrLoad(S)` to load the function call target and then generates an indirect call. Note that we need to save and restore the register used for performing the indirect call if it is live at this point in the code.

Deciding which strategy to apply

We have several criteria to determine which strategy to use for a modified feature. Based on the sign of δ_i, we decide whether we need to inject (see Table 6.1) or remove (see Table 6.2) features. Based on the address where the feature was extracted, we determine from which section the feature is extracted, including debug-time sections, code sections, or data sections.

For features extracted from code sections, we determine whether the feature describes a function call, loading a symbol, or loading data. If none of the three cases applies, the feature describes just instructions, and no other program property needs to be modified.

6.3 Evaluations

We evaluate several aspects of our attacks: (1) whether we can effectively perform untargeted attacks to evade authorship identification, (2) whether we can effectively perform targeted attacks to impersonate someone else, (3) which features are modified in our attacks and which binary modification strategies are commonly used, (4) whether our post-processing steps are effective for reducing the number of modified features, and (5) why some of our attacks failed. Our evaluations show that

- Our untargeted attacks are effective. We achieved 96% success rate in our experiments, showing that we can effectively suppress authorship signals.
- The success rate of our targeted attacks are 46% on average, showing that it is significantly more difficult to impersonate someone else.
- The top modified features describe function calls. This indicates that authorship identification classifiers heavily rely on function calls to identify authors. Therefore, inserting function calls that are associated with other authors is an effectively way to cause misprediction.
- Without our post-processing, there are 83 features to modify on average. With our post-processing, there are only 10 features to modify on average. Therefore, our post-processing procedure can

significantly reduce the number of changed features for launching a successful attack.

- For failed untargeted attacks, the lack of strategies for modifying CFG features and decompiled source features is the reason for failure. For failed targeted attacks, about a third of the cases are caused by lack of modification strategies for CFG and decompiled source features; the other two thirds of the cases failed because the targeted CW attack cannot generate a feature vector that both corresponds to a real binary and causes the required misprediction.

6.3.1 Evaluation Methodology

We evaluated our techniques by attacking classifiers trained with the techniques presented by Caliskan-Islam et al. [21] (described in Section 6.1.1). Our experiments consist of the following steps:

1. Randomly sample K authors from the Google Code Jam data set of around 1000 authors used by Caliskan-Islam et al. [21]. This data set consists of the source code of single-author programs, each with an author label.

2. Compile all the programs written by the sampled authors with GCC 5.4.0 and -O0 optimization. Each author had an average of 8 binary programs.

3. Split the binaries into a training set and a testing set, with a size ratio of about 7:1.

4. Train a random forest classifier with the training set.

5. Perform our attack on each binary in the testing set for which the target classifier makes the correct prediction. For each test binary, we

perform one untargeted attack, and $K-1$ targeted attacks. The targeted attacks attempt to cause misprediction for each of the incorrect authors.

We varied K from 5 to 100 to investigate how the number of training authors impact the effectiveness of our attacks. For each value of K, we repeated the experiments five times and report the averaged results. In their study of evading source code authorship identification, Simko et al. [116] varied K from 5 to 50, and commented that untargeted attacks become easier when the number of training authors increases, and the ease of targeted attacks does not have an obvious connection to the number of training authors. We expand the range for K in our experiments.

We used Scikit-learn [101] for training random forest classifiers, Tensorflow [124] for training substitute classifiers, and Dyninst [100] for implementing our binary modification strategies.

We use success rate to measure the effectiveness of our attacks, defined as

$$\frac{\#\ \mathit{of\ successful\ attacks}}{\#\ \mathit{of\ total\ attacks}} \tag{6.7}$$

An attack is successful if the binary generated by our attack caused the target classifier to make an incorrect prediction. For untargeted attacks, incorrect prediction means any of the incorrect authors. For targeted attacks, incorrect prediction means the specific targeted author.

6.3.2 Evaluation Results

The first question to answer in our evaluation is how effective is our untargeted attack. The results are shown in Table 6.3. In this table, the second and the third columns are the accuracy of the target classifiers and the substitute classifiers. The fourth and the fifth columns list the the success rate of untargeted and targeted attacks. Our untargeted attack has a *96% success rate on average, showing that we can effectively suppress authorship sig-*

nal. However, our targeted attacks did not enjoy the same success as the untargeted ones. Our targeted attack has a 46% success rate on average, showing that it is significantly more difficult to impersonate a specific programmer's style.

Table 6.3 also shows how the number of training authors K impacts the effectiveness of our attacks. For untargeted attacks, our success rate increases as K increases. This aligns with the observation made by Simko et al. [116]. Untargeted attacks only need to cause misprediction against any of the $\mathsf{K}-1$ incorrect authors. The larger the K, the more incorrect authors our attacks can work with, and the higher the success rate. For targeted attacks, our success rate decreases as K increases. This does not align with the observation made by Simko et al., where they concluded that the ease of targeted attacks does not have an obvious connection to K. In Simko et al.'s experiments, they always performed targeted attacks against only 5 of the K authors, regardless of the value of K. In another words, their experiments did not evaluate all possible scenarios of targeted attacks. On the other hand, we attempted to perform targeted attacks against each of the incorrect authors, covering all scenarios of targeted attacks.

The accuracy gap between the target classifier and the substitute classifier does not obviously impact the success rate of our attack. As shown in Table 6.3, The accuracy gap ranges from 0% to 20%. The success rates of both untargeted and targeted attacks do not exhibit an obvious correlation with the accuracy gap.

We then investigate what are the commonly used binary modification strategies and what are the commonly modified features. In Table 6.4, we list the number of times that a modification primitive is used in our untargeted attacks for $\mathsf{K} = 30$. The most frequently used primitive is `InsertCall`, indicating that the target classifiers heavily rely on function call features to identify authors. So, inserting function calls that are associated with other authors is an effectively way to cause misprediction. `SplitAddrLoad`

Table 6.3: **Evaluation results.** The second and the third columns list the accuracy of the target classifiers and the substitute classifiers. The fourth and the fifth columns show the success rate for untargeted and targeted attacks.

K	Target classifier accuracy	Substitute classifier accuracy	Untargeted attack success rate	Targeted attack success rate
5	100%	100%	88%	88%
15	100%	80%	93%	51%
30	89%	73%	98%	47%
50	86%	69%	100%	31%
100	82%	68%	100%	14%
Average	91%	78%	96%	46%

Table 6.4: **Number of times that a binary modification primitive is used in untargeted attacks**. The numbers are from the attacks for 30 training authors (K = 30). The rows are sorted in a decreasing order.

Modification primitive	Times used
`InsertCall`	586
`SplitAddrLoad`	292
`InsertFunction`	196
`Swap & InsertNop`	193
`ConvToIndCall`	115
`InsertNonCodeByte`	102
`Overwrite`	85
`InsertData`	68
`InsertSymbol`	45

ranks second, showing that the target classifiers also rely on features that describe the loading of a symbol to identify authors. `InsertFunction` ranks third, showing that inserting instructions that are typically seen in programs written by other authors is also effective for causing misprediction. `Swap` and `InsertNop` serve the purpose of removing instruction features. These two primitives have an effectiveness similar to `InsertFunction`, indicating that removing distinct instruction sequences associated with an author is effective for causing misprediction. Other strategies including editing debug sections, inserting data, and inserting symbols, all play important roles in our attacks.

Table 6.5: **The number of feature changed by our untargeted attacks.** The second column lists the average number of features changed by the L_0 CW attack. The third column shows the average number of features changed after our post-processing.

K	L_0 CW attack	Our post-processing
5	57	9
15	107	11
30	87	11
50	59	9
100	92	11
Average	80	10

Next, we investigate how many features we need to change to cause misprediction. In Table 6.5, the second column shows the number of changed features generated by the untargeted L_0 CW attack, and the third column shows the number of changed features after our post-processing step. Before post-processing, there are 80 features to modify on average. After the post-processing, there are only 10 features to modify on average. Our results show that our post-processing procedure can significantly reduce the number of changed features for performing a successful attack.

6.3.3 Analysis of Failed Attacks

Our attack contains two key steps: feature vector modification to generate a vector that both corresponds to a real binary and causes the required misprediction, and input binary modification to generate a new binary that matches the adversarial feature vector. A failure in either of the two steps would lead to a failed attack. Feature vector modification fails when it cannot find such an adversarial feature vector that corresponds to a real binary and causes the required misprediction. Input binary modification fails when it does not generate a new binary that causes the required misprediction. We found that feature vector modification accounts for all the failed attacks.

We break down the reasons of why our feature vector modification step would fail to generate an adversarial feature vector. Recall that our feature vector modification is based on the CW attack, which generates a feature vector that causes the required misprediction, without considering whether the generated vector would correspond to a real binary. We adapted the CW attack in three ways to generate vectors that correspond to a real binary. First, as we implemented binary modification strategies for only instruction features, the CFG features and decompiled source code features are not modified during feature vector modification. Second, as the value of an instruction feature represents the number of times that this feature appears in a binary, the feature value is an integer. However, the CW attack does not guarantee to generate integer values. So, we round the results of the CW attacks to the nearest integer values. Third, we capture feature correlation and merge correlated features. We can then divide failed feature vector modification into two categories:

Lack of modification strategies for CFG and decompiled source features: It may not be sufficient to modify only instruction features to evade authorship identification. Failed attacks in this category need binary modification strategies for CFG and decompiled source features.

Insufficient handling of finding vectors corresponding to real binaries: Our techniques for generating feature vectors that correspond to real binaries need further improvement. For example, we currently capture only linear correlation between features.

We found that all the failed untargeted attacks were due to not being able to modify CFG or decompiled source features. For failed targeted attacks, not being able to modify CFG or decompiled source features explained about 34% of the failed cases; not being able to find a feature vector that corresponds to a real binary explained the other failed cases. Our analysis shows that to improve untargeted attacks, we need to continue to design new modification strategies for CFG and decompiled source features. To improve targeted attacks, we also need to improve the targeted CW attack to find feature vectors that correspond to real binaries.

While our binary modification strategies were able to match all modified features, We found that they sometimes caused side effects and changed features that should not have been changed. Fortunately, such side effects did not impact the prediction results. The number of unintended changes ranged from 0 to 20. Most of the unintended changes were made to NDISASM instruction features. This is because our feature injection strategies often insert new code and data sections, which in turn requires changes to the program header of the binary. As NIDSASM disassembles all the bytes in the binary, the changes in the program header would cause unintended changes to NDISASM instruction features. It is not surprising that such unintended changes did not impact the prediction results as the program header is unlikely to carry authorship signals.

In summary, the our evaluations show that our attack framework is effective for untargeted attacks and we can practically suppress authorship signals. Performing targeted attacks is significantly more difficult than untargeted attacks. Our results also reveal weaknesses in current authorship identification techniques. Many features used in current authorship identification techniques are based on program properties that are easy to fabricate, such as function calls and symbols. We have shown that we can automatically modify these features, making such classifiers vulnerable to test time attacks.

6.4 Summary

We have presented our attack framework for performing authorship evasion. Our attack framework includes components for analyzing feature correlation, generating feature vectors to cause misprediction, and binary modification strategies to match the generated feature vectors. Our evaluations have shown that our attack framework is effective for untargeted attacks, which is to cause misprediction to any of the incorrect authors. Targeted attacks are significantly more difficult to achieve, which is to cause misprediction to a specific one among the incorrect authors.

Our attack experiences show that it is not secure to rely on features derived from program properties that are easy to modify, such as function calls, symbols, data, and instructions. Authorship identification techniques must consider the trustworthiness of the features.

7 CONCLUSION

Our goal of this dissertation has been to develop new fine-grained techniques for binary code authorship identification and new techniques for evading authorship identification. In this final chapter, we review our technical contributions and describe possible future research directions.

7.1 Contributions

To the best of our knowledge, our research is the first to perform fine-grained binary code authorship identification and the first to attempt evading binary code authorship identification. This dissertation has developed four techniques as the main contributions:

Deriving accurate source code authorship We defined two new line-level source code authorship models: the structural authorship, which represents the complete development history of a line of code, and the weighted authorship, which summarizes the structural authorship to a weight-vector of author contribution percentages. Our new models overcome the limitations of the prior methods that only report the last change to a line of code. We defined the repository graph as a graph abstraction for a code repository. We then designed two backward flow analyses to derive the structural authorship and the weighted authorship. We implemented our two authorship models in a new git built-in tool *git-author* and evaluated *git-author* in two experiments, which showed that our new authorship models can produce more information than the existing methods and that additional information is useful to build a better analysis tool.

Identifying multiple authors We developed new fine-grained techniques for identifying the author of a basic block. We performed an empirical study of three open source software to compare whether we should use the function or the basic block as the attribution unit. 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, and captured authorship relationships between adjacent basic blocks with CRF. 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.

Identifying multiple authors in multi-toolchain scenarios We studied the impact of the compilation toolchain on coding styles and developed new techniques to perform multi-toolchain, multi-author identification. We started with an extensive evaluation of existing multi-author identification techniques, showing that existing techniques did not work well in multi-toolchain scenarios. We designed two new techniques to overcome these weaknesses: the two layer approach that combines toolchain identification and then single-toolchain authorship identification, 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. Our techniques achieved 71% accuracy and 86% top-10 accuracy for classifying 700 authors, showing that analysts can effectively prioritize forensic investigations based on our prediction. Our results also showed that our deep learning models consistently outperformed traditional learning methods such as SVM. 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 interesting lessons from our investigation, such as that unoptimized code is more difficult to attribute. These lessons provide valuable insights on how compilation toolchains impact authorship styles.

Evading authorship identification We presented our attack framework for performing authorship evasion. Our attack framework includes components for analyzing feature correlation, generating feature vectors to cause misprediction, and binary modification strategies to match the generated feature vectors. Our evaluations show that our attack framework is effective for untargeted attacks. Targeted attacks are significantly more difficult to achieve. Our attack experiences show that it is not secure to rely on features derived from program properties that are easy to modify, such as function calls, symbols, data, and instructions. Authorship identification techniques must consider the trustworthiness of the features.

7.2 Future Directions

We see several future research directions that naturally follow this dissertation:

Open world techniques Authorship identification in nature is a closed world task, as it assumes that the target author is in a set of known authors. In real world applications, this assumption may not hold. Therefore, open world authorship analysis is necessary for dealing with unknown authors. Developing open world techniques can be divided into two steps. The first step is to perform the task of authorship clustering, with the goal of grouping the code from the same author together. The second step is to determine which authors are known and which authors are unknown.

Impact of code obfuscation Code obfuscation techniques are commonly used by bad actors to defend their malware from analysis, and runtime packers are a prevalent type of code obfuscation tool. They encrypt the original code into data and decrypt the code at runtime. We see three challenges in this area. First, it is difficult to use static binary analysis to extract code features and perform binary code authorship identification on packed binaries. So, it is essential to use existing unpacking techniques [68, 78, 109, 111] to unpack the code. Second, as the unpacked binaries contain code from both the original authors and the packer [127], authorship analysis must be able to distinguish the original author from the packer. Third, code from the original authors may be modified by the packer [110]. We must be able to identify the original authors despite of the changes made by the packer.

Evolving styles The programming style of a programmer may change through time, which can be caused by learning new programming patterns or incorporating new organizational programming guidelines. The basic idea is to compare the code written by the same author at different points in time. Open source software repositories record the complete develop history, so we can extract code snippets written by the same author at different times. Then, we can investigate whether the style of an author does indeed evolve through time and, if it is the case, what factors cause such change.

Evading fine-grained techniques In Chapter 6, we described our new techniques for evading single-author identification. The natural next step is to attempt evading our fine-grained techniques discussed in Chapter 4 and Chapter 5. The major challenge of evading fine-grained identification is to design fine-grained binary modification strategies to match adversarial feature vectors. As our fine-grained techniques predict authors at the basic block level, fine-grained modification strategies must inject or remove features within basic blocks.

Many of the feature injection and removal strategies presented in Chapter 6 are not applicable as they will introduce new basic blocks.

REFERENCES

[1] IBM, Compuware Reach $ 400M Settlement, 2005. Accessed: 2016-02-18, http://www.computerworld.com/article/2556607/technology-law-regulation/ibm–compuware-reach–400m-settlement.html.

[2] Abbasi, A., Weifeng Li, V. Benjamin, Shiyu Hu, and Hsinchun Chen. Descriptive analytics: Examining expert hackers in web forums. In *2014 IEEE Joint Intelligence and Security Informatics Conference (JISIC)*, Hague, Netherlands, Sep. 2014.

[3] Allodi, L., M. Corradin, and F. Massacci. Then and now: on the maturity of the cybercrime markets (the lesson that black-hat marketeers learned). In *IEEE Transactions on Emerging Topics in Computing*, 4, Feb. 2015.

[4] Alrabaee, Saed, Noman Saleem, Stere Preda, Lingyu Wang, and Mourad Debbabi. Oba2: An onion approach to binary code authorship attribution. In *Digital Investigation*, 11, Supplement 1:S94 – S103, May 2014.

[5] Alrabaee, Saed, Paria Shirani, Lingyu Wang, and Mourad Debbabi. Sigma: A semantic integrated graph matching approach for identifying reused functions in binary code. In *Digital Investigation*, 12, Supplement 1:S61 – S71, Mar. 2015.

[6] Andriesse, Dennis, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *25th USENIX Security Symposium (USENIX Security 16)*, Austin, TX, USA, Aug. 2016.

[7] Apache Software Foundation. Apache http server, http://httpd.apache.org.

[8] Arisholm, Erik, Lionel C. Briand, and Eivind B. Johannessen. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. In *Journal of Systems and Software*, 83(1):2–17, January 2010.

[9] Bengio, Yoshua. Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade*, pages 437–478. Springer, 2012.

[10] Benjamin, V., and Hsinchun Chen. Securing cyberspace: Identifying key actors in hacker communities. In *2012 IEEE International Conference on Intelligence and Security Informatics (ISI)*, Arlington, VA, USA, June 2012.

[11] Biggio, Battista, and Fabio Roli. Wild patterns: Ten years after the rise of adversarial machine learning. In *arXiv preprint arXiv:1712.03141*, 2017.

[12] Biggio, Battista, Blaine Nelson, and Pavel Laskov. Poisoning attacks against support vector machines. In *29th International Coference on International Conference on Machine Learning (ICML)*, Edinburgh, Scotland, June 2012.

[13] Biggio, Battista, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Šrndić, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. Evasion attacks against machine learning at test time. In *2013th European Conference on Machine Learning and Knowledge Discovery in Databases(ECMLPKDD)*, Prague, Czech Republic, Sep. 2013.

[14] Bird, Christian, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar Devanbu. Fair and balanced?: bias in bug-fix datasets. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM*

SIGSOFT symposium on The foundations of software engineering (ES-EC/FSE), Amsterdam, The Netherlands, Aug. 2009.

[15] Bird, Christian, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. Don't touch my code!: examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE)*, Szeged, Hungary, Sep. 2011.

[16] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Wadsworth and Brooks, Monterey, CA, 1984.

[17] Breiman, Leo. Random forests. In *Machine Learning*, 45(1):5–32, October 2001.

[18] Brown, Carson D, David Barrera, and Dwight Deugo. Figd: An open source intellectual property violation detector. In *21st International Conference on Software Engineering and Knowledge Engineering (SEKE)*, Boston, Massachusetts, USA, July 2009.

[19] Burrows, Steven. *Source code authorship attribution*. PhD thesis, Melbourne, Victoria, Australia, RMIT University, 2010.

[20] Caliskan-Islam, Aylin, Richard Harang, Andrew Liu, Arvind Narayanan, Clare Voss, Fabian Yamaguchi, and Rachel Greenstadt. De-anonymizing programmers via code stylometry. In *24th USENIX Security Symposium (SEC)*, Austin, TX, USA, Aug. 2015.

[21] Caliskan-Islam, Aylin, Fabian Yamaguchi, Edwin Dauber, Richard Harang, Konrad Rieck, Rachel Greenstadt, and Arvind Narayanan. When coding style survives compilation: De-anonymizing programmers from executable binaries. In *2018 Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, Feb. 2018.

[22] Canfora, Gerardo, Luigi Cerulo, and Massimiliano Di Penta. Identifying changed source code lines from version repositories. In *Proceedings of the Fourth International Workshop on Mining Software Repositories (MSR)*, Minneapolis, Minnesota, USA, May 2007.

[23] Carlini, Nicholas, and David Wagner. Towards evaluating the robustness of neural networks. In *2017 IEEE Symposium on Security and Privacy (SP)*, San Jose, CA, USA, May 2017.

[24] Chatzicharalampous, Evangelos, Georgia Frantzeskou, and Efstathios Stamatatos. Author identification in imbalanced sets of source code samples. In *2012 IEEE 24th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 790–797, Athens, Greece, Nov. 2012.

[25] Cortes, Corinna, and Vladimir Vapnik. Support-vector networks. In *Machine Learning*, 20(3), Sep. 1995.

[26] Croll, Paul R. Supply chain risk management-understanding vulnerabilities in code you buy, build, or integrate. In *2011 IEEE International Systems Conference (SysCon)*, Montreal, QC, Canada, Apr. 2011.

[27] CyActive. Cyber Security's "Infamous Five" of 2014: The Malware that Gave Hackers the Best Bang for their Buck. http://www.cyactive.com/wp-content/uploads/2014/12/Infamous-5-CyActive1.pdf, 2014. CyActive White Paper.

[28] D'Ambros, Marco, Michele Lanza, and Romain Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. In *Empirical Software Engineering*, 17(4-5):531–577, August 2012.

[29] Darmetko, Craig, Steven Jilcott, and John Everett. Inferring accurate histories of malware evolution from structural evidence. In *26th International Florida Artificial Intelligence Research Society Conference (FLAIRS)*, St. Pete Beach, FL, USA, May 2013.

[30] David, Yaniv, Nimrod Partush, and Eran Yahav. Statistical similarity of binaries. In *37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 266–280, Santa Barbara, California, USA, June 2016.

[31] de la Cuadra, Fernando. The geneology of malware. In *Network Security*, 4:17–20, May 2007.

[32] Edwards, Allen Louis. *Introduction to Linear Regression and Correlation*. W.H.Freeman & Co Ltd, 1976.

[33] Eyolfson, Jon, Lin Tan, and Patrick Lam. Do time of day and developer experience affect commit bugginess? In *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR)*, Waikiki, Honolulu, HI, USA, May 2011.

[34] Fan, Rong-En, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. Liblinear: A library for large linear classification. In *Journal of Machine Learning Research*, 9:1871–1874, June 2008.

[35] FireEye. Gazing into the cyber security: 20 predictions for 2015, 2015. FireEye White Paper, https://www2.fireeye.com/rs/fireye/images/wp-gazing-into%20the-cyber-security-future.pdf.

[36] Frantzeskou, Georgia, Efstathios Stamatatos, Stefanos Gritzalis, Carole E Chaski, and Blake Stephen Howald. Identifying authorship by byte-level n-grams: The source code author profile (scap) method. In *International Journal of Digital Evidence*, 6(1):1–18, 2007.

[37] Fritz, Thomas, Jingwen Ou, Gail C. Murphy, and Emerson Murphy-Hill. A degree-of-knowledge model to capture source code familiarity. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, Cape Town, South Africa, May 2010.

[38] German, Daniel M., and Ahmed E. Hassan. License integration patterns: Addressing license mismatches in component-based development. In *31st International Conference on Software Engineering (ICSE)*, Vancouver, British Columbia, Canada, May 2009.

[39] Git. https://git-scm.com/.

[40] GNU Project. Gcc: The gnu compiler collection, http://gcc.gnu.org, .

[41] GNU Project. The gnu image manipulation program, http://www.gimp.org, .

[42] Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[43] Grosse, Kathrin, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. Adversarial examples for malware detection. In *22nd European Conference on Research in Computer Security (ESORICS)*, Oslo, Norway, Sep. .

[44] Guilfanova, Ilfak, and DataRescue. Fast library identification and recognition technology, 1997. https://www.hex-rays.com/products/ida/tech/flirt/index.shtml.

[45] Guyon, Isabelle, and André Elisseeff. An introduction to variable and feature selection. In *Journal of Machine Learning Research*, 3:1157–1182, Mar. 2003. ISSN 1532-4435.

[46] Hamilton, Booz Allen. Managing risk in global ICT supply chains. 2012. Booz Allen Hamilton White Paper, https://www.boozallen.com/content/dam/boozallen/media/file/managing-risk-in-global-ict-supply-chains-vp.pdf.

[47] Hassan, Ahmed E. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, Vancouver, Canada, May 2009.

[48] Hata, Hideaki, Osamu Mizuno, and Tohru Kikuno. Bug prediction based on fine-grained module histories. In *Proceedings of the 2012 International Conference on Software Engineering (ICSE)*, Zurich, Switzerland, May 2012.

[49] Hemel, Armijn, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Dolstra. Finding software license violations through binary code clone detection. In *8th Working Conference on Mining Software Repositories (MSR)*, Waikiki, Honolulu, HI, USA, May 2011.

[50] Hendrikse, Steven. *The Effect of Code Obfuscation on Authorship Attribution of Binary Computer Files*. PhD thesis, Nova Southeastern University, 2017.

[51] Hex-Rays. Hex-Rays Decompiler, https://www.hex-rays.com/products/decompiler/, .

[52] Hex-Rays. IDA, https://www.hex-rays.com/products/ida/, .

[53] Ho, Tin Kam. Random decision forests. In *3rd International Conference on Document Analysis and Recognition (ICDAR)*, Montreal, Canada, Aug. 1995.

[54] Holt, Thomas J, Deborah Strumsky, Olga Smirnova, and Max Kilger. Examining the social networks of malware writers and hackers. In *International Journal of Cyber Criminology*, 6(1):891–903, Jan. 2012.

[55] HTCondor. High Throughput Computing, 1988. https://research.cs.wisc.edu/htcondor/.

[56] Hu, Xin, Tzi-cker Chiueh, and Kang G. Shin. Large-scale malware indexing using function-call graphs. In *16th ACM Conference on Computer and Communications Security (CCS)*, Chicago, Illinois, USA, Nov. 2009.

[57] Ilyas, Andrew, Logan Engstrom, and Aleksander Madry. Prior convictions: Black-box adversarial attacks with bandits and priors. In *arXiv preprint arXiv:1807.07978*, 2018.

[58] Jacobson, Emily R., Nathan Rosenblum, and Barton P. Miller. Labeling library functions in stripped binaries. In *10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools (PASTE)*, Szeged, Hungary, Sep. 2011.

[59] Jang, Jiyong, David Brumley, and Shobha Venkataraman. Bitshred: Feature hashing malware for scalable triage and semantic analysis. In *18th ACM Conference on Computer and Communications Security (CCS)*, Chicago, Illinois, USA, Oct. 2011.

[60] Jang, Jiyong, Maverick Woo, and David Brumley. Towards automatic software lineage inference. In *22nd USENIX Conference on Security (SEC)*, Washington, D.C., 2013.

[61] Jhi, Yoon-Chan, Xiaoqi Jia, Xinran Wang, Sencun Zhu, Peng Liu, and Dinghao Wu. Program characterization using runtime values and its application to software plagiarism detection. In *IEEE Transactions onSoftware Engineering*, 41(9):925–943, Apr. 2015.

[62] Kamei, Yasutaka, Shinsuke Matsumoto, Akito Monden, Ken-ichi Matsumoto, Bram Adams, and Ahmed E. Hassan. Revisiting common bug prediction findings using effort-aware models. In *Proceed-*

ings of the 2010 IEEE International Conference on Software Maintenance (ICSM), Sep. 2010.

[63] Kantchelian, Alex, J. D. Tygar, and Anthony D. Joseph. Evasion and hardening of tree ensemble classifiers. In *33rd International Conference on International Conference on Machine Learning (ICML)*, New York, NY, USA, June 2016.

[64] Khoo, Wei Ming, Alan Mycroft, and Ross Anderson. Rendezvous: A search engine for binary code. In *10th Working Conference on Mining Software Repositories (MSR)*, pages 329–338, San Francisco, CA, USA, May 2013.

[65] Kim, Sunghun, Thomas Zimmermann, E. James Whitehead Jr., and Andreas Zeller. Predicting faults from cached history. In *Proceedings of the 29th international conference on Software Engineering (ICSE)*, Minneapolis, Minnesota, USA, May 2007.

[66] Kingma, Diederik, and Jimmy Ba. Adam: A method for stochastic optimization. In *arXiv preprint arXiv:1412.6980*, 2014.

[67] Klambauer, Günter, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. Self-normalizing neural networks. In *arXiv preprint arXiv:1706.02515*, 2017.

[68] Kruegel, Christopher, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In *13th Conference on USENIX Security Symposium (USENIX Security)*, San Diego, CA, USA, Aug. 2004.

[69] Lafferty, John D., Andrew McCallum, and Fernando C. N. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *8th International Conference on Machine*

Learning (ICML), pages 282–289, Bellevue, Washington, USA, June 2001.

[70] Lakhotia, Arun, Mila Dalla Preda, and Roberto Giacobazzi. Fast location of similar code fragments using semantic 'juice'. In *2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop (PPREW)*, pages 5:1–5:6, Rome, Italy, Jan. 2013.

[71] Lange, Robert Charles, and Spiros Mancoridis. Using code metric histograms and genetic algorithms to perform author identification for software forensics. In *9th Annual Conference on Genetic and Evolutionary Computation (GECCO)*, London, England, July 2007.

[72] Larochelle, Hugo, Yoshua Bengio, Jérôme Louradour, and Pascal Lamblin. Exploring strategies for training deep neural networks. In *Journal of Machine Learning Research*, 10:1–40, June 2009.

[73] Lindorfer, Martina, Alessandro Di Federico, Federico Maggi, Paolo Milani Comparetti, and Stefano Zanero. Lines of malicious code: Insights into the malicious software industry. In *28th Annual Computer Security Applications Conference (ACSAC)*, Orlando, Florida, USA, Dec. 2012.

[74] Linux Kernel Project. Linux kernel, http://www.kernel.org.

[75] Luo, Lannan, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, Hong Kong, China, Nov. 2014.

[76] Mandiant. Mandiant 2013 Threat Report, 2013. Mandiant White Paper, https://www2.fireeye.com/WEB-2013-MNDT-RPT-M-Trends-2013_LP.html.

[77] Marquis-Boire, Morgan, Marion Marschalek, and Claudio Guarnieri. Big game hunting: The peculiarities in nationÂ--state malware research. In *Black Hat*, Las Vegas, NV, USA, Aug. 2015.

[78] Martignoni, L., M. Christodorescu, and S. Jha. Omniunpack: Fast, generic, and safe unpacking of malware. In *23rd Annual Computer Security Applications Conference (ACSAC)*, Miami Beach, FL, USA, Dec. 2007.

[79] Mathur, Arunesh, Harshal Choudhary, Priyank Vashist, William Thies, and Santhi Thilagam. An empirical study of license violations in open source projects. In *35th Anuual IEEE Software Engineering Workshop (SEW)*, Heraclion, Crete, Greece, Oct. 2012.

[80] Mei, Shike, and Xiaojin Zhu. Using machine teaching to identify optimal training-set attacks on machine learners. In *The Twenty-Ninth Conference on Artificial Intelligence (AAAI)*, Austin, Texas, USA, Jan. 2015.

[81] Mende, Thilo, and Rainer Koschke. Revisiting the evaluation of defect prediction models. In *Proceedings of the 5th International Conference on Predictor Models in Software Engineering (PROMISE)*, Vancouver, British Columbia, Canada, May 2009.

[82] Meng, Xiaozhu. Fine-grained binary code authorship identification. In *24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Student Research Competition Track (FSE-SRC)*, Seattle, WA, USA, 2016.

[83] Meng, Xiaozhu, and Barton P. Miller. Binary code is not easy. In *The International Symposium on Software Testing and Analysis (ISSTA)*, SaarbrÃ¼cken, Germany, July 2016.

[84] Meng, Xiaozhu, Barton P. Miller, William R. Williams, and Andrew R. Bernat. Mining software repositories for accurate authorship. In *2013 IEEE International Conference on Software Maintenance (ICSM)*, Eindhoven, Netherlands, Sep. 2013.

[85] Meng, Xiaozhu, Barton P. Miller, and Kwang-Sung Jun. Identifying multiple authors in a binary program. In *22nd European Conference on Research in Computer Security (ESORICS)*, Oslo, Norway, Sep. 2017.

[86] Menzies, Tim, Jeremy Greenwald, and Art Frank. Data mining static code attributes to learn defect predictors. In *IEEE Transactions on Software Engineering*, 33(1):2–13, January 2007.

[87] Menzies, Tim, Andrew Butcher, Andrian Marcus, Thomas Zimmermann, and David Cok. Local vs. global models for effort estimation and defect prediction. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Lawrence, Kansas, USA, Nov. 2011.

[88] Ming, Jiang, Dongpeng Xu, and Dinghao Wu. Memoized semantics-based binary diffing with application to malware lineage inference. In *30th International Information Security and Privacy Conference (IFIP SEC 2015)*, Hamburg, Germany, May 2015.

[89] Miyato, Takeru, Shin-ichi Maeda, Masanori Koyama, Ken Nakae, and Shin Ishii. Distributional smoothing with virtual adversarial training. In *arXiv preprint arXiv:1507.00677*, 2015.

[90] Moran, Ned, and James Bennett. Supply chain analysis: From quartermaster to sunshop, Nov. 2013. FireEye Labs White Paper, https://www.fireeye.com/content/dam/fireeye-www/global/en/current-threats/pdfs/rpt-malware-supply-chain.pdf.

[91] Moser, Raimund, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th international conference on Software engineering (ICSE)*, Leipzig, Germany, May 2008.

[92] Nagappan, Nachiappan, Brendan Murphy, and Victor Basili. The influence of organizational structure on software quality: an empirical case study. In *Proceedings of the 30th international conference on Software engineering (ICSE)*, Leipzig, Germany, May 2008.

[93] Nagappan, Nachiappan, Andreas Zeller, Thomas Zimmermann, Kim Herzig, and Brendan Murphy. Change bursts as defect predictors. In *Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering (ISSRE)*, San Jose, CA, USA, Nov. 2010.

[94] O'Gorman, Gavin, and Geoff McDonald. The elderwood project, Sep. 2012. Symantec White Paper, `http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/the-elderwood-project.pdf`.

[95] Okazaki, Naoaki. Crfsuite: a fast implementation of conditional random fields (crfs), 2007. URL `http://www.chokkan.org/software/crfsuite/`.

[96] Ouellette, J., A. Pfeffer, and A. Lakhotia. Countering malware evolution using cloud-based learning. In *8th International Conference on Malicious and Unwanted Software (MALWARE)*, Fajardo, Puerto Rico, USA, Oct. 2013.

[97] pancake. Unix-like reverse engineering framework and commandline tools security. http://www.radare.org/, visited, May 2018.

[98] Papernot, Nicolas, Patrick McDaniel, and Ian Goodfellow. Transferability in machine learning: from phenomena to black-box attacks using adversarial samples. In *arXiv preprint arXiv:1605.07277*, 2016.

[99] Papernot, Nicolas, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 372–387, Saarbrucken, Germany, Mar. 2016.

[100] Paradyn Project. Dyninst: Putting the Performance in High Performance Computing, http://www.dyninst.org.

[101] Pedregosa, F., G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. In *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[102] Qiu, Jing, Xiaohong Su, and Peijun Ma. 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, Mar. 2015.

[103] Rahimian, Ashkan, Paria Shirani, Saed Alrbaee, Lingyu Wang, and Mourad Debbabi. Bincomp: A stratified approach to compiler provenance attribution. In *Digital Investigation*, 14, Supplement 1, 2015.

[104] Rahman, Foyzur, and Premkumar Devanbu. Ownership, experience and defects: a fine-grained study of authorship. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, Waikiki, Honolulu, HI, USA, May 2011.

[105] Rieck, Konrad, Thorsten Holz, Carsten Willems, Patrick Düssel, and Pavel Laskov. Learning and classification of malware behavior. In *5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, Paris, France, July 2008.

[106] Roberts, Raymond. Malware development life cycle. In *Virus Bulletin Conference (VB)*, Oct. 2008.

[107] Rosenblum, Nathan, Barton P. Miller, and Xiaojin Zhu. Recovering the toolchain provenance of binary code. In *2011 International Symposium on Software Testing and Analysis (ISSTA)*, Toronto, Ontario, Canada, July 2011.

[108] Rosenblum, Nathan, Xiaojin Zhu, and Barton P. Miller. Who wrote this code? identifying the authors of program binaries. In *16th European Conference on Research in Computer Security (ESORICS)*, Leuven, Belgium, Sep. 2011.

[109] Roundy, Kevin A. *Hybrid Analysis and Control of Malicious Code, Doctoral disstertation, University of Wisconsin-Madison*. PhD thesis, Madison, WI, USA, University of Wisconsin-Madison, 2012.

[110] Roundy, Kevin A., and Barton P. Miller. Binary-code obfuscations in prevalent packer tools. In *ACM Computing Survey*, 46(1), Oct. 2013.

[111] Royal, Paul, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *22nd Annual Computer Security Applications Conference (ACSAC)*, Miami Beach, FL, USA, Dec. 2006.

[112] Ruttenberg, Brian, Craig Miles, Lee Kellogg, Vivek Notani, Michael Howard, Charles LeDoux, Arun Lakhotia, and Avi Pfeffer. 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.

[113] Sæbjørnsen, Andreas, Jeremiah Willcock, Thomas Panas, Daniel Quinlan, and Zhendong Su. Detecting code clones in binary executables. In *18th International Symposium on Software Testing and Analysis (ISSTA)*, pages 117–128, Chicago, IL, USA, July 2009.

[114] Schwarz, B., S. Debray, and G. Andrews. Disassembly of executable code revisited. In *Ninth Working Conference on Reverse Engineering (WCRE)*, Richmond, VA, USA, Oct. 2002.

[115] Security, RSA. Rsa 2012 cybercrime trends report :the current state of cybercrime and what to expect in 2012, 2012. RSA Security White Paper, `http://www.cs.toronto.edu/~lloyd/TKF/TKF11/11634_CYBRC12_WP_0112.pdf`.

[116] Simko, Lucy, Luke Zettlemoyer, and Tadayoshi Kohno. Recognizing and imitating programmer style: Adversaries in program authorship attribution. In *Proceedings on Privacy Enhancing Technologies*, 2018(1): 127–144, 2018.

[117] Śliwerski, Jacek, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *Proceedings of the 2005 International Workshop on Mining Software Repositories (MSR)*, St. Louis, Missouri, USA, May 2005.

[118] Spearman, Charles. The proof and measurement of association between two things. In *The American Journal of Psychology*, 15(1): 72–101, 1904.

[119] StojanoviÄ‡, SaÅ¡a, Zaharije RadivojeviÄ‡, and MiloÅ¡ CvetanoviÄ‡. Approach for estimating similarity between procedures in

differently compiled binaries. In *Information and Software Technology*, 58:259 – 271, July 2015.

[120] Sutton, Charles, Andrew McCallum, et al. An introduction to conditional random fields. In *Foundations and Trends® in Machine Learning*, 4(4):267–373, 2012.

[121] Szegedy, Christian, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In *arXiv preprint arXiv:1312.6199*, 2013.

[122] Tatham, S., and J. Hall. The netwide disassembler: NDISASM. http://www.nasm.us/doc/nasmdoca.html, visited, May 2018.

[123] Tennyson, Matthew F. On improving authorship attribution of source code. In *4th International Conference Digital Forensics and Cyber Crime (ICDF2C)*, Lafayette, IN, USA, Oct. 2012.

[124] TensorFlow. An open-source software library for machine intelligence, 2015. https://www.tensorflow.org/.

[125] Tian, Zhenzhou, Qinghua Zheng, Ting Liu, Ming Fan, Eryue Zhuang, and Zijiang Yang. Software plagiarism detection with birthmarks based on dynamic key instruction sequences. In *IEEE Transactions on Software Engineering*, 41(12):1217–1235, Dec. 2015.

[126] Tramèr, Florian, Nicolas Papernot, Ian Goodfellow, Dan Boneh, and Patrick McDaniel. The space of transferable adversarial examples. arXiv preprint arXiv:1704.03453, 2017. `https://arxiv.org/abs/1704.03453`.

[127] Ugarte-Pedrero, Xabier, Davide Balzarotti, Igor Santos, and Pablo G Bringas. Sok: deep packer inspection: a longitudinal study of the complexity of run-time packers. In *2015 IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, USA, May 2015.

[128] Wagner, Robert A., and Michael J. Fischer. The string-to-string correction problem. In *Journal of the ACM*, 21(1):168–173, Jan. 1974.

[129] Wu, Rongxin, Hongyu Zhang, Sunghun Kim, and Shing-Chi Cheung. Relink: recovering links between bugs and changes. In *19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC/FSE)*, Szeged, Hungary, Sep. 2011.

[130] Yamaguchi, Fabian, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy (SP)*, San Jose, CA, USA, May 2014.

[131] Yavvari, Chaitanya, Arnur Tokhtabayev, Huzefa Rangwala, and Angelos Stavrou. Malware characterization using behavioral components. In *6th International Conference on Mathematical Methods, Models and Architectures for Computer Network Security (MMM-ACNS)*, St. Petersburg, Russia, Oct. 2012.

[132] Yin, Zuoning, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE)*, Szeged, Hungary, Sep. 2011.

[133] Zeller, Andreas, and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. In *IEEE Transactions on Software Engineering*, 28(2):183–200, February 2002.

[134] Zhang, Guoming, Chen Yan, Xiaoyu Ji, Tianchen Zhang, Taimin Zhang, and Wenyuan Xu. Dolphinattack: Inaudible voice commands. In *2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 103–117, Dallas, TX, USA, 2017.

[135] Zhou, Bolei, Aditya Khosla, Agata Lapedriza, Aude Oliva, and Antonio Torralba. Object detectors emerge in deep scene cnns. In *arXiv preprint arXiv:1412.6856*, 2014.

[136] Zimmermann, Thomas, Rahul Premraj, and Andreas Zeller. Predicting defects for eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering (PROMISE)*, Minneapolis, Minnesota, USA, May 2007.