Extracting Compiler Provenance from Program Binaries

Nathan E. Rosenblum Barton P. Miller Xiaojin Zhu

Computer Sciences Department, University of Wisconsin–Madison

{nater,bart,jerryzhu}@cs.wisc.edu

Abstract

We present a novel technique that identifies the source compiler of program binaries, an important element of *program provenance*. Program provenance answers fundamental questions of malware analysis and software forensics, such as whether programs are generated by similar tool chains; it also can allow development of debugging, performance analysis, and instrumentation tools specific to particular compilers. We formulate compiler identification as a structured learning problem, automatically building models to recognize sequences of binary code generated by particular compilers. We evaluate our techniques on a large set of real-world test binaries, showing that our models identify the source compiler of binary code with over 90% accuracy, even in the presence of interleaved code from multiple compilers. A case study demonstrates the use of inferred compiler provenance to augment stripped binary parsing, reducing parsing errors by 18%.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—compilers,code generation; I.5.1 [*Pattern Recognition*]: Models—statistical

General Terms Languages, Security

Keywords program provenance, forensics, static binary analysis

1. Introduction

Program binaries are often the subject of analysis in the areas of computer security, performance modeling, and program instrumentation and debugging. Security analysts and antivirus vendors are confronted daily with malicious programs whose behavior they must analyze and understand, and whose origins and relationships to existing threats are of paramount importance. Developers of debugging, performance analysis, and instrumentation software must design tools useful in the face of myriad variations of source languages, programming idioms, and diverse compiler families and types of optimizations. The question of *binary program provenance*—of the characteristics of a program that derive from its path from source code to executable form—informs many aspects of binary analysis. The utility of program provenance can be straightforward, as in the case of software forensics where authorship and code similarity are key details [8]. In other domains it can be more subtly useful, aiding the development of binary tools and analysis capabilities targeted at specific compilers or source languages [2, 14]. When presented with only with a program binary or a snippet of binary code, details of program provenance are not readily apparent. The black box between the program author and the program binary affords little foothold for tailored tools or analyses.

We have developed a novel method for identifying the source compiler of program binaries, a major element of program provenance. We formulate compiler identification as a structured learning task, automatically building models that classify sequences of binary code by the generating compiler. Because our approach relies only on characteristics of the binary code and not on meta-data or other details of program headers, it is applicable even when such information has been stripped or is otherwise unavailable. Furthermore, because our method classifies sequences of code instead of whole binaries, it can be used even when codes produced by multiple compilers coexist within a program binary, such as statically linked library code. Our tool extracts compiler provenance with high accuracy even in such complex programs.

Our previous research into precise static parsing of stripped program binaries—a foundational technique for binary code analysis—used compiler-specific models of code at function entry points to extend traditional parsing techniques that perform poorly without debugging symbols [14]. The existing approach requires prior knowledge of the source compiler, which may not be available. We augment the binary code parser to relax the known-compiler requirement, and show that adding inferred source compiler labels improves the precision of analysis for binaries of unknown provenance. While the high accuracy of the existing parser allows only small improvement in absolute terms, adding compiler provenance reduces parsing errors by 18%.

The remainder of this paper is structured as follows. We first briefly describe characteristics of binary code that influence the way we approach the compiler inference problem. We then present a model of binary code that captures characteristics of compiler provenance and evaluate our approach first on single-provenance binaries, and then on binaries containing a mixture of code from multiple compilers. We present a case study of provenance-augmented static parsing, then conclude with a review of related work and a discussion of our approach.

2. Binary Code Characteristics

The compiler is a critical part of producing almost all program binaries. Although the exact sequence of translations between original program source code and the program binary is often unknown, a compiler was almost certainly part of that process. If the choice of compiler has an effect on the resulting binary—as compiler vendors have been at such pains to convince us—then these differences should be manifest in the executable code produced by the translation process. These code features reflect design and implementation decisions by compiler designers, and can be key indicators of program provenance. Compiled programs have several properties that make compiler-specific features prevalent. The Intel IA-32 ar-

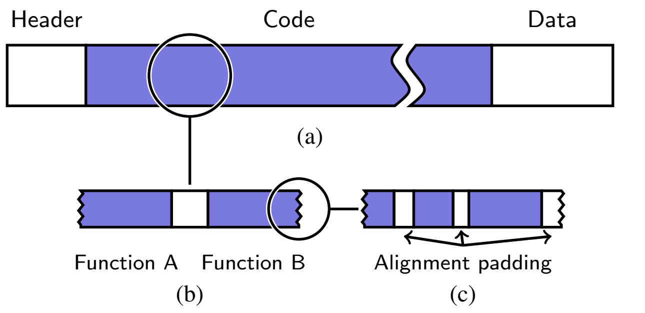


Figure 1. Layout of executable code in a typical binary. At the highest level of abstraction, the program's executable code resides in a contiguous segment of the binary, e.g. the `.text` segment in ELF binaries (a). Under closer consideration, there often exist non-executable bytes between functions (b) and even within functions (c).

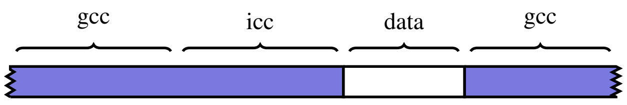
chitecture, our target platform, is a particularly rich source of such features.

The IA-32 has an expressive instruction set wherein multiple instruction sequences can perform the same operation. For example, a simple programmatic idiom like adding the constant 5 to a variable could be encoded by both the instruction `lea eax,[edi+5]` and the sequence `mov eax,edi; add eax,5`. The choice of one sequence or the other is ultimately determined by the compiler design choices. Factors like hardware characteristics or trade-offs in code size and performance optimizations influence these choices. Differences in compiler output thus depend on a specific set of often proprietary code generation and optimization policies. Systematic differences in binary code among different compilers reflect such distinguishing characteristics, providing evidence of program provenance.

Binary characteristics that reflect compiler provenance extend beyond the domain of executable instructions. Interpreting the executable code region of a binary solely as a sequence of instructions is often misleading. As illustrated in Figure 1, binaries frequently exhibit *gaps* between functions. These gaps may contain data such as jump tables or string constants, or they may contain regular padding instructions or arbitrary bytes. Such gaps even exist within individual functions, often due to performance-directed layout of branch targets. While the content of these gaps is sometimes dependent on the functionality of the program—for example, string constants or the addresses in jump tables—often the non-executable bytes in gaps or the very existence of gaps express compiler characteristics. One compiler might pad space for branch alignment with a sequence of `0x90`, the `nop` instruction, while another might use a different but semantically equivalent instruction of the appropriate length like `lea esi,[esi]`, use random byte values, or elide the padding altogether. The compiler provenance model we describe in the following section uses these characteristics to represent the content and structure of program binaries.

3. Source Compiler Modeling

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). The general idea of our approach is to learn the parameters of a probabilistic model of source compiler la-



beling in binaries, and to use this model to perform inference over a particular binary, labeling each byte with the most likely compiler.

The properties of binary code make probabilistic graphical models—models that capture the conditional dependence of many variables—well suited to our compiler inference task. 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.

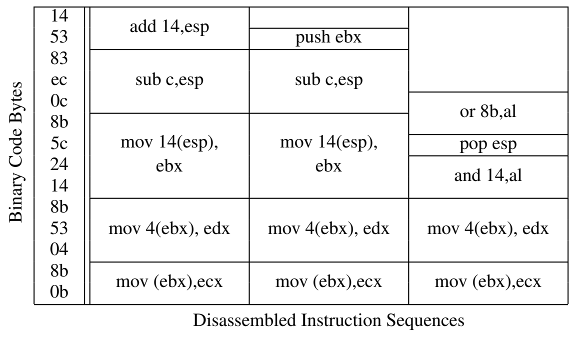
The program binary representation must capture the characteristics of the code and the non-code regions of the binary. While abstracting the program as a sequence of executable instructions is most natural, it is not sensible to label non-code regions consisting of data or random bytes as instructions. Furthermore, doing so requires statically identifying all instruction boundaries, itself a challenging task. We therefore model the binary uniformly as a sequence of bytes representing executable instructions intermingled with non-code. Our features are designed to capture the characteristics of the bytes underlying instructions and data in the same way.

Let the program binary $\mathcal{P}$ be a sequence of bytes at offsets $x_1 \ldots x_n$ in the binary. Our task is to assign labels $y_1 \ldots y_n$ to each byte, where each $y_i \in \mathcal{C}$ corresponds to a particular source compiler (e.g. `gcc`, `icc`, or `msvs`) or the special '`data`' label. The choice of features to represent the binary at each offset x_i has significant impact on the power of the model. Our previous experience with modeling *function entry idioms*—code sequences at and around function entry points—has demonstrated the power of *idiom features*: short sequences of instructions, similar to n-grams with optional single-instruction wildcards, that elide details such as literal arguments and memory offsets. For example, the idiom

$$u_1 = (\texttt{push ebp | * | mov esp,ebp})$$

would match offset x_i in the binary if disassembly from x_i yielded the sequence (`push ebp | push edi | mov esp,ebp`). In this case, we use idiom features to capture patterns indicative of compiler provenance. Importantly, idiom features can represent the binary not only at actual instruction offsets, but at any point including in data regions. This striking observation derives from the properties of the IA-32 instruction set that we described in the previous section. The architecture's opcode space is very dense; almost any byte is a valid opcode or beginning of a multi-byte opcode. This leads to the *self-repairing* property of IA-32 illustrated in Figure 2. Idiom features can be produced by disassembling from any offset in the binary, including in data regions. Spanning multiple bytes, these features implicitly incorporate information about multiple locations in the binary without the increased complexity of an arbitrarily structured graphical model.

We model the compiler label probability over the entire binary as a Conditional Random Field [9] with nodes $y_{1:n}$ representing the labels of every byte in the program. 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}. Each node is associated with one or more idiom features as



Binary Code Bytes			
14	add 14,esp		
53		push ebx	
83	sub c,esp	sub c,esp	
ec			
0c			or 8b,al
8b	mov 14(esp), ebx	mov 14(esp), ebx	
5c			pop esp
24			and 14,al
14			
8b	mov 4(ebx), edx	mov 4(ebx), edx	mov 4(ebx), edx
53			
04			
8b	mov (ebx),ecx	mov (ebx),ecx	mov (ebx),ecx
0b			

Disassembled Instruction Sequences

Figure 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.

indicated by a *feature function*. We define one such function for each idiom $u \in \mathcal{U}$ and compiler $c \in \mathcal{C}$ pair as

$$f_{I(u,c)}(x_i, y_i, \mathcal{P}) = \begin{cases} 1 & \text{if } y_i = c \text{ and idiom } u \text{ matches } \mathcal{P} \\ & \text{at offset } x_i \\ 0 & \text{otherwise.} \end{cases}$$

The transition relation between a label node and its neighbor is similarly defined as

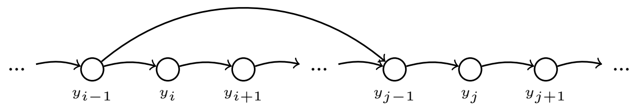
$$f_{T(c,c')}(y_i, \mathcal{P}) = \begin{cases} 1 & \text{if } y_i = c \text{ and } y_{i+1} = c' \\ 0 & \text{otherwise.} \end{cases}$$

Taken together, the unary idiom feature functions f_I and the binary transition features f_T define a *linear chain CRF* [11] over the binary, as illustrated in Figure 3. 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

$$P(y_{1:n}|x_{1:n}) = \frac{1}{Z} \exp\left(\sum_{i=1}^{n} \left[\sum_{u \in \mathcal{U}} \sum_{c \in \mathcal{C}} \lambda_{I(u,c)} f_{I(u,c)}(x_i, y_i) + \sum_{c,c' \in \mathcal{C}} \lambda_{T(c,c')} f_{T(c,c')}(y_i) \right] \right) \tag{1}$$

where $\lambda_{I(u,c)}, \lambda_{T(c,c')}$ are weights associated with each of the feature functions and Z is the *partition function* that normalizes (1) to a conditional probability distribution. We discuss the training of the weights in Section 4.

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 (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.



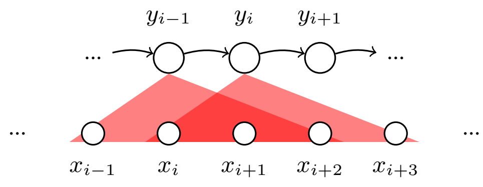


Figure 3. Byte labeling as a linear-chain Conditional Random Field. Each byte position x_i in the binary is paired with a label node y_i indicating its source compiler. The label nodes are associated with *idiom features* that represent the instructions spanning a range of bytes beginning at x_i (the 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.

We define a new binary feature function

$$f_{CF}(y_i, y_j, \mathcal{P}) = \begin{cases} 1 & \text{if } x_i \text{ branches to } x_j \text{ and } y_i \neq y_j \\ 0 & \text{otherwise.} \end{cases}$$

to encode these branch consistency constraints. Adding this feature to the model of (1) yields the final model

$$P(y_{1:n}|x_{1:n}) = \frac{1}{Z} \exp\left(\sum_{i=1}^{n} \left[\sum_{u \in \mathcal{U}} \sum_{c \in \mathcal{C}} \lambda_{I(u,c)} f_{I(u,c)}(x_i, y_i) + \sum_{c,c' \in \mathcal{C}} \lambda_{T(c,c')} f_{T(c,c')}(y_i) + \sum_{j=1}^{n} \lambda_{CF} f_{CF}(y_i, y_j) \right] \right) \tag{2}$$

where the weight λ_{CF} is expected to be strongly negative. 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 (1) and heuristically approximate the branch consistency component of this model as a hard constraint, as we describe in Section 5. This approximation is sensible as we expect the labeling consistency across intraprocedural branches to hold under all but the mosty highly contrived circumstances.

4. Model Training

The central insight of our research into extracting compiler provenance is that while most of the characteristics of binary code depend primarily on the functionality of the program, features that vary across compilers become apparent when sufficiently many binaries are examined. By defining templates that expand into many idiom feature functions and training our model on hundreds of binaries, we obtain parameters that closely capture the differences among several different compilers. The goal of training is to learn the model parameters $\lambda_{I(u,c)}$ such that the linear-chain CRF model (1) assigns high probability to correct compiler provenance labelings. The training process involves establishing a training set, selecting the features for the model, and estimating the model parameters from the training set.

We have collected a corpus 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). The GCC and MSVS data sets were collected from department Linux and Microsoft Windows workstations, respectively; we compiled a variety of open source software packages with the Intel C and C++ compiler to produce the ICC binaries. Debugging symbol information is available for all of the programs in our corpus. These binaries range in size from a few tens of kilobytes to 26MB.

All of the binaries except for the ICC data set are assumed to be generated by a single compiler. The ICC programs sometimes contain statically linked library code produced by the GCC compiler; this is an artifact of their compilation on platforms with GCC-compiled system libraries, and accounts for a tiny fraction of the code in each binary. These mixed compiler binaries have been hand annotated where necessary to reflect the ground truth compiler provenance.

We represent each program binary as a sequence of byte locations described by a set of idioms as follows. We first exhaustively disassemble each program using the Dyninst binary analysis and instrumentation tool [5, 13]. 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 ocurring 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 consitituting executable instructions) or the `data` label for all other bytes.

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 $\mathcal{U}$ and compiler labels $\mathcal{C}$

$$MI(\mathcal{U},\mathcal{C}) = \sum_{u\in U}\sum_{c\in\mathcal{C}} p(u,c)log\left(\frac{p(u,c)}{p(u)p(c)}\right),$$

where $p(u,c)$ is the joint probability of observing an idiom under compiler c and $p(u)$ and $p(c)$ are the marginal probabilities of observing particular idioms and compilers, respectively. These probabilities can be easily estimated from population statistics of the training set. Mutual information measures how dependent two random variables are on one another. In this case, it represents the co-occurrence of an idiom and a particular label: that is, how often code from a compiler displays that idiom. Mutual information is non-negative; it is zero only if two variables are statistically independent, and increases in value the more one value depends on the other. We select the 20,000 features with the highest compiler mutual information for training. See Appendix A for a discussion of features and learned model parameters.

We train the parameters of the linear-chain model (1) with the MALLET package [12]. 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.

5. Evaluation

We evaluate our compiler provenance inference techniques by measuring the accuracy of our models in labeling each byte of the binary. Our test corpus comprises two sets of binaries that exercise our technique in different ways. The first set is composed of the remaining 1,225 single-compiler binaries we held back from the training process. Evaluating our tool's accuracy on this set allows us to test the broad applicability of compiler inference over a large number of real-world binaries. The second data set was artificially constructed to evaluate our inference technique on the more difficult case of binaries containing code from several compilers. These binaries interleave code from the GCC and ICC compilers to simulate programs statically linked against libraries with varying provenance, such as may occur when using commericial or legacy libraries. We discuss the generation of multiple-compiler binaries below. Both testing sets are evaluated using the compiler provenance model trained as described in the previous section.

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 (1). We then heuristically approximate the control flow consistency component of equation (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 λ_{CF} in (2) to $-\inf$. Each inferred label is compared against the ground truth labeling to determine accuracy of our technique.

5.1 Single Source Compiler

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 in the previous section; the idiom feature representation and ground truth labels are also obtained as previously described. Statistics for this data set are listed in Table 1.

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 1. Single compiler data set.

Compiler provenance label accuracy over this data set is 0.925. The accuracies over binaries produced by each compiler are listed in Table 2. In addition to labeling accuracy, we list error rate broken down by the type of error: 'comp-comp' for erroneously labeling the source compiler of a byte, 'comp-data' for bytes labeled `data` incorrectly, and 'data-comp' for data bytes labeled as originating from a compiler. While the test set accuracy is a good metric for evaluating compiler provenance 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 Section 6.

As illustrated by the results in Table 2, 'data-code' mislabeling constitutes the majority of labeling errors, particularly on the

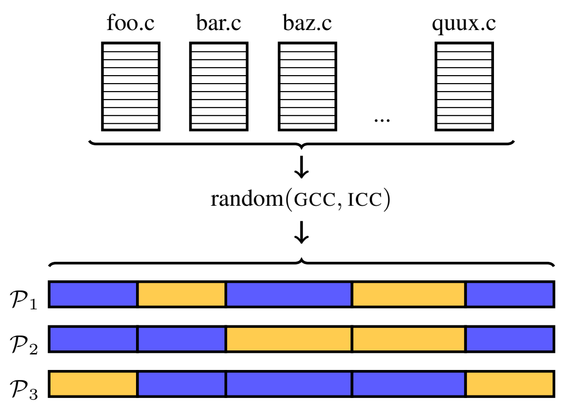


Figure 4. Multiple compiler binary code generation. Each source file is compiled by GCC (🟦) or ICC (🟨) at random and linked to form the program binary $\mathcal{P}_k$. Debugging information in the binary associates functions with a source file and compiler.

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.

5.2 Multiple Source Compilers

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 4. 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.

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

Table 2. Single compiler evaluation. Error rates are computed over the relevant subset of bytes (e.g., 'data-comp' 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.

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

6. Stripped Binary Parsing

Parsing a program binary—extracting the instructions and control flow graph from the underlying bytes—is foundational for any binary analysis. When the locations of functions are known through symbol or debug information this task is trivial. In circumstances where symbols are not available, such as malicious programs, commercial software, and legacy codes, the parsing task is more difficult. The standard approach of *recursive traversal parsing* [16] often fails to completely parse programs due to the prevalence of indirect control flow. We addressed this problem in previous work that used compiler specific *Function Entry Point (FEP)* models to automatically detect functions in stripped binary code [14]. Our approach offered significant improvements over existing state-of-the-art parsing techniques, but relied on knowledge of the source compiler that made it unsuitable for binaries of unknown provenance or those containing mixed-compiler code. We have addressed this limitation by relaxing the known-compiler requirement, augmenting the existing entry point modeling approach to apply to binaries regardless of source compiler. The compiler provenance inference technique described in this paper increases the precision of this more general stripped binary parser.

The FEP identification problem is similar to compiler inference, insofar as we represent the binary as a sequence of bytes and assign labels to each position. Whereas compiler inference models the binary as subsequences of byte offsets with the same label, this task is to find a very sparse set of byte offsets where functions begin. Each position in the binary $x_1 \ldots x_n$ will be assigned a label y_i of function entry (1) or non-entry (-1). Our goal, as with compiler inference, is to learn parameters of a probabilistic graphical model—in this case, one that will assign high likelihood to a correct labeling of function entry points.

Each point in the binary is characterized by idiom features identical to those described in Section 3. The idiom feature functions and associated weights λ used here, however, are parameterized by the source compiler label ℓ_i assigned by compiler provenance inference.

$$f_{I(u,\ell)}(x_i, y_i, \ell_i, \mathcal{P}) = \begin{cases} 1 & \text{if } y_i = 1 \text{ and idiom } u \text{ matches } \mathcal{P} \\ & \text{at offset } x_i \text{ and } \ell_i = \ell \\ 0 & \text{otherwise.} \end{cases}$$

Conceptually, this model uses the compiler labels to determine which subset of idioms to use for inference. We also define two features that capture structural characteristics of the binary. The call feature

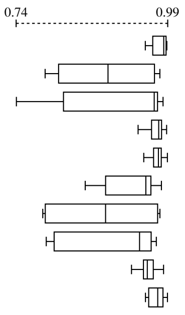
		Label Accuracy		Average error rate		
Program	Source files	Average	Spread	comp-comp	comp-data	data-comp
chcon	38	0.981		0.012	0.006	0.023
cp	74	0.893		0.126	0.007	0.019
date	25	0.907		0.109	0.008	0.046
df	48	0.972		0.021	0.010	0.015
dir	33	0.977		0.015	0.006	0.028
du	52	0.934		0.064	0.006	0.062
mv	64	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

Table 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 plot.

$$f_c(x_i, x_j, y_i, y_j, \mathcal{P}) = \begin{cases} 1 & \text{if } y_i = 1, y_j = -1 \text{ and the function starting at } x_i \text{ calls } x_j \\ 0 & \text{otherwise.} \end{cases}$$

discourages inconsistent assignment of negative labels to callees of a positively labeled function. The conflict feature

$$f_o(x_i, x_j, y_i, y_j, \mathcal{P}) = \begin{cases} 1 & \text{if } y_i = y_j = 1 \text{ and } x_i, x_j \text{ inconsistently overlap} \\ 0 & \text{otherwise.} \end{cases}$$

is intended to prevent inconsistent labelings of functions that overlap: if two candidate FEPs disassemble to overlapping instructions streams over the same bytes, they are mutually exclusive. Neither of these structural features depend on the inferred compiler label. For further details and a formal specification of the model, refer to our previous work [14].

The procedure for incorporating compiler provenance inference into FEP identification is to (a) learn weights $\lambda_{I(u,\ell)}$ 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 in the same manner as described in Section 4. We break from the expensive feature selection technique of our previous FEP identification work and adopt the mutual information approach. This approach allowed us to quickly select many thousands of features for a provenance-free reference model as we discuss below.

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 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.

Adding inferred compiler provenance to the FEP identification task has only modest impact in part because the FEP identification tool is already very powerful: by comparison, the industry standard IDA Pro disassembler [3] has an aggregate F_1 of 0.818 and performs significantly worse on the ICC data set ($F_1 = 0.540$). One open question is whether compiler provenance could be used to simplify the task while maintaining high accuracy. Without provenance labels, we were only able to extend our FEP identification techniques to the multiple-compiler setting by adding tens of thousands of features beyond our original approach, significantly increasing model complexity. More work is needed to determine whether adding inferred provenance labels could allow us to achieve comparable results with far fewer features.

7. Related work

To our knowledge, no other technique has been proposed to extract source compiler or other provenance characteristics from program binaries. Our approach bears some similarity to existing work in code comparison and malware identification that use probabilistic models of programs to attempt to classify malicious software or detect similarities between programs [6, 10]. These approaches represent binaries as n-grams of bytes and attempt to classify the entire binary; by contrast, the technique we use for compiler inference incorporates structural characteristics and disambiguates compiled code and non-code at a fine-grained level within a binary.

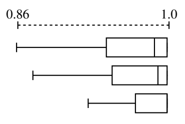
Experiment	FP	F_1	F_1 spread
No compiler labels	6,871	0.956	
Inferred labels	5,585	0.959	
Ground truth labels	2,414	0.969	

Table 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.

Instruction abstractions similar to those in our idiom features have been used to assign a score to candidate instruction sequences in obfuscated binaries [7]. Instructions or pairs of instructions are assigned a score based on the frequency of occurrence in a binary code corpus; the scores of all instructions in a sequence are added to form the sequence score. Our idiom sequence CRF differs in that the contribution of each idiom in the sequence is based on learned parameters that allow discrimination between different labels within a sequence of bytes rather than distinguishing between two alternative sequences of instructions.

One element of program provenance that continues to receive attention is authorship attribution [4, 15]. All existing work has focused on the problem of source code attribution, using course statistics such as comment to code ratio or average line length to differentiate among several authors. Whether distinguishing characteristics of the toolchain used to produce a binary code artifact could yield authorship-revealing information remains an open question.

8. Discussion

We have introduced the task of source compiler inference, and presented a novel technique that extracts the compiler provenance from program binaries using only the binary code and data bytes. Our technique represents the program as a sequence of idiom features, abstractions of instructions that are evaluated at each byte offset in the binary, not at the actual executable instruction locations. Using idiom features and structural characteristics of binary code such as source compiler consistency across intraprocedural control flow, we learn parameters of a conditional random field model of binary code. This model allows us to accurately label the most likely compiler of subsequences of code in binaries. Our tests demonstrate byte labeling accuracy above 90% for a large corpus of real-world binaries. The inference technique applies equally well to binaries containing code from several compilers, yielding results consistent with those of the single-compiler data set.

We applied our compiler inference technique to the task of function entry point identification in stripped binary code, which previously had required prior knowledge of the source compiler. First extending this technique to elide the known-compiler requirement, we then showed that adding inferred provenance labels acheives a modest but significant improvement in parser precision. Our extension allowed the function entry modeling algorithm to associate code features with particular compilers in a single model, taking advantage of compiler-specific characteristics without requiring compiler-specific models. More research is needed to determine whether the information supplied by source compiler inference can be used to decrease the complexity of function entry point identification, for example by reducing the number of features needed.

The extent to which differences in the code produced by different versions of the same compiler impacts the compiler identification task is outside the scope of this study. Given that the code generated by different compiler families exhibits significant variation, it is reasonable to expect that binaries generated by substantially different versions of a compiler (e.g. GCC 3.x vs GCC 4.x or different editions of Microsoft Visual Studio) would differ under the representations we present here. We have completed a preliminary study of code variance in the GCC compiler that supports this assumption; however, the extent to which such version-dependent code characteristics impacts the compiler family identification task is an open question.

The ability to infer compiler provenance from the contents of a program binary enables tool-specific binary analysis like our function entry point identification case study or recognition and understand of compiler-specific idioms like the implementation of switch statements [1]. Source compiler inference also can provide information relevant to forensic analysis and reverse engineering by giving insight into a major component in the binary production toolchain. If we pull back from our focus on the source compiler and instead view the entire binary production process—compiler, compiler optimizations, linker, link-time optimizations, post-link processing—as a black box, it is reasonable to ask whether other program provenance characteristics might be extracted besides the identity of the source compiler. The extent to which this question can be answered is itself an open question, and is the focus of our ongoing work.

A. Idiom Feature Details

The feature selection procedure we used for the experiments described in this paper simply ranked idiom features by the mutual information between the idiom and class variables. The following idioms represent the top in our training data for the single-compiler data set when ranked by mutual information.

Idiom	$MI(\mathcal{U}, \mathcal{C})$
test ebp,esp	0.0004485
test ebp,esp |adc ebx,ebx	0.0003902
* |adc ebx,ebx	0.0003502
adc esi,esi	0.0002985
adc esi,esi |test ebp,esp |adc esp,esp	0.0002741
adc esi,esi |test ebp,esp	0.0002741
adc esi,esi |* |adc esp,esp	0.0002741
* |test ebp,esp |adc esp,esp	0.0002643
* |* |adc esp,esp	0.0002528
* |test ebp,esp	0.0002454

When we train the compiler inference model, each idiom is assigned a weight for each label transition such as `gcc` → `data` or `icc` → `icc`. The following are pairs of label transitions and idiom features with the largest absolute value weights in the single-compiler model.

Idiom	Label transition	Weight
adc	`data` → `data`	3.94
test ebp,esp |* |adc esp	`data` → `data`	3.548
* |* |and	`data` → `data`	2.624
and	`data` → `data`	2.522
test ebp,esp	`data` → `data`	-2.415
* |adc ebx,ebx	`data` → `data`	-2.249
* |* |push	`gcc` → `data`	2.19
* |and	`data` → `data`	2.113
* |* |push mem,eax	`data` → `data`	-1.992
adc edx,mem |sbb	`data` → `gcc`	1.984
adc ecx,mem |* |sbb ecx	`gcc` → `data`	1.972
* |* |sub	`data` → `data`	1.941
* |* |adc esi,esi	`data` → `data`	-1.94
* |* |xor	`data` → `data`	1.882
sbb eax |sbb	`data` → `gcc`	1.841
* |and |sbb	`data` → `data`	-1.786
* |add |adc esi,esi	`data` → `data`	-1.757
sbb |* |push	`gcc` → `data`	1.683
adc esp	`data` → `gcc`	1.673
* |adc	`data` → `data`	-1.669

Acknowledgments

This work is supported in part by Department of Energy grants 93ER25176, 07ER258000 and 09ER25940, and National Science Foundation Cybertrust grants CNS-0627501 and CNS-0716460.

The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

[1] C. Cifuentes and M. V. Emmerik. Recovery of jump table case statements from binary code. In *Seventh International Workshop on Program Comprehension (IWPC '99)*, page 192, Pittsburgh, PA, May 1999. ISBN 0-7695-0179-6.

[2] C. Cifuentes and K. J. Gough. Decompilation of binary programs. *Softw. Pract. Exper.*, 25(7):811–829, 1995. ISSN 0038-0644. doi: http://dx.doi.org/10.1002/spe.4380250706.

[3] Data Rescue. IDA Pro Disassembler: Version 5.5 `http://www.datarescue.com/idab ase`, 2007.

[4] J. H. Hayes and J. Offutt. Recognizing authors: an examination of the consistent programmer hypothesis. *Software Testing, Verification and Reliability*, 2009.

[5] J. K. Hollingsworth, B. P. Miller, and J. Cargille. Dynamic program instrumentation for scalable performance tools. Technical Report CS-TR-1994-1207, University of Wisconsin-Madison, 1994. URL `citeseer.ist.psu.edu/75570.html`.

[6] J. Z. Kolter and M. A. Maloof. Learning to detect and classify malicious executables in the wild. *Journal of Machine Learning Research*, 7:2721–2744, 2006.

[7] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *Thirteenth USENIX Security Symposium*, pages 18–18, San Diego, CA, August 2004.

[8] C. Krügel, E. Kirda, D. Mutz, W. K. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *Eighth International Symposium on Recent Advances in Intrusion Detection (RAID 2005)*, pages 207–226, Seattle, WA, September 2005.

[9] J. Lafferty, A. McCallum, and F. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proc. 18th International Conf. on Machine Learning*, 2001.

[10] W.-J. Li, K. Wang, S. J. Stolfo, and B. Herzog. Fileprints: identifying file types by n-gram analysis. In *Sixth IEEE Information Assurance Workshop (IAW '05)*, pages 64–71, June 2005.

[11] A. McCallum. Efficiently inducing features of conditional random fields. In *Proceedings of the 19th Conference in Uncertainty in Artificial Intelligence*, Acapulco, Mexico, August 2003.

[12] A. K. McCallum. Mallet: A machine learning for language toolkit. http://www.cs.umass.edu/ mccallum/mallet, 2002.

[13] Paradyn Project. Dyninst: An application program interface for runtime code generation. http://www.paradyn.org, 2010.

[14] N. Rosenblum, X. Zhu, B. P. Miller, and K. Hunt. Learning to analyze binary computer code. In *Proceedings of the twenty-third conference on Artificial Intelligence (AAAI-08)*, Chicago, IL, July 2008.

[15] E. H. Spafford and S. A. Weeber. Software forensics: Can we track code to its authors? Technical Report Purdue Technical Report CSD-TR-92-010 / SERC Technical Report SERC-TR-110-P, 1992. URL `citeseer.ist.psu.edu/spafford92software.html`.

[16] H. Theiling. Extracting safe and precise control flow from binaries. In *RTCSA '00*, page 23, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0930-4.