THE PROVENANCE HIERARCHY OF COMPUTER PROGRAMS

by

Nathan E. Rosenblum

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

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2011

To Annie, Mary, and Sarah.

Acknowledgments

It is my great pleasure to thank the people who, one way or another, helped to bring this all about.

My advisor, Bart Miller, taught me what it means to be a computer scientist, how to approach ideas critically, and how to sift through extraneous details to find the actual message in a paper—even in my own. Jerry Zhu was like a second advisor, patiently helping me to pick my way across the machine learning landscape. The two of them kept me (more or less) on track, for which I am eternally grateful.

The other members of my committee, Tom Ristenpart, Karen Hunt, and Vicki Bier, through their thoughtful comments and suggestions, helped make this dissertation what it is. I owe further thanks to Karen for her collaboration over the course of this research. Somesh Jha and Tom Reps provided guidance and insight from the prelim all the way up to my defense; thank you both.

My research would have been much more difficult, if not impossible, without the efforts of all of the researchers, past and present, on the Paradyn project. There are too many of you to name here; special thanks to my office mates Alex, Kevin, Todd, and David, for tolerating my many idiosyncrasies. Emily has been a pleasure to collaborate with. Drew and Matt patiently answered my innumerable questions about Dyninst when I was a dewy-eyed first year grad student, and patiently helped me solve the bugs I introduced in my first large commit. Dorian, thank you for joining me in typographic arguments at practice talks and for your extremely practical advice about the dissertation process. It was spot-on.

I have the good fortune to count several more members of the Computer Sciences department at Madison as colleagues, mentors, and friends. My thanks to Remzi Arpaci-Dusseau, Charlie Fischer, Ben Liblit, Jude Shavlik, Mike Swift, Mary Vernon, and Stephen Wright, from whom I've learned so much. Of graduate students there are too many, but special thanks to Joe Meehean, Trevor Walker, Matt Fredrikson, Greg Cipriano, Scott Diehl, Joe Chabarek, Nick Penwarden, and Ameet Soni for all

the shop-talk and the non-. Will Benton provided the very template in which I have written this document, and recommends that I thank the Academy.

My colleagues at IDA/CCS gave me interesting problems and the room to work on them, providing timely distraction from my dissertation research. I would like to thank Francis Sullivan for providing me the opportunity, and Dan Ridge for showing me the meaning of creative problem solving.

Vic Zandy and Meg Twiddy are two of the best people I know, or hope to.

Emily Blem, Ryan Bannen, and Sarah Cunningham Bannen made Madison home, again and again.

My parents, Charlie and Mary Rosenblum, and my brother Jacob, have provided support and amused tolerance in equal measure over the course of my lengthy education. Annie's parents, John MacLeod and Chickie Massa, her sister Lizzy, and Derek Oldham have welcomed me with open hearts. Few people have such families.

Annie Massa-MacLeod, whom I love intensely, made this possible. I hope to someday repay the favor.

Contents

List of Tables

List of Figures

Abstract

Where did this binary come from? How was it compiled? What language did the programmer choose? Who wrote this code? These questions rarely occur to most computer users, but for analysts working in forensics, reverse engineering, and software theft, they are of paramount importance. The provenance of a program binary—the specific process through which an idea is transformed into executable code—can provide valuable insight, yet it is in the very domains where such information would be most useful that it is least likely to be available.

The thesis of this dissertation is that characteristics of a program's provenance are inherently preserved during translation from source code to an executable form. We model program provenance as a hierarchy, and show that it is possible to recover details of a program's path through this hierarchy by combining evidence extracted from the program with models derived from large binary code data sets using machine learning techniques. In addition, we show that recoverable provenance characteristics extend beyond the tool chain used to produce a binary; we demonstrate that programmers can be identified based solely on characteristics of executable code, and introduce techniques to cluster programs according to stylistic (as opposed to functional) similarity.

1

Introduction

Justly or not, program binaries exist below the attention threshold of most computer users, even software engineers and computer scientists; we are more often concerned with what a program *does* than with the details of its executable format. This indifference persists despite the fact that a program binary can have substantially different properties than the original source program [4]. Recognition of the importance of understanding program binaries has driven the development of binary instrumentation and analysis tools [37], as has the necessity of analyzing binaries when program source code is unavailable, as is the usual case when studying viruses and other malicious programs.

The properties of a program binary are not only a function of the source code, but also of the compiler and other tools used to create the executable program. While binaries are increasingly the subject of analysis, little attention has been paid to the program production process; this despite the fact that in domains like security and software forensics, knowing how a program was produced (or who produced it) can be as important as understanding its purpose. One explanation for a lack of interest in this *program provenance* is that binaries do not typically come annotated with a manifest of such things as their authors or compilation options, rendering moot the question of how to exploit such information.

This dissertation seeks to fill in the blanks of the provenance problem, investigating whether and to what extent a program's provenance can be recovered from binary code. Our central hypothesis is that provenance is intrinsically encoded in the program binary: that variations in different stages of the production process—from stylistic flourishes of the programmer to the use of particular compiler optimizations—are reflected systematically in the resulting executable. We have developed novel program representations and techniques to model the provenance hierarchy, and have developed a framework based on machine learning techniques to recover a program's provenance using information derived from other, unrelated programs. Our studies show that properties of program provenance often can be accurately recovered from binary code;

counter-intuitively, even high-level properties like the identity a program's author can be inferred from low-level binary characteristics.

1.1 Why provenance?

There are two ways in which questions of program provenance arise: either because specific details of the production process are sought, or because such details reveal something about some other property of the program. The former case covers more than just identifying a particular stage of production ("to what degree was this code optimized?"), encompassing problems like identifying programs of mixed provenance ("does this binary contain statically linked library code?"); such provenance inquiries may not even deal in specifics at all ("is this program the work of several authors?"). A program's provenance may also indirectly relate to non-provenance questions: knowing that a buggy compiler was used, for example, may help to diagnose a crash or performance problem.

In the remainder of this section, we describe two broad areas where program provenance has direct or indirect significance. Both of these areas have the defining characteristic that provenance details are most useful exactly when they are least likely to be available. The provenance recovery techniques that we introduce in this dissertation address this fundamental tension.

1.1.1 Malware analysis and software forensics

The contemporary computing security environment is characterized by a preponderance of malicious software—malware—created and spread throughout personal computers and servers to support a variety of illicit activities. The propagation of new malware is rampant; by some estimates 63,000 new malware variants arose every day in 2010 [85]. Analyzing these threats involves not only understanding the programs—identifying such things as their functionality and command and control channels—but also investigating how they arose, their relationship to existing, known malware, and who might be responsible for their production. Program provenance questions intersect with these goals in several areas:

Reverse engineering Understanding a malicious program requires interpreting its binary code, running the program and observing it, or frequently both; malware authors appear reluctant to provide source code for their programs. Provenance details can assist in such reverse engineering, for example by helping to find code during static analysis [75, 76] or providing information about the compilation process to improve *decompilation*, or source-code recovery [95].

Recovering production process Like any program, malware binaries are the product of a transformation process characterized by the specific tools—source programming language, compiler, and (generally unique to the malware domain) post-compilation obfuscation or packing methods. The provenance techniques we develop in this dissertation can recover the identity of these tools [77], aiding forensic investigation and allowing similarities between specific production toolchains to be discovered.

Identifying program authors Malware, or the use to which it is put, is often illegal; a natural goal of malware analysts is to find evidence that points to the responsible parties. We have developed authorship attribution techniques that can discriminate among code produced by different authors [78]; such techniques could be used to identify known malware authors, to detect stylistic similarities between several malicious programs, or to track the sharing of program components and programmer expertise within the underground malware economy.

1.1.2 SOFTWARE ENGINEERING AND RELIABILITY

Supporting software deployments is complicated by the fact that computer programs do not exist in isolation: they frequently rely on external dependencies over which designers have little control, and—in the case of open-source projects—the deployed programs may have been built with a variety of different compilation toolchains. For engineers diagnosing bugs or performance problems in deployed software, the provenance characteristics of library dependencies or the deployed program itself can enable compiler-specific analysis [72]. Fine-grained provenance details of the program and its dependencies could augment crash reports for bug detection, so that the precise deployment environment could be recreated [63].

1.2 ORGANIZATION OF THE DISSERTATION

Our work has been experimentally driven, organized around a series of investigations into the provenance hierarchy. While these investigations have been largely independent of one another, they share a common philosophy and many of the same techniques and methodologies. The structure of this dissertation reflects the commonalities among the areas on which we focus. It is divided into four parts: *introduction*, *design*, *experimentation*, and a *conclusion*. The introduction includes this motivating chapter and background material, which has been split into two parts: a discussion of related work in binary code analysis and authorship attribution, and a background chapter on the machine learning concepts used in this dissertation, which can be skipped by those

familiar with the field. The design part describes the mechanisms we introduce for provenance recovery, while the experimentation part presents a series of evaluations of the utility and effectiveness of our techniques. The conclusion summarizes the key contributions of this dissertation and presents a vision for future work.

1.2.1 Design

The design part forms the heart of the dissertation, motivating and formally describing our framework for investigating the provenance hierarchy and introducing the main technical contributions: representations and models of program provenance. Chapter 4 gives a high-level overview of the provenance recovery framework, which is consists of two main components: representation and extraction of code characteristics, and modeling techniques that learn to infer program provenance.

Chapter 5 describes the descriptive *features* through which we sift for patterns of provenance, and evaluates these features' suitability for capturing a variety of provenance properties. In Chapter 6 we develop *provenance models*—primarily based on *conditional random fields* [50] and *support vector machines* [18]—that incorporate these features, and describe the machine learning techniques that we use to learn model parameters and to infer program provenance. The techniques introduced in this part provide a unifying foundation for the experimental evaluation of provenance recovery in later chapters.

1.2.2 Experimentation

The models we have developed provide an an expressive mechanism for representing binary code, but the extent to which such models are useful for recovering program provenance is best evaluated experimentally. We have implemented several tools based on provenance models that infer the specific details of various stages of the production process. Evaluating these tools on large corpora of test programs demonstrates that our techniques are effective for provenance recovery. These experiments, which involve exploration of often large design spaces and depend on frequently costly machine learning techniques, incur substantial computational expense. Rather than pre-optimizing by limiting the scope of our experiments, we choose to throw cycles at the problem; a fundamental aspect of our work has been devising ways to adapt our techniques to a distributed computing infrastructure. Using large-scale computing resources, we have been able to perform and evaluate provenance recovery on data sets whose size would have otherwise been prohibitive.

We bring a variety of machine learning techniques to bear on the provenance prob-

lem. Our most common approach is to frame program provenance using probabilistic graphical models [43]. Chapter 7 describes a system to discover function entry points in program binaries using approximate inference in a model based on speculatively discovered control flow graphs, and its implementation as part of the Dyninst binary instrumentation and analysis toolkit [65]; this chapter also describes our use of the Condor distributed computing system for learning and experimentation at scale [55]. In Chapter 8, we introduce techniques based on conditional random fields to recover details of the compiler toolchain, including the programming language, compiler version and specific options used to produce a binary; these techniques have been implemented as an analysis tool, ORIGIN [67]. Chapters 9 and 10 adopt alternative learning techniques to evaluate several solutions related to program authorship. In the former, we show that programmer style is preserved throughout the compilation process, and that the author of a program can frequently be identified from properties of the binary code using the machinery of support vector machines; in the latter, we address the problem of detecting stylistically similar programs when little is known *a priori* about their authors using unsupervised clustering techniques and *transfer learning*.

1.3 CONTRIBUTIONS

To the best of our knowledge, ours is the first study of program provenance. This dissertation describes techniques and methods to automatically recover provenance properties from binary code, making the following contributions:

1. We show that evidence of a program's provenance is encoded in binary code, and can be represented using simple, uninformed features.

2. We develop a framework for modeling program provenance, including novel program representations and probabilistic models that capture the relationship between provenance properties and features of binary code.

3. We introduce methods to automatically infer program provenance by learning model parameters using large corpora of program binaries, and develop tools to recover provenance properties based on these models.

4. We demonstrate that programmer style has a marked impact on the binary code despite the complex transformations of the compilation process, and that our provenance recovery techniques can be used to identify the author of a program or to detect stylistically similar programs using binary code features.

1.4 Guiding principles

Whether and to what extent a program's provenance can be recovered has been, until now, an open question; as with any exploratory research, we have been presented with a vast design space to explore. We have identified several principles that serve to prune this space, or at least to focus our attention onto (we feel) productive areas:

Naïveté first Our foremost principle could also be phrased "don't try to be clever". A program's provenance impacts the binary code representation in complex ways, with multiple elements of the provenance hierarchy interacting with one another. While it is tempting to try to construct equally complex features informed by expert domain knowledge, the design space is vast and sacrifices generalizability to new provenance elements. Instead, we rely on a the fundamental power of a machine learning-based approach, designing simple code features and letting the provenance characteristics in the data speak for themselves.

Provenance, not functionality However we represent programs, we are describing an artifact whose properties are determined both by its functionality and by the way in which it was produced. The fundamental challenge in recovering provenance is to learn to distinguish between these properties. Our learning infrastructure and the data from which we construct models of program provenance are designed with this dichotomy in mind. The distinction is not always clear-cut, however; the visible characteristics of a program exist on a spectrum of influence between provenance and program functionality, particularly for high-level provenance properties like programmer style and authorship.

Mixed program provenance Programs are but strings of bytes assembled by some process, and what is true of one part of the program—whether the settings used by the compiler or the identity of its author—may not be true of another. For example, a program may be composed of code produced by multiple compilers due to static linking against a library; representing the program as having a single, uniform provenance is nonsensical in this case. The possibility of mixed provenance within a program heavily influences the design of our system.

These principles have guided our investigation of program provenance, from the design of our provenance recovery techniques to the methodology of our experiments. The success of our approach will be illustrated in the experimental part of this dissertation; whether this approach is necessary or merely sufficient is beyond our ability to determine.

1.5 A NOTE ON METHODOLOGY

The defining characteristic of our approach to provenance recovery is that it is *data-driven*. Our methodology is simple: we construct data sets of programs that reflect variations in a provenance property of interest (e.g., authorship) and apply a variety of machine learning techniques to the problem of extracting from the data those features that are characteristic of specific provenance properties. The data-driven automation of provenance recovery using machine learning does not mean that our approach is wholly uninformed by domain knowledge, however. The representations and features that we define reflect domain expertise, encoding our expectation of how certain provenance properties might be reflected in binary code. Nevertheless, using data to infer the relationship between these features and program provenance is the most important aspect of our approach. The specific machine learning techniques that we use—which we introduce in Chapter 3 and describe further in the experimental part of this dissertation—are a second-order concern.

An alternative approach to provenance recovery would be to rely more strongly on domain knowledge, for example by using the expertise of specialists to construct rule-based expert systems [9] for identifying particular provenance properties. Using expert knowledge to construct pattern-based or other rules for provenance recovery offers the possibility of highly precise results, as we show in our evaluation of code identification in the Dyninst system in Chapter 7. However, expert systems have several limitations that make the data-driven approach better suited for provenance recovery. Obtaining expert knowledge can incur significant cost, such as developing human expertise about a novel provenance property or eliciting knowledge from existing experts. By contrast, the data-driven approach relies only on computational resources, which can be quickly and easily scaled. The rules encoded by expert systems are typically highly specific (leading to precise results), but have long been recognized to exhibit poor generalizability compared to data-driven learning approaches [25], a phenomenon exemplified in Chapter 7.

The two approaches are not incompatible; in machine learning, it is commonly understood that domain knowledge is as important as learning algorithms, and there has been significant work to integrate data-driven approaches with human expertise [3, 21, 87]. A plausible hybrid provenance recovery system might incorporate a feedback loop in which the inferred provenance properties are provided to a domain expert, who then could indicate invalid or ambiguous results to the system for further refinement. This approach could take advantage of hypothetically available expertise that was not available during the design of the program representations and features that we introduce here. Evaluation of this hybrid methodology and comparison of domain experts to our approach is hampered, however, by the general lack of expertise in

program provenance, which is a largely unexplored area of research. In this dissertation, we focus on the data-driven approach and leave the question of incorporating expert knowledge for future work.

2

Related Work

This dissertation investigates program provenance, the overarching question of which is how to mine specific properties from an executable program. This question is deeply rooted in binary code analysis; to understand our approach, it is necessary to understand fundamentals of this field and those techniques that are most closely related to provenance recovery. In this chapter, we survey analyses that extract meaningful properties from binaries. We focus on two main classes of techniques: derivation of *binary code representations* that capture code details at various levels of abstraction, and techniques that build on these representations to capture *code properties*. We conclude with a review of literature on program authorship attribution, a particularly high-level provenance problem and one of the key contributions of this dissertation.

Binary code analysis depends on having code to analyze; we begin with a brief discussion of *binary code parsing*, or extraction of code from program binaries or other binary code artifacts. For brevity, we constrain our discussion to *static* parsing; *dynamic* techniques such as execution tracing are a complementary source of program code, but have not been the focus of this dissertation. There are many good sources for further information about parsing binary code; Li [52] provides a detailed overview.

2.1 PARSING BINARY CODE

Program binaries are, at the lowest level, a sequence of bytes; the role of a parser is to convert these bytes into a representation of the executable instructions that make up the program code. How the parser accomplishes this conversion is largely orthogonal to the higher-level representations we discuss below, but different parsing techniques have different limitations that may impact analysis results.

The two main classes of parsers, depicted in Figure 2.1, use the *linear sweep* and *recursive traversal* methods. Linear sweep, exemplified by the GNU `objdump` disas-

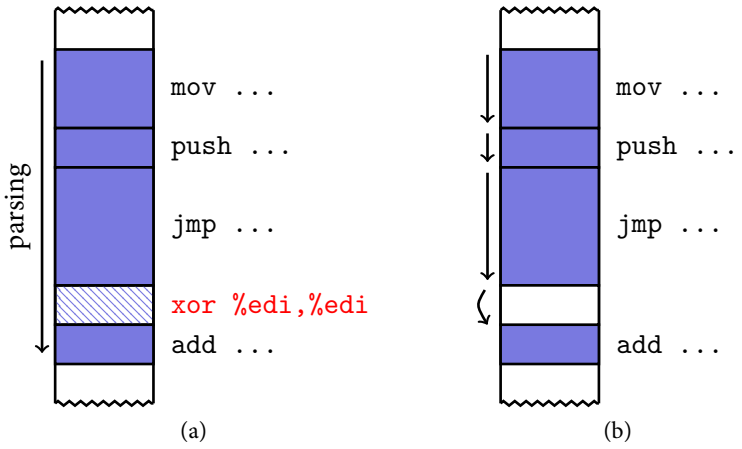


Figure 2.1: Parsing techniques. Linear sweep methods (a) assume that all instructions are contiguous and may incorrectly disassemble unreachable bytes (▨); recursive traversal (b) follows feasible control flow paths and avoids these regions.

sembler,[1] assumes that the sequence of instructions beginning at a given starting offset in the binary are contiguous. Linear sweep parsers can produce erroneous results when this assumption is violated, for instance when padding bytes are introduced to align branch targets or when data is interleved with executable code [93]. Recursive traversal parsing instead follows the *control flow* implied by the instructions, only parsing bytes that are successors of previously-parsed instructions [15, 84, 89].

While recursive traversal avoids the chief limitation of the linear sweep method, it can fail to discover code that is reachable only through data-dependent control flow paths like indirect branches or calls through function pointers. This limitation can be significant: we have found that, on average, 40% of functions in stripped binaries are unreachable through static control flow analysis [75]. Schwarz et al. [84] describe a hybrid of linear sweep and recursive traversal that seeks to overcome this problem, but this technique makes strong assumptions about the availability of relocation information to disambiguate code and data in the binary, significantly limiting its applicability.

A significant amount of work has focused on improving static parsing techniques, either by using static analysis to resolve indirect control flow targets [12, 13, 93, 94] or by incorporating compiler- or platform-specific *normal forms* to help identify the targets of particular sources of indirection like multi-way branches [13] and virtual function invocations [94].

[1] `objdump` is part of the GNU Binutils suite of software [29]

Speculative disassembly has also been used to expand parsing coverage by assuming that *gaps* in between statically parsed code also contain code. Such techniques are distinguished by how broadly they apply this assumption: the Dyninst and RAD tools search for known, compiler-specific instruction patterns [31, 70], while the UQBT system takes a more liberal approach, treating any gaps in the binary as executable code [15].

Static binary parsing, despite substantial efforts at improvement, remains an error-prone process, and the best existing methods rely on significant tool-specific or expert knowledge. The problem with such approaches arises when assumptions are violated: for example, the function discovery heuristics used by Dyninst and IDA Pro [36] are heavily tuned towards the GCC compiler on Linux binaries and have error rates of approximately 50% [75] when applied to binaries produced by the Intel compiler. In Chapter 7 we describe a technique that uses program provenance to improve parsing; nonetheless, analyses based on static parsing—including provenance recovery—must be resilient to errors in parser output.

2.2 REPRESENTING BINARY CODE

Program analyses operate on a representation of the binary code. The appropriate representation is one that captures code properties that are essential to the analysis without introducing extraenous detail—what constitutes "essential" may vary depending on the goal of the analysis. The primary distinguishing characteristic of code representations is the degree to which they abstract away low-level details of the binary code. We illustrate these distinctions with a running example using the following toy C functions:

```
int foo(int a) {
  int tmp = 0;
  while(tmp++ < a)
    --a;
  return bar(tmp);
}

int bar(int a) {
  return (int)sqrt(a);
}
```

```
55 89 e5 83 ec 28 db 45 08 dd 5d f0 dd 45
f0 d9 fa dd 5d e8 dd 45 e8 dd e8 df e0 9e
7a 02 74 0e dd 45 f0 dd 1c 24 e8 e5 fe ff
ff dd 5d e8 dd 45 e8 d9 7d fe 0f b7 45 fe
b4 0c 66 89 45 fc d9 6d fc db 5d f8 d9 6d
fe 8b 45 f8 c9 c3 55 89 e5 83 ec 18 c7 45
fc 00 00 00 00 eb 04 83 6d 08 01 8b 45 fc
3b 45 08 0f 9c c0 83 45 fc 01 84 c0 75 eb
8b 45 fc 89 04 24 e8 85 ff ff ff c9 c3
```

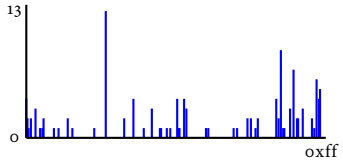


Figure 2.2: Byte frequency distribution for the running code example. The code was produced by the GCC compiler on a 32-bit Linux platform. Byte distributions have been used to distinguish between executable binaries and text files and images.

2.2.1 Instruction-level representation

The lowest level representation of binary code are the bytes that form the executable instructions. Such a representation has no semantics; no boundaries between instructions are imposed, and no distinction is drawn between those that specify the opcode an instruction and those making up its parameters. Even so, the underlying bytes can reveal properties of the binary. Zhang and White [101] use histograms of byte frequency to distinguish executable code from several other file types with good (90%) accuracy, even when only an initial fraction (10 network packets) of the files' contents are considered. The success of this technique is due in large part to the distinctive byte distributions of the non-executable files (JPEG and GIF images and PDF and Microsoft Word documents); its utility for distinguishing between executables with different provenance is not explored. The efficacy of this technique may also depend on regularities in the initial portion of the files examined, such as header metadata in executable file formats; the technique was not evaluated on arbitrary segments of the different files. Our experimental results suggest that byte-level representations such as this one are insufficient for modeling provenance.

An instruction-based representation is more expresive. The `foo` example function comprises seventeen instructions when compiled by the GCC compiler targetting the 32-bit Intel x86 architecture:

```
55                      push   %ebp
89 e5                   mov    %esp,%ebp
83 ec 18                sub    $0x18,%esp
c7 45 fc 00 00 00 00    movl   $0x0,0xfffffffc(%ebp)
eb 04                   jmp    8048453 <foo+0x13>
83 6d 08 01             subl   $0x1,0x8(%ebp)
8b 45 fc                mov    0xfffffffc(%ebp),%eax
3b 45 08                cmp    0x8(%ebp),%eax
```

```
0f 9c c0              setl   %al
83 45 fc 01           addl   $0x1,0xfffffffc(%ebp)
84 c0                 test   %al,%al
75 eb                 jne    804844f <foo+0xf>
8b 45 fc              mov    0xfffffffc(%ebp),%eax
89 04 24              mov    %eax,(%esp)
e8 85 ff ff ff        call   80483f4 <bar>
c9                    leave
c3                    ret
```

Parsing the bytes into instructions allows analyses to distinguish between an instruction's operation and its operands. Combined with specifications of instruction semantics like those provided by the ROSE framework [82], the instruction-level representation enables static analyses like symbolic evaluation [41]. Various tools exist to extract machine instructions from the underlying bytes; in our work, we use the InstructionAPI library [66]. Further representations can be built on top of machine instructions. For example, Saebjornsen et al. [80] use *normalized* forms of instructions that abstract away memory- and register-specific information; this representation is substantially similar to the *idioms* we previously developed for stripped binary parsing [74] and which we describe in Chapter 5.

The chief difference between byte- and instruction-level representations, from a provenance perspective, is that instructions collapse the information contained in several bytes into a single unit. Besides decreasing the resolution of the representation, instructions are a first step in generalizing some of the specifics of the program bytes: depending on how detailed the instruction representation is, several distinct byte patterns may have the same instruction representation. For example, the byte sequences `48 a1 00 00 00 00 00 00 00 00` and `48 8b 04 25 00 00 00 00` both represent instructions that move data from the same memory location to the `rax` register on 64-bit Intel architecture; both disassemble to exactly `mov 0x0, %rax`. Byte- and instruction-based representations can be useful in provenance models; our data-driven approach frequently leads us to use code features based on both.

2.2.2 Control Flow Representation

A *control flow graph* (CFG) is a structural representation of a program that is derived from the underlying instructions [61]. The nodes of the CFG represent *basic blocks*, which are sequences of instructions with the following properties:

(1) instructions in the sequence are *contiguous*, and
(2) each instruction after the first *postdominates* its predecessor.

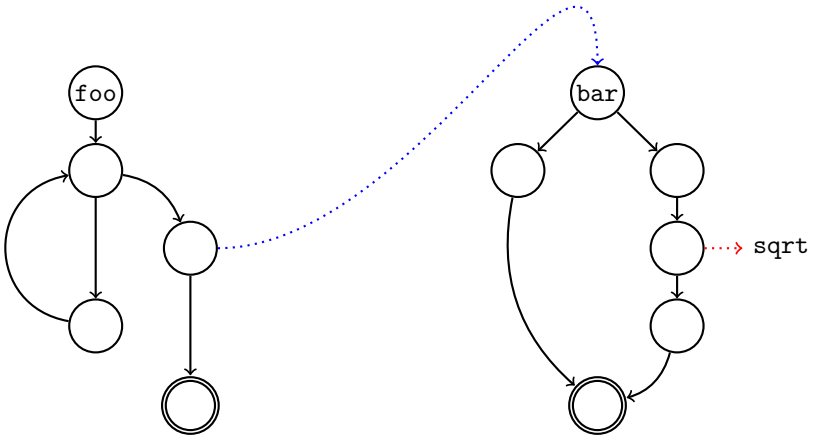


Figure 2.3: The *interprocedural* control flow graph for the `foo` and `bar` functions of the running example.

Basic blocks are convenient for characterizing program structure because the instructions they comprise can be considered an atomic unit. The edges in the graph correspond to possible execution paths between basic blocks. In the CFG of our running example, depicted in Figure 2.3, we distinguish *intraprocedural* control flow edges ($\longrightarrow$, such as conditional branches) from *interprocedural* edges ($\cdots\rightarrow$) that correspond to control transfer between *functions*.

Functions comprise a collection of basic blocks. Unlike basic blocks and control flow edges, functions are a mapping of a source code concept into the binary domain, and are somewhat ambiguously defined; while compilers must adhere to an Application Binary Interface for externally visible functions, no such contraint is imposed on private functions (e.g. C functions declared `static`) and the binary code generated to implement procedural program structures may vary. A common choice, used in the UQBT and Dyninst analysis frameworks [14, 37], is to define functions as the set of blocks reachable through intraprocedural control flow (i.e. branches) from an *entry* block . Entry blocks are identified by the program *symbol table* or as targets of interprocedural `call` instructions.

The control flow graph is a powerful representation, because it allows characterization of arbitrarily large portions of the program, or relationships between different program components. We use the CFG not only to represent code characteristics, but to structure some of our models, for example to ensure that functions in call relationships have particular provenance properties in common (Chapter 7). There is a tension

between the expressiveness of representations and models based on the CFG and their computational cost, however, which we return to in later chapters.

2.2.3 REPRESENTATIONS FOR PROVENANCE

To the best of our knowledge, this dissertation makes the first attempt to characterize and model aspects of program provenance; it is difficult to survey existing approaches for representing code in provenance applications when no such approaches exist. The most closely related work involves characterizing malicious software (*malware*). As we describe below, techniques to identify malware have used representations at the byte [35, 44, 83], instruction [80], and control flow levels [88], as well as hybrids of several representations [46]. The techniques we introduce in Chapter 5 incorporate aspects of all of the code representations discussed above.

2.3 EXTRACTING CODE PROPERTIES

The central goal of this dissertation is to develop methods to discover details that reveal aspects of the provenance of binary code. The majority of existing work has been applied towards two somewhat distantly related tasks: explicitly identifying similarities between programs to identify code duplication or theft, and malicious software identification [98]. Both tasks require techniques to represent the defining features of binary programs and methods that use these features to establish relationships between different binaries.

In the following sections, we survey various methods for extracting code features from different binary code representations. We devide these methods into three classes: those based on simple *N-gram* patterns (e.g., of bytes or instructions), techniques that use more complicated instruction patterns, and control flow-based approaches.

2.3.1 N-GRAMS

A common way to characterize a program binary is as a collection of *N-grams* derived from a byte- or instruction-level representation. N-grams are (usually short) sequences of tokens of length N. For example, if $N = 3$, the initial bytes in the `foo` function (`55 89 e5 83 ec...`) can be represented as

$$\langle \texttt{55 89 e5} \rangle, \langle \texttt{89 e5 83} \rangle, \langle \texttt{e5 83 ec} \rangle, \ldots$$

with an N-gram representation based on bytes. Byte-level N-grams have been used to distinguish betweeen different file types [53], and to identify malicious programs [35, 44, 83].

N-grams do not directly capture specific properties of code, such as the way a compiler encodes a programming idiom or sequences of instructions that are distinctive of a particular malware family. Machine learning and data mining algorithms must be used to learn which N-grams are representative of a particular property, such as whether a sequence of bytes is more likely to be code or text. One limitation of N-gram–based code representations is their lack of generalizability. In malware classification, for example, *polymorphic* malware that changes its byte-level representation as it propagates can evade detection based on byte paterns [83]. N-grams also fail to to incorporate long-range information within the binary: significant relationships between disparate elements that cannot be captured by short N-grams.

structured patterns

When more is known about how the binary representation reflects a property of interest, more specific pattern-based techniques are applicable. Such structured patterns are usually based on an instruction-level representation. The UQBT binary translation framework uses an extension of the SLED instruction specification language [73] to recognize specific procedure abstractions in compiled code [16]. For example, the specification

```
CALLEE_PROLOGUE std_entry locals=0, regs IS
    PUSHod (EBP);
    MOVrmod (EBP, Reg(ESP));
    { SUBiodb (Reg (ESP), locals) };
```

describes a common IA-32 function preamble that saves a stack frame. The Dyninst instrumentation framework uses a similar specification of common preamble patterns to detect code in stripped binaries [31].

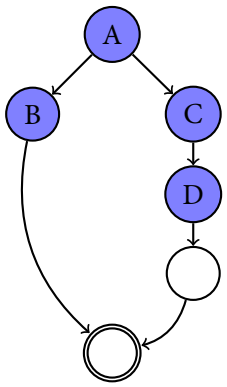
Both of these techniques are distinguished by the use of specific prior knowledge to design patterns that reveal particular properties about binary code. In the provenance domain, we lack such knowledge; our approach is to instead define broad classes of code patterns and to allow the data to determine which are related to aspects of program provenance.

graphical patterns

Low-level techniques based around byte and instruction representations are often too sensitive to minor binary code variations such as specific registers used or instruction ordering, obscuring higher-level properties. Detecting salient changes between versions of a program (to analyze security patches, for example) can be hampered by irrelevant

low-level differences in the binary code, such as encoding of relative offsets. Flake [24] describes a comparison technique that uses control flow characteristics of functions to detect changes between two versions of the same program. Further extensions to this technique incorporate characteristics such as basic block size and some instruction-level information such as `call` instruction targets [22].

Structural characteristics of programs have been used to increase resilience to polymorphic malware. Kruegel et al. [46] introduce a malware fingerprinting approach based on *K-subgraphs* of program control flow graphs. The general approach is to select a connected subset of nodes ( )



whose adjacency matrix defines a fingerprint

$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} \rightarrow (0110\ 0000\ 0001\ 0000)$$

that is used to search for identical structures in new binaries. The fingerprints also incorporate node colors based on the instructions present in each basic block. Fingerprints based on these subgraphs were shown to have low error rates for detecting malware instances in network streams, both in terms of falsely identifying benign data as malware and in terms of confusing different malware instances. These results suggest that the fingerprints capture specific properties of a particular program, which may limit their utility for capturing general provenance properties; the authors note the fingerprints' rigidity as a possible limitation. Some of the *graphlet* features we describe in Chapter 5 are similar to these fingerprints, though they incorporate substantially more properties of the control flow graph; the way we apply graphlet features mitigates the generalizability problem.

Recent malware research has focused on higher-level structural characteristics, such as the interaction of programs with the operating system [6, 26, 28]. These approaches use dynamic analyses such as tracing to observe system calls at runtime, abstracting away most of the underlying code details. Such techniques are not generally suitable for modeling program provenance, which is usually intimately concerned with details of the code. However, representations that capture high-level properties of system interaction may be useful for provenance problems like authorship attribution. We use some *statically* derived properties of system interaction in our study of programmer style and attribution (Chapters 9 & 10); the dynamic techniques used in malware detection and clustering may be applicable but are outside the scope of this dissertation.

2.4 Authorship Attribution

Program authorship attribution has immediate implications for the security community, particularly in its potential to significantly impact applications like plagiarism detection [81] and digital forensics [64]. The central thesis of authorship attribution is that authors imbue their works with an individual style; while attribution research has historically focused on literary documents [40], computer programs are no less the product of a creative process, one in which opportunities for stylistic expression abound.

Previous studies of program authorship attribution have been limited to source code. Spafford and Weeber [90] introduced the topic with an exploration of possible characteristics of source code that could be used to identify its author, such as indentation and formatting style and variable naming conventions. Motivated by this discussion and early work on plagiarism detection [100], MacDonell et al. [56] designed a neural network-based system that learned to distinguish between programs written by a small set of authors with fair accuracy (80-85%). The code features used in this system are stylistic metrics similar to those suggested by Spafford and Weeber [90], such as proportion of operators with whitespace on both sides. Schleimer et al. [81] describe a document comparison algorithm based on character N-grams that can be used to detect source code plagiarism with a very low false positive rate, but this technique is designed to detect copying of specific code, rather than to capture the style of a particular author.

Hayes and Offutt [33] attempt to evaluate experimentally the thesis of authorship attribution, which they term the "consistent author hypothesis". The authors measured the variance of eleven facets of source programs elicited from five programmers. They conclude that several high-level characteristics, such as the average number of operators used in a program or the average number of unique language constructs, e.g. `for` and `while` loops, can be used to distinguish the five authors. However, the small size of

the experiment makes these results difficult to generalize from; moreover, the authors found no distinguishing facets in programs written by a larger group of fifteen graduate students.

Characterization of programmer style in source code relies on surface characteristics like spacing and variable naming, both of which reflect the essentially textual nature of program source. In many domains, such as analysis of commercial software or malware, source code is usually unavailable. Program binaries, however, retain none of the surface characteristics used in source code attribution; such details are stripped away in the compilation process. In addition, compilers can introduce potentially extensive structural changes during optimization and code generation, such as removing unused code or reordering code blocks [4]. Attribution for program binaries has remained an open problem. In Chapters 9 and 10 we show that individual programmer style is preserved in binary code, and introduce models that can accurately discriminate among programs written by distinct authors. Although this dissertation is concerned with the provenance of binary code, we also describe techniques based on our binary code models that can be applied to source code, yielding highly accurate authorship attribution.

2.5 SUMMARY

Prior program analysis work has explored many of the issues involved in representing and describing binary code, but has not emphasized features that are tailored toward investigating program provenance. Existing representations are insufficiently general—they are sensitive to variation that is irrelevent for particular provenance properties—and at the the same fail to capture high-level code characteristics that are necessary to reveal aspects of provenance like programmer style. In Chapter 5, we describe techniques that integrate existing and novel code representations and present methods to automatically tailor these representations to recover particular aspects of provenance.

3

Machine Learning Background

The techniques developed in this dissertation are heavily informed by statistical machine learning concepts. Recovering program provenance is at heart an *inference* problem, amenable to a wide variety of statistical modeling approaches. Although the focus of this dissertation is on applications rather than on the issues surrounding learning and inference algorithms, some familiarity with machine learning concepts is necessary. Our hope is for this chapter to provide an overview of these concepts so that the reader can better understand the methodology and experimental results that we present in this dissertation.

Machine learning, broadly speaking, is an area of study that focuses on how a program's performance on some task can be automatically improved with experience [60]. In the case of statistical machine learning, the 'experience' comes in the form of the statistical properties, or *features*, of data; statistical machine learning is concerned with making quantitative predictions about data based on these features [32]. In the following sections we introduce they key ideas that are necessary to understand how machine learning techniques are applied, focusing on how such problems are formulated, different approaches to the learning problem, and *feature selection*, a solution to several issues that arise in learning applications. We conclude with a more detailed overview of two models that are used frequently in this dissertation.

3.1 DESCRIBING THE PROBLEM

Machine learning problems are often described in terms of making predictions about *random variables*: given input variables (or *evidence*) X, the goal is to predict the value of an output variable Y. The evidence is made up of *features* that describe the data; features could be qualitative properties (e.g., the color or shape of an object), or quantitative measurements (e.g., a person's height or weight). In general, a feature is just some arbitrary property of the data in question, which we represent with *feature*

functions $f : x \to \mathbb{R}$. For example, the function

$$f_{red}(x) = \begin{cases} 1 & \text{if } x \text{ is red} \\ 0 & \text{otherwise} \end{cases}$$

could be one of a great many features functions that describe a colored ball; the function

$$f_{ecnt}(x) = \sum_{\ell \in \text{LETTERS}(x)} \mathbb{1}_{[\ell = \text{'e'}]}$$

is a feature function that represents the number of the times that the letter 'e' occurs in this sentence (28).

A convenient convention is to define the evidence X to be a vector of d random variables $X_1, \ldots, X_d$, and to let the value of each variable be a *feature vector*

$$\mathbf{x} = \begin{pmatrix} f_1(x) \\ f_2(x) \\ \vdots \\ f_d(x) \end{pmatrix}.$$

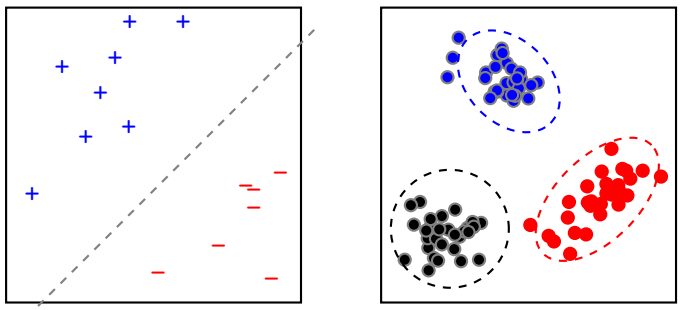
In other words, each element x_i of the feature vector is the value of the i^{th} feature function applied to x. Data can be thus described as points in a d-dimensional *feature space*.

The output variable Y is likewise a real-valued random variable, and is assumed to be a function of the inputs (plus possibly some random noise ϵ), i.e.,

$$Y = f(X) + \epsilon. \tag{3.1}$$

When the output is a quantitative measure of the inputs, the prediction problem is called a *regression problem*. In this dissertation, we are more interested in the case of qualitative (i.e., discrete or categorical) output labels; these define a *classification problem*.

Classification problems center on predicting the value of *labels* $Y \in \mathcal{Y}$ based on the features of the evidence X. This task can be thought of in terms of the conditional distribution $\Pr(Y|X)$. The function $f(X)$ in (3.1) is $\mathrm{E}(Y|X = x)$, the expected value of this distribution given specific inputs; if $f(x)$ can be computed, then the output label for a data point can be generated. Of course, the entire point of machine learning is that the true function $f(x)$ is not known *a priori*; the learning problem is to find some function $\hat{f}(x)$ that approximates the true distribution $\Pr(Y|X)$.



(a) classification (b) clustering

Figure 3.1: Examples of *classification*, a supervised learning problem, and *clustering*, an unsupervised problem. In classification (a), the objective is to assign *labels* (e.g. +,−) from a known set of possibilities to the data, for example by partitioning the feature space. In clustering (b), the objective is to group together data points that are nearby in the feature space.

3.2 SUPERVISED AND UNSUPERVISED LEARNING

Machine learning problems can be roughly divided into two categories, illustrated by Figure 3.1. The first are *supervised* learning problems, which correspond to the task that we described in the previous section: discovering the properties of the relationship $\Pr(Y|X)$ between the input and output variables. Supervised problems are so-called because they involve a *training* process that uses *examples* of input and output variables to learn how they are related. The second type of problems are *unsupervised* learning problems, which are distinguished by the lack of differentiation between the variables; the goal of unsupervised learning is to discover the properties of the distribution $\Pr(X)$, without resorting to labeled training data. Our approaches to provenance recovery consist of classification, a supervised learning problem, and unsupervised *clustering*.

3.2.1 CLASSIFICATION

As described in the previous section, classification problems seek to form a predictive model of *label* output variables given the input features by finding an approximation $\hat{f}(X)$ that models the underlying conditional distribution. The approximation is usually defined in terms of a set of parameters θ; given N training examples (x_i, y_i), learning can be thought of as an optimization over the parameter space that minimizes an

objective function that relates the true labels y_i to predicted labels $\hat{f}(x_i)$, i.e.

$$\underset{\theta}{\operatorname{argmin}}\, L\left(y_i, \hat{f}(x_i)\right),$$

where the objective is defined by a *loss function* $L(\cdot)$ and by the form of $\hat{f}$.[1] The loss function measures errors between the predicted and true labels, for example the zero-one loss $\mathbb{1}_{\left[y \neq \hat{f}(x)\right]}$; by minimizing the loss, an algorithm can find a set of parameters such that the approximation captures the true relationship between input variables and output labels, as observed in the data set. The approximate model and its parameters are called a *classifier*.

More detailed discussion of the general problem of supervised learning strays into the territory of statistical decision theory and is beyond the scope of this dissertation. Hastie et al. [32] provide a good introduction for the interested reader. For the purposes of this chapter, what is important to understand is that classifiers are trained by estimating parameters that fit a model to training data. We construct classifiers based on two models, *support vector machines* [18] and *conditional random fields* [50], which we describe in Chapter 6 and in the experimental part of this dissertation. We also describe these models in more detail in Section 3.4; this section may offer additional insight into later material, but is not required for understanding.

3.2.2 Clustering

Clustering is an unsupervised learning problem in which the goal is to group together data that are similar to one another based on their features. More precisely, clustering algorithms partition N data points $\mathbf{x}_1, \ldots, \mathbf{x}_n$ into k subsets $\{S_1, \cdots, S_k\}$ according to some function of their distance from one another in the feature space. Unlike classification, there are no label variables that we want to predict, and much of the following discussion on model training is inapplicable. We defer further details to Chapter 10, where we describe a clustering application that groups binary programs by programmer style.

[1] This objective corresponds to the *empirical risk*; practical machine learning systems usually include an additional regularization term $\Omega(\theta)$ that penalizes model complexity to avoid overfitting. The need for regularization is a practical consequence of the *bias-variance tradeoff*; for more information, refer to § 2.9 of Hastie et al. [32].

3.3 FEATURE SELECTION

As discussed above, supervised machine learning problems involve a training process by which the model parameters are estimated based on a *training set* of observations

$$\mathcal{T} = (\mathbf{x}_1, y_1), \cdots, (\mathbf{x}_N, y_N),$$

where $\mathbf{x}_i$ are data points in a d-dimensional feature space. It is often the case that learning a model that incorporates the entire feature space is not the best option, either because the feature space is so large that computational or storage overheads are excessive, or because doing so can lead to *overfitting*. Overfitting occurs when a model's parameters reflect some random property of the training data and not the output variable [32]; it is often a sign of excessive *model complexity*, measured in terms of features and model parameters. Models that overfit the training data will fail to generalize to new data points, leading to poor label predictions. Overfitting can be mitigated by penalizing model complexity in the learning algorithm's objective function; alternatively, the number of features (and thus model parameters) can be reduced through *feature selection*.

Feature selection is a procedure by which some subset d' of the d-dimensional feature space is retained, and the rest of the features are thrown out (usually $d' \ll d$). There are a variety of strategies for feature selection. In this dissertation, we employ *forward feature selection*, a greedy, agglomerative technique that iteratively builds up a set of features based on the feature that most improves classifier performance at each iteration, and a heuristic approach based on *mutual information*.

3.3.1 FORWARD FEATURE SELECTION

Forward feature selection iterates over a set of candidate features, evaluating how much each one would improve the performance of the classifier. We assume a d-dimensional feature space with features $f_1, \cdots, f_d$ and let S and R be the indices of selected and remaining features, respectively. Given a set of m data points $X = \mathbf{x}_1, \cdots, \mathbf{x}_m$ with labels $Y = y_1, \cdots, y_m$, let the data using only the the selected features be written X_S. The forward feature selection procedure is as follows:

```
S ← ∅, R ← {1, ···, d}
repeat
    max ← 0
    best ← null
    for i = 1 to |R| do
        T ← S ∪ r_i
        eval ← CVEval(X_S, Y)
        if eval > max then
            best ← i
            max ← eval
    S ← S ∪ best
    R ← R ∖ best
until termination condition met
```

The CVEval(X_S, Y) method trains and evaluates a classifier using the selected features using cross validation, returning the chosen evaluation criterion (e.g. accuracy). There are several options for when to terminate the feature selection procedure, such as when the improvement for the evaluation criterion between two iterations falls below a threshold (as we do in this dissertation), or when decreasing performance on a held-out *tuning* set indicates overfitting.

Forward feature selection may not discover the optimal combination of features; it is a greedy hill-climbing technique that does not evaluate every possible combination of features. What it lacks in optimality it makes up for in expediency: as we show in Chapter 7, forward feature selection is parallelizable and can be performed in a reasonable amount of time even for data sets with hundreds of thousands of features.

3.3.2 Mutual Information

While the forward feature selection process is tractable, it can be slow, especially if the feature space is very large or if a sizable fraction of the feature space is useful (i.e., contributes information about target variable). In such cases, we use a feature selection approach based on the mutual information between features and class labels.

Let X and Y be discrete random variables. The mutual information between X and Y is defined to be

$$I(X, Y) = \sum_{x \in X} \sum_{y \in Y} p(x, y) \log \left(\frac{p(x, y)}{p(x)p(y)} \right), \tag{3.2}$$

where $p(x)$ and $p(y)$ are the probability distributions of each random variable and $p(x, y)$ is the joint probability distribution of X and Y. Mutual information measures

how much the uncertainty about the value of one random variable is reduced by knowing the value of another.

In this dissertation, mutual information is used to describe the relationship between binary code features and provenance class labels. Let the provenance label of a binary code artifact (e.g., a program) be the Y random variable and let a single binary code feature be the X random variable. We compute the mutual information between class and feature by computing the empirical joint and marginal distributions over the M binary code artifacts in the data set

$$p(x) = \frac{1}{M}\sum_{i=1}^{M} \mathbb{1}_{[x_i=x]} \quad p(y) = \frac{1}{M}\sum_{i=1}^{M} \mathbb{1}_{[y_i=y]} \quad p(x,y) = \frac{1}{M}\sum_{i=1}^{M} \mathbb{1}_{[x_i=x \wedge y_i=y]}.$$

Plugging the empirical distributions into (3.2), we can compute the mutual information between features and provenance properties and use it as a simple feature selection technique by using only the top k features ranked by mutual information in our models. Unlike forward feature selection, this is a heuristic method; it does not seek a set of features that directly optimize the learning problem's objective function, and it is not capable of measuring the contribution of two or more features taken together. However, feature selection by mutual information is inexpensive to compute, and we use it frequently for provenance modeling.

3.4 CLASSIFIER MODEL DETAILS

We use two types of model for classification in this dissertation: *conditional random fields* [50] and *support vector machines* [18]. The following sections provide details that may be helpful in interpreting the provenance models that we describe in Chapter 6 and in our experiments.

3.4.1 CONDITIONAL RANDOM FIELDS

The majority of the provenance classification models that we use in this dissertation are based on *conditional random fields* (CRFs), a type of *probabilistic graphical model*. Probabilistic graphical models are a graph-based representation of probability distributions [43], as illustrated in Figure 3.2. The nodes in a graphical model correspond to random variables, and the edges indicate statistical dependencies. The connectedness of a graphical model indicates the factorization of the distribution: the distribution can be defined as the product of functions over fully-connected node *cliques*. Any distribution can be described as a graphical model; even if all of the variables are independent, the model is simply a graph with no edges. Graphical models are a convenient representa-

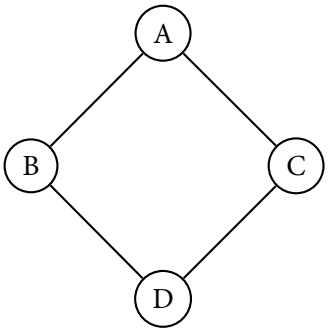


Figure 3.2: An undirected probabilistic graphical model with four variables. An edge between two nodes reflects conditional dependence. The probability distribution defined by this graph factorizes as $p(A, B, C, D) = \frac{1}{Z}\Psi_1(A, B)\Psi_2(A, C)\Psi_3(B, D)\Psi_4(C, D)$, where Ψ are functions over the variables and Z is a normalization term.

tion for complex distributions, and provide a structure for inference algorithms that compute properties of the distribution.

More formally, let a graphical model be a collection of vertices and edges $G = (V, E)$, where the vertices are the variables $X \cup Y$. Given a set of cliques $A \subset V$, the distribution over the variables can be written

$$p(Y, X) = \frac{1}{Z} \prod_A \Psi_A(X_A, Y_A), \tag{3.3}$$

where $\{\Psi_A\}$ are *factors* mapping from the variables in a clique to $\mathbb{R}^+$, and Z is a normalization term

$$Z = \sum_{X,Y} \prod_A \Psi_A(X_A, Y_A)$$

that makes this expression a distribution.

Conditional random fields are a specific type of graphical model, where the factors Ψ_A are defined in terms of a set of feature functions f_A and parameters λ_A

$$\Psi_A(X_A, Y_A) = \exp\{\lambda_{Ak} f_A(X_A, Y_A)\},$$

resulting in a conditional probability distribution of the form

$$p(Y|X) = \frac{1}{Z(x)} \prod_{\Psi_A \in G} \exp\{\lambda_A f_A(X_A, Y_A)\}, \tag{3.4}$$

where, importantly, the normalization term is conditioned on the evidence, i.e.

$$Z(x) = \sum_{Y} \prod_{A} \Psi_A(x_A, Y_A).$$

Equation 3.4 is just a refinement of the general graphical model where we specified the form of the factors. In practice, the model is typically further broken down in terms of sets of cliques with *tied parameters*. Consider the CRF depicted in Figure 3.3, which is a specialization called a *linear-chain conditional random field*. The edges between the Y variables indicate dependencies between adjacent variables; there is a factor $\Psi_{i,i+1}$ for every adjacent pair. If we are interested in learning the model parameters that determine whether, e.g., $y_i = y_{i+1}$ or $y_i \neq y_{i+1}$ is more likely, it is more practical to estimate a single parameter for all of the adjacency factors, rather than $n-1$ parameters (one for each factor) [91].

Let $\mathcal{C} = \{C_1, \ldots, C_p\}$ be a partition the graphical model G, and let each factor $\Psi_c \in C_p$ be defined over both the variables and a set of partition-specific parameters Λ_p, i.e.

$$\Psi_c(X_c, Y_c, \Lambda_p) = \exp\left\{ \sum_{k=1}^{K(p)} \lambda_{pk} f_{pk}(X_c, Y_c) \right\}.$$

Expanding (3.4) with this factorization, CRFs are defined as

$$p(Y|X) = \frac{1}{Z} \prod_{C_p \in \mathcal{C}} \prod_{\Psi_c \in C_p} \Psi_c(X_c, Y_c; \Lambda_p), \tag{3.5}$$

where

$$Z = \sum_{Y'} \prod_{C_p \in \mathcal{C}} \prod_{\Psi_A \in C_p} \Psi_c(X_c, Y_c'; \Lambda_p)$$

again normalizes the distribution. In this model, the parameters Λ are divided into p groups that are associated with subsets of factors; parameter estimation in such a model will learn the same parameters for each factor in a group C. For example, in the linear-chain CRF of Figure 3.3, there are two groups of factors: one for the factors corresponding to edges between Y variables, and one for the edges between Y and X variables. The linear-chain CRF has the form

$$p(Y|X, \Lambda_a, \Lambda_b) = \frac{1}{Z} \exp\left\{ \sum_{k=1}^{K(a)} \lambda_{ak} f_{ak}(y_t, y_{t-1}) + \sum_{k=1}^{K(b)} \lambda_{bk} f_{bk}(y_t, X_t) \right\}.$$

Conditional random fields are useful for program provenance modeling because they allow us to incorporate arbitrary feature functions that describe properties of our

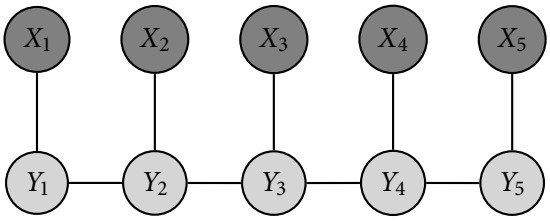


Figure 3.3: A linear-chain *conditional random field* with feature variables (●) and label variables (○). Each label is associated with its own feature variable; the labels are linked by edges indicating first-order Markov dependencies.

data. Inference and parameter estimation for these models can be expensive, however, depending on the model structure. For linear- and tree-structured graphical models, efficient inference algorithms are known; for loopy graphs, however, exact inference is intractable in general and approximate methods are required [91].

3.4.2 Support vector machines

Support vector machines (SVMs) are a type of *maximum margin* classifier: they seek to find an optimal d-1–dimensional hyperplane in $\mathbb{R}^d$ that maximally separates data points of two different classes, as depicted in Figure 3.4. A linear, two-class SVM is usually formulated with labels $y \in \{-1, +1\}$, and is parameterized by a weight vector $\mathbf{w}$ that solves the following optimization problem:

$$\min_{\mathbf{w},\xi,b} \frac{1}{2} \parallel \mathbf{w} \parallel^2 + C \sum_i^n \xi_i \qquad \text{s.t.} \quad y_i(\mathbf{w}^T\mathbf{x} - b) \geq 1 - \xi_i, \quad \xi_i \geq 0 \forall\, i = 1 \ldots n$$

The *slack variables* ξ allow for solutions even when there is no separating hyperplane, as depicted in Figure 3.5. Two-class SVMs can be easily extended to the case of K classes by training K different binary classifiers with weight vectors $\mathbf{w}_1, \cdots, \mathbf{w}_K$; the classifier assigns a new example the label $k \in [1, K]$ that leads to the largest margin, i.e.

$$\operatorname*{argmax}_k \mathbf{w}_k^T \mathbf{x}.$$

In Chapters 8 and 9, we use linear support vector machines for provenance recovery. Linear SVMs are adequate for our problem because the provenance properties we are interested in are roughly linearly separable in the binary code feature space that we introduce in this dissertation. Much of the power of support vector machines stems

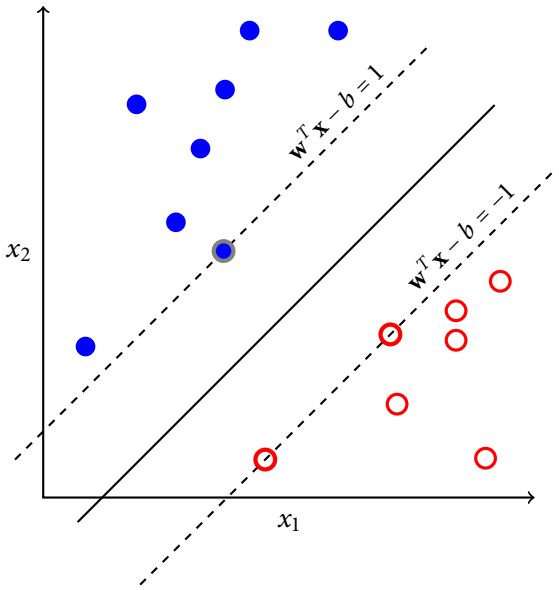


Figure 3.4: The maximum margin hyperplane (—) and margins (- -) for a two-class SVM. The points that lie on the margins are the *support vectors*.

from their use of *kernels* that enlarge the feature space and can separate otherwise inseparable data. Hastie et al. [32] provide a thorough introduction to non-linear SVMs.

3.5 SUMMARY

Machine learning, for the purposes of this dissertation, can be thought of as a way to learn a mapping between arbitrary attributes of a thing and some property that we are interested in: the shape and color of a mole and cancer risk, for example; the proportion of cloud cover and the chance of rain; or the occurrence of particular binary code constructs and the provenance of that program. Supervised machine learning techniques underlie much of our provenance recovery work. The concepts and notation that we introduced in this chapter will help to understand the provenance models that we introduce in Chapter 6, and the methodology that we use in our experiments.

Our discussion has been necessarily incomplete; a thorough description of even the few concepts that we described above is well beyond the scope of this dissertation. For further details on statistical decision theory and machine learning, Hastie et al. [32] provide an invaluable resource; Koller and Friedman [43] give a comprehensive

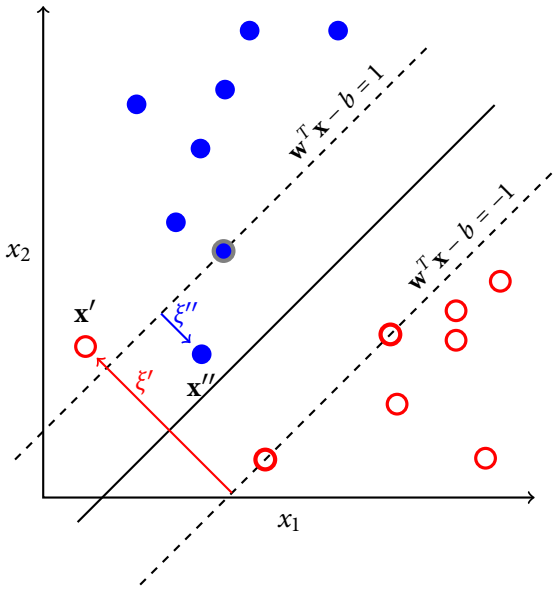


Figure 3.5: Slack variables allow solutions to the SVM optimization problem when no separating hyperplane exists. Here, point $\mathbf{x}'$ is misclassified, though $\mathbf{x}''$ is not.

treatment of probabilistic graphical models.

As we state in the introduction, the focus of this dissertation is on the high-level, data-driven approach to provenance recovery, not on the specific modeling and classification techniques that underlie our system. We chose conditional random fields because they ease the integration of multiple forms of knowledge (like the program structural features that we describe in Chapter 8), and because they have been shown to perform well on a variety of learning problems, from text processing [69, 86] to computer vision [34]. Support vector machines have also been widely adopted for classification applications; the tutorial by Burges [11] provides an overview. Investigating alternative modeling techniques may be of interest for incremental improvements of provenance recovery.

4

Overview of Provenance Recovery

Program provenance recovery amounts to establishing a mapping between characteristics of binary code and the details of the process though which the code was produced. There are potentially many methods by which one could arrive at such mapping: by careful, painstaking examination of binary code; by exhaustive study of the various compiler toolchain components; or perhaps by intuitive leap. In this dissertation, however, we adopt a pragmatic approach based on statistical machine learning. By turning provenance recovery into a machine learning problem, we bring to bear powerful statistical modeling techniques with which to discover the relationship between binary code and the properties of its provenance. The story of this dissertation is not only that of how we arrive at this relationship, but also of the nature of program provenance—the *provenance hierarchy*—and the characteristics that encode that provenance in program binaries.

Machine learning is, in essence, a way of automatically establishing a model that describes the relationship between a set of variables. In the case of provenance recovery, these variables come in two classes: a program's *provenance properties*, or its path through the provenance hierarchy, and the *code characteristics* that describe program binaries. The "learning" comes in when we estimate the *parameters* of this model so that it can be used to answer queries about one or more variables. The elements of the provenance recovery problem are depicted in Figure 4.1. The parameters of provenance models are learned in a *training* process that uses observed code characteristics and provenance properties; the application of the learned parameters to infer the provenance of novel binary code is called, appropriately, *inference*.

In this chapter we provide an overview of each element of this learning framework, beginning with a description of the provenance hierarchy that establishes the scope of our research. We then outline the issues of program representation and of developing provenance models, which are developed respectively in Chapters 5 and 6.

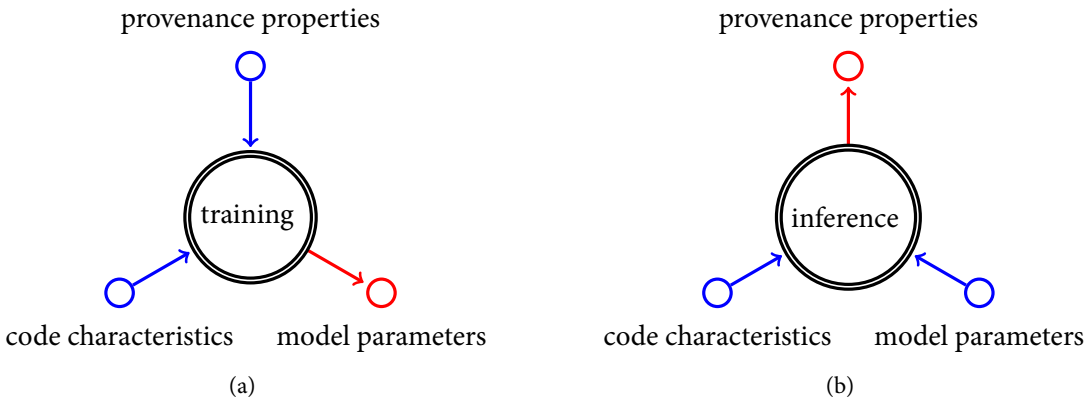


Figure 4.1: Relationship of the components of provenance recovery.

4.1 The Provenance Hierarchy

Program binaries are created through a process, a series of stages wherein an idea is instantiated and transformed into machine-interpretable code: the programmer selects a language, writes some code, chooses a compiler, sets build options, and so forth. Because the output of each stage determines the input to the one that follows it, we view provenance hierarchically: the possible decisions form a branching tree, and a program's eventual form is determined by its path through that tree. Exactly how to characterize the provenance hierarchy is somewhat arbitrary; for this dissertation, we establish the following levels:

1. *Authorship:* the identity and stylistic characteristics of the individual individuals who wrote the program,

2. *Functionality:* the algorithms and implementation techniques realized by the program,

3. *Language:* the programming language and APIs used to write the program,

4. *Environment:* the build and target environments, including the operating system properties, system libraries, and machine architecture, and

5. *Toolchain:* the toolchain components, such as the compiler, linker, or post-compilation tools.

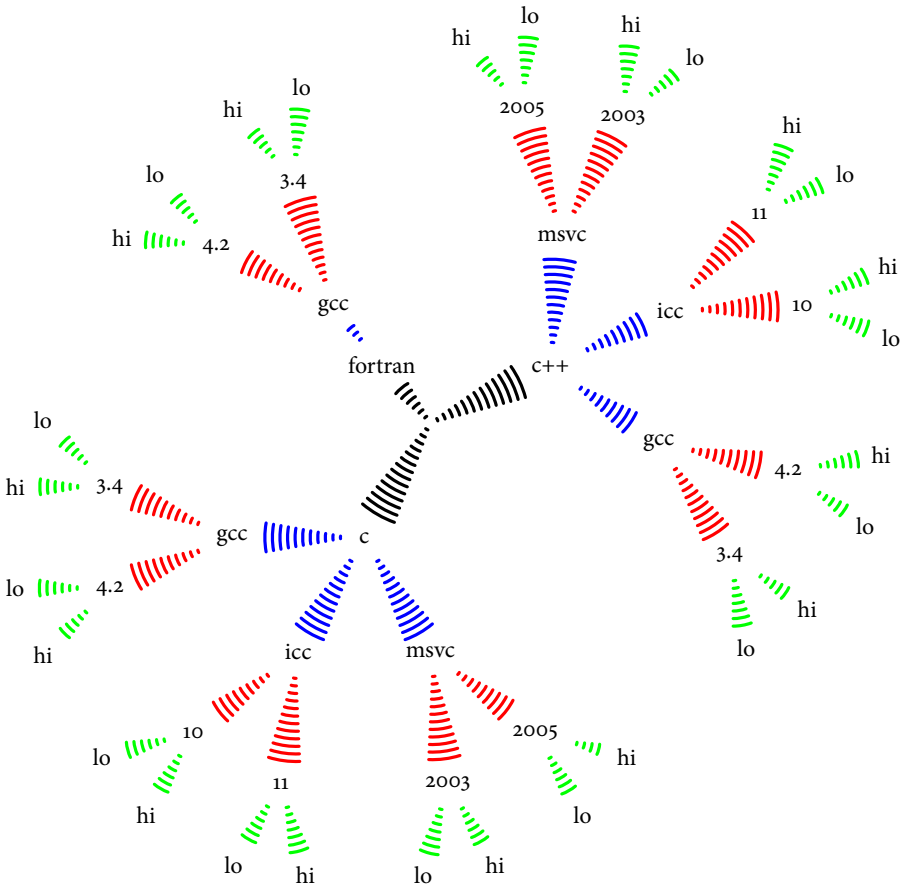


Figure 4.2: An example of a partial provenance hierarchy similar to one that we have used in toolchain experiments [77]. The concrete values of the source language, compiler family, compiler version, and optimization level are depicted. Note that particular provenance values are duplicated across subtrees of the hierarchy; the GCC compiler applies to all three languages and all programs can be compiled with low or high optimization.

The possible choices at each level of the provenance hierarchy define the provenance properties component of provenance recovery. The focus of this dissertation has been on the authorship, language, and toolchain levels, which we explore with a series of experiments in Chapters 7–10. Figure 4.2 depicts an example of concrete provenance properties that might make up the language and toolchain portions of the hierarchy for a collection of programs.

4.2 Program Representations

The other side of the provenance modeling relationship comprises the details of the binary code that encode the program's path through the provenance hierarchy. As we discussed in Chapter 2, there are many ways of representing binary code and of extracting detailed code properties. For example, the same code can be viewed as a sequence of bytes, a series of machine instructions, or as a control flow graph, as depicted in Figure 4.3. Different abstractions may be more or less suitable for revealing the mapping between code characteristics and program provenance.

Our intuition is that higher-level abstractions (such as control flow graphs) are more suitable for capturing the properties of higher levels of the provenance hierarchy: it seems reasonable to expect characteristics of a program's control flow to be more indicative of programmer style, for example, than would be the distribution of arithmetic instructions. This intuition has not always been borne out by experience, however [78]. What makes our machine learning–based approach pragmatic is that we need not choose the best program characteristics *a priori*, but only to define large classes of characteristics that capture provenance when taken as a whole.

The code characteristics that we use for provenance recovery are described in detail in Chapter 5. From a machine learning perspective, these characteristics define a high-dimensional *feature space* in which we construct provenance models. A program or other binary code element can be thought of as a point in that space; the mapping problem that is provenance recovery revolves around the relationship of these points to the values of properties of the provenance hierarchy.

4.3 Learning the Mapping

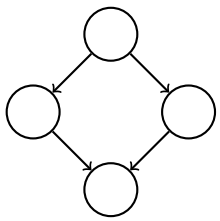
The preceding discussions of the provenance hierarchy and program representations are intrinsic to the provenance recovery problem, regardless of how it is solved; the machine learning aspect of our approach comes up in how we relate these two elements to one another. This task, which we describe in Chapter 6, involves defining provenance models and estimating their parameters. We approach these issues in two different ways,

```
55 89 e5 83 ec
04 83 7d 08 01
7e 09 c7 45 fc
01 00 00 00 eb
07 c7 45 fc ff
ff ff ff 8b 45
fc c9 c3
```

(a)

```
  push   %ebp
  mov    %esp,%ebp
  sub    $0x4,%esp
  cmpl   $0x1,0x8(%ebp)
  jle    L1
  movl   $0x1,-0x4(%ebp)
  jmp    L2
L1:
  movl   -$0x1,-0x4(%ebp)
L2:
  mov    -0x4(%ebp),%eax
  leave
  ret
```

(b)



(c)

Figure 4.3: Three views of a simple binary code artifact. Different types of code characteristics can be extracted from each view, for example register operands (b) or the shape of the control flow graph (c).

depending on whether we make a *closed* or *open world* assumption: that is, whether or not we can assume that the possible values of the provenance properties in question are known *a priori*.

The typical problem we are trying to solve in provenance recovery is to identify the specific provenance properties of a binary—for example, in order to determine whether a buggy compiler produced some code—out of a known set of possibilities (the *closed world* assumption). This task can be characterized as a *classification* problem: the provenance properties (e.g., compiler versions) are classes, and the provenance modeling problem involves finding a mapping between *features* and the *class labels* (e.g., GCC 4.4 or MSVC 2005). Classification is a *supervised* learning problem, where model parameters are estimated from *training* data. There are many different ways to define classifiers—Chapter 3 provides an overview—but our emphasis has been on the use of *conditional random fields* [50] and *support vector machines* [18] as, respectively, probabilistic and non-probabilistic classifiers. Regardless of the specific algorithm used, classification techniques share a set of common problems:

Model definition Provenance models are defined by the code representation (e.g., byte- or control flow–level) from which we generate features and the granularity to which we assign provenance; in the chapters ahead, we describe experiments in which we model the provenance of individual bytes all the way up to entire

programs. Depending on the type of model we use, we may also incorporate structural properties of the program like the layout of code in the binary.

Feature selection The features with which we describe code elements ultimately determine how well our models capture provenance. Many binary code features may be redundant, and can add needless complexity or even adversely impact model performance. We describe a simple, heuristic feature selection approach in Chapter 5 that allows us to inexpensively prune the feature space prior to model training.

Parameter estimation At heart, machine learning boils down to an optimization problem. The model definition specifies an *objective function* that relates provenance labels, features, and parameters. The learning problem is to explore the parameter space in order to minimize or maximize that objective function for some *training data* for which both features and provenance labels are known. For example, the *weight* parameters of support vector machines are trained by minimizing the weight vector subject to constraints that enforce class separation in the feature space (refer to Chapter 3 for details). The choice of training data is crucial: only relationships between provenance properties and code characteristics that appear in the training data can be learned.

The classification approach to provenance recovery is a fundamentally closed world approach. There are many problems for which this assumption is inappropriate; for example, security analysts may want to group stylistically similar malware programs together, despite not knowing who the possible authors are. For these types of *open world* provenance problems, and when training data are otherwise not readily available, we rely on *unsupervised* learning techniques like *clustering*.

Unlike classification, which assigns provenance labels to binary code, clustering techniques compute the similarity between code based on its features. Clustering techniques have model definition and feature selection requirements similar to those of classifiers, but introduce a new and substantially harder problem: without training data, an unsupervised clustering algorithm may find similarities for the wrong binary code property, for example grouping programs based on whether they use floating point math rather than similar programmer style. A variety of *constrained clustering* methods have been developed to address this problem [5]; in Chapters 6 and 10 we describe an approach based on learning *distance metrics* and *transfer learning* to compensate for the lack of training data [99].

4.4 SUMMARY

We have established a framework for provenance recovery that is based on a hierarchical representation of stages in the production process and on a mapping between those stages and concrete characteristics of binary code at several levels of abstraction. We adopt a machine learning approach to this problem, automatically learning the relationship between provenance and code characteristics by specifying classification models and estimating their parameters through the use of training data, or by using unsupervised clustering approaches. In the following chapters, we describe the code representations and modeling techniques we use to perform provenance recovery.

5

Representing Program Provenance

This chapter focuses on representing and extracting the code features used to model provenance. Our techniques for provenance recovery are highly dependent on how we extract features; like most techniques based on machine learning, good feature engineering determines to a large degree the success of our approach. In this chapter, we give an overview of our approach to designing code features and describe both our features and the extraction process.

5.1 DESIGNING CODE FEATURES

For provenance recovery, a good feature is one that is characteristic of a particular element in the provenance hierarchy. For example, one might expect different compilers to generate systematically different instruction sequences for a particular source-level programming idiom; instruction patterns that reflect these differences would make good features for discriminating among possible compilers. The design space for code features is large, spanning the code representations described in Chapter 2. Feature design involves two main subproblems: choosing code details that are likely to reflect a particular provenance property, and evaluating how well the features are likely to perform on a particular data set.

The chief difficulty in feature design is the first subproblem: it is often unclear, *a priori*, how a provenance property will be reflected in the code. The power of our machine learning–based approach is the way in which it sidesteps this problem. As we describe in the following sections, we define broad feature categories or *templates* that may (but are not guaranteed to) capture characteristics of various stages of the provenance hierarchy; the model training process automatically determines which features are actually useful. This approach is central to our "don't try to be clever" philosophy.

There is an essential tension, however, between the inclination to try every possible code feature and the performance of the provenance recovery system. At the very least,

incorporating an excessive number of features may introduce scalability problems in terms of training or inference cost; a large number of redundant or overly expressive features may also increase the risk of modeling failures like overfitting. One possible way to evaluate different feature choices is to implement and experiment with various combinations for a provenance recovery application. This approach can be expensive, particularly for exploring a large number of possible features. We use an alternative criterion for evaluating feature designs based on *mutual information* [32].

Mutual information measures how much uncertainty about the value of one random variable is reduced by knowing the value of another. For evaluating candidate code features, mutual information can be thought of as measuring both positive and negative correlation of particular features and provenance properties. For example, if we are interested in *source language provenance* and a particular feature occurs frequently in programs compiled from C++ code but never in Fortran and only rarely in C, then the feature and the source language property will have high mutual information. On the other hand, if a feature is observed uniformly often in a given data set, then it has low mutual information with provenance properties. More precisely, given the set of values for a particular feature Φ and a set of provenance property values $\mathcal{Y}$, the mutual information between the feature and provenance property values is defined to be

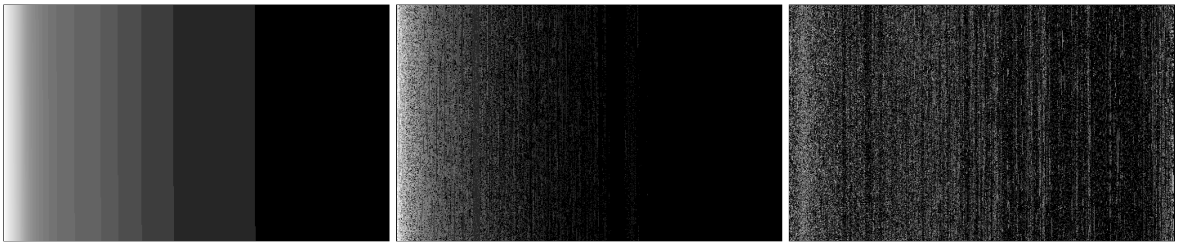
$$I(\Phi, \mathcal{Y}) = \sum_{f \in \Phi} \sum_{y \in \mathcal{Y}} p(f, y) \log \left(\frac{p(f, y)}{p(f)p(y)} \right),$$

where $p(f)$ and $p(y)$ are the empirically estimated probabilities of feature values and provenance properties, respectively, and $p(f, y)$ is the probability of co-occurrence of these variables. In each of the following sections, we evaluate classes of features in terms of their mutual information with several data sets reflecting several different provenance properties.

5.2 N-grams of bytes

Of all of the features we use to capture provenance details, those that directly capture pattens at the byte level rely on the fewest assumptions about the how program provenance is reflected in the code. As we describe in Chapter 2, the program bytes comprise a mixture of different information, including the machine instructions, data reference offsets, program data, and padding or random junk values. Despite being almost trivially simple, byte N-grams are a natural choice of code feature: a program's bytes are necessarily dependent on its path through the provenance hierarchy.

We define byte N-grams to be an N-byte sequence in a program or other binary



(a) version (b) family (c) authorship

Figure 5.1: The distribution of provenance–feature mutual information (MI) for N-grams on data sets reflecting (a) compiler version, (b) compiler family, and (c) programmer style. Each pixel represents an N-gram; the intensity (white–black) reflects the mutual information for that feature, normalized over the particular data set (1–0). Each N-gram occupies the same position in each subfigure; features are ranked by their value on the compiler version data set (a). This figure shows that the feature MI values for the compiler version and family data sets are recognizably similar, with many of the same features having high mutual information; by contrast, there is little overlap between important features on the authorship and compiler data sets.

code object $\mathcal{P}$, i.e.

$$\text{N-gram}(a, \mathcal{P}) = \mathcal{P}[a, a + N - 1],$$

where a is an offset within the binary code object. Often we want to describe not only a particular offset in a binary, but a larger region of code such as a basic block or a function. N-gram features extend naturally to contiguous regions of bytes: the region is said to *contain* all length-N sequences of bytes in the program between starting offset s and ending offset e, inclusive:

$$\text{N-grams}(s, e, \mathcal{P}) = \bigcup_{a=s}^{e-N} \{\mathcal{P}[a, a + N - 1]\}.$$

We frequently use this notion of containment to describe high-level code constructs (such as functions) using low-level, local features, as we discuss in in the following chapter.

Particular N-gram features provide can greater or lesser amounts of information about provenance, depending on the data set. Figure 5.1 depicts the distribution of mutual information between byte N-grams and, respectively, *compiler version*, *compiler family*, and *authorship* properties for three data sets that we describe in more detail in Chapters 8 and 9. The figures were created by ordering N-grams by decreasing mutual

information on the compiler version data set (a) and creating a matrix by folding the data into columns. Each point in the figures corresponds to a single feature. Features with higher mutual information have higher intensity. These figures are meant to provide the reader with a qualitative sense for how the importance of features varies with respect to particular provenance properties, not to be a quantitative assesment of feature quality. We provide similar comparisons based on the same data sets for several other feature types, below.

5.3 Instruction Idioms

Byte N-grams are an extremely low-level representation of binary code, describing at most a few bytes of the program. By contrast, an instruction-level representation organizes the underlying bytes into semantically meaningful units that can represent somewhat larger regions of the binary; in the Intel IA-32 instruction set, a single instruction can span up to fifteen bytes [38]. We have developed instruction-based features, which we call *idioms*, designed to capture provenance characteristics that are reflected in the instruction-level representation. Idioms are short sequences of instructions, possibly with wildcards, that abstract away some of the byte-level details of the instruction set. Idioms have the following properties:

1. specific opcodes are collapsed into related *mnemonics*,
2. the values of immediate operands are abstracted away,
3. memory references are represented as a single catch-all token, and
4. wildcard tokens (`*`) match a single instruction.

To be precise, idioms are tuples of instruction mnemonics and operands specified by the grammar in Figure 5.2. There are some arbitrary and instruction set–specific details in how we choose to collapse instructions into mnemonics and in how immediates are elided (for example, in the Intel x86 instruction set some interrupt-generating instructions implicitly encode the interrupt number in the opcode). The specific rules we use to generate idioms have work well for our applications, but there are certainly other approaches that we did not explore.

We designed idiom features to flexibly tolerate minor code variations that would otherwise mask regularities in a provenance property. For example, the idiom

$$\langle \texttt{push ebp} \mid \texttt{mov esp,ebp} \mid \texttt{sub [imm],esp} \rangle$$

matches both the x86 instruction sequence

```
IDIOM   ::= <INSN-W> , <INSN> | <INSN>
INSN-W  ::= <INSN> | <INSN>,"*" | "*",<INSN> | "*","*"
INSN    ::= <MNEMONIC> | <MNEMONIC> <OPS>
OPS     ::= <OP> | <OP>,<OP>
OP      ::= <REG> | "[IMM]" | "[MEM]"
```

Figure 5.2: A grammar for idiom features in Backus Naur Form [42]. The special symbol `MNEMONIC` represents the set of instruction mnemonics (e.g., `mov` or `jmp`); `REG` is the set of architectural registers.

```
55                  push   %ebp
89 e5               mov    %esp,%ebp
83 ec 18            sub    $0x18,%esp
```

and the sequence

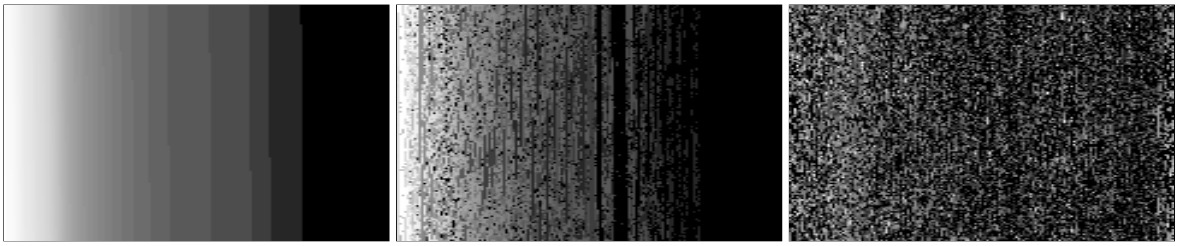
```
55                  push   %ebp
89 e5               mov    %esp,%ebp
83 ec 04            sub    $0x04,%esp
```

allowing it to represent a common function entry preamble while ignoring variation in the amount of storage allocated for local variables (the immediate value subtracted from `esp`). Idioms have proven particularly effective for representing intra-program provenance properties, as we describe in Chapter 7. Figure 5.3 depicts the idiom features' mutual information for several data sets.

5.4 GRAPHLETS

The combination of byte N-grams and idioms are effective for capturing many low-level details that are characteristic of particular elements of the provenance hierarchy, as we demonstrate in later chapters. However, these features do not capture higher-level characteristics of program structure. Our experience has taught us that variations in the provenance hierarchy frequently lead to variations in the layout and structure of the binary code; different versions of a compiler, for example, may make systematically different choices when ordering the basic blocks of a function. Based on a higher-level abstraction of the binary code, structural features take a further step in hiding inessential byte-level details; a similar intuition has motivated researchers working on malware analysis [10, 46]. We have designed a collection of features based on properties of the program control flow graph.

The key challenge in designing control flow-based features is finding the balance between feature expressiveness—the extent and level of detail—and generalizability.



(a) version (b) family (c) authorship

Figure 5.3: The mutual information between idiom features and compiler version, compiler family, and authorship provenance properties, with features ordered by their rank in (a) as in Figure 5.1. Compared to Figure 5.1, the mutual information density is distributed over a larger number of features. Again, and as expected, the features that are closely associated with program authorship differ widely from those that indicate compiler provenance.

Features that precisely describe large segments of the control flow graph are likely to fall towards the functionality side of the functionality–provenance spectrum. Efficiency is also a concern. Testing whether a program's control flow graph contains a particular structural pattern is equivalent to the *subgraph matching problem*, which is known to be NP-complete [17]. Both generalizability and efficiency argue for smaller, simpler, control flow–based features.

We have developed features that we call *graphlets*, which are based on structural representations used in genetic sequence modeling [71]. Our graphlets are connected 3-subgraphs defined over the program control flow graph. To be precise, let a CFG be a directed graph $G = (V, E, \tau)$ over the *basic blocks* of the binary that is defined by:

- the set V of vertices corresponding to basic blocks,
- the set $E \subseteq V \times V$ corresponding to control flow edges between blocks, and
- the labeling function $\tau : E \to \mathcal{T}$ that associates a particular edge in the graph with a type (such as `branch` or `call`).

Graphlets are three-node, non-isomorphic, annotated subgraphs of the CFG.[1] These subgraphs $G' = (V', E', \tau', \sigma)$ extend the CFG with a labeling function $\sigma : V \to \Sigma$ that

[1] Our use of three nodes stems in part from the use of similar featuers by Pržulj et al. [71], and also reflects our desire for small subgraphs that can be efficiently encoded. 3-subgraphs work well for our applications; however, this does not rule out the possibility that different sizes of graphlets (e.g., two or four nodes) may also be effective.

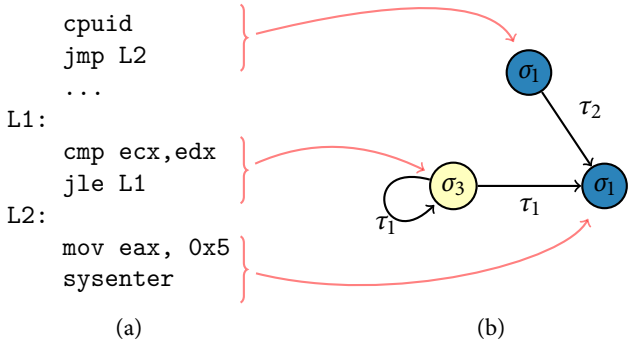


Figure 5.4: A code example and a corresponding graphlet. The vertex colors σ and edge labels τ are determined by the particular graphlet feature mapping functions (for example, both of the blocks represented by (●) vertices contain system instructions).

assigns a *color* to vertices (basic blocks). The edge labeling function τ' may also map to a different set of edge types than those used in the control flow graph. The details of the color and edge mapping depend on particular classes of graphlet feature, which we describe below.

5.4.1 INSTRUCTION SUMMARY GRAPHLETS

Instruction *summary graphlets* are inspired by a binary code representation used in polymorphic worm detection [46], where basic blocks were colored according to fourteen *instruction classes* such as string operations or branches. Following this scheme, we define fifteen instruction classes (Table 5.1), where the extra class is due to an implementation detail in the library we use for instruction decoding [66] and is not a principled decision motivated by assumptions about provenance properties. The color of a vertex in summary graphlets is a fifteen-bit number encoding whether instructions of each class are present in a block. More formally, summary graphlets supply a labeling function $\sigma : V \to [0, 2^{15} - 1]$; we have found that basic blocks rarely include instructions from more than a few classes, so the set of vertex colors in a given program is sparse. This representation captures differences in the arrangement of code while being less sensitive to the particular instructions used, reducing redundancy with idiom features. Figure 5.4 depicts an example code sequence and a corresponding summary graphlet.

Table 5.1: Instruction classes used in summary graphlets.

Instruction class	Example(s)
Arithmetic	`add`, `imul`, `shl`
Conditional branch	`jb`, `jle`
Call	`call`, `ret far`
Comparison	`test`, `setb`
Flags register operation	`pushf`, `stc`
Floating point	`fimul`, `fld`
Halt / Illegal	`hlt`, `ud`
Direct branch	`jmp`
Load effective address	`lea`
Logical	`not`, `xor`
Move	`mov`, `lds`
Stack operation	`pop`, `pushad`
String	`stosb`, `cvtdq2pd`
System	`cpuid`, `int`
Test & set	`cmpxch`, `bound`

5.4.2 Branch graphlets

Our experience analyzing code emitted by different compilers suggests that the particular branch instructions used to direct control flow are highly indicative of provenance properties like compiler family or version. For example, one version of the GNU C compiler might frequently use the `jge` (jump if greater-than or equal) instruction to test a loop condition, while a different version might re-order the block layout and condition tests and use a `jl` (jump if less-than) instruction for the same source code. We designed *branch graphlets* to explicitly capture this phenomenon. Branch graphlets are similar to instruction summary graphlets, but supply a color labeling $\sigma : V \to \mathcal{B}$, where $\mathcal{B}$ is the set of 22 unique branching and looping instructions in the IA-32 instruction set [38].

5.4.3 Supergraphlets

Instruction summary and branch graphlets are a bridge between the instruction-level representation (using colors based on instruction classes) and the program structure (the local control flow); however, these features may not capture provenance properties

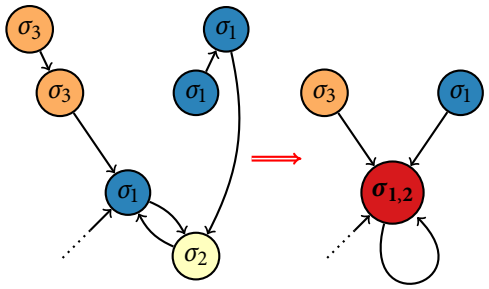


Figure 5.5: Supergraphlets represent control flow relationships in a graph where the neighbors of the middle three nodes have been *collapsed*; the color of one collapsed node (●) reflects the union of two nodes with different colors.

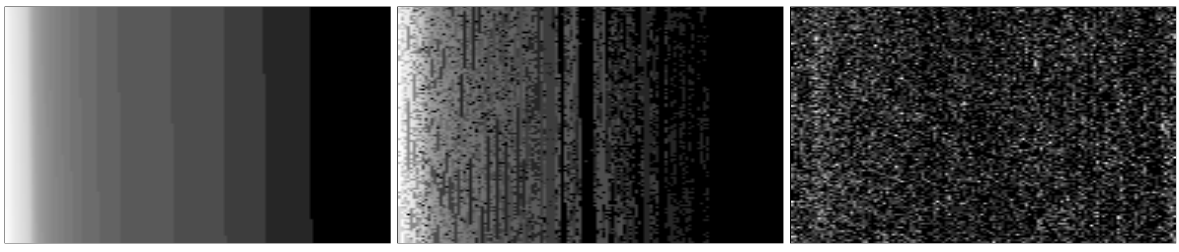
that are visible only in long-range program structure. Because using patterns based on larger subgraphs is computationally infeasible, we developed *supergraphlet* features that are defined over a transformation of the original control flow graph.

Supergraphlets are analogous to instruction summary graphlets defined over a *collapsed* control flow graph, as illustrated in Figure 5.5. The graph collapse operation merges each node in the graph with a random neighbor. The edge set and color of the collapsed node represent the union of the edge sets and colors of the original nodes. A three-node graphlet instantiated from the collapsed graph is thus an approximate representation of six nodes in the original CFG. This process can be repeated recursively to obtain the desired long-range structural coverage. Note that because random neighbors are selected, we do not obtain all possible supergraphlets of the original graph; in keeping with our general approach to feature design, we rely on the vast number of instantiated supergraphlet features to adequately capture details that are characteristic of a given provenance property. Supergraphlets have proven useful in the recovery of stylistic provenance details, as we show in Chapter 9.

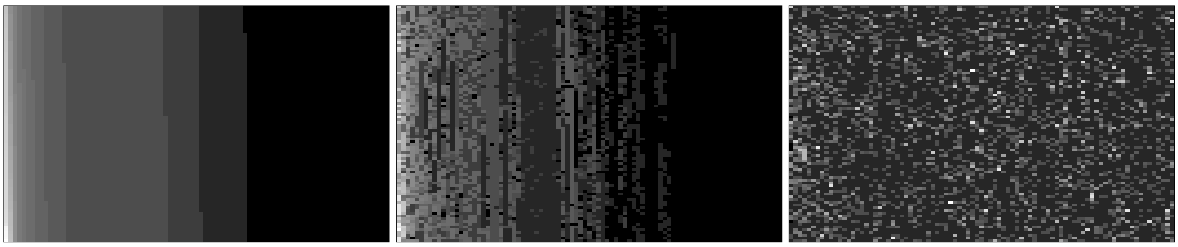
Figure 5.6 depicts the mutual information distribution for the instruction summary graphlet and supergraphlet features. The provenance information is concentrated in a smaller fraction of the supergraphlet features (d–f) compared to the instruction summary graphlets (a–c).

5.4.4 GRAPHLET QUERIES

We test for specific graphlets in a CFG by computing a *canonical labeling* that is identical for the isomorphisms of size three under a particular graphlet coloring function σ.



(a) version (insn) (b) family (insn) (c) authorship (insn)



(d) version (super) (e) family (super) (f) authorship (super)

Figure 5.6: Provenance–feature mutual information for instruction summary graphlets (a–c) and supergraphlets (d–f). Note that the fraction of features with high mutual information is substantially larger for the instruction summary graphlets in the version data set (a) than for supergraphlets (d), indicating that only a small number of supergraphlets carry most of the information in that data set.

Producing a canonical labeling is equivalent to the graph isomorphism problem, for which no polynomial time algorithm is known. However, canonical labelings can often be efficiently computed in practice, particularly for small graphs such as ours. A labeling is the concatenation of the graph's adjacency matrix, annotated with node and edge colors; the canonical labeling is the minimum labeling under a lexicographic ordering. The general graphlet matching and canonical labeling algorithms are presented in Figure 5.7.

As an example, consider the set of possible colors $\Sigma = \{\sigma_1, \cdots, \sigma_m\}$ that are the output of the σ coloring function, and the possible edge types $\mathcal{T} = \{\tau_1, \cdots, \tau_n\}$ output by the τ edge type function, such as those used in instruction summary graphlets. Assume arbitrary total orderings over node colors $\sigma_1 \prec \cdots \prec \sigma_m$ and the types $\tau_1 \prec \cdots \prec \tau_n$. According to our canoicalization rules, the graphlet is the concatenation of nodes and edges, with each part ordered first by their colors and then by edge types. The graphlet

```
function MatchGraphlets(F = (V, E),τ,σ,C)
    M ← ∅
    for all v ∈ V do
        for all {n_a, n_b} ∈ Neighbors(v) do
            V_s ← {v, n_a, n_b}
            E_s ← V_s × V_s ⊆ E
            c ← Canonical(V_s, E_s, τ, σ)
            if c ∈ C then
                M ← c
    return M

function Canonical(V,E,τ,σ)
    c ← Sort(V, σ(V))
    for d ← 1 to max deg(v ∈ S) do
        S_d ← {v ∈ S| deg(v) = d}
        E_d ← S_d × S_d ⊆ E
        for all v ∈ Sort(S_d, τ(E_d)) do          ▷ See caption
            c ← c || τ((∗, v) ∪ (v, ∗) ∈ S_d)
    return c
```

Figure 5.7: An algorithm for finding graphlets in a function. The canonical label for every connected triple of blocks is computed and tested against the set of graphlet features. The $\textsc{Sort}(V, \prec)$ function sorts a set of vertices V using the given ordering function (e.g., vertex colors or edge types). The canonical ordering of vertices is over vertex color σ, vertex degree, and edge color τ. The 3-graphlets we use require testing at most six permutations to find the canonical ordering, but vertex degree ordering frequently reduces that number.

depicted in Figure 5.4 has the annotated adjaceny matrix

$$\begin{pmatrix} \cdot & \cdot & \cdot \\ \cdot & \tau_2 & \cdot \\ \cdot & \tau_1 & \tau_1 \end{pmatrix}$$

where $\cdot$ denotes a missing edge; its canonical labeling is

$$\sigma_1 \cdots \sigma_1 \cdot \tau_2 \cdot \sigma_3 \cdot \tau_1 \tau_1.$$

In general, computing the canonical labeling requires examining $K!$ permutations for a K-vertex graph; in practice our algorithm reduces this search space by partitioning the set of nodes based on vertex degree, which is invariant to isomorphism [49].

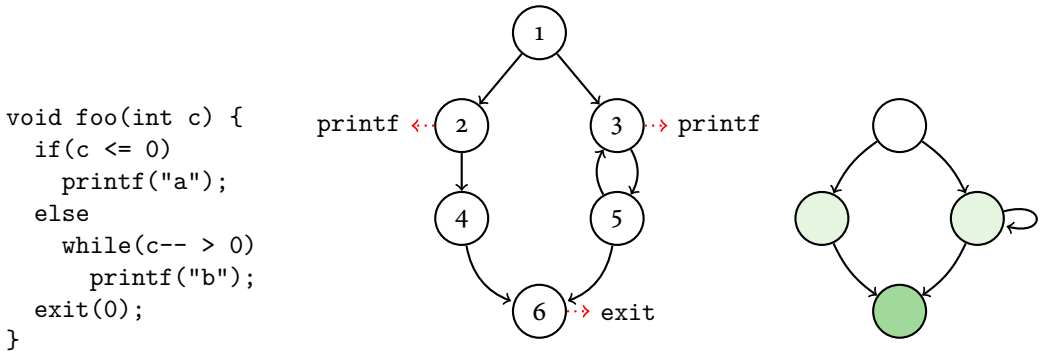


Figure 5.8: Transforming a control flow graph to produce call graphlets. Only nodes 1, 2, 3, and 6 from the control flow graph (b) are preserved in the transformation (c), which is colored by the calls invoked at each node.

5.5 Call Graphlets

So far our focus has been on features that represent elements of the control flow graph of a program, which describes the program structure in terms of basic blocks. An alternative view of program structure is that of the relationship between *functions*, both those that constitute the program binary (*local functions*) and those that are part of external libraries (*external functions*). The *call graph* of a program describes the relationship between local and external functions at a high level, but does not describe the ordering of *interprocedural* control flow.

Call graphlets are defined over a transformation of the control flow graph G^c that contains only those vertices that invoke `call` instructions. This graph is constructed by creating edges $E^c = \{(v, v') : v \rightsquigarrow v'\}$, where $\rightsquigarrow$ indicates the existence of a path in the original control flow graph. Figure 5.8 depicts a simple function and its original and transformed CFGs. The vertices in this representation use the coloring function $\sigma_c : V^c \to \{\mathcal{L}, \textsc{local}\}$, where $\mathcal{L}$ is a predefined set of external library functions and LOCAL is a special value meaning any local function within the program binary. Internal functions receive a single, generic color because, unlike calls to external libraries, they are not comparable across different programs.

The call graphlet features themselves are extracted from the transformed graph in the same way that other graphlet-based features are. Figure 5.9 depicts the mutual information for these features.

(a) version (b) family (c) authorship

Figure 5.9: Mutual information for call graphlet features. Note that only a few features contain all of the information about compiler family and version properties, while many features are informative about program authorship.

5.6 EXTERNAL LIBRARIES

A program's dependence on particular external libraries can reveal provenance properties. Call graphlets encode characteristics of a program's interaction with external libraries, but each graphlet feature is still quite "local": capturing the structure of function invocations prevents considering the aggregate usage of external libraries for the whole program. For some provenance properties—particularly programmer style and authorship—the identity of external functions alone may be telling. For example, in Chapter 9 we show that groups of programmers can be coarsely partitioned by their preference for output functions.

External library features indicate whether—or how frequently—a program binary invokes a particular external library function. Because these features tend toward the functionality end of the functionality–provenance spectrum, we have used them primarily for high-level provenance applications, like authorship attribution.

5.7 SUMMARY

The features described in this chapter capture binary code details at several levels of abstraction, but have not been designed to reflect *a priori* notions of how program provenance is preserved in program binaries. Our approach is to define a large number of simple, uninformed features and to let the data determine which are indicative of variations in how a program was produced. The following chapter shows how we use these features to form models of program provenance. In the experimental chapters, we make the discussion of code features more concrete, evaluating the contribution of

various types of feature to several provenance recovery problems.

6

Modeling Program Provenance

Our approach to provenance recovery uses machine learning techniques to model the relationship between code features and elements of the provenance hierarchy. Provenance models are defined by the level of code abstraction to which provenance is assigned (e.g. basic blocks, functions, entire programs), and by the way in which particular code features map to provenance properties. The modeling part of our framework involves the specification of these models, and the training and inference methods that learn to associate code characteristics with program provenance. Provenance models can be divided into two classes: those that consider the provenance of individual code elements independently, which we call *simple provenance models*, and those that integrate structural characteristics of programs to capture relationships between code elements, which we call *complex models*.

6.1 SIMPLE PROVENANCE MODELS

A provenance model defines a relationship between evidence in the binary and a provenance property. In simple provenance models, the evidence consists of the code features (Chapter 5) associated with a single code element, be it a basic block, function, or entire program. Models are determined by three components:

1. A provenance property or set of properties, such as compiler family or source language. Each property has a set of possible values (e.g., GCC, ICC, MSVS for compilers, or C++, C, and Fortran for languages), which we refer to as *provenance labels*.

2. *Feature functions* $f(\cdot)$ that describe the evidence, or features, that are associated with a particular code element. The domain and range of feature functions vary according to the particular model, as we describe below.

3. Model parameters that determine the relationship between components (1) and (2). These parameters are not specified *a priori*, but are derived from training data using statistical machine learning techniques (Section 6.3).

Provenance models are further characterized by the model structure that determines how the parameters, feature functions, and labels relate to one another. Most of the techniques we use take the form of *probabilistic models*, wherein we define the relationship between features and provenance properties in terms of a conditional probability distribution

$$P(\,\mathit{provenance}\,|\,\mathit{features}\,).$$

More formally, we define the distribution over the possible provenance labels $y \in \mathcal{Y}$ for a given code element to be

$$P(y|\mathbf{x}, \Lambda) = \frac{1}{Z}\exp(\lambda_y^T \mathbf{x}) \tag{6.1}$$

where x is a *feature vector* representing the evaluation of d feature functions for the code element

$$\mathbf{x} = \begin{pmatrix} f_1(\cdot) \\ f_2(\cdot) \\ \vdots \\ f_d(\cdot) \end{pmatrix}$$

and $\lambda_y = (\lambda_{1,y}, \lambda_{2,y}, \ldots, \lambda_{k,y})^T$ is the y^{th} column of the parameter matrix

$$\Lambda = \begin{pmatrix} \lambda_{1,y_1}, & \lambda_{1,y_2}, & \ldots & \lambda_{1,y_m} \\ \lambda_{2,y_1}, & \lambda_{2,y_2} & \ldots & \lambda_{2,y_m} \\ \vdots & \vdots & \ddots & \vdots \\ \lambda_{d,y_1}, & \lambda_{d,y_2} & \ldots & \lambda_{d,y_m} \end{pmatrix},$$

where each column of Λ corresponds to a provenance label $y \in \mathcal{Y}$. The factor Z is a normalization term that ensures that Equation 6.1 defines a probability distribution. A program or other binary code artifact is a collection of code elements $e_1, \ldots, e_n$ described by n feature vectors. We model such a collection, for example when independently describing the provenance of all functions in a binary, as

$$P(\{y\}_{1:n}|\{\mathbf{x}\}_{1:n}, \Lambda) = \frac{1}{Z}\exp\left(\sum_{i=1}^{n} \lambda_{y_i}^T \mathbf{x}_i\right). \tag{6.2}$$

In the following sections, we make these model components more concrete by describing the feature functions that we use to represent code features for code elements at several levels of abstraction.

6.1.1 CODE ELEMENT OVERVIEW

We have created models of provenance at the level of bytes, basic blocks, functions, contiguous code regions, and programs. Having models of different granularities allows our framework to be applied flexibly to a variety of provenance recovery scenarios, as we show in the experimental part of this dissertation; for example, we can model the provenance of individual functions to find portions of a binary emitted by different compilers, or we can model programs as a whole to identify programmer style.

Let e stand in for a particular type of code element; rewriting Equation 6.1 slightly, simple provenance models for code elements $e \in \mathcal{P}$ are defined generically as

$$P(y_e|\mathbf{x}_e, \Lambda, \mathcal{P}) = \frac{1}{Z} \exp\left(\sum_{i=1}^{k} \lambda_{y_e,i} \times f_i(e, \mathcal{P})\right)$$

(we expand the parameter and feature vectors for clarity). Simple provenance models incorporate only *unary* feature functions $f : e \to \mathbb{R}$; we include the argument $\mathcal{P}$ explicitly below, but it can be treated as an implicit parameter of all feature functions. Feature functions indicate whether the code element in question exhibits a particular feature:

$$f_k(e, \mathcal{P}) = \begin{cases} 1 & \text{if feature } k \text{ exists for code element } e \text{ in } \mathcal{P}, \\ 0 & \text{otherwise.} \end{cases}$$

Exactly what is meant by "exists" depends on the particular feature. Feature functions are defined for all of the possible feature types that we define in Chapter 5; we write idiom feature functions as $f_\iota(\cdot)$, graphlet feature functions as $f_\gamma(\cdot)$, and so forth.

The form of simple provenance models is insensitive to the type of code element; only the feature functions vary with respect to the granularity of the model. The feature functions for higher-level abstractions (e.g., functions) are defined recursively in terms of the feature functions for lower-level abstractions, beginning with those associated with byte offsets in the program.

6.1.2 BYTE OFFSETS

The finest-grained provenance models we have developed assign labels to specific offsets in the binary—that is, to the location of a single byte. Our work on stripped binary parsing [74, 75] uses such models to recognize the patterns of function entry points, as we describe in Chapter 7. In these models, provenance labels are associated with each byte in the program or binary code artifact; that is, for an n-byte binary $\mathcal{P}$,

we model each offset $a \in [0, n)$ as

$$P(y_a|\mathbf{x}_a, \Lambda, \mathcal{P}) = \frac{1}{Z} \exp\left(\sum_{i=1}^{k} \lambda_{y_a,i} \times f_i(a, \mathcal{P})\right) \tag{6.3}$$

The feature functions are parameterized by the program and offset, and indicate whether the code at that offset exhibits a particular feature. For features like N-grams and instruction idioms that are a direct interpretation of the code bytes, an offset exhibits a feature if the feature begins at that offset, e.g.

$$f_\iota(a, \mathcal{P}) = \begin{cases} 1 & \text{if idiom } \iota \sim \textsc{Decode}(\mathcal{P}, a) \\ 0 & \text{otherwise} \end{cases}$$

where Decode returns the instructions represented by the bytes starting at offset a and $\sim$ represents a matching operation. Other offset-based feature functions such as graphlets test whether a feature spans the offset. Recall that a graphlet γ consists of a set of vertices V_γ defined by some transformation over the control flow graph; a graphlet matches a given offset if it is within the region of code spanned by any of the vertices:

$$f_\gamma(a, \mathcal{P}) = \begin{cases} 1 & \text{if } \exists v = [s, e] \in V_\gamma : s \leq a \leq e \\ 0 & \text{otherwise.} \end{cases}$$

Feature functions for the other types of features in Chapter 5 are defined similarly. These feature functions form the basis for those that are used with more abstract code elements.

6.1.3 basic blocks

The feature functions used in basic block–based models are derived from offset-based feature functions. To model provenance at the basic block level, we define the model

$$P(y_b|\mathbf{x}_b, \Lambda, \mathcal{P}) = \frac{1}{Z} \exp\left(\sum_{i=1}^{k} \lambda_{y_b,i} \times f_i(b, \mathcal{P})\right),$$

which only differs from Equation 6.3 in that it is defined over the domain of basic blocks $b \in \mathcal{B}$ in the program's control flow graph. The feature functions are built up from offset-based feature functions; evaluating an idiom feature function over a basic block

$$f_\iota(b, \mathcal{P}) = \bigcup_{a \in b} f_\iota(a, \mathcal{P})$$

recursively applies that feature function over the instruction offsets within the block. Alternatively, the feature function can return the number of times a feature occurs within a block

$$f_\iota^+(b, \mathcal{P}) = \sum_{a \in b} f_\iota(a, \mathcal{P}),$$

which can be used to define models that depend on the cardinality of features. Other block-level feature functions are defined similarly.

6.1.4 FUNCTIONS

Notwithstanding programming language features like inline assembly, functions are the base unit of compilation and so are particularly suitable for modeling provenance, especially for provenance properties related to the compilation toolchain. We define a function $\mathcal{F}_i$ in terms of an *entry point* e and a collection of basic blocks $\mathcal{B}$ reachable through *intraprocedural* control flow (e.g. branches) from that point. The feature functions that characterize function-level provenance models are defined in terms of basic block–based feature functions; we use both indicator and cardinality feature functions in function-based models:

$$f_k(\mathcal{F}_k = \langle e, \mathcal{B} \rangle, \mathcal{P}) = \bigcup_{b \in \mathcal{B}} f_k(b, \mathcal{P}) \qquad f_k^+(\mathcal{F}_k = \langle e, \mathcal{B} \rangle, \mathcal{P}) = \sum_{b \in \mathcal{B}} f_k^+(b, \mathcal{P}).$$

6.1.5 PROGRAMS AND CODE REGIONS

In some cases, the provenance of an entire program or a large region of code as a whole may be of interest. The feature functions we use in such models are derived from finer-grained feature functions as appropriate. For example, we usually specify whole-program models in terms of the functions contained in the program

$$f_k^+(\mathcal{P}) = \sum_{\mathcal{F} \in \mathcal{P}} f_k^+(\mathcal{F}, \mathcal{P}).$$

If the model applies to potential non-code bytes interspersed with the code (Figure 6.1), arbitrary code regions defined by an offset range $[s, e]$ can be characterized by offset-based feature functions

$$f_k^+(\mathcal{R} = [s, e], \mathcal{P}) = \sum_{a \in [s,e]} f_k^+(a, \mathcal{P})$$

or, if only code is desired, by block-based feature functions

$$f_k^+(\mathcal{R} = [s, e], \mathcal{B}, \mathcal{P}) = \sum_{\substack{b \in \mathcal{B}, \\ s \le b \le e}} f_k^+(b, \mathcal{P}).$$

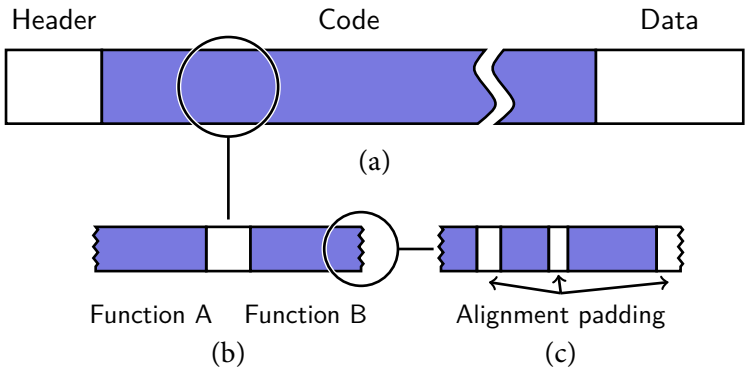


Figure 6.1: Layout of executable code in a typical binary. At the highest level of abstraction, the program's executable code resides in a contiguous segment of the binary, e.g. the `.text` segment in ELF binaries (a). Under closer consideration, there often exist non-executable bytes between functions (b) and even within functions (c).

6.2 Complex provenance models

The simple provenance models introduced in the previous section treat individual binary code elements, such as basic blocks or functions, independently; they assume that provenance is only reflected in code features like the existence of instructions or in local control flow structure. Complex provenance models, by contrast, seek to capture higher-level relationships between code elements. For example, we might reasonably assume that adjacent functions in a program binary are most likely produced by the same compiler and with the same compilation options, as compilation units (source files) tend to comprise many functions. Complex provenance models allow us to represent these and other intuitions about the structure of program provenance; moreover, the models allows us to determine the validity and importance of these structural features from training data in the same way that we learn the predictive value of other code features.

Complex models are distinguished by the use of *binary* feature functions, in addition to the unary feature functions described in the previous section. Binary feature functions indicate whether a relation holds between two code elements. For example, the function

$$f_{\text{ADJ}}(\mathcal{F}_i, \mathcal{F}_j) = \begin{cases} 1 & \text{if } \mathcal{F}_i, \mathcal{F}_j \text{ are adjacent} \\ 0 & \text{otherwise} \end{cases}$$

indicates whether two functions are adjacent to one another in the binary. Unlike their

unary counterparts, binary feature functions describe a relationship between two code elements that may have different provenance labels; models that incorporate binary feature functions incorporate additional parameters M that are indexed by the relation and the provenance labels of both elements, i.e.,

$$\mu_{r,y_i,y_j} \in \mathrm{M}$$

where $r \in \mathcal{R}$ is a particular type of relation (e.g., adjacency) and y_i and y_j are provenance labels. Extending Equation 6.2 to incorporate binary feature functions, we define a *probabilistic graphical model* of the form

$$P(\{y\}_{1:n}|\{\mathbf{x}\}_{1:n}, \Lambda, \mathrm{M}) = \frac{1}{Z} \exp\left(\sum_{i=1}^{n} \lambda_{y_i}^T \mathbf{x}_i + \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{r\in\mathcal{R}} \mu_{r,y_i,y_j} f_r(e_i, e_j)\right) \tag{6.4}$$

where e stands in for the code element type (e.g., function) appropriate to the model. Models based on binary feature functions can capture all $N \times N$ possible pairwise relations between all code elements, but we typically base our models on the much sparser relationships defined by code layout and program control flow.

6.2.1 SEQUENCES

We frequently use sequential provenance models that capture the adjacency relationships of code elements as laid out in a program binary. Modeling provenance as a sequence follows from our intuition that locality is an important factor in code element provenance: given the nature of the compilation process, for example, it is likely that adjacent functions are part of the same translation unit and therefore were emitted by the same compiler. As with unary features, the extent to which adjacency is an important factor in sequential provenance models is determined empirically through model training.

When the only binary feature is adjacency ($\mathcal{R} = \{\text{ADJ}\}$), the model in Equation 6.4 defines a *linear-chain conditional random field* [50]; for any code element e_i, only $f_{\text{ADJ}}(e_i, e_{i+1})$ and $f_{\text{ADJ}}(e_{i-1}, e_i)$ are non-zero. Depicted in Figure 6.2, linear-chain CRFs are graphical models in which the labels nodes y depend on their immediately adjacent neighbors and on the evidence nodes x which represent the unary feature vector. Linear-chain CRFs are practical for provenance modeling and recovery applications because efficient algorithms exist for parameter estimation and inference. The techniques we use to infer compiler toolchain provenance in Chapter 8 build on sequential models.

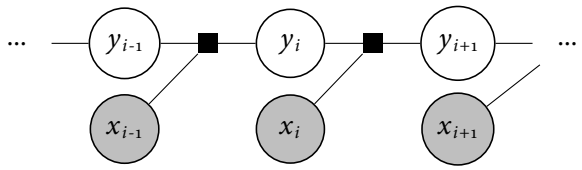


Figure 6.2: A linear-chain conditional random field, depicted as a *factor graph*. Label nodes (○) are related to one another and to evidence nodes (●) by *factors* (■) that represent feature functions.

6.2.2 Program Structure

Another reason to introduce dependencies between model components is to capture program structure in the model. Unlike the local structural elements represented by graphlet-based features, directly incorporating control flow information into the model allows us to represent relationships between the provenance properties of distant code elements. For example, one might hypothesize that a caller–callee relationship between two functions might be a weak indicator of similar provenance, based on the further hypothesis that placing related code in the same source file is common practice. Adding pairwise feature functions that represent program structure allows the training process to automatically evaluate this hypothesis. The feature function

$$f_{\text{CALL}}(\mathcal{F}_i, \mathcal{F}_j) = \begin{cases} 1 & \text{if } \mathcal{F}_i \text{ calls } \mathcal{F}_j \\ 0 & \text{otherwise} \end{cases}$$

encodes a caller–callee relationship; combining this feature function with adjacency features results in graphical model structure such as that depicted in Figure 6.3. Other structural features are possible; in the experimental part of this dissertation, we describe models that make use of call relationships, intraprocedural control flow, and byte-range overlap. The power of CRFs is in their ability to represent arbitrary relationships between model components by defining new binary feature functions.

Incorporating binary features based on control flow structure is likely to introduce circular probabilistic dependencies into the model, as in Figure 6.3. These edges encode potentially important information about the provenance of connected code elements, but they come at a cost: in general, efficient algorithms for exact inference in loopy graphical models do not exist. To incorporate this structure, we must rely on *approximate* inference techniques for both parameter estimation and when applying the models to provenance recovery. There are two ways to approach this problem, both of which we discuss below and in more detail in the following experimental chapters: we

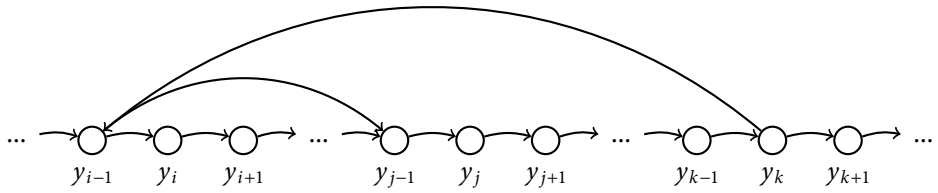


Figure 6.3: A general CRF that incorporates long-range relationships. Inference in loopy graphical models is not tractable in general, and approximate methods are required.

can use an existing algorithm like Loopy Belief Propagation [27], or we can use domain knowledge to derive an approximate algorithm for a specific provenance property [75].

6.3 LEARNING AND INFERENCE

Provenance models provide a skeleton for recovering program provenance, but the model parameters—the sets of weights Λ and M associated with each feature function—determine how these models assign provenance properties to program binaries. Our provenance recovery framework learns these parameters automatically through a parameter estimation or *training* process. Once learned, the parameterized model can be used to assign labels to binary code elements, recovering provenance properties.

6.3.1 MODEL TRAINING AND CLASSIFICATION

Model training involves three components: (1) training data that reflect the provenance property or properties of interest, (2) a model definition such as the examples used in the previous sections, and (3) a machine learning algorithm suitable for parameter estimation for the particular model.

Training data Model parameters are learned from training data that reflect the relationship between a particular provenance property of interest (e.g., compiler family) and example programs or other binary code elements. The training data consist of label and data tuples $\langle y_i, e_i, \mathcal{P}_k \rangle$, where e_i is some code element such as a basic block or function and $\mathcal{P}_k$ is the program containing that element; except for when $e_i \equiv \mathcal{P}_k$, there is a many-to-one mapping from code elements to programs. We use a wide variety of training data for the various provenance investigations described in the experimental part of the dissertation.

Model definitions Provenance models in our framework are specified by the following parameters:

- The type of code element (such as offset or function) with which provenance is associated.
- Any dependencies between code elements, such as sequential or structural dependencies. Our framework can incorporate arbitrary dependencies by specifying pairwise feature functions.
- The unary code features that are part of the model. Typically we select a subset of possible code features with which to perform both parameter estimation and inference; the model specification lists the particular feature functions that should be evaluated for all code elements.

Algorithms The algorithms we use for model training and inference vary according to the form of the model (simple or complex) and according to whether or not a probabilistic interpretation of model output is needed. In most cases, we are able to make use of existing machine learning software:

- Models incorporating only unary feature functions (Equation 6.2) are equivalent to *logistic regression* [32]—the provenance labels assigned to code elements are statistically independent of the labels assigned to other elements. We use the training and inference implementation provided by the LIBLINEAR software package for these models [23].
- Models that incorporate binary structural features are defined as conditional random fields, for which we employ several different training and inference techniques. For sequential models, we use the MALLET software package [59], which implements Sum-Product and Max-Product algorithms for parameter estimation and inference, respectively (see Chapter 3 for details). For more complicated, arbitrarily structured models, the GRMM package implements approximate inference through Loopy Belief Propagation [27] and variants like Tree-based Reparameterization [97]. In Chapter 7 we introduce a heuristic algorithm that enforces pairwise structural constraints, approximating probabilistic inference in loopy graphical models.
- The models we describe above are stated in probabilistic form; however, our framework also incorporates models similar to Equation 6.2 that are not probabilistic in nature. For independent classification of provenance, we also use the machinery of *support vector machines* (SVMs) [18], which employ a maximum-margin approach to classifying objects in a feature space (see Chapter 3). The

LIBLINEAR package implements algorithms for training and classification with SVMs.

6.3.2 CLUSTERING

The preceding discussion focuses on classification, a *supervised* learning problem in which the objective is to assign a particular provenance label to a novel binary code object. Alternatively, we may be interested not in the specific provenance of a program binary, but in how the binary's provenance relates to others in a collection. For example, a security analyst may be interested in determining whether different malware instances contain stylistically similar code—without knowing the identities or stylistic attributes of possible program authors, or even how many authors may be represented.

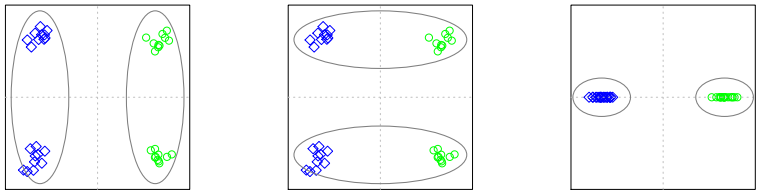
Resolving queries of this type involves *clustering*, an *unsupervised* learning problem where no training data are available to build models. Clustering algorithms typically rely on some notion of distance between elements, for example treating feature vectors as coordinates in a d-dimensional *feature space*. We consider several clustering methods in Chapter 10, where we examine the problem of grouping programs by stylistic similarity.

One of the primary challenges in unsupervised learning problems is to design techniques that group entities by the property of interest (e.g., programmer style) and not by some other property (program functionality). Our approach, illustrated in Figure 6.4, is to transform the feature space such that binary code elements with similar provenance properties are close to one another. We define a $d \times d$ *distance metric* A such that the *Mahalanobis distance* [57] between two feature vectors $\mathbf{x}_a, \mathbf{x}_b \in \mathbb{R}^d$ is

$$D_{\mathrm{A}}(\mathbf{x}_a, \mathbf{x}_b) = \sqrt{(\mathbf{x}_a - \mathbf{x}_b)^T \mathrm{A}(\mathbf{x}_a - \mathbf{x}_b)}.$$

If a metric can be found such that code elements with similar provenance are close under that metric, then clustering techniques will do better at forming provenance clusters.

We observe that features associated with a particular provenance property (e.g., programmer style), if they are general, can be learned from *any* set of training data that reflects that property, even if the programs used in training do not share the same concrete provenance as those that we desire to cluster. Our approach is to use the labeled training data from one domain to learn the provenance distance metric. More precisely, consider two sets of programs $\{\mathcal{P}_1, \cdots, \mathcal{P}_\ell\}$ and $\{\mathcal{P}_{\ell+1}, \cdots, \mathcal{P}_u\}$, with known author labels $\{y_1, \cdots, y_\ell\} \in \mathcal{Y}$ and unknown labels $\{y_{\ell+1}, \cdots, y_u\} \in \mathcal{Y}'$, with $\mathcal{Y} \setminus \mathcal{Y}' \neq \varnothing$; that is, at least some of the concrete provenance labels are unique to each set. We define a two-part algorithm for transferring provenance knowledge from the labeled data to the unlabeled data:



(a) (b) (c)

Figure 6.4: The hazards of unsupervised clustering. Let the vertical dimension indicate one property (e.g., functionality) and the horizontal dimension indicate another (e.g., style). Assuming that the data belong to true classes y_1 (◇) and y_2 (○)—i.e., the desired grouping is by style—and two clusters are formed, the correct cluster partition (a) is no more likely than the alternative (b). Using the distance metric $\left(\begin{smallmatrix} 1 & 0 \\ 0 & 0 \end{smallmatrix}\right)$ is equivalent to transforming the data as in (c), where the differences in terms of functionality have zero value and the clustering decision is unambiguous.

1. Learn a metric A over ℓ labeled programs $\mathcal{P}_1, \cdots, \mathcal{P}_\ell$ such that the distance in the feature space between two programs with the same label y is always less than the distance between two programs with different provenance.
2. Cluster u unlabeled programs $\mathcal{P}_{\ell+1}, \cdots, \mathcal{P}_{\ell+u}$ using the distance function D_{A}.

In Chapter 10, we use the *large margin nearest neighbors* (LMNN) algorithm [99] to learn metrics that reflect programmer style.

6.4 Summary

Provenance models define a relationship between binary code elements and provenance labels based on *feature vectors* that describe the code and *model parameters* that are learned from training data. The power and flexibility of our provenance recovery system are due to the use of a variety of probabilistic and discriminative that can incorporate a wide variety of code features that capture provenance properties; these models allow us to assign provenance labels to novel binary code elements, or to group code by similarity in one or more aspects of its provenance. In the experimental part of this dissertation that follows, we present a series of experiments in which we use this framework to resolve a variety of program provenance questions, from identifying the patterns of function entry points that are characteristic of particular compilers to modeling the evidence of programmer style that survives the compilation process.

7

Code Discovery in Stripped Binaries

Binary code analysis is a foundational technique in the areas of computer security, performance modeling, and program instrumentation that enables malicious code detection, formal verification, identification of performance bottlenecks, and many other areas. Because program binaries contain both code and non-code bytes, precisely locating executable code within programs is required before any analysis can proceed. The usual approach is to identify the start of each function (the *function entry points*, FEPs), and then to parse the binary code using the recursive traversal or other parsing methods described in Chapter 2. This approach is suitable when debugging symbols are available or when FEPs are otherwise explicitly specified; however, malicious programs, commercial software, and legacy codes all commonly lack debugging symbols.

Recursive traversal parsing can be used to find code reachable from the program entry point of such *stripped* binaries, but typically only recovers a subset: code reachable through indirect (pointer-based) control flow often cannot be located statically. Indirect control flow is common in binaries; in the real-world data set we describe later in this chapter, approximately 40% of functions are unrecoverable through recursive traversal parsing. The remaining functions lie in *gaps* between statically discovered functions. Existing tools discover functions in gaps by searching for manually-specified patterns of instructions that are recognizable as function preambles [31, 36], or use simple unigram and bigram instruction models to augment pattern-based heuristics [47]. These heuristics and simple statistical methods cannot adapt to variations in the compilation toolchain that significantly perturb or even optimize away expected instruction sequences at FEPs.

In this chapter, we bring our provenance recovery framework to bear on the problem of finding code in stripped binaries. While code discovery in stripped binaries is not fundamentally a provenance problem, we use modeling techniques that are analogous to those that we developed for provenance recovery to determine whether or not a particular offset within the binary is the start of a function. This approach overcomes the limitations of existing tools, which depend on expert domain knowledge

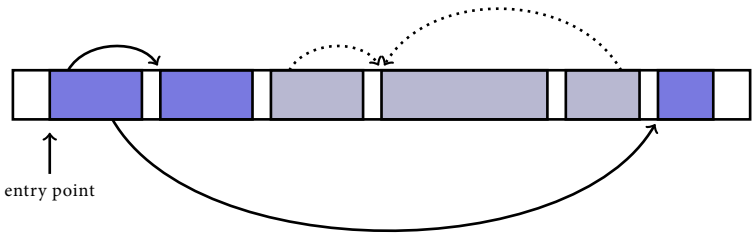


Figure 7.1: Stripped binary program model. The binary is assumed to contain functions reachable through statically analyzable control flow (⟶) from the entry point (■) and other *gap functions* that are only reachable through indirect control flow that cannot be discovered statically (■).

to develop pattern-based heuristics; our learning-based techniques automatically adapt FEP variations introduced by different compilers. The compiler-specific nature of this approach serves as a motivation for provenance recovery, and was the point of departure for our research into program provenance. In the following sections, we describe our provenance model and the algorithms we use for parameter estimation and FEP identification, and present a set of experiments that evaluate the efficacy of this technique. Our approach to stripped binary parsing significantly outperforms industry standard disassembly and binary analysis tools.

7.1 Problem Domain

We assume a binary code model as depicted in Figure 7.1, where some amount of code is statically reachable and the remainder resides in gaps. When analyzing the gap regions of a binary, it is not generally known how many functions exist, or whether all gap contents are actually code. To find FEPs in a gap, it is necessary to treat every byte offset in the gap as a candidate FEP. This technique is known as *exhaustive disassembly* [47]. Characteristics of the binary code—determined largely by the instruction set architecture—influence how we approach this task, which code features we consider, and how we design our provenance model.

7.1.1 Self-Repairing Disassembly

In this work, we consider binaries compiled on the Intel x86 architecture [38]. Intel x86 is a variable length instruction set with an opcode space that is quite dense: almost any

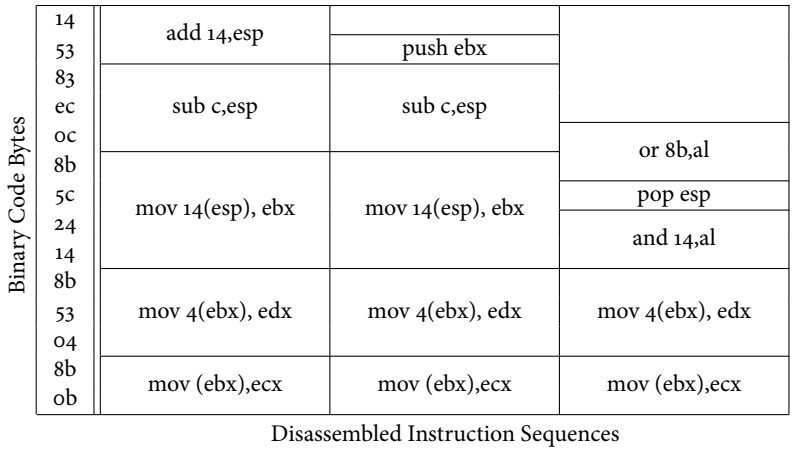


Figure 7.2: Self-repairing disassembly. Each instruction sequence (column) is produced by parsing from a particular offset within the bytes depicted on the left. Note that two of the sequences align within one instruction, and all three align within three instructions.

value is a valid opcode or the start of a valid multiple-byte opcode. Consequently, every byte offset in the gap might be the start of an instruction, and it is likely that disassembly from that point will produce a valid instruction sequence of some length. Furthermore, and somewhat non-intuitively, x86 code commonly exhibits *self-repairing disassembly*: the tendency for instruction streams disassembled by starting at different offsets in the binary to *align* or sync up with one another. Figure 7.2 depicts this phenomenon for parses from three offsets near the start of a function. Others have observed informally that disassembled x86 instruction streams offset by a few bytes tend to align quite quickly [54]; we provide a formal analysis that applies to all variable length instruction sets in Appendix A.

The consequences of self-repairing disassembly for FEP identification are twofold: (1) because the parse from an address that is not the boundary of an actual instruction quickly aligns with the actual instruction stream, it is unlikely that an incorrect FEP candidate will produce an illegal instruction or other obvious clues; and (2) the rapid alignment limits the utility of classifiers based on n-gram models of instruction streams [47]. Several candidate FEPs offset by a few bytes will likely have similar likelihood under an n-gram model, making it difficult to differentiate among them to identify the actual FEP.

7.1.2 program structure

The multiple instruction streams obtained by parsing from candidate function entry points in the binary induce a collection of candidate control flow graphs. In the provenance model we describe below, we make use of two types of structural features that follow from observations about binary code:

1. An instruction at byte-offset a within the binary can span several bytes. If so, the locations a and $a+1$ represent conflicting (overlapping) parses, and are unlikely to both be FEPs.

2. The disassembly starting from a can contain a `call` instruction that calls offset a'. If we believe that a is an FEP, then a' probably is too.

The first observation can be extended to define the *consistency* of a pair of candidate parses of a binary. Consider two control flow graphs G_i and G_j produced by parsing a binary from offsets a_i and a_j, respectively. We say that the control flow graphs are consistent if, for every pair of instructions $I \in \textsc{Instructions}(G_i)$ and $I' \in \textsc{Instructions}(G_j)$, the byte ranges $a_{I:sz(I)}$ and $a_{I':sz(I')}$ are disjoint. While it is possible to craft x86 binary code that violates consistency [54], it is not typically encountered in compiled code.

7.2 model formulation

We formulate function entry point identification as a provenance classification problem. Let $\mathcal{P}$ be a program binary where $a_1, \ldots, a_n$ represent the offsets of each byte within the binary's gaps (these offsets are not necessarily contiguous). For each offset a_i, we can generate the disassembly starting at that byte. The provenance question we seek to answer is whether the binary code at a_i was produced to implement a function entry point. We use $y_1, \ldots, y_n$ to denote the labels: $y_i = 1$ if a_i is an FEP, and $y_i = -1$ otherwise. We use both idiom and structural features in this model.

7.2.1 idiom features

Idioms are well-suited for this type of provenance question because function entry points are frequently well-characterized by the specific patterns of instructions (hence the existing pattern-based approaches to FEP identification). The idiom feature function

f_ι is defined over idiom features and offsets in the binary

$$f_\iota(a, \mathcal{P}) = \begin{cases} 1 & \text{if idiom } \iota \sim \textsc{Decode}(\mathcal{P}, a) \\ 0 & \text{otherwise.} \end{cases}$$

In addition to the standard idioms we described in Chapter 5, we introduce a variant called *prefix idioms*. Like standard idioms, prefix idioms are a short sequence of instructions, possibly with wildcards; they differ in that the prefix idiom feature function matches when the instruction sequence *ends*, rather than begins, at the given offset:

$$f_\phi(a, \mathcal{P}) = \begin{cases} 1 & \text{if } \exists j \text{ s.t. } j < i,\ \phi \sim \textsc{Decode}(\mathcal{P}, a_j), \text{ and } a_i - a_j = |\phi| \\ 0 & \text{otherwise.} \end{cases}$$

Our evaluation shows that prefix idioms significantly increase the precision of FEP identification.

Using only idiom features $\mathcal{I}$ and prefix features Φ, we formulate a simple provenance model that is equivalent to logistic regression [32], as introduced in Chapter 6:

$$P(y_i | a_i, \mathcal{P}) = \frac{1}{Z} \exp\left(\sum_{\iota \in \mathcal{I}} \lambda_{y_i,\iota} f_\iota(a_i, \mathcal{P}) + \sum_{\phi \in \Phi} \lambda_{y_i,\phi} f_\phi(a_i, \mathcal{P}) \right). \tag{7.1}$$

Recall that the model parameters $\lambda_{y,\iota}$ and $\lambda_{y,\phi}$ are indexed by class label and learned, not specified. Although the definition of f_ϕ makes computing this expression for all offsets a appear to be quadratic in the size of the binary, limiting idioms to three instructions means that the prefix idiom term is a constant factor of at most 45;[1] in practice we perform far fewer disassemblies.

7.2.2 IDIOM FEATURE SELECTION

There are tens of thousands of idiom features represented in our data sets. To reduce the complexity of our models and avoid overfitting, we perform forward feature selection on the candidate idiom features (refer to Chapter 3 for details). Our training data consist of several large corpora of binaries which we describe in the evaluation, below. The distribution of actual to candidate FEPs is highly skewed towards candidates (there are many more byte locations in a binary than there are functions); using accuracy to evaluate a classifier on such a skewed data set places excessive weight on rejections of

[1] Intel x86 instructions span at most fifteen bytes [1].

non-FEPs (*true negatives*, TN), making it difficult to detect errors stemming from *false positives* (FP). We therefore use *precision*

$$P = \frac{TP}{TP + FP}$$

and *recall*

$$R = \frac{TP}{TP + FN}$$

to evaluate the classifier. Because our application domain is much more sensitive to false positives than false negatives, we use the $F_{0.5}$-measure: $F_{0.5} = 1.5PR/(0.5P + R)$, the weighted harmonic mean of precision and recall, as our objective during feature selection to emphasize precision. We reserve 20% of the data as a tuning set to decide when to stop adding features. Feature selection and parameter learning are performed separately for each of the three compilers used in the experiments, as the models are expected to vary significantly depending on the source compiler. At each iteration of the feature selection process, we select the idiom that maximizes the F-measure over the training set, recording also performance on the reserved tuning set. We terminate feature selection when adding additional idioms to the model causes decrease or only negligible increase in performance on the tuning set.

Figure 7.3 compares the tuning set performance of a models with and without prefix idioms as a function of the number of features used in the model. These curves were generated over the Intel Compiler (ICC) data set (described below), and are truncated to 35 iterations iterations; the full feature selection run for ICC binaries with prefix features enabled plateaus at 41 features. The ICC data set represents the hardest compiler in our experiments; the behavior of feature selection on other compilers is similar. The performance increase due to including prefix idiom features is quite significant. In all experiments that follow, all references to the idiom-based model include prefix idioms.

7.2.3 Structural features

We use the observations about binary code structure in the previous section to expand the model in Equation 7.1 to a complex provenance model. Although none of gap functions are reachable from the program entry point by statically resolvable calls (otherwise they would not be in the gaps, by definition), some may make statically resolvable calls to other gap functions. If a candidate FEP is targeted by a call instruction (i.e., it is the *callee*), this can be taken as an additional piece of evidence that it is actually an FEP. The binary feature function

$$f_c(a_i, a_j, \mathcal{P}) = \begin{cases} 1 & \text{if the function starting at } a_i \text{ calls } a_j \\ 0 & \text{otherwise} \end{cases}$$

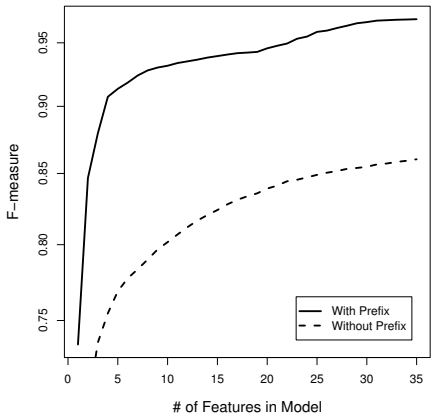


Figure 7.3: Inter-round tuning set improvement during feature selection with and without prefix idiom features. $F_{0.5}$ measure performance on the tuning set is markedly better when prefix idioms are considered in addition to standard entry point idioms.

captures the pairwise caller–callee relationship between all candidate functions; we call this the *call-consistency* feature. Similarly, our observation about disassembly overlap consistency leads us to specify the *function overlap* feature function:

$$f_o(a_i, a_j, \mathcal{P}) = \begin{cases} 1 & \text{if the functions starting at } a_i \text{ and } a_j \text{ overlap inconsistently} \\ 0 & \text{otherwise.} \end{cases}$$

With these structural features, our model is a *conditional random field* (CRF) [50], with nodes $a_{1:n}$ and pairwise connections. This is a *structured classification* problem, since the labels are correlated and must be inferred together. Using the alternative feature vector notation introduced in Chapter 6, let Λ be the parameter matrix

$$\Lambda = \begin{pmatrix} \Lambda_{+1} & \Lambda_{-1} \end{pmatrix}$$

associated with the unary idiom and prefix idiom features and let $\mathbf{x}_i$ be the feature vector of all of the idiom and prefix idiom feature functions evaluated at a_i. We define the joint probability of labels as

$$P(y_{1:n}|\mathbf{x}_{1:n}, \Lambda, \mathrm{M}, \mathcal{P}) = \frac{1}{Z} \exp\left(\sum_{i=1}^{n} \Lambda_{y_i}^T \mathbf{x}_i + \sum_{i,j=1}^{n} \mu_{c,y_i,y_j} f_c(a_i, a_j, \mathcal{P}) + \sum_{i,j=1}^{n} \mu_{o,y_i,y_j} f_o(a_i, a_j, \mathcal{P}) \right), \quad (7.2)$$

where $\mu_{o,*,*} and \mu_{c,*,*} \in$ M are the parameters associated with the binary structural features. Note that the terms containing the pairwise structural features may induce a graph with many loops.

The CRF formulation of Equation 7.2 allows us to incorporate heterogeneous structural and idiom features to define our objective. However, standard inference methods like loopy belief propagation [27] are expensive for large-scale analysis on this large (up to 937,865 nodes in a single binary in our test data sets), highly connected graph. In the following section, we describe an approximate inference procedure that considerably speeds up classification by artificially clamping the parameters $\mu_{o,1,1}$ and $\mu_{c,1,-1}$ to $-\infty$ and all other μ_* parameters to 0—effectively turning the call–consistency and function overlap features into hard constraints. The result is an efficient, approximate model that can handle such large binaries in under fifteen seconds.

7.3 Large-scale binary analysis

In a real-world setting, targets for binary analysis may be tens or even hundreds of megabytes in size. In the data sets we use for evaluation of our techniques, we have binaries that vary in size from a few kilobytes to 26MB. While not all of the contents of a binary are code, with an average of 40% of functions unrecoverable through static analysis, the sizes of the gaps remaining are significant. In one binary we may have to perform inference over nearly one million candidate FEPs. We have approached the scaling problem in two ways: by distributing the work of feature selection and model training, and by approximating the conditional random field model for efficient inference.

Because we perform idiom feature selection over the entire data set, composed of tens of millions of training examples, selecting among the tens of thousands of idioms is a costly enterprise. Fortunately, each iteration of a forward feature selection search lends itself easily to loosely coupled distributed computation. We use the Condor *High Throughput Computing* framework [55, 92] to distribute feature selection and experimental evaluation jobs to a large number of compute notes.

For each compiler-specific data set, subsets of features are distributed to each worker machine in the Condor pool. The worker selects the best feature from this subset (that is, the one with the best $F_{0.5}$-measure) and returns that value to the controlling system; all results from worker machines are synchronized at the end of each iteration. Feature selection for all three data sets consumed over 150 compute-days of machine computation, but took less than two days in real time.

Although the cost of idiom feature selection and model training is large, it is only a one time cost for a particular training data set. Much more important to our particular

application domain is the cost of inference. We efficiently approximate the complex provenance model (7.2) by breaking down inference into three stages:

1. We start with only the unary idiom features in the CRF. We train the model using the selected idioms, equivalent to model (7.1). We then fix the parameters λ_u for each idiom. Our classifier considers every candidate FEP in the gap regions of the binary, assigning to each the probability that it is an actual FEP (i.e., we compute $P(y_i = 1|\mathbf{x}_i, \mathcal{P})$).

2. We then approximate the overlap feature by computing the *score* s_i of $\mathbf{x}_i$. Initially, $s_i = P(y_i = 1|\mathbf{x}_i, \mathcal{P})$, where the probability was computed in the previous step. If $\mathbf{x}_i$ and $\mathbf{x}_j$ inconsistently overlap and $s_i > s_j$, we simply force the weaker contender $y_j = -1$ by setting $s_j \leftarrow 0$.

3. We add call-consistency. The target of a call instruction is at least as likely to be a valid function as the function that originated the call. Therefore, if $\mathbf{x}_j$ is called by $\mathbf{x}_{i1}, \ldots, \mathbf{x}_{ik}$, we set $s_j \leftarrow \max(s_j, s_{i1}, \ldots, s_{ik})$.

FEPs are considered in order of ascending address, and the last two stages are iterated until a stationary solution of s is reached. Then s_i is treated as the approximate marginal $P(y_i = 1|\mathbf{x}_{1:n}, \mathcal{P})$, and is thresholded to make a binary prediction.

7.4 EVALUATION

We tested our classifier on three separate IA-32 binary data sets, corresponding to three compilers: i) GCC: a set of 616 binaries compiled with the GNU C compiler on Linux. These were obtained in compiled form with full symbol information (indicating the location of all functions) from our department Linux server. ii) MSVS: a set of 443 system binaries from the Windows XP SP2 operating system distribution, compiled with Microsoft Visual Studio. We obtained their symbol information from the Microsoft Symbol Server. iii) ICC: a set of 112 binaries that we compiled with the Intel C Compiler on Linux, again with full symbol information.

Training data were extracted from the binaries by first copying the target binary and stripping all symbols, leaving only the main entry point of the binary as a starting point for static disassembly. We then used the Dyninst tool to parse from these starting points, obtaining a set of all functions reachable through static analysis. Dyninst's pattern-based FEP heuristic was disabled for this process. The entry points of these functions represent the positive training examples for idiom feature selection and training the weights of idiom features in our CRF. Negative examples were generated by

Table 7.1: Size of training and test sets for FEP identification, in terms of candidate FEP locations. Because the GCC and MSVS data sets were collected rather than generated, they may contain code emitted by several different versions of the compiler. We used ICC version 10.1 to generate the ICC data set.

	Training Set Examples		Test Set Examples	
Compiler	Positive	Negative	Positive	Negative
GCC	115,686	4,081,268	85,870	22,720,579
MSVS	29,710	8,025,036	70,717	13,237,424
ICC	34,229	16,893,535	47,841	13,121,646

parsing from every byte within these functions (excluding the initial byte) to generate spurious functions.

The gaps remaining in the stripped binaries after static disassembly constitute the test data. Because the original binaries had full symbol information, we have a *ground truth* to which we can compare the output of our classifier on the candidate FEPs in these gaps. The sizes of training and test sets for each compiler are listed in Table 7.1.

We automatically select idiom features separately for each of the data sets, using the LIBLINEAR package [23] to build our logistic regression–based FEP model. The number of features selected reflects the varying complexity of function entry points for binaries from each compiler. While the GCC model contains only 12 idiom features, the MSVS model contains 32 and the ICC model contains 41. The latter two compilers optimize away much of the regularity at function entry that is found in code produced by GCC.

As described above, we terminated feature selection when the $F_{0.5}$ measure failed to increase on the tuning set. To ascertain whether this stopping criterion was overly aggressive, we continued feature selection to 112 iterations on ICC, our most difficult data set. The extended model, when incorporated into our classifier, improves the AUC of the precision-recall curve by .004 for this data set. This modest improvement is probably not practically significant, so we elect to use the smaller model.

Table 7.2 lists the top five features for the two Linux data sets in the order they were selected. Although there are individual instructions common to both sets of chosen idioms, the difference between the two models reflects the difference in code generated by the two compilers. In particular, note that the first two idioms selected for the GCC model are similar to the `(push ebp | mov esp,ebp)` heuristic used by Dyninst. While this pattern is selected as a positive feature in the ICC model as well, its later selection and lower weight (not depicted here) reveal that it is a less reliable indicator

Table 7.2: Top features of FEP models. The +/- column indicates whether a particular feature is a positive or negative indicator of FEP status.

GCC		ICC		MSVS	
Idiom	+/-	Idiom	+/-	Idiom	+/-
push ebp	+	PRE: nop	+	mov edi,edi	+
* | mov ebp,esp	+	push edi	+	PRE: ret	+
PRE: daa | add | add	+	PRE: ret	+	PRE: ret n	+
PRE: nop | nop | nop	+	nop	-	int3	-
PRE: ret | lea	+	push ebp | mov ebp,esp	+	PRE: int3 | * | int3	+

of FEPs emitted by the ICC compiler.

Prefix idioms tend to have relatively larger importance on the GCC and MSVS data sets that on the ICC data set; 50% and 53%, respectively, of selected features were prefix idioms. By comparison, only 34% of the features selected from the ICC data set were prefix idioms. There are several factors that may contribute to a larger number of prefix features being selected. For example, common entry idioms that also occur relatively frequently at other points in the binary increase false positives; prefix idioms can help eliminate these types of errors. Also, when FEPs show significant variation the immediately preceding bytes may be better indicators of function location. Compilers may pad the space between the end of one function and the beginning of the next for

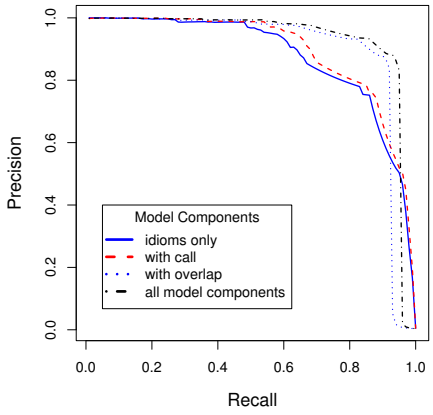


Figure 7.4: Contribution of various model components toward performance on the ICC test set.

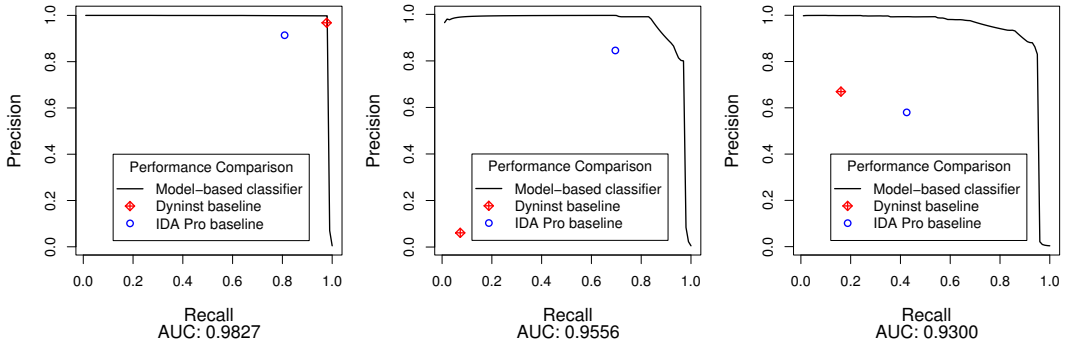
Table 7.3: Contribution of model components (idioms, call, overlap) to the classifier evaluated by the $F_{0.5}$ measure. Our classifier works best with all components enabled; in all cases it outperforms the baseline tools.

Component(s)	GCC	ICC	MSVS
Idioms	.986	.785	.893
Idioms + Call	.986	.797	.922
Idioms + Overlap	.981	.850	.894
Idioms + Call + Overlap	**.989**	**.859**	**.923**
Dyninst	.971	.326	.067
IDA Pro	.876	.517	.789

alignment purposes; this padding frequently consists of a few recognizable instructions (e.g. a series of one-byte `nop` instuctions, or longer instructions with similarly null effect, such as `mov %esi,%esi`). Our analysis suggests that the large number of prefix idioms chosen for GCC is due to the prevalence of entry idioms at non-entry locations, while the MSVS prefix idioms are largely due to the latter factor. The large number of non-prefix idioms in the ICC model reflects the preponderance of common entry sequences in these binaries, as well as the relative dearth of repeated prefix idioms.

We implemented our classifier as an extension to the Dyninst tool, replacing the heuristic function detection functionality with our structured classifier. Our implementation allows us to individually enable components of the model (idiom, call, and overlap features). Figure 7.4 depicts the relative contribution of each model component on the ICC test set (other data sets have similar behavior). Both of the structural features increase the area under the curve, but the contribution from the overlap feature is greater. Table 7.3 shows the $F_{0.5}$ measure for each model component and the full model. In all cases, adding structural information increased this measure on the test set.

To calibrate our classifier's performance, we obtained baseline results from two existing tools that attempt to find functions in the gaps of stripped binaries. Unaugmented Dyninst can scan for simple known entry patterns, as can the IDA Pro tool, the industry standard in interactive disassembly tools. IDA Pro is best suited for use on Windows binaries, using a signature package to identify library code that has been linked into binaries. In all of our experiments, the baseline tools had access to the same stripped binaries that our classifier did. In the case of the Windows binaries, we explicitly disabled automatic retrieval of symbol information from the Microsoft Symbol Server. We graphically compare the Dyninst and IDA Pro tools to our classifier in Figure 7.5. In each case, our implementation of the classifier dramatically outperforms



(a) GCC Data Set (b) MSVS Data Set (c) ICC Data Set

Figure 7.5: Precision-recall curves comparing our model and the baseline classifiers. The GCC data set exhibits the most regular FEPs and is the easiest. Poor performance of the Dyninst baseline on the MSVS data set was due to a mismatch between Dyninst's heuristic patterns and the MSVS compiler's output.

existing tools.

7.5 SUMMARY

We have evaluated the use of a program provenance recovery framework for improving the discovery of code in stripped binaries. By treating the characteristics of *function entry points* as a provenance property, we are able to construct a provenance model that can automatically learn to distinguish FEPs from other locations in the binary, allowing analysis of previously unreachable code. Our model is based on both idiom and structural features, including a specialized *prefix* idiom feature function that extends the idioms introduced in Chapter 5. We formalize this model as a conditional random field and describe an approximate inference algorithm for efficient computation in the large, loopy graphs defined by the model. A parser based on this model can quickly classify many candidate FEPs in large program binaries. Unlike previous approaches to parsing stripped binaries, our parser does not rely on hand-coded patterns. Our extensions to the Dyninst tool allow it to adapt to future variations in compiler-emitted code, as well as code that does not conform to expected patterns. Experiments show that our CRF formulation with both idiom and structure features performed well on a large number of real-world data sets, outperforming existing, standard tools.

The feature selection methodology described in this chapter differs from that used

in those chapters that follow. For this experiment, we performed a standard *forward feature selection* process, in which the set of model features is built up, one at a time, by choosing the feature that offers the best performance on a tuning data set. In later chapters we abandon this principled approach in favor of a significantly less expensive, albeit heuristic, feature selection process. Our decision has been driven by two factors:

1. the expense of querying training data (and thus of parameter estimation and model evaluation) for more complex feature types is greater, and

2. as we will show, function entry point characteristics are a particularly easy provenance property to model; in later provenance studies, we use thousands of features, making even distributed forward feature selection intolerably slow.

The FEP models we produce for stripped binary parsing are tied to a specific compiler, which raises the question of how to choose which model to use when analyzing a binary of unknown provenance. The variations in idiom features among the three data sets used here suggests that different compilers produce markedly different code, at least at function entry points. Building provenance models that can determine the properties of the compiler toolchain is the subject of the following chapter.

8

The Production Toolchain

In the previous chapter, we showed how our binary code modeling techniques can be used to recover the compiler-specific patterns of instructions at function entry points. In this chapter, we turn our attention to the compiler toolchain itself, developing new models to recover detailed components of the compiler toolchain, such as the specific version of compiler and the optimization level of the code. Fine-grained compiler details could augment crash reports for bug detection in distribution scenarios where vendors do not control the compilation environment for the software or external dependencies, such as open source projects. Higher level program properties like the original source code language can assist in reverse engineering, decompilation, and other binary analyses that are tailored to specific languages [15, 72, 75]. From a security perspective, program provenance reflects the set of explicit and implicit user choices made in producing a program and is directly relevant to investigators in the field of digital forensics [64].

In this chapter, we formulate several simple and complex provenance models, based respectively on the machinery of support vector machines [18] and conditional random fields [58]. These models allow us to address several different program provenance questions, and to explore trade-offs between model complexity and program representation on real-world binaries. Following a section that describes the toolchain provenance problem domain, we turn our attention to the following topics:

- We formulate a *sequence-based* provenance model that labels the bytes underlying a binary according to the most likely *compiler family* that generated them. This model is also capable of distinguishing between code and data; we describe an experiment that uses the inferred source compiler to implement a compiler-agnostic version of the stripped binary parsing tool that we describe in Chapter 7.

- We introduce models that capture detailed toolchain provenance at *function-level* granularity, allowing us to recover the source language and specific compiler

```
int bar(int foo) {
    int i, j;

    for(i=0;i<foo;++i) {
        i = j + i;
        j *= i;
    }
    return j;
}
```

(a) source

```
test   edi,edi
jle    4004ae <bar+0x16>
mov    eax,0x0
lea    eax,[rdx+rax]
imul   edx,eax
add    eax,0x1
cmp    edi,eax
jg     4004a1 <bar+0x9>
mov    eax,edx
ret
```

(b) GCC 4.4

```
xor    edx,edx
test   edi,edi
jle    400989 <bar+0x11>
add    edx,eax
imul   eax,edx
inc    edx
cmp    edx,edi
jl     40097e <bar+0x6>
ret
```

(c) ICC 11

Figure 8.1: Comparing the assembly generated by two different compilers. Both compilers were run at their 'high' optimization levels. The assembly displays differences in idioms for adding two variables, ordering of independent operations, and incrementing counter variables. The expressiveness of the x86 instruction set allows compilers great flexibility even in such small code snippets.

version and optimization levels used to produce binary code. We develop algorithms for efficiently extracting *graphlet-based* features of program control flow to incorporate in these models.

- We evaluate provenance recovery on a large set of real-world software across several compiler families, versions, optimization levels and source languages. Our results show that toolchain provenance can be recovered accurately (approaching 100% for some components) even when the code distinctions between component variations are extremely subtle, or where the binaries contain code of mixed provenance.

8.1 Problem Domain

Toolchain provenance encompasses those parts of the production process that transform the program from source code to an executable binary. These transformations determine the form and contents of the resulting binary code; identifying those characteristics that are strongly related to particular components should allow us to infer the composition of the toolchain process. The crucial role of the compiler toolchain in producing a binary makes it likely that toolchain components leave a strong signal in the code; Figure 8.1 illustrates how two different compilers can produce markedly different assembly from a simple code snippet.

The approach we take depends on how detailed we want the provenance properties to be, and at what level of binary code abstraction we wish to model provenance. We have developed two toolchain provenance models: one that represents the compiler family that generated code to facilitate compiler-agnostic stripped binary parsing, and one that recovers details of the compiler version and optimization level as well as the program's source language.

8.1.1 SOURCE COMPILER RECOVERY

Recall from Chapter 7 that when parsing stripped binaries, the primary challenge is to find the offset at which a code sequence begins. If we know the identity of the compiler, we can model the patterns of *function entry points*; however, compiler identity may not be available and programs may contain code produced by multiple compilers. Predicting the compiler for different regions of the program would allow compiler-specific stripped parsing techniques to be applied to unknown or mixed-provenance binaries.

Our objective is to accurately label the source compiler of subsequences of the binary. We assume that binaries are composed of interleaved sections of either code produced by one or more compilers or of non-code in the form of data or random bytes. For example, a binary containing statically linked code from several libraries might contain code from both the Intel C compiler (icc) and the GNU C compiler (gcc), as in Figure 8.2. We develop a sequential model of compiler provenance that classifies

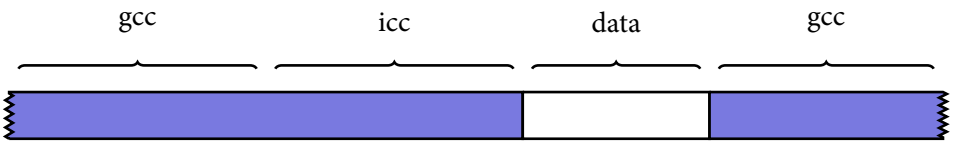


Figure 8.2: Program model for byte-level source compiler recovery with multiple compilers.

bytes as code produced by a particular compiler or as non-code, and introduce model components that lead to consistent labeling of code regions.

The source compiler recovery problem has the following characteristics, which we make more concrete in the modeling sections below:

- compiler family–level provenance property
- provenance assignment at byte-offset granularity

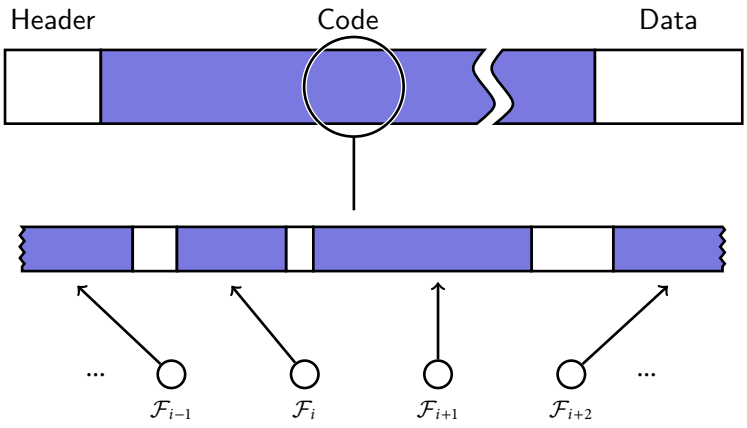


Figure 8.3: The *function-provenance* abstraction over a typical binary code artifact, such as a Linux ELF binary. Header information is only used to find functions if available. The binary code is represented as a linear sequence but is not necessarily contiguous: there may be gaps containing non-executable data, padding, or random bytes interspersed among machine instructions. Note that such gaps may even exist between the basic blocks of a single function.

- identification of data in code regions
- sequential and control flow–based consistency

8.1.2 Detailed toolchain provenance

The detailed toolchain provenance model extends set of toolchain provenance properties to include the specific compiler version and the degree of code optimization, as well as the source language. Because this model is not motivated by the stripped binary parsing problem, we adopt a function-level abstraction for provenance recovery, as depicted in Figure 8.3. This representation has the advantage that it entails significantly less computation than modeling every byte of code explicitly, and it is a more *consistent* representation from a provenance standpoint: functions are the smallest unit of output for the compilation toolchains that we consider. The model remains flexible enough to represent binaries of mixed provenance, though by trading off the power of intra-function provenance labeling (e.g., modeling data interleaved with code).

The detailed toolchain provenance problem has the following characteristics:

- compiler family, compiler version, optimization level, and source language prove-

nance properties

- provenance assignment at function granularity
- control flow–based consistency (implicit)

In the following sections, we describe the models for both toolchain provenance problems.

8.2 SEQUENTIAL COMPILER MODEL

The properties of binary code make probabilistic graphical models well-suited to the compiler inference task. Modeling both interleaved code and non-code requires a common representation of the features that describe each byte in the binary. Subsequences of the binary containing executable instructions should be labeled consistently: adjacent instructions are likely generated by the same compiler. This consistency extends beyond immediate neighbors; we expect code connected through *intraprocedural* control flow (i.e., branches) to originate from the same compiler. This combination of independent local features and dependency relationships between adjacent and distant labels encourages viewing compiler inference as a structured classification problem.

The program binary representation must capture the characteristics of the code and the non-code regions of the binary. While abstracting the program as a sequence of executable instructions is most natural, it is not sensible to apply labels to non-code regions consisting of data or random bytes as though they contain instructions. Furthermore, doing so requires statically identifying all instruction boundaries (i.e., parsing), itself a challenging task. We therefore model the binary uniformly as a sequence of bytes representing executable instructions intermingled with non-code.

We use idiom features to characterize the binary at each byte offset. We chose idiom features because, unlike graphlets or other high-level features, they require only local interpretation of the program bytes (rather than construction of a control flow graph), making them suitable for describing data bytes as well. Byte N-grams are an alternative that we did not investigate; we chose idioms over N-grams due to the way the former hide details like immediate operands, and to ease integration of the source compiler model with function entry point detection, as we describe in Section 8.4.1.

Let the program binary $\mathcal{P}$ be a sequence of bytes at offsets $a_1, \ldots, a_n$. Our task is to assign labels $y_1, \ldots, y_n$, where each $y_i \in \mathcal{Y}$ corresponds to a particular source compiler (e.g., `gcc`, `icc`, or `msvs`) or the special '`data`' label. We model the compiler label probability over the entire binary as a *conditional random field* [50] with nodes $y_{1:n}$ representing the labels of every byte in the program.

Each node is associated with one or more idiom features $\iota \in \mathcal{I}$ as indicated by the idiom feature function

$$f_\iota(a, \mathcal{P}) = \begin{cases} 1 & \text{if idiom } \iota \sim \textsc{Decode}(\mathcal{P}, a) \\ 0 & \text{otherwise.} \end{cases}$$

Because the binary consists of subsequences of bytes produced by individual compilers (or non-code bytes), each node y_i is connected to its neighbors y_{i-1} and y_{i+1} using the adjacency feature function

$$f_{ADJ}(a_i, a_j, \mathcal{P}) = \begin{cases} 1 & \text{if } |a_i - a_j| = 1 \\ 0 & \text{otherwise.} \end{cases}$$

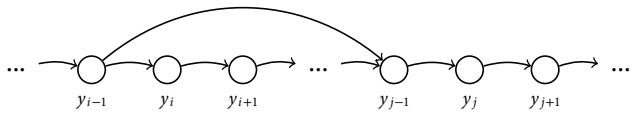
Taken together, the unary idiom feature functions $f_{\iota \in \mathcal{I}}$ and the binary adjacency feature f_{ADJ} define a *linear chain CRF* [58] over the binary, as illustrated in Figure 8.4.

Recall from Chapter 6 that the conditional probability of each label node is determined by the idiom features present at that node and by the labels of adjacent nodes. We define the joint probability of these labels (eliding the program parameter $\mathcal{P}$ for clarity) as

$$P(\{y\}_{1:n} | \{\mathbf{x}\}_{1:n}, \Lambda, \mathrm{M}) = \frac{1}{Z} \exp\left(\sum_{i=1}^{n} \lambda_{y_i}^T \mathbf{x}_i + \sum_{i=1}^{n} \sum_{j=1}^{n} \mu_{ADJ, y_i, y_j} f_{ADJ}(a_i, a_j) \right) \tag{8.1}$$

where $\mathbf{x}_i = (f_{\iota 1}(a_i), \ldots, f_{\iota |\mathcal{I}|}(a_i))^T$ is the idiom feature vector for offset a_i and Λ, M are the sets of weight parameters associated with the feature functions.

Formulating our model as a linear chain CRF is attractive because it fulfills most of our modeling requirements while allowing tractable parameter estimation and inference, which is not true of general structure graphical models. However, the model in (8.1) does not capture the intraprocedural labeling consistency that we expect from compiled code. That is, the compiler label assigned to a branch instruction has no impact on the label assigned to its target. We therefore extend our model with long range edges between intraprocedural control flow instructions and their targets.



We define a new binary feature function

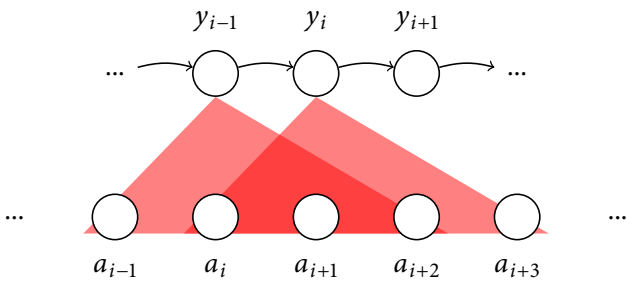


Figure 8.4: Byte labeling as a linear-chain conditional random field. Each byte position a_i in the binary is paired with a label node y_i indicating its source compiler. The label nodes are associated with *idiom features* that represent the instructions spanning a range of bytes beginning at a_i (the shaded areas). The bytes constituting idiom features overlap for nearby label nodes. The model captures the association both between idioms and labels and between adjacent labels.

$$f_{CF}(a_i, a_j, \mathcal{P}) = \begin{cases} 1 & \text{if in the CFG } G \text{ of } \mathcal{P}, \text{ there are blocks } b_i, b_j \text{ s.t.} \\ & a_i \in b_i, a_j \in b_j \text{ and } b_i \rightsquigarrow b_j \\ 0 & \text{otherwise} \end{cases}$$

to encode these branch consistency constraints (recall that $\rightsquigarrow$ indicates reaching control flow). Adding this feature to the model of (8.1) yields the final model

$$P(\{y\}_{1:n}|\{\mathbf{x}\}_{1:n}, \Lambda, \mathrm{M}) = \frac{1}{Z} \exp\left(\sum_{i=1}^{n} \lambda_{y_i}^T \mathbf{x}_i + \sum_{i=1}^{n}\sum_{j=1}^{n}\left[\mu_{ADJ,y_i,y_j} f_{ADJ}(a_i, a_j) + \mu_{CF,y_i,y_j} f_{CF}(a_i, a_j)\right]\right) \quad (8.2)$$

where the weights μ_{CF,y_i,y_j} are expected to be strongly positive for $y_i = y_j$ and strongly negative for $y_i \neq y_j$. Note that with the introduction of of the control flow edges, this model no longer has the form of a linear chain CRF with its efficient parameter estimation and inference algorithms. In practice, we use model (8.1) and heuristically approximate the branch consistency component of this model as a hard constraint, as we describe in Section 8.4.1. This approximation is sensible as we expect the labeling consistency across intraprocedural branches to hold under all but the most highly contrived circumstances.

8.3 Detailed compiler provenance model

The models we use for detailed toolchain provenance classification are derived from those used in the previous section, but incorporate more binary code features and different provenance labels; furthermore, they assign provenance at function, rather than byte, granularity. To be precise, let a program binary $\mathcal{P}$ be a sequence of functions $\mathcal{F}_1, \ldots, \mathcal{F}_k$ ordered by their entry addresses $a_1, \cdots, a_k$. The task of the classifier is to assign labels $y_1, \cdots, y_k$ to the sequence of functions, where each $y_i \in \mathcal{Y}$ is the identity of some provenance component (such as source language) or set of components (such as both compiler family and version), depending on the model formulation. Classification can be applied to a function $\mathcal{F}_i$ independently of any others (predicting y_i) or *jointly* over the entire binary sequence (predicting $y_1, \cdots, y_k$).

Toolchain components influence not only the instructions that make up a program, but also the way those instruction combine to form the control flow graph. We use idiom, instruction summary graphlet, and branch graphlet feature functions to model toolchain details. The idiom feature functions are equivalent to those defined in the previous section, but evaluated at the function level as we describe in Chapter 6. The summary graphlet feature function is defined over the set of summary graphlets $s \in G_S$:

$$f_s(\mathcal{F}_i) = \begin{cases} 1 & \text{if } s \in \textsc{MatchGraphlets}(\mathcal{F}_i, G_S)^1 \\ 0 & \text{otherwise.} \end{cases}$$

Our hope is that summary graphlets can capture differences in the arrangement of code without being sensitive to the use of particular instructions, thereby avoiding redundancy with the idiom features. Branch graphlet feature functions, which encode the specific instructions used to implement branches, are defined over the set of branch graphlets $b \in G_B$:

$$f_b(\mathcal{F}_i) = \begin{cases} 1 & \text{if } s \in \textsc{MatchGraphlets}(\mathcal{F}_i, G_B) \\ 0 & \text{otherwise.} \end{cases}$$

We use these feature functions to form two types of detailed toolchain provenance classifier, depending on whether we incorporate program structure into the model.

8.3.1 Independent classification

Since functions are the smallest unit of code that can be associated with a particular toolchain component—for example, a single C-language function could be compiled

[1] Recall from Chapter 5 that $\textsc{MatchGraphlets}(F, G)$ returns graphlets $G' \subseteq G$ found in F.

and linked into a binary comprising mostly C++ code—we first model provenance over individual functions. This model assumes that functions are statistically independent, and uses as evidence the feature vectors we described in the previous section. Each feature is associated with a parameter, and the learning task is framed as choosing parameter values that minimize a *loss function* that measures the fit between the model and the data. There are many probabilistic and non-probabilistic models that are applicable to this problem formulation. We use linear support vector machines (SVMs) due to their good performance on high-dimension data sets and the availability of a robust implementation.

SVMs operate by finding a weight vector $\mathbf{w}$ that defines a *decision boundary* in the feature space that best separates two different classes; the distance from a particular example to that boundary is the *margin* and is defined as $\mathbf{w}^T\mathbf{x}$, where $\mathbf{x}$ is the feature vector. The weight vector in SVMs plays the same role as the parameters Λ of the probabilistic models we have previously described. In such a binary classifier, an example is assigned to class +1 or −1 depending on the sign of the margin. Such a classifier can be extended to K classes through a simple procedure:

1. Train K weight vectors $\mathbf{w}_1 \cdots \mathbf{w}_K$ by repeatedly partitioning the data into two groups: one for the current class, and one for everything else.

2. For each input to the classifier, choose the label $k \in [1, K]$ subject to

$$\arg\max_k \mathbf{w}_k^T \mathbf{x}.$$

We use the LIBLINEAR linear support vector machine library [23] to independently model function provenance. We scale the values of each feature across all functions to the interval $[0, 1]$; scaling prevents frequently occurring features from drowning the contribution of rarer ones. As we discuss in the evaluation section, this model can accurately recover some provenance components, but is outperformed on others by more sophisticated models.

8.3.2 JOINT CLASSIFICATION

While our provenance recovery techniques are designed to accommodate binaries that contain code of different provenance, our intuition is that there should be a good deal of provenance consistency from one function to the next: compilation units (source files) rarely consist of single functions. To capture this expected local consistency, we introduce a simple notion of *adjacency* into our feature representations: two functions within a binary are considered adjacent if they are adjacent in the ordering imposed

by the function offsets returned by the binary parser. Clearly this is a weak definition of adjacency—two functions could be separated by megabytes of data and still be considered adjacent—but our evaluation shows that it is a powerful tool for improving provenance models.

We can incorporate adjacency into the model using the idiom and graphlet features we have previously defined and introducing an adjacency feature function

$$f_{ADJ}(\mathcal{F}_i, \mathcal{F}_j) = \begin{cases} 1 & \text{if } \mathcal{F}_i \text{ is adjacent to } \mathcal{F}_j \\ 0 & \text{otherwise,} \end{cases}$$

introducing only an additional $|\mathcal{Y}| \times |\mathcal{Y}|$ parameters over those required by the independent functions model. We have found that parameter estimation for such models converges slowly in practice, possibly due to the contribution of the vastly more unary feature terms dominating the objective function during both inference and optimization. Instead, we introduce a *modified idiom feature function*

$$f_\iota(\mathcal{F}_i, \mathcal{F}_j) = \begin{cases} 1 & \text{if } \iota \sim \textsc{Decode}(\mathcal{F}_i) \text{ and } \mathcal{F}_i \text{ is adjacent to } \mathcal{F}_j \\ 0 & \text{otherwise} \end{cases}$$

to test for adjacency; the graphlet features are modified similarly. Importantly, the actual test for idiom existence still only evaluates one function. This formulation *multiplies* the putative number of model parameters by $|\mathcal{Y}| \times |\mathcal{Y}|$; we have not found this to be a problem in practice, as many combinations of provenance labels are unsupported in our data sets. The trade-offs between this formulation and the previous may be different if the target data set contains many programs in which the majority of provenance properties are represented; in that case, the explosion in parameters may outweigh the improved rate of convergence of the model based on modified idiom features.

8.3.3 Joint model structure

So far we have said little about the nature of the provenance labels $\mathcal{Y}$ and how they relate the components of the toolchain: source language, compiler family, compiler version and code optimization level. Every function in a program has a specific combination of provenance corresponding to each of these components. If we allow each component to take on a set of values—$\mathcal{S}$ for source language, $\mathcal{C}$ for compiler, $\mathcal{V}$ for version, and $\mathcal{O}$ for optimization—then a natural choice is to define labels as tuples $\langle s, c, v, o \rangle$. If we train classifiers using unique tuples as classes, then the adjacency feature induces a linear-chain conditional random field.

As we have mentioned previously, linear-chain CRFs are useful because they allow efficient inference. However, fixing labels to a particular combination of provenance

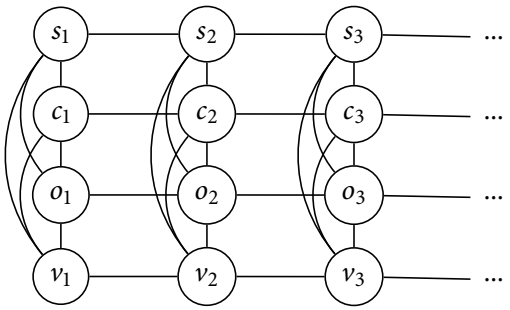


Figure 8.5: A conditional random field with a grid structure. Data nodes are omitted for clarity.

components can be problematic; it can be difficult to interpret classifier output when a subset of components are ambiguous. If we allow each of the toolchain components to be labeled independently this problem is ameliorated, at the cost of increased model complexity. Such a CRF can be visualized as a collection of linear sequences where cotemporal label nodes are fully connected, as depicted in Figure 8.5. This general CRF corresponds to the model

$$P(\{y\}_{1:n}|\{\mathbf{x}\}_{1:n}, \mathrm{M}) =$$
$$\frac{1}{Z}\exp\left(\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{\ell\in\mathcal{L}}\left[\mu_{ADJ,\ell_i,\ell_j}\cdot f_{ADJ}(\mathcal{F}_i,\mathcal{F}_j)+\sum_{\iota\in\mathcal{I}}\mu_{\iota,\ell_i,\ell_j}\cdot f_\iota(\mathcal{F}_i,\mathcal{F}_j)\right]+\sum_{i=1}^{n}\sum_{\substack{\ell\in\mathcal{L}\\ \ell'\neq\ell}}\mu_{p,\ell_i,\ell'_i}\right) \tag{8.3}$$

that uses a set of terms over $\mathcal{L} = \{s, c, v, o\}$ to represent the individual provenance components. The inner summation over $\ell \in \mathcal{L}$ computes the contribution of the idiom features and the adjacency features for each provenance property (the horizontal connections in Figure 8.5); the last term computes the relationship between the provenance properties for each function (the arcs). Exact inference in this kind of loopy graphical model is intractable in general; nonetheless good approximate inference algorithms are known, and our evaluation suggests that such approximations are appropriate for provenance recovery.

8.4 Evaluation

We evaluated our toolchain provenance recovery techniques on several corpora of real-world program binaries generated from software written in several programming languages and compiled with various toolchain components. Our evaluation shows that:

- Byte-level provenance models can effectively differentiate between code and data, precisely identifying the locations of the binary that correspond to each. The compiler sequence model achieves 92.5% accuracy on our test set. Furthermore, these results generalize to the case of programs of mixed provenance; on a different data of programs containing code produced by two compilers, our classifier assigns the correct provenance 93% of the time.[2]

- Compiler provenance models can augment the stripped binary parsing task that we introduced in Chapter 7. Inferring the source compiler allows automatic application of compiler-specific function entry point models, and reduces FEP identification error rates by 18%.

- The provenance models we define in this chapter effectively capture the characteristics of finer-grained toolchain properties like compiler version. We achieve classification accuracy of 80% when all component labels are predicted jointly; individual provenance recovery accuracy for source language, compiler family, and code optimization level exceeds 95%.

In the following sections, we evaluate the compiler sequence model and the detailed provenance models independently, describing the data sets, evaluation methodologies, and experimental results of each.

8.4.1 Compiler sequence model

We evaluate the compiler sequence model in terms of three criteria: how accurately it can differentiate between code and data, how accurately it assigns compiler labels in the case of mixed-provenance binaries, and the extent to which it improves the FEP identification model from Chapter 7.

[2] The single-compiler and multiple-compiler data sets comprise different programs; classifier accuracy on these data sets is not directly comparable.

Evaluation Data Set

We use two corpora in our evaluation. The first is an expanded version of the corpus we use in Chapter 7, consisting of 1,285 binaries from three different compilers: 616 from the GNU C Compiler (GCC), 226 from the Intel C Compiler (ICC), and 443 from the Microsoft compiler (MSVS). We use this corpus for both training and testing. The second data set was artificially constricted to evaluate our technique on the more difficult case of binaries with mixed provenance. These binaries interleaved code from the GCC and ICC compilers to simulate programs that have been statically linked against libraries with varying provenance, such as may occur when using commercial or legacy libraries. We discuss the generation of this data set below; this data set is used only to test the classifier.

All of the binaries in the first data set are assumed to be generated by a single compiler, so we refer to it as the single-compiler data set. This assumption may not hold for the ICC programs; these programs sometimes contain statically linked code that was produced by GCC, as an artifact of their compilation on systems with GCC-compiled system libraries. We have attempted to annotate this code, but it is possible that our annotations are imperfect. The noise introduced into our training and testing sets due to incorrect annotations may artificially decrease our reported results, but will never cause us to overestimate the accuracy of provenance recovery.

We extract representations suitable for use in our models as follows. We first exhaustively disassemble each program using the Dyninst binary analysis and instrumentation tool [37, 65]. Exhaustive disassembly decodes an instruction at each byte offset, rather than following a linear chain of instructions or traversing the instruction control flow. Each position in the binary is then described by the idiom feature abstractions occurring at that point. We establish *ground truth* labels at each point by parsing the binaries in our corpus using traditional recursive traversal parsing with full symbol information. The ground truth label at each point is either the particular compiler (for bytes constituting executable instructions) or the `data` label for all other bytes.

Methodology

In principle we would like to use as much training data as possible, as doing so leads to more precise parameter estimates. However, the scale of our task constrains how much data we can use, given that a single example (that is, a binary) can comprise millions of idiom features. The primary limiting factor for our experiments is memory usage. To keep space requirements and training time reasonable, we randomly select a subset of 20 binaries from each compiler, reserving the remainder for evaluation.

While additional training data could be used (at the cost of additional resources), the improvement would likely be subject to diminishing returns.

There are over two million unique idiom features represented in our training data. While the power of the CRF model that we selected is in its ability to represent many features, many of these features are redundant or have little predictive power for our compiler inference task. We perform a simple *feature selection* process to reduce the number of idiom features in the model. The goal of feature selection is to choose the features that give the model the most predictive power—those that appear very frequently in one compiler class, for example, and not others. Our process makes use of the *mutual information* between idioms $\mathcal{I}$ and compiler labels $\mathcal{Y}$. We select the 20,000 features with the highest compiler mutual information.

We train the parameters of the linear-chain model (8.1) with the MALLET package [59]. MALLET is a Java-based framework that supports parameter estimation and inference in linear-chain conditional random fields. Parameter estimation over the 60 training binaries takes approximately 220 minutes on a 2.27GHz Intel Xenon workstation. Training is a one-time cost in our compiler provenance inference system.

We begin testing by using MALLET to label each byte in the binary with the most likely compiler label, one of `gcc`, `icc`, `msvs`, or the special `data` label using the parameters estimated during training of the linear chain CRF given by Equation 8.1. We then heuristically approximate the control flow consistency component of Equation 8.2 by propagating compiler labels across control flow as determined by the Dyninst tool. This approximation is equivalent to setting the control flow consistency weight μ_{CF,y_i,y_j} in Equation 8.2 to $-\inf$ for all $y_i \neq y_j$. Each inferred label is compared against the ground truth labeling to determine accuracy of our technique.

Single-Compiler Evaluation

The binaries in the single compiler testing set are drawn from the same corpus as the training binaries. Each binary is assumed to contain code generated by a single compiler, except for a limited portion of the ICC data set as noted above; the idiom feature representation and ground truth labels are also obtained as previously described. Statistics for this data set are listed in Table 8.1.

Compiler provenance label accuracy over this data set is 0.925. The accuracies over binaries produced by each compiler are listed in Table 8.2. In addition to labeling accuracy, we list error rate broken down by the type of error: 'compiler-compiler' for erroneously labeling the source compiler of a byte, 'compiler-data' for bytes labeled `data` incorrectly, and 'data-compiler' for data bytes labeled as originating from a compiler. While the test set accuracy is a good metric for evaluating compiler provenance

Table 8.1: Programs and code-data breakdown for one random fold of the single-compiler experiment.

Compiler	Binaries	Code bytes	Data bytes
GCC	559	32,102,222	9,030,390
ICC	174	14,490,581	5,265,195
MSVS	386	15,952,368	5,045,413

Table 8.2: Single compiler evaluation. Error rates are computed over the relevant subset of bytes (e.g., 'data-compiler' is the error rate over all ground truth `data` bytes). The error rates for different types of labeling errors may have greater or lesser impact depending on the use of inferred compiler provenance.

		Error rate		
Compiler	Accuracy	compiler-compiler	compiler-data	data-compiler
GCC	.932	.044	.001	.147
ICC	.969	.006	.000	.101
MSVS	.870	.036	.035	.315

inference, the type of error may have more or less impact depending on applications of this provenance.

For example, if compiler provenance is used to group programs by toolchain characteristics or to distinguish library code from program code, mislabeling code as data or vice versa may be less important than mislabeling the specific compiler. We discuss the impact of labeling errors further in the case study of FEP identification, below.

As illustrated by the results in Table 8.2, mislabeling data as code constitutes the majority of labeling errors, particularly on the MSVS data set. The reasons for this are twofold. First, there is a fundamental disparity between the number of 'code' and 'data' bytes in program binaries; in our data sets the ratio falls between 1.8:1 and 4.2:1 for most programs. This population disparity manifests itself as a bias in the learned models.

The higher error rate for the MSVS data set is likely due to noise in our testing data. The public symbol files we use to parse the ground truth for the MSVS data set are partially stripped to hide type information, resulting in potentially incomplete parses of these binaries. It is likely that some regions of the binaries with ground truth label `data` are incorrect, contributing to the inflated error 'data-code' error rate. This noise in the ground truth can only lead to under-reporting of classifier accuracy, both by

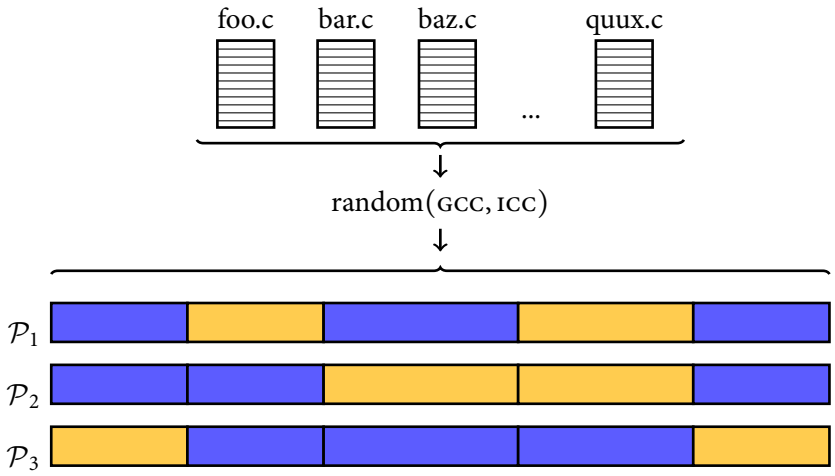


Figure 8.6: Generating mixed-provenance binaries. Each source file is compiled by GCC (■) or ICC (■) at random and linked to form the program binary $\mathcal{P}_k$. Debugging information in the binary associates functions with a source file and compiler.

causing us to count a correctly labeled region as incorrect and by polluting the training data with mislabeled examples, making parameter estimation harder. Complete debug information was available for the other data sets, obviating this potential risk.

Multiple-Compiler Model

To evaluate our compiler provenance inference techniques in a true multiple-compiler setting, we implemented a build system that can generate binaries containing code from two or more compilers. We replace the compiler in standard Makefile-based build environments with a utility that invokes a compiler chosen at random; for this evaluation we chose the GCC and ICC compilers. Our utility records which compiler was used for each source file. After the build process completes, we extract mappings from functions to source files from the compiler-emitted DWARF debugging information. Merging this mapping with the source file record, we derive precise ground truth provenance for multiple compiler binaries as illustrated in Figure 8.6. One limitation of this approach is that statically linked library code or other system code may not be directly associated with the compiler record generated by our utility. We therefor omit all regions of the binary for which we lack precise ground truth provenance for testing.

We constructed a test set of 10 program for evaluation from the GNU `coreutils` distribution. Each program was compiled 10 times with the random compiler system.

Table 8.3: Multiple compiler evaluation. Each program was compiled 10 times with GCC or ICC randomly chosen for each source file. The chosen compiler has an impact on code size and the availability of debug information used for ground truth labeling, contributing to the variability of label accuracy depicted by the box-and-whisker plot. The box shows the range between the first and third quartile of the data values, with the interior line indicating the median value. The whiskers extend to cover the data range.

		Label Accuracy		Average error rate		
Program	Source files	Average	Spread	C/C	C/D	D/C
			0.74 0.99			
chcon	38	0.981		0.012	0.006	0.023
cp	74	0.893		0.126	0.007	0.019
date	25	0.907		0.109	0.008	0.046
df	48	0.972		0.021	0.010	0.015
dir	33	0.977		0.015	0.006	0.028
du	52	0.934		0.064	0.006	0.062
mv	64	0.887		0.129	0.008	0.022
sort	44	0.899		0.013	0.003	0.213
stat	20	0.961		0.015	0.021	0.048
tail	29	0.971		0.022	0.005	0.033

Compiler provenance labels were applied to each binary using the same model and procedure described in the previous section, with those regions of the binary with unknown provenance omitted. Compiler provenance label accuracy over the entire data set was 0.938. Table 8.3 summarizes labeling accuracy and error types over this data set.

The accuracy of our compiler provenance inference techniques for both the single- and multiple-compiler data sets demonstrates the feasibility of extracting provenance from program binaries. The low rate of mistaken compiler errors—where executable code created by one compiler is assigned the label of another—allows us for the first time to attribute sequences of code to a particular compiler. In the following section, we present a case study that uses inferred compiler provenance to extend a previously compiler-specific approach to stripped binary parsing.

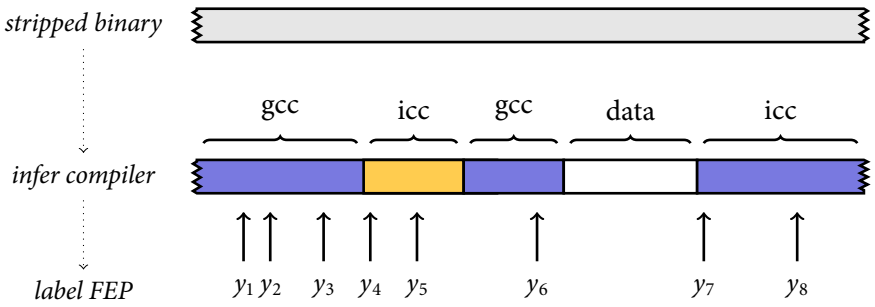


Figure 8.7: Compiler-agnostic FEP identification. We first infer the source compiler for regions of the binary, then apply an FEP identification model that is parameterized by the source compiler to find function entry points.

Stripped Binary Parsing

Our provenance-based approach to stripped binary parsing in Chapter 7 relied on knowledge of the source compiler to choose the correct function entry point (FEP) model to search for code. The compiler provenance model allows us to apply the FEP model to stripped parsing without knowing the source compiler *a priori*. We augment our FEP identification system by splitting the problem into two tasks, as depicted in Figure 8.7: first, we use the compiler sequence model to label the program regions according to the most likely source compiler, and then we use a modified FEP model to label function entry points.

Recall that the FEP model assigns labels $y \in \{-1, +1\}$ to program offsets based on idiom features $\iota \in \mathcal{I}$, and that the model structure was defined by the control flow of the program (approximated heuristically). We expand the space of idiom parameters $\lambda_{y,\iota}$ to include the extra dimension $\ell \in \mathcal{L}$, where $\mathcal{L}$ are the compiler labels assigned by the compiler sequence model:

$$\Lambda = \{\lambda_{y,\ell,\iota}\}.$$

Conceptually, this model incorporates a different set of parameters for each possible compiler label.

The procedure for incorporating compiler provenance inference into FEP identification is to (a) learn weights $\lambda_{y,\ell,\iota}$ using *ground truth* compiler labels over training binaries; (b) label test binaries with inferred compiler provenance; and (c) label function entry points using the augmented FEP model. We perform feature selection and training as described above.

We evaluated the provenance-augmented FEP identification tool on 972 of the binaries in our data set. Because this is a two-class classification task, evaluation in

terms of *precision* and *recall* for the positive entry class is appropriate. Furthermore, the populations of the entry and non-entry classes are extremely skewed, as there are many more bytes than functions in a binary. In our data set there are 256,832 FEPs out of 84,188,324 bytes. Accuracy is a poor metric for classifier performance in such a skewed data set, as it can be very high while admitting many false positive function identifications. We therefore adopt the commonly-used F_1 measure, which represents the harmonic mean of precision and recall.

Table 8.4 summarizes the results of our experiments. In addition to comparing FEP identification with and without compiler provenance inference, we also evaluated the tool with the ground truth compiler labels to quantify the maximum contribution of provenance inference for this task. We obtain a modest increase in the F_1 measure when using the inferred provenance labels. Though slight, this increase is statistically significant across our sample population and represents a more than 18% decrease in the number of false positives returned.

8.4.2 DETAILED COMPILER PROVENANCE MODEL

We evaluated several models of detailed compiler provenance in terms of how accurately each property—compiler family, compiler version, optimization level, and source language—were recovered. We used a different corpus for this evaluation because we need source code to generate controlled training and testing sets for the various provenance properties.

Table 8.4: Evaluation of stripped binary parser performance with compiler inference. Using imperfectly inferred compiler provenance contributes a small but significant increase in the precision of the tool. Total false positive errors are reduced by 18% over the no-provenance case.

Experiment	FP	F_1	F_1 spread
			0.86 1.0
No compiler labels	6,871	0.956	
Inferred labels	5,585	0.959	
Ground truth labels	2,414	0.969	

Table 8.5: Variations of compiler toolchains used in this study. For the GCC and ICC families, an arbitrary revision (e.g., 4.4.2) was selected to represent the compiler version. For the MSVC family, unpatched installations of the indicated Visual Studio development environments were used. The compiler family and version values are used as provenance labels in our learning framework; we condense the different optimization level options to 'low' and 'high' classes.

		Optimization Level $\mathcal{O}$	
Compiler Family $\mathcal{C}$	Version $\mathcal{V}$	Low	High
GNU Compiler Collection (GCC)	3.4.x	-O0,-O1	-O2,-O3
	4.2.x	-O0,-O1	-O2,-O3
	4.3.x	-O0,-O1	-O2,-O3
	4.4.x	-O0,-O1	-O2,-O3
Intel Compilers (ICC)	10.x	-O0	-O2,-O3
	11.x	-O0	-O2,-O3
Microsoft Visual C++ (MSVC)	VS 2003	/Od	/O2
	VS 2005	/Od	/O2
	VS 2008	/Od	/O2

Evaluation Data Set

We collected source code for 175 programs written in C, C++, and Fortran. The programs were collected from eight open source software packages: the GNU binutils and coreutils utilities, GNU grep, the GNU groff typesetting package, Mozilla Firefox, LAPACK, and Xpdf (a free PDF viewer). For the compilation toolchain, we obtained several compiler versions from each of the GNU Compiler Collection (GCC), the Intel C Compiler (ICC), and the Microsoft Visual C Compiler (MSVC). Table 8.5 lists the compiler versions and optimization options we used to construct our experimental dataset.

We generated the binaries that make up our dataset by compiling the source packages with all applicable combinations of compiler versions and optimization options.[3] The resulting data set comprises 2,686 binaries containing in total over 955,000 functions. For each binary, we record the source language, compiler family, version, and optimization options used to generate it; these form the *ground truth* label tuples $y = \langle s, c, v, o \rangle$ that we use for training and evaluation.

[3] Some combinations of source package and compiler are invalid; for example, the GNU software cannot be compiled with MSVC due to operating system dependencies.

Methodology

The performance of any classifier depends on both the training data used for parameter estimation and the testing data; any particular selection of data may not be representative of another selection. To mitigate the possibility that results may be biased by the particular choice of training and testing data, it is common practice to repeat training and evaluation over multiple *folds* of the data, where each fold consists of disjoint sets of training and testing examples randomly selected from the entire corpus. We generated ten experimental folds as follows:

1. Randomly select 30 training programs without replacement from the source corpus.

2. Randomly select 30 testing programs without replacement from the remaining programs.

3. For each selected program, add binaries with all combinations of toolchain components to the training or testing set, as appropriate.

The remaining evaluation steps were repeated independently over each of the folds.

To select a subset of significant features, we used the ParseAPI parsing library to obtain the control flow graphs for each binary in the training set. We then exhaustively enumerate all idioms and graphlet-based features that occur in the training data, using the occurrences of these features along with the provenance labels to compute the mutual information score for each feature. There are typically over one million features in a given training set; we selected the top 20,000 to reduce the size of the feature space.

For each function in each binary, we used the selected features to construct a sparse feature vector representing the output of the appropriate feature functions for each model (feature counts for the SVM classifier, boolean values for the CRFs), as well as the ground truth label tuple. The label that we provide to the learning algorithms depends on the kind of provenance modeling in which we are interested: as discussed in Section 8.3.3, we can concatenate the label components into a single class, use only single components or a concatenated subset, or allow all of the labels to be considered individually (the latter applies only to the general-structure CRF model). All of the functions were aggregated for the SVM classifier based on LIBLINEAR; the sequences of functions in each training binary were constructed separately for the CRF implementations based on MALLET and GRMM. All three of the learning packages automatically perform parameter estimation over the training data.

Testing data was formatted using the selected features in the same way as training data, except that the ground truth provenance labels are retained only as reference for

Table 8.6: Classification accuracy for individual functions. The compiler version component of provenance is difficult to capture with this independent function model.

Component	Labels	Accuracy	Spread
			0.53 1.0
Compiler family	3	.987	
Optimization	2	.971	
Compiler version	9	.616	
All components	18	.604	

evaluation. We used the parameters estimated in the training process to assign the most probable provenance labels to the testing data, based on the features present. Our evaluation focused not only on classification accuracy, but also on the types of errors encountered.

Independent Classification Results

We trained several provenance models over independent functions as described in Section 8.3.1 using the SVM-based classifier. Table 8.6 lists classifier accuracy for models trained to recover various provenance components; here and in the following discussion, results are averaged over the ten experimental folds unless otherwise noted. The results reported for "all" in Table 8.6 use the concatenation of all provenance components as labels.

The independent classification results show that for most of the toolchain components we consider, individual functions contain sufficient details to correctly determine their provenance. The version of compiler used to produce a program appears to be significantly more difficult to determine. Consider the version labeling errors made on three representative binaries, below, where errors that confuse versions within a single compiler family are shaded blue (■) and those that confuse different compiler families are shaded red (■):

Label	Error rate	Error distribution
$\langle gcc, 34, lo \rangle$	.130	
$\langle icc, 11, hi \rangle$	.088	
$\langle msvs, 2005, lo \rangle$	.576	

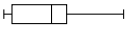
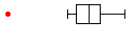
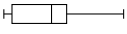
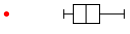
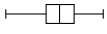
	Linear CRF		Linear CRF (concat.)		General CRF	
Component	Acc.	Spread	Acc.	Spread	Acc.	Spread
		0.87 1.0		0.81 1.0		0.73 1.0
Compiler	.999		.998		.992	
Optimization	.999		.993		.982	
Version	.919		.910		.845	
Joint	.918		.905		.831	

Table 8.7: Classification accuracy for provenance models incorporating function adjacency. The joint accuracy for the linear CRF (first column) was computed by concatenating the labels assigned by the individual component classifiers. The individual component accuracies for the concatenated-label CRF were computed by considering only those portions of the label tuple from the joint classification.

The histograms show the distribution of errors in each binary. Note that the level of detail is insufficient to resolve errors at the function level; the shading indicates the presence of a classification error in that segment of the binary, not contiguous errors. There are several important details to note in these error distributions. First, the classifier tends to rarely mislabel a function with a version associated with a different compiler family; such errors make up only 4% of all version classification errors. This matches our intuition that code emitted by one version of a compiler bears more similarity to code emitted by a different version than to code produced by a different compiler family.

Note also that while the average error rate for labeling the version provenance component is high, it is not uniform across the test set and the compiler version can be accurately inferred for many binaries. A small set of binaries account for the majority of errors; of these, the Microsoft Visual C data set is disproportionately represented. The data suggest that different compiler families have varying rates of "churn" across versions, with the GCC and ICC compilers producing significantly more varied code between versions than the MSVC compiler. We found that up to 70% of the functions in our data set are bitwise identical when generated by the Visual Studio 2003 or 2008 versions with the optimization level held constant. In other words, the code generator in Visual Studio has remained relatively fixed between these versions. This invariance poses a fundamental limitation for provenance recovery techniques that treat functions independently.

Joint Classification Results

We incorporated intra-binary function adjacency into the models based on both the linear chain and general conditional random fields that are presented in Section 8.3.3. For the linear chain models, we evaluated inference of individual provenance components (source language, compiler family, version, and optimization level), as well as recovery of all components simultaneously using concatenated-tuple labels as in the previous section. The general CRF takes the grid structure of Equation 8.3 with fully connected cotemporal label nodes. The linear chain models are learned using MALLET's exact inference mode; we use approximate Tree Reparameterization [97] for inference during learning and classification for the grid models. Table 8.7 lists classifier accuracy on the test set.

Incorporating the adjacency features significantly increases the accuracy of provenance recovery, particularly for the compiler version component. Both the individual component classifiers and the classifier based on concatenated labels accurately recover provenance on our test set. Despite the single outlier fold for the second CRF, the difference in classifier accuracy of the two models is statistically insignificant. The distinction between the two arises in runtime cost: retrieving all three of the reported provenance components with the individual component CRFs requires training three separate models and running inference three times; the concatenated-label model achieves comparable results at one third of the cost.

The grid-structured conditional random field has the poorest performance on our test set, though again its accuracy for the compiler and version provenance components is comparable to the other models. The output of this model may be easier to interpret, however. While the linear chain CRFs provide a single estimate for a particular label likelihood, the grid-structured CRF provides estimates for each component of the label tuple while still representing their dependencies. This can make interpreting uncertainty in the version component easier, for example: the labels for compiler family and optimization level might be assigned high confidence values by the model, while the version would be lower. By contrast, the concatenated-label CRF would assign low confidence to the entire tuple, giving no indication as to where the ambiguity lies.

The types of errors made by these classifiers offer further insight into the provenance recovery problem. The distribution of errors is quite skewed: on average across the experimental folds, the concatenated-label CRF makes no errors on 84% of test set binaries. The remaining binaries exhibit errors in three different modes, typified by mislabeled version (■) and optimization level (■) in the following examples:[4]

[4] These are not the same examples presented in the previous section.

Label	Error rate	Error distribution
$\langle icc, 10, lo \rangle$	.048	
$\langle msvs, 2008, lo \rangle$	.433	
$\langle msvs, 2008, lo \rangle$	1.00	

The latter two examples reflect the difficulty of inferring the compiler version, even in these composite models with adjacency features. In some cases only a subsequence of the binary is incorrectly labeled; for a small number of others, almost the entire binary is assigned the incorrect label for the version component. This error mode is more common in binaries from the Microsoft data set, due to the relatively few differences between different compiler versions.

The first example exhibits the most common error mode on our testing set, and occurs more frequently in the GCC and ICC binaries. Further analysis of these errors reveals that they arise due to the existence of *statically linked* library code appended to the end of these binaries by the compiler. Binaries produced by the Intel compiler tend to include more of such code, in the form of optimized support routines specific to that compiler. These functions are counted as errors because we produce ground truth labels at the binary level—a limitation of how we generated our corpus, but not of our technique. Indeed, these "errors" demonstrate that the classifier is capable of detecting regions of the binary with varying provenance.

Source Language

Evaluating the source language provenance component is challenging because many of the programs in our data set are written in a mixture of languages (e.g., both C and C++). While mixed provenance poses no intrinsic challenge for our technique, it can be difficult to automatically establish a ground truth labeling without laborious human analysis. We therefore evaluate the source component on a subset of the corpus consisting of 28 programs written in C, C++ and Fortran, which subset we have examined by hand to ensure a (mostly) uniform source language. Our ground truth labelings are likely to still be imprecise for this reduced data set, so classification accuracy may be artificially understated.

Independent classification of the source component using SVMs achieves an average accuracy of 91%. The results for classification of the source language component and for joint classification using the the linear chain CRF with concatenated labels are listed in Table 8.8. These results are not directly comparable to the larger study of the compiler family, version, and optimization level components from the previous section due to the use of a different training and evaluation corpus; nevertheless, our

Table 8.8: Classification accuracy on a corpus incorporating source language labels.

Component	Accuracy	Spread
		0.96 1.0
Language	.967	
Joint	.990	

evaluation suggests that the source language of a program can be accurately inferred with our provenance recovery technique.

8.5 SUMMARY

We have evaluated the recovery of detailed toolchain provenance properties from program binaries, both at the byte-level and at the level of functions. Our provenance recovery framework achieves on average 90% accuracy when jointly inferring the source language, compiler family and version, and optimization level options used to produce a binary. We developed provenance models based on support vector machines and conditional random fields, and showed how these models can differentiate between code and data, and identify compiler properties even in binaries of mixed provenance. The results of our evaluation strongly support our claim that toolchain provenance can be recovered solely from the characteristics of the executable code in program binaries.

One limitation of the toolchain provenance recovery techniques that we have presented is that they assume that all possible variations in the compiler toolchain are known *a priori*. Even for commercial compilers this is a tenuous assumption (new, radically different compiler versions could be introduced at any time); if we expand our scope to possible custom toolchain components and hand-written assembly, the task of enumerating all variations in toolchain provenance appears to be intractable. One possible approach to this expanded problem is to use statistical *outlier detection* methods [7]. Alternatively, the unsupervised clustering and knowledge-transfer approach for authorship clustering that we describe in Chapter 10 is directly applicable to this problem, offering a means by which to determine the existence of related but unknown toolchain components without access to training data.

The work in this chapter builds on the representations and models used in Chapter 7, incorporating program structure both in the model itself and in the use of branch- and instruction summary graphlets. In the following chapters, we continue to expand the set of features and modeling techniques that we apply to provenance recovery,

exploring the question of whether programmer style is reflected in binary code.

9

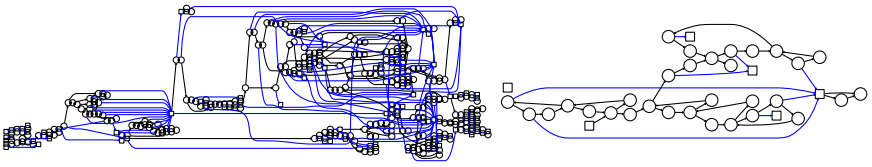
Style and Author Identification

Program authorship attribution has immediate implications for the security community, particularly in its potential to significantly impact applications like software plagiarism or theft [81] and digital forensics [64]. The central thesis of authorship attribution is that authors imbue their works with an individual style. While attribution research has historically focused on literary documents [40], computer programs are no less the product of a creative process, one in which opportunities for stylistic expression abound. Previous studies of program authorship attribution have been limited to source code [33, 45], and rely on surface characteristics like spacing and variable naming, both of which reflect the essentially textual nature of program source. In many domains, such as analysis of commercial software or malware, source code is usually unavailable. Program binaries, however, retain none of the surface characteristics used in source code attribution; such details are stripped away in the compilation process. Adapting program authorship attribution to the binary domain—to identify known malware authors or detect new ones, e.g., or to discover theft of commercial software—requires new ways to recognize the style of individual authors.

In this chapter, we apply our provenance framework to authorship attribution and the discovery of stylistic characteristics of binary code. Using our framework's automatic feature selection and machine learning approach, we avoid the problem of choosing good stylistic features *a priori*, which has been the focus of source code attribution [90], and which is the primary challenge for attribution in the binary domain. In this chapter, we focus on *authorship classification*, or identifying a particular author out of a set of candidates; the following chapter addresses *authorship clustering*, its unsupervised analog.

The studies we have conducted incorporate the full range of program representations that we introduced in Chapter 5. We evaluate binary code authorship attribution on several large sets of programs from the Google Code Jam programming competition[1] and from student projects from an undergraduate operating systems course at

[1] `http://code.google.com/codejam/`



(a) (b)

Figure 9.1: The control flow graphs for two implementations of the same program by different authors. Program (a) is implemented as many small subroutines and makes use of several C++ STL classes; program (b) is almost entirely implemented as a monolithic C function.

the University of Wisconsin. Our experiments demonstrate that programmer style is reflected in binary code, and lay the groundwork for authorship attribution in a variety of domains.

Although our focus in this dissertation is on binary code provenance, our methodology is equally applicable to the source code domain. In Section 9.5 we present the results of a preliminary study of authorship attribution at the source level, showing that simple textual features can lead to highly accurate author classification. Our results suggest that a machine learning–based approach to source-level authorship attribution may significantly outperform existing techniques.

9.1 Problem Domain

Program authorship attribution is predicated on the hypothesis that programmer style is reflected in characteristics of binary code. Figure 9.1 lends credence to this hypothesis, depicting the control flow graphs of two binaries that implement the same functionality but that were written by different programmers. Our goals in this chapter are twofold: to determine whether authorship attribution is possible, and to evaluate our provenance recovery framework's suitability for the task. Because this is an exploratory experiment—to the best of our knowledge, ours is the first attempt at attribution of binary code—we have limited the problem scope to better control for confounding variables.

We restrict our attention to programs with a single author, easing the problem of establishing a ground truth author labeling for code regions in the compiled binary. Even in single-author programs, the binary is likely to contain code that was not written by the author, such as statically linked utility code or instantiations of templates from the C++ Standard Template Library. We annotated the data sets described in Section 9.3.1 by hand to eliminate most of such external code; however, our annotations

function COLLAPSE($G = (V, E, \tau, \sigma)$)
  $V' \leftarrow \varnothing, E' \leftarrow \varnothing$
  for all $v \in V$ **do**
    choose $n \in$ NEIGHBORS(v) ▷ random n
    $\sigma'(v') \leftarrow \sigma(v) \cup \sigma(n)$
    $V' \leftarrow V' \cup v'$
    $E' \leftarrow E' \cup \{(v', n') : (v, n') \in E \vee (n, n') \in E\}$
    $E' \leftarrow E' \cup \{(n', v') : (n', v) \in E \vee (n', n) \in E\}$
  return (V', E', τ, σ')

Figure 9.2: Graph collapse for supergraphlet features. While the node colors are merged in the collapsed graph, the edge degree and types are preserved.

are likely imperfect. To mitigate the impact of external code, we infer provenance at the granularity of whole programs; by aggregating the features across the entire program, we relegate improperly included code to background noise.

In the provenance recovery experiments described in previous chapters, we have used both instruction- and control flow–based features to characterize provenance. Our initial experiments with authorship attribution using idiom and instruction summary graphlet features were encouraging, but we suspected that the locality of the graphlet features was a limiting factor. We designed the call graphlet and supergraphlet features that we describe in Chapter 5 to capture stylistic characteristics that are visible only in high-level program structure (recall that supergraphlets are induced by collapsing the control flow graph using the algorithm depicted in Figure 9.2). Table 9.1 lists the feature types we use in authorship provenance models and the number of each type that are present on one of our experimental data sets.

9.2 MODEL FORMULATION

In authorship attribution, we assume that there exists a known set of programmers of interest, and that training data are available in the form of samples of programs written by each programmer. We describe programs in terms of the number of occurrences of each binary code feature. To be precise, given a known set of program authors $\mathcal{Y}$ and a set of M training programs $\mathcal{P}_1, \cdots, \mathcal{P}_M$ with author labels $y_1, \cdots, y_M$, the task of the classifier is to learn a decision function that assigns a label $y \in \mathcal{Y}$ to a new program, indicating the identity of the most likely author.

A program $\mathcal{P}_m$ is represented by a integral-valued feature vector $\mathbf{x}_m$ describing the features that occur in the program. Feature vectors summarize a set of feature functions

Table 9.1: The number of concrete features instantiated by each feature template for a representative corpus of 1,747 C and C++ binaries comprising 27MB of code. Each template captures one or more instruction-level, control-flow, or external library interaction properties of the code.

		Code Property		
Feature	#	Instruction	Control flow	External
N-grams	391,056	✓		
Idioms	54,705	✓		
Graphlets	37,358	✓	✓	
Supergraphlets	117,997	✓	✓	
Call graphlets	8,062		✓	✓
Library calls	152			✓

$f \in \Phi$ that indicate the presence of that feature evaluated over a feature-specific domain in the binary. For example, the function

$$f_{\text{FPRINTF}}(c_j, \mathcal{P}_m) = \begin{cases} 1 & \text{if call site } c_j \text{ in } \mathcal{P}_m \text{ calls FPRINTF} \\ 0 & \text{otherwise} \end{cases}$$

tests for a particular library call and is defined over the domain of *call sites* in the program; idiom feature functions

$$f_\iota(a_j, \mathcal{P}_m) = \begin{cases} 1 & \text{if idiom } \iota \text{ exists at instruction offset } a_j \text{ in } \mathcal{P}_m \\ 0 & \text{otherwise} \end{cases}$$

are defined over the domain of instruction offsets in the binary. The domains of each feature type are listed in Table 9.2. The feature vector $\mathbf{x}_m$ for a program counts up the $n = |\Phi|$ features

$$\mathbf{x}_m = \begin{pmatrix} \sum_{Dom(f_1)} f_1(\cdot, \mathcal{P}_m) \\ \sum_{Dom(f_2)} f_2(\cdot, \mathcal{P}_m) \\ \cdots \\ \sum_{Dom(f_n)} f_n(\cdot, \mathcal{P}_m) \end{pmatrix}$$

evaluated at every point in the domain $Dom(f_i)$ of the particular feature.

As in the independent classification experiment for detailed toolchain provenance (Section 8.3.1), we perform authorship classification with linear support vector machines (SVMs), which scale well with high-dimensional data. The number of feature functions in Φ is quite large; using feature vectors that summarize all possible features would

Table 9.2: Domains for feature functions used in authorship classification.

Feature type	Domain $Dom(f)$
N-grams	all offsets $0 \leq a \leq n$ in n-byte $\mathcal{P}$
Idioms	all instruction offsets $0 \leq a \leq n$ s.t. $a \in \textsc{Decode}(\mathcal{P})$
Summary Graphlets	all connected 3-subgraphs of CFG G of $\mathcal{P}$
Supergraphlets	all connected 3-subgraphs of $G' = \textsc{Collapse}(G)$, $G'' = \textsc{Collapse}(G')$, and $G''' = \textsc{Collapse}(G'')$
Call graphlets	all connected 3-subgraphs of the call-CFG described in Section 5.5
Library calls	all call instructions $C \subseteq \textsc{Decode}(\mathcal{P})$

increase both training cost and the risk that the learned parameters would *overfit* the data—that is, that the resulting classifier would fail to generalize to new programs. As in previous experiments, we use a selection process based on mutual information to reduce the feature space. We scale the feature vectors to the interval [0,1]; scaling prevents frequently occurring features from drowning the contribution of rarer ones, while preserving the sparsity of the feature vectors. In the evaluation section, we examine the contribution of each feature type to overall classifier performance.

9.3 EVALUATION

We investigated several aspects of authorship attribution: (1) the extent to which our techniques recover author style in program binaries, (2) the trade-offs involved in imprecise classification (i.e., tolerating some *false positives*), and (3) how different types of features contribute to authorship attribution. Our evaluation shows that:

- The binary code features we introduce effectively capture programmer style. Our classifier achieves accuracies of 81% for ten distinct authors (10% accuracy is expected for labels selected by random chance) and 51% when discriminating among almost 200 authors (0.5% for random chance). These results show that a strong author style signal survives the compilation process.

- The authorship classifier offers practical attribution with good accuracy, if a few false positives can be tolerated. The correct author is among the top five 95% of the time for a data set of 20 authors, and 81% of the time when 100 authors are represented.

9.3.1 evaluation data set

Obtaining appropriate data sets is a fundamentally limiting problem for the development of authorship attribution techniques. At the minimum, the collection of programs (the *corpus*) used to develop an authorship model must have author labels. A *parallel* corpus—one in which each author has written programs with the same functionality—helps to control for confounding variables in the evaluation. Previous program authorship studies have obtained parallel corpora by eliciting program implementations from a small number of human subjects [33, 45]. We have constructed substantially larger corpora from programs written for the Google Code Jam programming competition and from student projects from an operating systems course at the University of Wisconsin.

Google Code Jam

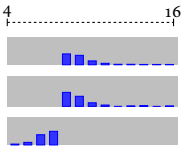
The Code Jam competition is an annual world-wide programming contest conducted over multiple rounds and representing the efforts of thousands of contestants. Each round of the competition involves writing a program to solve a small number (usually 3 to 6) of problems; contestants who pass a threshold are advanced to later rounds. The Code Jam website provides access to all correct solutions for each contestant, meeting both of our criteria for a high-quality authorship attribution data. We use the 2009 and 2010 contest data in our evaluation.

Code Jam solutions are not required to be written in any particular programming language. We restrict our attention to those solutions that were written in C or C++ and that could be compiled with GCC 4.5. We further filter the data set to only include programs by authors who submitted solutions to eight or more problems over the course of a given year's competition. Importantly, we note that each contestant submits *two* solutions for a given problem: one that is tested against a "short" test case and one that is run against a (presumably harder) "long" test case. In almost all cases the second solution is identical to the first (or has minor bugfixes); we therefore eliminate the second solution from our data set to avoid artificially inflating our evaluation accuracy by including many identical programs.

Class Projects

Our second corpus consists of programs written by students enrolled in an undergraduate operating systems course (CS537) at the University of Wisconsin. The students completed seven programming assignments during the course, including a shell implementation, several memory and scheduling libraries, and a simple web server. This

Table 9.3: Corpora used for model training and evaluation. Each binary is the implementation by a particular author of one of the program types for a given corpus. The Program/Author distribution reflects how many program types are available for each author; authors in the CS537 data set produced at most seven programs.

Corpus	Authors	Program Types	Binaries	Program/Author Dist.
				
Code Jam 2010	191	23	1,747	
Code Jam 2009	93	22	834	
CS537 Fall 2009	32	7	203	

data set has several properties that make it more difficult to use than the Code Jam database. Students were allowed to work on programs alone or with partners, so a program's authorship can be ambiguous in some cases; also, in many assignments skeleton code or fully implemented modules were provided to the students. These uncertainties in the ground truth may impact our evaluation. We attempt to mitigate this possibility by eliminating one author out of every partnership, and by analyzing the original assignments by hand and constraining our training and evaluation to code that does not appear to have been provided. This second approach in particular is likely to be imperfect; we discuss the implications for our results below.

Table 9.3 summarizes our experimental corpora. The two different data sets differ significantly, both in the distribution over program sizes and in that of the number of different programs written by each author. The fewer programs per author in the CS537 corpus increase the difficulty of the learning problem.

9.3.2 METHODOLOGY

To create a data representation suitable for learning and evaluation, we process the binaries in each corpus with the ParseAPI parsing library [68] to obtain control flow graphs and the underlying instructions. We eliminate statically linked library functions and other known binary code snippets that are unrelated to the program author. We then exhaustively enumerate all of the features types over their domains listed in Table 9.2, using the occurrence of these features along with the known authorship labels to compute the mutual information for each feature. We select a subset of features using cross-validation to avoid overfitting. Cross-validation selected 1,900 features for modeling and evaluation of the Code Jam data; 1,700 features are used for CS537.

Our evaluation methodology involves both standard ten-fold cross-validation and

random subset testing, depending on the experiment:

- For classification of the entire data set (e.g. 191-way classification for the Code Jam 2010 data), we use ten-fold cross-validation.
- When evaluating how classification accuracy behaves as a function of the number of authors represented in the data, we randomly draw a subset $\mathcal{Y}_s \subseteq \mathcal{Y}$ of authors and use their programs in the test. We cannot test all possible combinations of $|\mathcal{Y}_s|$ authors; instead, we repeat the test 20 times and expect relatively high variance for small sets of authors. We approach the clustering evaluation similarly.

9.3.3 Classification

We evaluate authorship classification to determine (1) how much each feature type contributes to attribution, and (2) how accurately the identity of a particular author can be inferred using a model based on our feature templates. For the former question, our experience led us to expect that simple feature types that instantiate large numbers of features (e.g., idioms) would be more useful in authorship modeling. For the latter question, we hypothesized that discriminating among authors would become increasingly difficult with larger author populations.

Figure 9.3 depicts the cross-validation accuracy of models trained with varying numbers of features from each feature type, as well as the accuracy of a model trained with all feature types. Our intuition is borne out by these results: the individual contributions of simple idiom and N-gram features exceed those of the other templates. The best classifier uses a combination of all of the feature templates, achieving 51% accuracy on the 191-way classification problem of the full Code Jam 2010 data set.

Experiments confirm our hypothesis that author classification becomes harder for larger populations. Figure 9.4 depicts classifier performance as a function of the number of authors included in a subset of the data; classifier accuracy decreases as the author population size grows. In cases where precise author identification is infeasible, predicting a small set of likely authors can help to focus further investigation and analysis. In Figure 9.4, this relaxed accuracy measure is plotted for a classifier that returns the top five most likely authors.

Table 9.4 lists exact and relaxed cross-validation accuracy for authorship classification on each corpus. The CS537 data present a significantly harder challenge for authorship attribution, due to two factors. First, there are fewer programs per author (4–7) than in the other data sets (8–16), making this a fundamentally harder learning problem. More importantly, the programs in this data set do not reflect only the work of individual programmers; students in the course were often provided with substantial

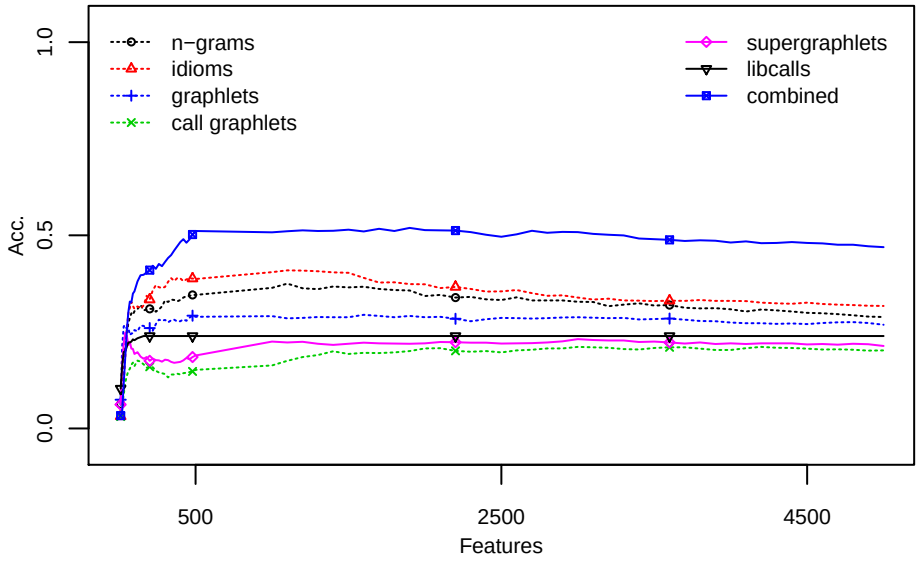


Figure 9.3: Individual contribution of each feature type to authorship attribution. The accuracy of 191-way authorship classifiers trained with varying numbers of a single type of feature are depicted, as is the accuracy of the aggregate classifier. Note that all feature types, but particularly idioms and n-grams, lead to some degree of over-fitting.

Table 9.4: Classification results averaged over 20 randomly selected subsets of 20 authors.

	Code Jam 2009		Code Jam 2010		CS537	
	Acc.	spread	Acc.	spread	Acc.	spread
Exact	.778		.768		.384	
Top 5	.947		.937		.843	

amounts of partially implemented skeleton code, and also worked closely with the course professor following an often rigid specification at the sub-module level. Despite these challenges, our attribution techniques recover significant stylistic characteristics in this data set.

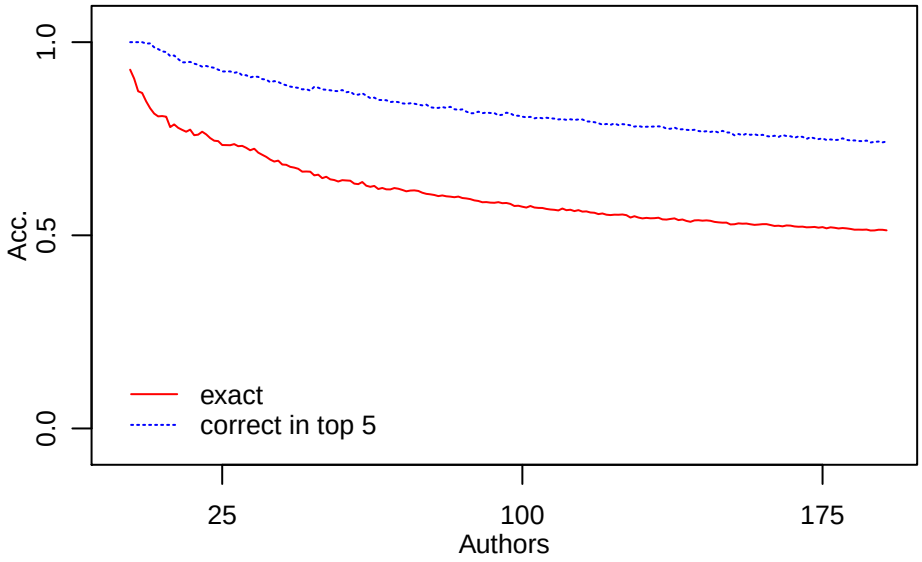


Figure 9.4: Evaluation of authorship classification on the Code Jam 2010 data set. The classifier uses the top 1,900 features by mutual information; its accuracy is depicted as a function of the true number of authors in the data set for both the exact (—) and relaxed (····) evaluations.

9.4 Discussion

Our evaluation shows that programmer style is preserved in program binaries, and can be recovered using techniques that automatically select stylistic code features with which to model program authorship. The SVM-based classifier we introduce can identify the correct author out of tens of candidates with good accuracy, though discriminating among a large number of authors is likely to be more limited. Nonetheless, we argue that our techniques offer a practical solution to program author identification: when discriminating among programs written by 100 authors, the correct author is ranked among the top five most likely 81% of the time, reducing the number of candidates by 95%.

The conclusions we draw are subject to limitations inherent in empirical studies. In particular, threats to internal validity apply to our claim that our techniques isolate programmer style, rather than some other program property like program functionality. We addressed this issue by using a parallel corpus, where each author implemented the same programs; the fact that our authorship classifier is able to learn to recognize

an author's programs despite differing functionality mitigate this threat. The domain transfer results for authorship clustering that we present in the following chapter provide further evidence that our techniques recover programmer style.

In this study, we assume that a program has a single author. This assumption may be violated in many scenarios, such as when programmers collaborate or when programs are assembled from commodity components. The binary code representation we use is not inherently restricted to representing the program as a single unity; our features could just as easily describe individual compilation units, functions, or arbitrary sequences of binary code, for example using the sequential model we have previously used to recover toolchain provenance [76, 77]. The extension of authorship attribution to multiple authors and a sub-program model is an open question.

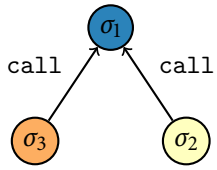
Interpreting Features

The nature of the features underlying our authorship models suggests several additional directions for future research. Our use of many simple, uninformed binary code features provides much of the power of our approach, but makes understanding the resulting models difficult: it is not clear how to map from instruction idioms and control flow graphlets to conceptual notions of high-level programmer style. To illustrate the difficulty of this problem, consider the following idiom features, ranked by the magnitude of their weights w in an SVM-based classifier that discriminates among ten different program authors (these idioms are among the top twenty features by magnitude):

Idiom	w
`push rbx | * | lea rax, [mem]`	0.213638
`push rbx | sub rsp, [imm] | lea rax, [mem]`	0.213638
`sub rsp, [imm] | * | mov [mem], rsi`	0.180291
`mov rdi, rax | * | mov rax, [imm]`	0.179203
`call rip | mov xmm, rax | mov rdi, rax`	0.173018
`jnz rip | mov xmm, rax | mov rdi, rax`	0.165779
`mov rax, [imm] | call rip | mov xmm, rax`	0.153061
`sub rsp, [imm] | * | mov rsi, [imm]`	0.148890
`mov xmm, eax | * | mov rdi, [imm]`	-0.133057

While some patterns emerge (IP-relative branching and calls appear to be a telling characteristic), it is difficult to translate from simple instruction patterns to a meaningful, high-level understanding of programming style; our models require hundreds or thousands of these features to capture program authorship. Other highly-weighted

features offer similar barriers to interpretation; there are several three- and four-byte N-grams among the most discriminative features in this model, and the highest-weighted feature is the instruction summary graphlet



where $\sigma_1 = \{\texttt{jump}\}$, $\sigma_2 = \{\texttt{call},\texttt{lea},\texttt{mov}\}$, and $\sigma_3 = \{\texttt{arith}, \texttt{call}, \texttt{mov}, \texttt{stack}\}$, which is hardly sufficient to reach any conclusion about programming style.

The contribution of some features is easier to judge, such as the external library features. The use of `sprintf` and several of the C++ `iostream` operators are heavily weighted (within the top ten), suggesting that programmer preference for particular APIs or standard library functions may be a good indicator of style. Nonetheless, this feature type alone is insufficient (see Section 9.3.3) for authorship attribution and extracting high-level interpretations of programmer style from low-level code features is an open problem.

9.5 Source Code Attribution

The focus of this dissertation is on recovery of program provenance from binary code; nonetheless, we developed a simple, proof-of-concept authorship classifier for source code to determine just how much easier source-level analysis is. Our goal was to establish a baseline for how much programmer-specific detail could be gleaned from programs. Surprisingly, a simple classifier achieved state-of-the-art results on our evaluation data set.

Previous source code attribution has relied on a small number of carefully crafted features. The use of code metrics like variable naming conventions, comment style, and program organization has been proposed several times [30, 90]; Krsul and Spafford [45] show the feasibility of this approach in a small pilot study. More recently, Hayes and Offutt [33] found further evidence that programmers can be distinguished through aggregate textual characteristics like average use of particular operators, placement of semicolons, and comment length.

Most of the features that we use in our provenance framework do not translate well to the source domain. We created a linear support vector machine classifier based on *character 3-grams*, analogous to the byte N-gram features that we use in binary

Table 9.5: Classifier accuracy for source code attribution on the Code Jam 2010 data set. The 'vanilla' programs were stripped of comments but otherwise unmodified; classifiers were also trained and evaluated on programs where whitespace formatting had been removed and variable names anonymized. The classifier is highly accurate, even for large numbers of distinct authors.

	10 authors		50 authors		100 authors	
	Acc.	spread	Acc.	spread	Acc.	spread
		.9 1		.9 1		.9 1
Vanilla	.991		.986		.978	
Unformatted	.990		.980		.970	
Anon. vars	.985		.964		.951	
Both	.976		.940		.920	

provenance modeling. We represented a program's source code with a feature vector that indicates the number of occurrences of each 3-gram in the source file.

We evaluated this simple classifier on the Code Jam 2010 data used in the binary experiment. We preprocessed the source code to remove comments, reducing the possibility that uniquely identifying text (such as names) would appear in the program, though random hand inspection of the data found no examples of such distinguishing comments. In addition to this sanitized data set, we prepared three other data sets for evaluation:

1. We collapsed all repeated whitespace characters (spaces, tabs, carriage returns) to a single space character, eliminating most of the impact of whitespace formatting. This data set was intended to test whether whitespace was a major facet of programmer style.

2. We replaced all variables with anonymous "myN" tokens, testing whether variable names were a major contributing factor to classifier performance.

3. We combined both the whitespace removal and variable anonymization.

We trained and tested classifiers over multiple folds on each data set to evaluate (1) how classifier accuracy varies for the different transformations, and (2) how well we can perform attribution for data sets representing different numbers of programmers. The results are depicted in Table 9.5. The characteristics of programmer style appear to depend only slightly on whitespace and variable names; even when both facets have

Table 9.6: The top 40 source code N-grams by mutual information. Some (highlighted) seem to be associated with the use of particular library functions, while others (highlighted) capture whether or not a space was inserted between the `for` keyword and the opening parenthesis.

pen	␣fr	reo	eop
fre	tdo	",␣	t);
ut)	or(	n("	r␣(
9>␣	)␣#	ope	,␣&
8>␣	en(	,my	or␣
=my	;my	f␣l	␣(i
++m	y9>	7>␣	r(i
;␣#	",m	ype	typ
=0;	ped	ef␣	␣ty
ede	y8>	)␣(	dou

been eliminated from consideration, the classifier can distinguish among 100 different authors with 92% accuracy.

The classifier's performance is surprising, exceeding the best reported authorship attribution accuracy of 80-85% [56] by a comfortable margin and on a significantly larger data set; the previous work only considered collections of programs by 5–10 authors. Interpreting these results is difficult; as with byte N-grams in the binary domain, it can be hard to understand the importance of an arbitrary triplet of characters. Table 9.6 lists the top features ranked by mutual information. Some seem more meaningful than others. For example, the use of `freopen` appears to be strongly associated with some authors and not others; whether whitespace is inserted between keywords that are followed, syntactically, by parenthetic expressions also appears to be telling. There are approximately 16,000 unique N-grams in the un-formatted and variable-anonymized data set, however; understanding how these features capture programmer style is beyond the scope of this study.

9.6 Summary

We have presented techniques to extract stylistic characteristics from program binaries to perform authorship attribution. Our authorship attribution techniques identify the correct author out of a set of 20 candidates with 77% accuracy, and rank the correct author among the top five 94% of the time. These techniques enable analysts to

determine, for example, whether a new program sample is likely to have been written by a person of interest. Framing authorship attribution as a provenance recovery problem, we developed instruction- and structure-based representations of binary code that automatically capture binary code details that reflect programmer style. The results of our evaluation strongly support our claim that programmer style is preserved through the compilation process, and can be recovered from characteristics of the code in program binaries.

Previous work on program authorship attribution has focused exclusively on source code-level attribution, using textual characteristics like variable naming conventions and comment style [30, 33, 45, 90]. All of these approaches use code characteristics that are targeted at the source domain, and cannot be applied to binary code. Furthermore, our preliminary investigation of source-level authorship attribution using N-gram text features suggests that an approach based on statistical machine learning may significantly outperform existing, heuristic techniques. Our source code authorship classifier achieves 92% accuracy when discriminating among the 191 programmers of one of our data sets, and is over 97% accurate when discriminating among ten programmers.

The experiments we describe in this chapter have focused on identifying a specific author when training data are available. In the following chapter, we direct our attention to the more challenging problem of grouping programs by stylistic similarity when little or no training data are available.

10

Style and Similarity

In the previous chapter, we demonstrated that programmer style is preserved in binary code, and that stylistic characteristics can be used to model a particular programmer. The author identification techniques we described discriminate among code written by known authors, but are limited: as classification problems, they require training data for each candidate programmer and cannot be applied to broader authorship questions, such as determining whether a set of programs represents the work of two or more authors or whether several programs were written by a single, though anonymous, programmer. In this chapter, we describe *authorship clustering* techniques that automatically group programs by stylistic similarity in the absence of author-specific training data.

10.1 PROBLEM DOMAIN

Authorship clustering is a generalization of, and bears many similarities to, our authorship classification work. We use the same binary representations and features as in Chapter 9, but instead of learning a mapping between author labels and binary code features, our goal is to cluster programs that were written by the same author. Clustering is an unsupervised learning technique that groups data by similarity. For program authorship, clustering corresponds to the task of finding stylistically similar programs without assuming particular authors are present. In many ways, clustering is harder than classification: without training data, it is generally not possible to tell whether particular features are more or less useful for relating the data, which leads to the possibility that clustering algorithms will arrive at clusters that reflect a different property than what was desired. This issue is particularly challenging for authorship clustering, where we have a large number of features and no assurance that they reflect only programmer style and not, for example, program functionality.

10.1.1 k-means clustering

There are many possible algorithms that can be used to cluster data. Since our goal is to develop a technique to encourage the formation of stylistic clusters (rather than to identify the best clustering algorithm for our domain), we selected the simple and well-known k-means clustering algorithm [8] for our experiments. The objective of k-means is to divide a set of n data points into k subsets $S_1, \cdots, S_k$ such that for any subset S_i, the distance between any data point $\mathbf{x}_i$ to the mean $\boldsymbol{\mu}_i$ of the subset is minimized. More precisely, our objective is a solution to the problem

$$\underset{S}{\operatorname{argmin}} \sum_{i=1}^{k} \sum \mathbf{x}_j \in S_i \|\mathbf{x}_j - \boldsymbol{\mu}_j\|^2,$$

which is known to be NP-hard [2]. k-means is an approximate, iterative algorithm consisting of an *assignment step* that places each data point in the nearest cluster, and an *update step* that refines the cluster means $\{\boldsymbol{\mu}\}$. The steps of the algorithm are as follows:

1. Choose an initial k data points $\mathbf{x}_1, \ldots, \mathbf{x}_k$ and let them be the centroids of the initial clusters, i.e. set $\boldsymbol{\mu}_i = \mathbf{x}_i$.

2. Assign each data point to the cluster S_i with the closest mean, i.e.

$$S_i = \left\{ \mathbf{x}_j : \|\mathbf{x}_j - \boldsymbol{\mu}_i\| \leq \|\mathbf{x}_j - \boldsymbol{\mu}_{i^*}\| \, , \forall \, i^* \neq i \right\}$$

3. Compute the new means:

$$\boldsymbol{\mu}_i = \frac{1}{|S_i|} \sum_{\mathbf{x}_j \in S_i} \mathbf{x}_j$$

4. Iterate (2) and (3) until convergence.

The cluster assignments returned by k-means are heuristic, and are sensitive to the initial assignments in step (1). In the experiments we describe below, we run the algorithm repeatedly and evaluate our stylistic clustering technique in terms of the mean cluster quality measurements.

10.1.2 Distance metrics

The heart of any clustering approach is the *distance metric* that specifies the similarity of any pair of data elements in the d-dimensional feature space. In the k-means algorithm

depicted above, we used the euclidean distance

$$D(a,b) = \|\mathbf{x}_a - \mathbf{x}_b\| = \sqrt{(\mathbf{x}_a - \mathbf{x}_b)^T(\mathbf{x}_a - \mathbf{x}_b)}$$

in the d-dimensional feature space. The euclidean distance may not be ideal for stylistic clustering, however: as discussed above, the binary code features reflect more properties than just programmer style, for example toolchain provenance or program functionality. Depending on how strongly each property is reflected in the feature space, the k-means algorithm may produce clusters that reflect something other than authorship.

One way to encourage the formation of authorship clusters is to transform the feature space such that stylistically similar programs are closer to one another; that is, to define a $d \times d$ *transformation matrix* L such that, if $\mathbf{x}_a$ and $\mathbf{x}_b$ are programs written by the same author, then

$$\sqrt{(L\mathbf{x}_a - L\mathbf{x}_b)^T(L\mathbf{x}_a - L\mathbf{x}_b)} < \sqrt{(\mathbf{x}_a - \mathbf{x}_b)^T(\mathbf{x}_a - \mathbf{x}_b)}. \tag{10.1}$$

Equivalently, we can replace the euclidean distance with the *Mahalanobis distance* [57], a generalized distance metric in $\mathbb{R}^d$ that is defined as

$$D_A(\mathbf{x}_a, \mathbf{x}_b) = \sqrt{(\mathbf{x}_a - \mathbf{x}_b)^T A(\mathbf{x}_a - \mathbf{x}_b)}$$

where $A = L^T L$.

Introducing this alternative distance metric raises several problems. First, it is clear that (10.1) underspecifies the desired transformation: a trivial solution

$$L = \begin{pmatrix} .5 & 0 & \cdots & 0 \\ 0 & .5 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & .5 \end{pmatrix}$$

fulfills the inequality but does not change the similarity of the data points with respect to any particular property. What we need is a way to specify a transformation over the data that brings stylistically similar data points closer together while moving dissimilar points further apart. An additional constraint is that, for authorship clustering, we do not have training data for the set of programs to be clustered. We address these two problems in the following sections, first by giving a brief overview of *distance metric learning* and the *large margin nearest neighbors* (LMNN) algorithm [99], and then by describing our *transfer learning* approach to stylistic clustering.

10.2 Learning a distance metric

Our objective is to find a metric D_A such that the distance between any similar pairs of programs (i.e., programs with the same author) $(\mathbf{x}_i, \mathbf{x}_j) \in \mathcal{S}$ is small, while the distance between any dissimilar pair of programs $(\mathbf{x}_i, \mathbf{x}_j) \in \mathcal{D}$ is large. Distance metric learning is an area of active research [48]; a full review of proposed techniques is beyond the scope of this dissertation. We chose the LMNN algorithm for our study due to its efficiency, good reported results on a wide variety of clustering problems [99], and because it fits well with the k-means clustering algorithm.

In general, metric learning problems can be stated in terms of an optimization problem similar to

$$\begin{aligned} \min_{A} \quad & \sum_{(i,j)\in\mathcal{S}} D_A(\mathbf{x}_i, \mathbf{x}_j) \\ \text{s.t.} \quad & D_A(\mathbf{x}_i, \mathbf{x}_j) > 1 \quad \forall (i,j) \in \mathcal{D} \\ & A \succeq 0; \end{aligned}$$

that is, problems that minimize the distance of similar data points while ensuring that dissimilar data points are at least a minimum distance apart, and that the matrix A remains positive semidefinite. Requiring a metric that fulfills these constraints globally for all pairs of programs may be overly restrictive for our stylistic clustering experiments, however; because we use k-means, a local clustering algorithm, we only need program similarity within a local neighborhood to be reflected by the distance metric. The LMNN algorithm learns precisely such a metric.

The LMNN algorithm is defined in terms of a set of similar points $\mathcal{S}$ with the further restriction that any pair of points $(i, j) \in \mathcal{S}$ are in a local *neighborhood* of k points; following the original LMNN paper [99], we set the neighborhood size to three. The optimization is constrained by triplets of points $(i, j, k) \in \mathcal{R}$, where all three points are in a local neighborhood and k has a different label than i and j. The optimization problem is

$$\begin{aligned} \min_{A} \quad & \sum_{(i,j)\in\mathcal{S}} D_A(\mathbf{x}_i, \mathbf{x}_j)^2 \\ \text{s.t.} \quad & D_A(\mathbf{x}_i, \mathbf{x}_k)^2 - D_A(\mathbf{x}_i, \mathbf{x}_j)^2 \geq 1 \quad \forall (i,j,k) \in \mathcal{R} \\ & A \succeq 0, \end{aligned} \tag{10.2}$$

which defines a *semidefinite program* (SDP). The implementation of LMNN introduces *slack variables* that allow (but penalize) constraint violations; we refer the interested reader to the paper by Weinberger and Saul [99] for more details.

One downside of metric learning is its cost, which scales quadratically with with the dimensionality of the feature space. We apply *principal component analysis* (PCA),

an unsupervised dimensionality reduction technique, to allow metric learning over a smaller feature space. Briefly, given a matrix representing a set of data points in the feature space, PCA computes an orthogonal linear transform that projects the data onto a new coordinate system, with each coordinate ranked by how much of the data variance it explains. By selecting the first $k < d$ *principal components*, we obtain a lower-dimensional representation of the data that is guaranteed to explain more of the variance in the feature space than any other lower-dimensional representation. In the experiments below, we first transform the feature space using PCA, selecting the k components that explain 95% of the data variance.

10.3 STYLISTIC TRANSFER

The LMNN algorithm gives us a way to learn a distance metric that reflects programmer style, but only if we have training data with authorship labels; by itself it provides no additional leverage for stylistic clustering on a set of programs for which no training data exist. However, with metric learning it may be possible to *transfer* stylistic knowledge from one data set to another. We observe that if there exist a subset of features that truly reflect programmer style in a general way, then they can be learned from programs written by *any* set of authors; although the programs that we wish to cluster may have no training data, we can derive a metric from a different collection of programs with author labels.

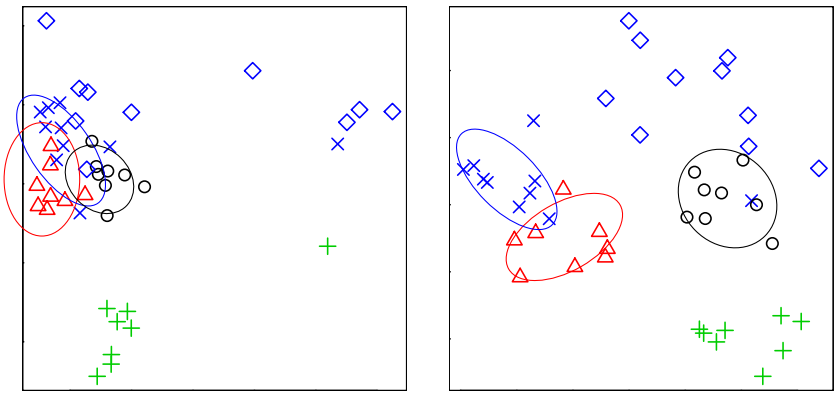
More precisely, consider two sets of programs $\mathcal{P}_1, \cdots, \mathcal{P}_\ell$ and $\mathcal{P}_{\ell+1}, \cdots, \mathcal{P}_{\ell+u}$, with known author labels $y_1, \cdots, y_\ell$; the authors for the unlabeled programs may or may not coincide with those of the labeled programs. If both sets of programs have the same representation (i.e., they use the same features), then we can learn a stylistic distance metric from the labeled programs and apply it to clustering the unlabeled programs.

We define a two part algorithm for transferring stylistic knowledge from the labeled data to the unlabeled data:

1. Learn a metric A using LMNN over ℓ labeled programs $\mathcal{P}_1, \cdots, \mathcal{P}_\ell$ such that the distance in the feature space between two programs with the same author is always less than the distance between two programs with different authors.

2. Cluster u unlabeled programs $\mathcal{P}_{\ell+1}, \cdots, \mathcal{P}_{\ell+u}$ using the distance function D_A.

Figure 10.1 depicts a set of five authors' programs before and after applying the transformation $\mathbf{x}' = L\mathbf{x}$, where L is derived from running LMNN on a different set of authors' programs (recall that $A = L^T L$). Only the first two dimensions of the data are plotted. In the un-transformed plot, several authors' programs are close



(a) (b)

Figure 10.1: Visualizing the first two principal components of programs written by five authors. The data in (a) are unmodified; in (b), the points have been transformed using a distance metric derived from a separate set of author-labeled programs.

together, making them difficult to distinguish. After the transformation based on the stylistic distance metric, the programs form more recognizably distinct groups. In the evaluation, below, we quantify the extent to which this knowledge transfer improves stylistic clustering.

As with authorship classification, there is a risk that the metric learning procedure might overfit the labeled author set, learning a metric that is too specific to the training data. We therefore define the *interpolated distance metric*

$$D'_A(\mathbf{x}_a, \mathbf{x}_b, \lambda) = \sqrt{(\mathbf{x}_a - \mathbf{x}_b)^T(\lambda A + (1-\lambda)I)(\mathbf{x}_a - \mathbf{x}_b)},$$

where λ is a parameter controlling how much the metric is allowed to diverge from the identity matrix. Clearly, when λ is 1 the data space is fully transformed by the learned metric; when λ is 0 D'_A reduces to euclidean distance in the original feature space. We choose $\lambda \in (0, 0.5, \ldots, .95, 1)$ through cross validation on the labeled training set. Our experimental results suggest that overfitting is rarely an issue, at least with our program representation and evaluation data set: the λ selected through cross validation is always in the interval $[.9, 1]$.

10.4 EVALUATION

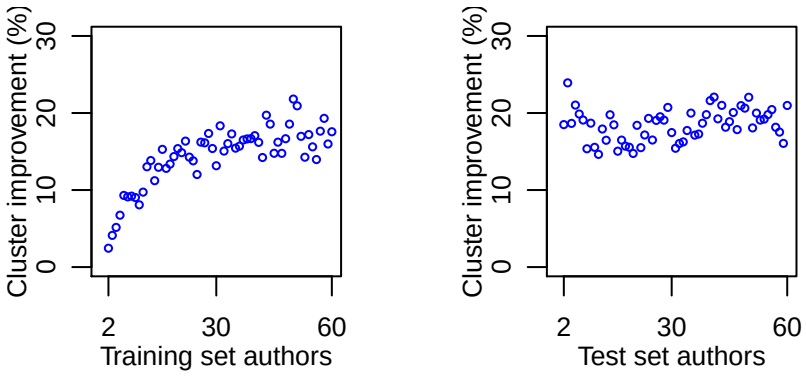
We evaluated authorship clustering to determine how well the clusters reflect the ground truth program authorship, and whether stylistic characteristics learned from one set of authors can improve the clustering of programs written by different authors (i.e., how well stylistic knowledge generalizes). Unlike classifiers, clustering algorithms have no notion of candidate labels, so cluster assignments are evaluated against the ground truth authors with measures based on cluster *agreement*: whether programs by the same author are assigned to the same cluster, and programs by different authors are assigned to different clusters. We computed several common measures of cluster agreement, including Adjusted Mutual Information (AMI), Normalized Mutual Information (NMI), and the Adjusted Rand Index (ARI); we prefer AMI because it is stable across different numbers of clusters, easing comparison of different data sets [96]. All of the measures we use take values in the range $[0, 1]$, where higher scores indicate better cluster agreement.

The cluster agreement measures that we use can be difficult to interpret, compared to the straightforward accuracy criterion by which we evaluate authorship classification. We therefore compute cluster assignment accuracy based on *majority vote*: after clustering, we use the ground truth authorship labels to compute the author with the most programs represented in each cluster, and assign that author label to every other program in the cluster.[1] The combination of clustering and majority vote is similar to applying the k-nearest-neighbors algorithm [32], which classifies a new example based on the majority label of its k nearest neighbors. Evaluated this way, cluster assignments are more readily comparable to classifier accuracy, which may ease interpretation of our results.

We performed several experiments to evaluate authorship clustering:

1. We randomly selected N authors from the Code Jam 2010 corpus and used LMNN to learn a distance metric over the feature space. We then randomly selected 30 different authors and clustered their programs using k-means with and without transforming the data with the learned metric. Since there are multiple sources of randomness in this experiment (both in selecting the data sets and in the k-means clustering algorithm), we repeated the experiment 20 times and computed the average AMI. Figure 10.2a depicts clustering improvement over the un-transformed data as a function of N. As expected, using more training authors to compute a metric leads a greater improvement. We conclude that stylistic information derived from one set of authors can be transferred to improve clustering of programs written by a different set of authors.

[1] This is more properly termed a 'plurality' vote, though 'majority' is the term used in the literature.



(a) (b)

Figure 10.2: Clustering with metric learning. The improvement over the original clustering $(\text{AMI}_{metric} - \text{AMI}_{orig.})/\text{AMI}_{orig.}$ is illustrated as a function of the number of training authors (a) and the true number of testing authors (b).

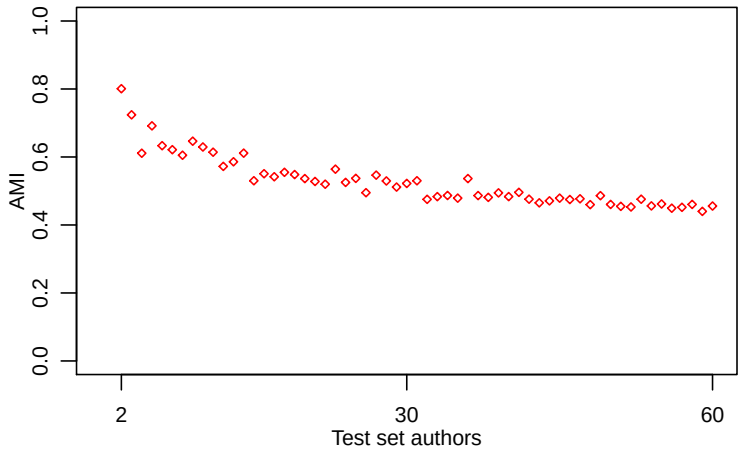


Figure 10.3: Cluster score (AMI) decreases as the number of represented authors increases. The authors' programs were clustered using a metric learned from 30 different training authors.

2. We performed a similar set of experiments with the number of authors used to compute the metric fixed at 30 to evaluate whether the clustering improvement is affected by the number of test set authors. Figure 10.2b shows that that the improvement due to incorporation of the stylistic metric is nearly invariant for a range of test set sizes.

3. Figure 10.3 depicts the AMI measure for metric-transformed cluster assignments as a function of test set sizes, with the metric training set fixed at 30 authors; clustering 60 authors using the learned metric achieves an average AMI of .456. Cluster evaluation measures can be difficult to interpret. For comparison, the labels predicted by a 60-author supervised classifier that achieves 64% accuracy receive an AMI of .503.

4. We compared the majority-vote accuracy of cluster assignment to the accuracy of a supervised classifier applied to a data set representing the same number of authors. The accuracies of the two approaches are depicted in Figure 10.4. These results are not directly comparable for two reasons. First, the although the same data set was used, the random way that we generate data subsets means that the authors in each classification and clustering experiment varied. We mitigate the impact of this issue by repeating the experiment multiple times, as described above, and reporting the average accuracy.

   More importantly, the classifier was trained on a reserved subset of data for the same authors on which it is evaluated. The clustering results, on the other hand, rely on a stylistic metric that was learned from an entirely different set of authors' programs; despite this, the cluster assignment labels are only 19% less accurate than those produced by the supervised classifier.

Table 10.1 compares the results of clustering 10 authors' programs with and without metric transformation. The cluster quality measures we compute are highly variable, due to the random nature of training and test set selection and the inherent randomness in the clustering algorithm; nonetheless, the improvement offered by the learned metric is significant at a 95% confidence level for all measures.

10.5 SUMMARY

Authorship clustering, or grouping programs by stylistic similarity, is challenging because unsupervised clustering techniques may identify groups of programs that reflect some other provenance property. We have presented a technique based on metric learning that transfers stylistic knowledge from a data set for which we have

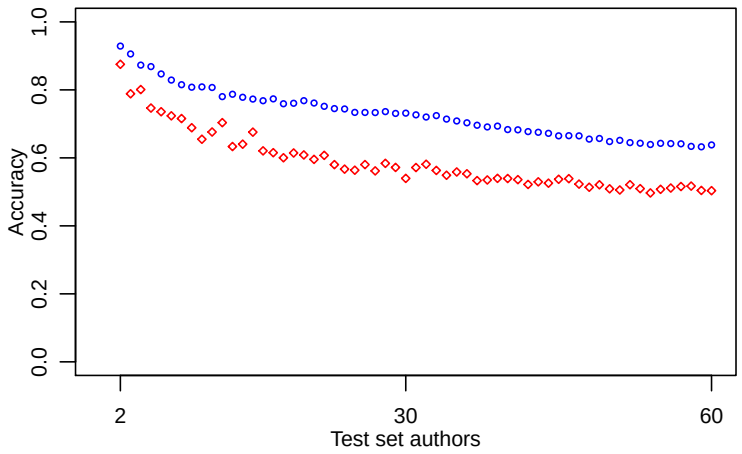


Figure 10.4: Accuracy of majority-vote labeling based on ground truth (◇), compared to the accuracy of a classifier evaluated against a data set of the same size (○). The results are not directly comparable; the classifier is trained on data from the same set of authors that it is tested on, while the stylistic clustering algorithm learns a metric from a different set of authors. Clustering after knowledge transfer underperforms classification by only 19% on average.

Table 10.1: Cluster evaluation measures for 10 test authors, using metrics learned from 30 different authors.

	AMI	spread	NMI	spread	ARI	spread
		0 1		0 1		0 1
no transformation	.510		.637		.406	
learned metric	.606		.723		.480	

author labels to another, unlabeled data set. We performed a series of experiments showing that this knowledge transfer improves the performance of the simple k-means clustering algorithm by 20%. The clustering results further support our conclusion from Chapter 9 that programmer style is preserved in binary code.

The techniques we described in this chapter were part of an exploratory study, and leave many issues open. In particular, the choice of k-means as a clustering algorithm was made in the interest of simplicity; k-means has several drawbacks, in particular its sensitivity to the choice of initial cluster centroids, and different clustering algorithms

are worth studying for this problem. Furthermore, k-means depends on the *a priori* choice of the target number of clusters. In a real application, a reasonable estimate for this value may or may not be available. Authorship clustering may benefit from the application of nonparametric Bayesian methods that replace such a fixed parameter with a prior probability distribution [39].

11

Conclusion

Our goal in this dissertation has been to explore computer program provenance, the details of the process through which a program is transformed from source code to a program binary. Our work arose from the hypothesis that program binaries reflect the various components of the production process—from the programmer's stylistic decisions down to the optimization options passed to the compiler—and that characteristics of binary code can be used to infer how a program was produced. The specifics of a program's provenance have applications in computer security, software forensics, and software engineering. In this chapter, we review our technical contributions and suggest some possible directions for future provenance research.

11.1 CONTRIBUTIONS

To the best of our knowledge, ours is the first study of program provenance recovery, and the first to formulate program provenance as a hierarchy of stages that impart specific characteristics to the eventual program binary. We have established contributions in three main areas:

Provenance models We developed models that capture program provenance in terms of features of binary code. Our models are based on representations of code at several levels of abstraction: as a string of bytes, as a sequence of instructions, as an interprocedural control flow graph, and as a collection of functions. These representations allowed us to define a large set of binary code features that, taken together, reflect the details of program provenance. The models map these features to provenance properties, providing a framework with which to automatically learn the binary code characteristics that are determined by a program's provenance.

Provenance recovery We constructed a system that recovers provenance properties from program binaries. Our system uses statistical machine learning techniques

to estimate the parameters of provenance models, and uses these models to infer the provenance of binary code. We used provenance recovery for a variety of tasks: to recognize the function entry preambles that are characteristic of several compilers to improve a stripped binary parser; to identify the compiler family, compiler version, and optimization level of binary code; to recognize the source language that program was written in; and to perform authorship attribution, identifying the programmer based on binary code features alone.

Programmer style We provided the first evidence that a programmer's stylistic traits are preserved throughout the compilation process. Using our provenance recovery framework, we automatically identified code features associated with variations in programming style and used these features to construct classifiers that can identify the author of a program binary. We extended this work to stylistic clustering, a technique in which programs are grouped by stylistic similarity, despite the lack of training data or knowledge of which specific authors are represented in the set of programs to be clustered.

We evaluated our provenance recovery techniques with experimental studies of large corpora of programs with a variety of provenance properties. Our experimental results demonstrate the both soundness of our central hypothesis—that program provenance is reflected in binary code—and the efficacy of our techniques.

11.2 Future Directions

We see several opportunities to build on the ideas we have presented here. Our work has focused on techniques to recover the properties of several levels of the provenance hierarchy; we believe that viable avenues of research exist for expanding the scope of provenance recovery and in extending the modeling and recovery techniques that we developed.

There are many more aspects of program provenance than what we have presented here. In the provenance hierarchy we established in Chapter 4, we have only addressed the authorship, language, and toolchain levels, leaving open the question of what properties of programs functionality or of the program build environment could be recovered. From a security and forensics perspective, we believe that inferring build environment characteristics like the versions of system libraries could be valuable information. Even within the levels of the provenance hierarchy that we explored, there is room for further research. For example, binary obfuscation tools are a post-compilation part of the binary toolchain that are frequently used by malicious programmers; recovering the identity of obfuscation tools could ease techniques that analyze obfuscated code [79].

The provenance recovery techniques that we developed were largely based on well-studied supervised learning models, in particular support vector machines and conditional random fields. These models served us well for the most part, due to the *closed world* nature of the provenance problems that we addressed, where all of the possible values for a provenance property are known in advance. Our stylistic clustering work exposed a deficiency in these tools, however; although we were able to derive a technique to transfer information about what constitutes programmer style to a target domain for unsupervised clustering, the k-means algorithm we used required *a priori* specification of the desired number of clusters. In a truly *open world* problem, no such guess would be feasible: the objective of stylistically clustering malware binaries, for example, might be to both find programs with similar style and to determine the number of actors operating in the market. We believe that fully extending provenance recovery to open world domains offers a rich set of research challenges, particular in terms of developing new machine learning models that apply to provenance properties with unbounded values.

Given our experience with provenance recovery and the open questions above, we believe that the following directions are likely to be fruitful for future program provenance research:

Open-world provenance Eliminating the assumed knowledge of the identity or number of provenance property variations will be necessary to extend provenance recovery to open domains. The machinery of nonparametric Bayesian modeling [39] seems particularly suitable for open-world provenance recovery. For example, the authorship clustering task could eliminate the *a priori* specification of the number of authors by using a nonparametric approach like the Dirichlet process mixture model (DPMM) [62], where a set of programs is assumed to represent the work of some portion of an infinite number of unique authors.

Our initial experiments with DPMM-based authorship clustering (not reported in this dissertation) have been inconclusive. These experiments assumed that authors are described by a multinomial distribution over binary code features, which simplifies implementation of the models but may not be realistic. Further investigation of authorship clustering using non-conjugate priors may be productive. Alternatively, future work could dispense with the assumption that programmers have a single "style" altogether, instead describing programs as the product of programming with a mixture of different styles; this approach may also be useful for representing programs written by several authors. Such a model could be based on the Indian Buffet Process, a nonparametric prior for infinite collections of latent properties [20].

Non-code features The features we used for provenance modeling were all based on the binary code, building either on the machine instructions or on higher level abstractions like control flow. Features based on data-oriented characteristics may increase the accuracy of existing provenance recovery techniques or open a path to recover new provenance properties. For example, techniques to recover the use of particular data structures [51] might improve the accuracy of authorship classification or clustering models.

Program provenance and code "social networks" One interesting avenue for future research is the use of provenance recovery to connect programmers or entities (e.g., companies or other organizations) based on provenance recovered from a set of programs. For example, applying authorship clustering techniques at the sub-program level (e.g., to code segments determined through other means to implement specific functionality, such as command and control or encryption) could highlight connections between programmers participating in the underground malware economy. Numerous challenges exist in this domain—for example, even analyzing malicious code is difficult [79], and obtaining ground truth authorship data for evaluation is likely to be challenging—but results in this area could have high impact.

The preceding research directions are by no means comprehensive. In this dissertation, we have provided the results of an initial foray into a wide-ranging set of open questions. Further investigations will doubtlessly follow paths that we cannot predict; our only hope is that the problems we have introduced will prove sufficiently interesting to motivate future researchers.

A

Self-repairing disassembly

The x86 and x86-64 instruction sets, like all variable-length instruction sets, exhibit a *self-repairing disassembly* property: the instruction sequences produced by disassembling from conflicting offsets in the binary will frequently align quickly to the same instruction stream. Consider a disassembly sequence A starting at some arbitrary byte offset, and a second disassembly sequence B starting d bytes later. It is sufficient to consider $0 < d < K$, where K is the length in bytes of the longest instruction (including its operands, if any) in the instruction set, because otherwise we can always remove the first few instructions in A. When will A's and B's disassembly instruction align?

We define the instruction-length distribution $p(l), l = 1 \ldots K$ as the probability that a random instruction has a length of l bytes. The distribution $p(l)$ can be estimated through exhaustive disassembly on a large set of programs. Then $p(l)$ is the fraction of disassembled instructions with length l. Obviously $p(l) \geq 0$, $\sum_{l=1}^{K} p(l) = 1$. We define $2K - 1$ states $s_{-(K-1)}, \ldots, s_0, \ldots, s_{K-1}$. We define a probabilistic transition matrix T between the states, where the probability of move from state s_i to s_j is

$$T_{ij} = \begin{cases} p(i-j) & \text{if } i > 0 \text{ and } j \in \{i-K, \ldots, i-1\} \\ p(j-i) & \text{if } i < 0 \text{ and } j \in \{i+1, \ldots, i+K\} \\ 1 & \text{if } i = j = 0 \\ 0 & \text{otherwise} \end{cases}$$

Let Q be the $(2K-2) \times (2K-2)$ submatrix of T by removing the row and column corresponding to s_0. Finally, let the *fundamental matrix* be $N = (I - Q)^{-1}$. Note both Q and N are indexed by $-(K-1), \ldots, -1, 1, \ldots, K-1$.

Theorem A.1. *If disassembly sequence B starts* $0 < d < K$ *bytes after sequence A, then the expected number of disassembled* instructions *in A, before A and B align, is* $\sum_{j=1}^{K-1} N_{dj}$.

Proof. We can view self-repair as a game between players A and B. Whichever player is behind can disassemble the next instruction in its sequence, which is equivalent to

throwing a biased, K-sided die with probability for side l being $p(l)$. That player then advances l bytes. The game repeats until A and B arrive at the same byte, i.e., align. The number of moves A takes until the game is over is the number of disassembled instructions in sequence A before alignment.

The signed distance d of how far A is behind B is the "state" of the game. Initially A is d bytes behind B, so the game is in state s_d. A will advance $l \sim p(l)$ bytes ahead. The distance becomes $d - l$. We say that the game made a transition from state s_d to state s_{d-l}. The general transition rule is $s_d \to s_{(d-\mathrm{sgn}(d)l)}$ with probability $p(l)$, where $\mathrm{sgn}(d) \in \{-1, 1\}$ is the sign function. It is easy to verify that this defines a proper Markov chain random walk on states $s_{-(K-1)}, \ldots, s_0, \ldots, s_{K-1}$. Importantly, s_0 is a special state—the game ends upon reaching s_0. We can model it by turning s_0 into an *absorbing state* in the Markov chain. The resulting absorbing Markov chain is precisely the transition matrix T.

Given T, it is well known that the corresponding fundamental matrix N contains the length of random walks [19]. Specifically, N_{ij} is the expected number of times a random walk starting at state s_i would visit s_j before absorption. Here i, j are in $\{-(K-1), \ldots, -1, 1, \ldots, K-1\}$. Our random walks always start at s_d. Since it is A's turn to move whenever a random walk visits a state $j > 0$, the total expected moves A will make is

$$\sum_{j>0} N_{dj} = \sum_{j=1}^{K-1} N_{dj}.$$

□

Corollary A.2. *The expected number of* bytes *in A, before A and B align, is*

$$\left(\sum_{j=1}^{K-1} N_{dj}\right)\left(\sum_{l=1}^{k} l\,p(l)\right).$$

Proof. Each time A moves, it advances $\sum_{l=1}^{k} l\,p(l)$ bytes by expectation. □

We obtained an estimate of $p(l)$ by exhaustively disassembling the data we use in the evaluation section of Chapter 7. Our analysis indicates that disassembly from nearby byte offsets will align very quickly. Disassembled sequences offset from one other by a single byte are expected to align in 2.2 instructions; sequences offset by three bytes align in 2.7 instructions. Observations of self-repairing disassembly in the real binaries agree closely with these figures.

References

[1] *AMD64 Architecture Programmer's Manual*, volume 3. Advanced Micro Devices, September 2007.

[2] Aloise, Daniel, Amit Deshpande, Pierre Hansen, and Preyas Popat. NP-hardness of euclidean sum-of-squares clustering. In *Machine Learning*, 75(2), May 2009.

[3] Andrzejewski, David, Xiaojin Zhu, Mark Craven, and Ben Recht. A framework for incorporating general domain knowledge into Latent Dirichlet Allocation using First-Order Logic. In *The Twenty-Second International Joint Conference on Artificial Intelligence (IJCAI)*, Barcelona, Catalonia, Spain, July 2011.

[4] Balakrishnan, Gogul, and Thomas Reps. WYSINWYX: What you see is not what you eXecute. In *ACM Transactions on Programming Languages and Systems*, 32 (6), August 2010.

[5] Basu, Sugato, Ian Davidson, and Kiri Wagstaff, editors. *Constrained Clustering: Advances in Algorithms, Theory, and Applications*. Chapman & Hall/CRC Press, Boca Raton, Flordia, 2009.

[6] Bayer, Ulrich, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Krügel, and Engin Kirda. Scalable, behavior-based malware clustering. In *Network and Distrtributed System Security Symposium (NDSS)*, San Diego, California, February 2009.

[7] Ben-Gal, Irad. Outlier detection. In Maimon, Oded, and Lior Rokach, editors, *Data Mining and Knowledge Discovery Handbook*. Springer-Verlag, New York, New York, 2005.

[8] Bishop, Christopher M. *Pattern Recognition and Machine Learning*. Springer-Verlag, New York, 2006.

[9] Blum, B. I. A simple expert system. In *SIGBIO Newsletter*, 10(1), March 1988.

[10] Bruschi, Danilo, Lorenzo Martignoni, and Mattia Monga. Detecting self-mutating malware using control-flow graph matching. In *Third international Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, Berlin, Germany, July 2006.

[11] Burges, Christopher J. C. A tutorial on support vector machines for pattern recognition. In *Data Mining and Knowledge Discovery*, 2(2), June 1998.

[12] Cifuentes, Christina, and Dough Simon. Procedure abstraction recovery from binary code. In *Fourth European Conference on Software Maintenance and Reengineering*, Zurich, Switzerland, February 2000.

[13] Cifuentes, Cristina, and Mike Van Emmerik. Recovery of jump table case statements from binary code. In *Seventh International Workshop on Program Comprehension (IWPC)*, Pittsburgh, Pennsylvania, May 1999.

[14] Cifuentes, Cristina, and Mike Van Emmerik. UQBT: Adaptable binary translation at low cost. In *Computer*, 33(3), March 2000.

[15] Cifuentes, Cristina, and K. John Gough. Decompilation of binary programs. In *Software–Practice and Experience*, 25(7), July 1995.

[16] Cifuentes, Cristina, Mike van Emmerik, Norman Ramsey, and Brian Lewis. Procedure abstraction recovery. In *The University of Queensland Binary Translator (UQBT) Framework*. The University of Queensland, Sun Microsystems, Inc, 2001.

[17] Cook, Stephen A. The complexity of theorem-proving procedures. In *Third Annual ACM Symposium on Theory of Computing*, Shaker Heights, Ohio, May 1971.

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

[19] Doyle, Peter G., and J. Laurie Snell. *Random Walks and Electrical Networks*. Mathematical Association of America, Washington, D.C., 1984.

[20] Driffiths, Thomas L., and Zoubin Ghahramani. Infinite latent feature models and the indian buffet process. Technical Report GNCU TR 2005-001, Gatsby Computational Neuroscience Unit, Univerity College London, May 2005.

[21] Druck, Gregory, Gideon Mann, and Andrew McCallum. Learning from labeled features using generalized expectation criteria. In *Thirty-first Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Singapore, July 2008.

[22] Dullien, Thomas, and Rolf Rolles. Graph-based comparison of executable objects. In *Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC)*, Rennes, France, June 2005.

[23] 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(Aug), August 2008.

[24] Flake, Halvar. Structural comparison of executable objects. In *Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, Dortmund, Germany, July 2004.

[25] Fogel, David, John C. Hanson, Russell Kick, Heidar A. Malki, Charles Sigwart, Michael Stinson, and Efraim Turban. The impact of machine learning on expert systems. In *ACM Conference on Computer Science*, Indianapolis, Indiana, February 1993.

[26] Fredrikson, Matt, Somesh Jha, Mihai Christodorescu, Reiner Sailer, and Xifeng Yan. Synthesizing near-optimal malware specifications from suspicious behaviors. In *IEEE Symposium on Security and Privacy*, Oakland, California, May 2010.

[27] Frey, Brendan J., and David J. C. Mackay. A revolution: Belief propagation in graphs with cycles. In *Advances in Neural Information Processing Systems (NIPS)*, Denver, Colorado, December 1997.

[28] Giffin, Jonathon T., Somesh Jha, and Barton P. Miller. Efficient context-sensitive intrusion detection. In *Network and Distrtributed System Security Symposium (NDSS)*, San Diego, California, February 2004.

[29] GNU Project. GNU binutils, 2011. URL `http://www.gnu.org/software/binutils`.

[30] Gray, Andrew, Philip Sallis, and Stephen MacDonell. Software forensics: Extending authorship analysis techniques to computer programs. In *3rd Biennial Conference of the International Association of Forensic Linguists*, Durham, North Carolina, September 1997.

[31] Harris, Laune C., and Barton P. Miller. Practical analysis of stripped binary code. In *SIGARCH Computer Architecture News*, 33(5), December 2005.

[32] Hastie, Trevor, Robert Tibhirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer-Verlag, New York, fifth edition, 2001.

[33] Hayes, Jane Huffman, and Jeff Offutt. Recognizing authors: An examination of the consistent programmer hypothesis. In *Software Testing, Verification and Reliability*, 20(4), December 2010.

[34] He, Xuming, Richard S. Zemel, and Miguel Á. Carreira-Perpi nán. Multiscale conditional random fields for image labeling. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Washington, DC, June 2004.

[35] Henchiri, O., and N. Japkowicz. A feature selection and evaluation scheme for computer virus detection. In *Sixth International Conference on Data Minding (ICDM)*, Hong Kong, December 2006.

[36] Hex-Rays. IDA Pro disassembler, 2011. URL `http://www.hex-rays.com/idapro`.

[37] Hollingsworth, Jeffrey K., Barton P. Miller, and Jon Cargille. Dynamic program instrumentation for scalable performance tools. Technical Report CS-TR-1994-1207, University of Wisconsin, 1994. URL `ftp://ftp.cs.wisc.edu/paradyn/papers/Hollingsworth94Dynamic.pdf`.

[38] *IA-32 Intel Architecture Software Developer's Manual*, volume 2. Intel Corporation, March 2006.

[39] Jordan, Michael I. Bayesian nonparametric learning: Expressive priors for intelligent systems. In Dechter, Rina, Hector Geffner, and Joseph Y. Halpern, editors, *Heuristics, Probability and Causality*. College Publications, 2010.

[40] Juola, Patrick. Authorship attribution. In *Foundations and Trends in Information Retrieval*, 1(3), December 2006.

[41] King, James C. Symbolic execution and program testing. In *Communications of the ACM*, 19(7), July 1976.

[42] Knuth, Donald E. Backus normal form vs. backus naur form. In *Communications of the ACM*, 7(12), December 1964.

[43] Koller, Daphne, and Nir Friedman. *Probabilistic Graphical Models: Principles And Techniques*. The MIT Press, Cambridge, Massachusettes, 2009.

[44] Kolter, J. Zico, and Marcus A. Maloof. Learning to detect and classify malicious executables in the wild. In *Journal of Machine Learning Research*, 7, December 2006.

[45] Krsul, Ivan, and Eugene H. Spafford. Authorship analysis: Identifying the author of a program. In *Computers & Security*, 16(3), 1997.

[46] Kruegel, C., E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *International Symposium on Recent Advances in Intrusion Detection (RAID)*, Seattle, Washington, September 2005.

[47] Kruegel, Christopher, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In *Thirteenth USENIX Security Symposium*, San Diego, California, August 2004.

[48] Kulis, Brian. ICML 2010 tutorial on metric learning, June 2010. URL `http://www.eecs.berkeley.edu/~kulis/icml2010_tutorial.htm`.

[49] Kuramochi, Michihiro, and George Karypis. Frequent subgraph discovery. In *International Conference on Data Mining (ICDM)*, San Jose, California, November 2001.

[50] Lafferty, John, Andrew McCallum, and Fernando Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Eighteenth International Conference on Machine Learning (ICML)*, Williamstown, Massachusettes, June 2001.

[51] Lee, JongHyup, Thanassis Avgerinos, and David Brumley. TIE: Principled reverse engineering of types in binary programs. In *Network and Distributed System Security Symposium (NDSS)*, San Diego, California, February 2011.

[52] Li, Shengying. A survey on tools for binary code analysis, 2004. URL `http://www.ecsl.cs.sunysb.edu/tr/BinaryAnalysis.doc`.

[53] Li, Wei-Jen, Ke Wang, Salvatore J. Stolfo, and B. Herzog. Fileprints: identifying file types by n-gram analysis. In *Sixth IEEE Information Assurance Workshop (IAW)*, June 2005.

[54] Linn, Cullen, and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Tenth ACM conference on Computer and Communications Security (CCS)*, Washington, DC, October 2003.

[55] Litzkow, Michael J., Miron Livny, and Matt W. Mutka. Condor - a hunter of idle workstations. In *Eighth International Conference on Distributed Computing Systems (ICDCS)*, San Jose, CA, June 1988.

[56] MacDonell, S.G., A.R. Gray, G. MacLennan, and P.J. Sallis. Software forensics for discriminating between program authors using case-based reasoning, feedforward neural networks and multiple discriminant analysis. In *Sixth International Conference on Neural Information Processing (ICONIP)*, Perth, Australia, November 1999.

[57] Mahalanobis, Prasanta Chandra. On the generalised distance in statistics. In *Proceedings of the National Institute of Sciences of India*, 2(1), April 1936.

[58] McCallum, Andrew. Efficiently inducing features of conditional random fields. In *Nineteenth Conference in Uncertainty in Artificial Intelligence*, Acapulco, Mexico, August 2003.

[59] McCallum, Andrew Kachites. MALLET: A machine learning for language toolkit, 2002. URL `http://www.cs.umass.edu/~mccallum/mallet`.

[60] Mitchell, Tom M. *Machine Learning*. McGraw-Hill, New York, New York, 1997.

[61] Muchnick, Steven S. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, California, 1997.

[62] Neal, Radford M. Markov chain sampling methods for dirichlet process mixture models. Technical Report 9815, University of Toronto, September 1998.

[63] Orso, Alessandro. Monitoring, analysis, and testing of deployed software. In *FSE/SDP Workshop on Future of Software Engineering Research*, Santa Fe, New Mexico, November 2010.

[64] Palmer, Gary. A road map for digital forensic research. Technical Report DTR-T001-01 FINAL, Digital Forensics Research Workshop (DFRWS), August 2001.

[65] Paradyn Project. Dyninst: An application program interface for runtime code generation, 2011. URL `http://www.paradyn.org`.

[66] Paradyn Project. InstructionAPI: An application program interface for instruction parsing, 2011. URL `http://www.paradyn.org/html/instruction7.0.1-features.html`.

[67] Paradyn Project. ORIGIN: Toolchain identification for program binaries, 2011. URL `http://www.paradyn.org/origin`.

[68] Paradyn Project. ParseAPI: An application program interface for binary parsing, 2011. URL `http://www.paradyn.org/html/parse7.0.1-features.html`.

[69] Peng, Fuchun, and Andrew McCallum. Accurate information extraction from research pappers using conditional random fields. In *Human Language Technology Conference and North American Chapter of the Association for Computational Linguistics (HLT-NAACL)*, Boston, Massachusettes, May 2004.

[70] Prasad, Manish, and Tzi-cker Chiueh. A binary rewriting defense against stack based buffer overflow attacks. In *2003 USENIX Annual Technical Conference*, San Antonio, Texas, June 2003.

[71] Pržulj, N., D.G Corneil, and I. Jurisca. Modeling interactome: Scale-free or geometric? In *Bioinformatics*, 20(10), July 2004.

[72] Quinlan, Daniel, and Thomas Panas. Source code and binary analysis of software defects. In *Workshop on Cyber Security and Information Intelligence Research (CSIIRW)*, Oak Ridge, Tennessee, April 2009.

[73] Ramsey, Norman, and Mary F. Fernández. Specifying representations of machine instructions. In *ACM Transactions on Programming Languages and Systems*, 19 (3), May 1997.

[74] Rosenblum, Nathan E., Xiaojin Zhu, Barton P. Miller, and Karen Hunt. Machine learning-assisted binary code analysis. In *NIPS Workshop on Machine Learning in Adversarial Environments for Computer Security*, Whistler, British Columbia, Canada, December 2007.

[75] Rosenblum, Nathan E., Xiaojin Zhu, Barton P. Miller, and Karen Hunt. Learning to analyze binary computer code. In *Twenty-third conference on Artificial Intelligence (AAAI)*, Chicago, Illinois, July 2008.

[76] Rosenblum, Nathan E., Barton P. Miller, and Xiaojin Zhu. Extracting compiler provenance from program binaries. In *Nineth ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE)*, Toronto, Ontario, Canada, June 2010.

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

[78] Rosenblum, Nathan E., Zhu Xiaojin, and Barton P. Miller. Who wrote this code? Identifying the authors of program binaries. In *Sixteenth European Symposium on Research in Computer Security (ESORICS)*, Leuven, Belgium, September 2011.

[79] Roundy, Kevin, and Barton P. Miller. Hybrid analysis and control of malware binaries. In *Recent Advances in Intrusion Detection (RAID)*, Ottowa, Canada, September 2010.

[80] Saebjornsen, Andreas, Jeremiah Willcock, Thomas Panas, Daniel Quinlan, and Zhendong Su. Detecting code clones in binary executables. In *International Symposium on Software Testing and Analysis (ISSTA)*, Chicago, Illinois, July 2009.

[81] Schleimer, Saul, Daniel S. Wilkerson, and Alex Aiken. Winnowing: Local algorithms for document fingerprinting. In *ACM SIGMOD International Conference on Management of Data*, San Diego, CA, June 2003.

[82] Schordan, Markus, and Dan Quinlan. A source-to-source architecture for user-defined optimizations. In *Joint Modular Languages Conference (JMLC)*, Klagenfurt, Austria, August 2003.

[83] Schultz, M.G., E. Eskin, F. Zadok, and S.J. Stolfo. Data mining methods for detection of new malicious executables. In *IEEE Symposium on Security and Privacy*, Oakland, California, May 2001.

[84] Schwarz, Benjamin, Saumya Debray, and Gregory R. Andrews. Disassembly of executable code revisited. In *Ninth Working Conference on Reverse Engineering (WCRE)*, Richmond, Virginia, October 2002.

[85] Security, Panda. PandaLabs annual report, January 2010. URL `http://pandalabs.pandasecurity.com/pandalabs-annual-report-2010`.

[86] Settles, Burr. ABNER: An open source tool for automatically tagging genes, proteins, and other entity names in text. In *Bioinformatics*, 21(14), July 2005.

[87] Settles, Burr. Closing the loop: Fast, interactive semi-supervised annotation with queries on features and instances. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Edinburgh, Scotland, July 2011.

[88] Shafiq, M. Zubair, S. Momina Tabish, Fauzan Mirza, and Muddassar Farooq. Peminer: Mining structural information to detect malicious executables in realtime. In *Twelfth International Symposium on Recent Advances in Intrusion Detection (RAID)*, Saint-Malo, France, September 2009.

[89] Sites, Richard L., Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. Binary translation. In *Communications of the ACM*, 36(2), February 1993.

[90] Spafford, Eugene H., and Stephen A. Weeber. Software forensics: Can we track code to its authors? Technical Report CSD-TR-92-010, Purdue University, February 1992.

[91] Sutton, Charles, and Andrew McCallum. An introduction to conditional random fields for relational learning. In Getoor, Lise, and Ben Taskar, editors, *Adaptive Computation and Machine Learning*. The MIT Press, Cambridge, Massachusettes, 2007.

[92] Thain, Douglas, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the Condor experience. In *Concurrency and Computation: Practice & Experience*, 17(2–4), February 2005.

[93] Theiling, Henrik. Extracting safe and precise control flow from binaries. In *Seventh international Workshop on Real-time Computing Systems and Applications (RTCSA)*, Cheju Island, South Korea, December 2000.

[94] Troger, J., and C. Cifuentes. Analysis of virtual method invocation for binary translation. In *Ninth Working Conference on Reverse Engineering (WCRE)*, Richmond, Virginia, October 2002.

[95] Van Emmerik, Mike, and Trent Waddington. Using a decompiler for real-world source recovery. In *Eleventh Working Conference on Reverse Engineering (WCRE)*, Delft, The Netherlands, November 2004.

[96] Vinh, Nguyen Xuan, Julien Epps, and James Bailey. Information theoretic measures for clusterings comparison: Is a correction for chance necessary? In *26th Annual International Conference on Machine Learning (ICML)*, Montreal, Quebec, Canada, June 2009.

[97] Wainright, Martin J., Tommi Jaakkola, and Alan S. Willsky. Tree-base reparameterization for approximate inference in loopy graphs. In *Advances in Neural Information Processing Systems (NIPS)*, Vancouver, British Columbia, Canada, December 2001.

[98] Walenstein, Andrew, and Arun Lakhotia. The software similarity problem in malware analysis. In *Dagstuhl Seminar on Duplication, Redundancy, and Similarity in Software*, Dagstuhl, Germany, July 2006.

[99] Weinberger, Kilian Q., and Lawrence K. Saul. Distance metric learning for large margin nearest neighbor classification. In *Journal of Machine Learning Research*, 10, February 2009.

[100] Whale, Geoff. Software metrics and plagiarism detection. In *Journal of Systems and Software*, 13(2), October 1990.

[101] Zhang, Like, and Gregory B. White. An approach to detect executable content for anomaly based network intrusion detection. In *International Parallel and Distributed Processing Symposium (IPDPS)*, Long Beach, California, March 2007.