Recovering the Toolchain Provenance of Binary Code

Nathan Rosenblum and Barton P. Miller and Xiaojin Zhu
Computer Sciences Department
University of Wisconsin
Madison, WI 53706
{nater,bart,jerryzhu}@cs.wisc.edu

ABSTRACT

Program binaries are an artifact of a production process that begins with source code and ends with a string of bytes representing executable code. There are many reasons to want to know the specifics of this process for a given binary—for forensic investigation of malware, to diagnose the role of the compiler in crashes or performance problems, or for reverse engineering and decompilation—but binaries are not generally annotated with such *provenance details*. Intuitively, the binary code should exhibit properties specific to the process that produced it, but it is not at all clear how to find such properties and map them to specific elements of that process.

In this paper, we present an automatic technique to recover *toolchain provenance*: those details, such as the source language and the compiler and compilation options, that define the transformation process through which the binary was produced. We approach provenance recovery as a classification problem, discovering characteristics of binary code that are strongly associated with particular toolchain components and developing models that can infer the likely provenance of program binaries. Our experiments show that toolchain provenance can be recovered with high accuracy, approaching 100% accuracy for some components and yielding good results (90%) even when the binaries emitted by different components appear to be very similar.

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 created through a process, a series of stages wherein an idea is instantiated and transformed into machine-interpretable code. Binaries are not merely the end result of this process, but are also a record of it, exhibiting characteristics that are particular to the set of decisions made and tools used in building the program. In most circumstances, these characteristics are interesting only insofar as they influence the program as it is used—how much faster an optimizing compiler makes it run, or the support for concurrency available in a particular language. For particular disciplines, however, the production process of a program—its provenance—is of primary importance. Developers and researchers in the security, software engineering, testing and performance analysis communities can benefit from knowledge of how a program binary was produced; the difficulty lies in obtaining details of this process when only the end result is available.

We have previously shown that the compiler family that produced a particular program can be recovered solely by examining the properties of the program's executable code [24]. This information helped to solve the basic analysis task of finding code in binaries stripped of debugging symbols, and is a major facet of a program's provenance. The compiler family, however, is hardly the limit of interesting provenance details. Identifying finer-grained components of the compiler toolchain, such as the specific version of compiler, the optimization level of the code, and other options can aid analyses that are sensitive to compiler-specific code transformations [21]. Fine-grained compiler details could augment crash reports in deployments where vendors do not control the compilation environment for the software or external dependencies [17], such as open source projects, to help diagnose crashes due to compiler bugs or incompatibilities. 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 [4, 23]. 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 [18].

We have developed novel techniques that extend our earlier compiler work to *toolchain provenance*: the compiler family, versions, optimization options and source languages that characterize the production process for a given binary. We rely only on observable properties of the executable code; by ignoring meta information such as program headers or debugging symbols, our techniques are broadly applicable even when such information is missing, corrupt, or inapplicable such as when applied to incomplete snippets of binary code. We formulate provenance recovery as a machine learning task, constructing models of binary code that reflect its generation through a multiple step production process. These models allow us to infer the process that produced new binaries, revealing their provenance. We explore several methods of representing programs and modeling provenance, showing how trade-offs between model complexity and program representation provide different benefits for provenance inquiries on real-world binaries.

Our paper makes the following contributions:

- We define the problem of *provenance recovery* as two tasks: (1) building representations of program binaries that capture properties that are characteristic of a particular source language and production toolchain, and (2) defining and learning the parameters of models that allow inference of the provenance of previously unseen binaries. We introduce algorithms for both of these tasks, building on techniques for finding patterns in binary code [23] and the machinery of support vector machines [5] and conditional random fields [15], respectively.

- We implement a tool called ORIGIN that extracts significant features from binary programs and classifies the programs according to their toolchain provenance. ORIGIN is designed around several different classifiers and can be applied to whole programs or smaller snippets of binary code. To our knowledge, ours is the first tool for recovering detailed toolchain provenance from program binaries.

- 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.

2. OVERVIEW

The toolchain provenance of a program can be thought of as those components of its provenance that are involved in the transformation 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 that produced a particular binary. The machine code in a binary is not solely determined by the toolchain, of course, but is also highly dependent on the intended functionality of the program, the author's use of particular algorithms and programming practices, and so forth. In order to use code characteristics to infer toolchain provenance, we must choose a program representation that captures those details that are particular to the toolchain. Using this representation, our goal is to *discriminate* between code produced by various toolchain components. This task is essentially a *classification* problem: we need to establish decision criteria that allow us to label an example—some binary code—as belonging to the class of code produced by a particular toolchain component. We approach the selection of these criteria as a machine learning problem, which informs our technique's high-level workflow:

1. We collect a set of programs compiled with the toolchain components of interest, and divide these programs into classes corresponding to individual component varieties. For example, open source software in various programming languages is readily available, and can be compiled with various compilers under diverse compilation settings.

2. Using existing techniques for recursive traversal parsing [29], a control flow graph of each binary is constructed. Both the control flow graph and the underlying instructions are used to form a novel representation, or *features*, of the code, which representation we describe below.

3. A subset of the features describing the binary are selected. The features we use are designed to richly represent the binary code, as we do not know *a priori* what properties will be useful for toolchain component classification. This approach leads to a preponderance of candidate features. While it is the job of the selected machine learning algorithm to choose, in some sense, the "right" features, the cost of learning can often increase with the size of the *feature space.* We therefore perform *feature selection* to choose a subset of features that are likely to be valuable for classification.

4. A *training set* of programs represented by the selected features is used to build a classifier for toolchain components. There are many different discriminative machine learning algorithms that we could apply to this task, each having varying strengths and weaknesses. We consider two examples of broad classes of algorithms: support vector machines [5], which are applicable to classification of independent example data; and conditional random fields [15], which are a type of *probabilistic graphical model* that can capture rich dependencies between examples.

2.1 Binary Code Representation

At the lowest level of abstraction, absent the program header metadata or other formatting, a binary is simply a sequence of bytes, some of which represent machine interpretable instructions. While it is certainly possible to work with binaries at this level [24]—that is, without considering any *structural* properties—we overlay a function abstraction on this linear view, dividing the program into a sequence of functions as depicted in Figure 2. This representation has the advantage that (1) it entails significantly less computation than modeling every byte of code explicitly, and (2) it is a more *consistent* representation from a provenance standpoint: for the compilation toolchains that we consider, functions are the smallest unit of output.[1] Furthermore, by recovering provenance at the function level, our technique is flexible enough to represent binaries of mixed provenance; such binaries frequently arise when programs are statically linked against precompiled libraries.

[1]Inline assembly embedded in source code is an exception, as it can be thought of as a sub-function level output of some other toolchain.

```
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 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 IA-32 instruction set allows compilers great flexibility, even in such small code snippets.

After dividing the binary into functions, we are left with the choice of how to represent those functions in a way that captures properties specific to their toolchain provenance. Because our technique uses only the executable code, these properties are manifest solely in the instructions of the program and their layout. The Intel IA-32 instruction set, our target platform, provides ample opportunity for variations in the toolchain to produce differences in the machine instructions. For example, simple operations like adding a constant to a variable can be encoded by either an explicit pair of store and addition instructions (`mov eax,edi; add eax,5`) or a curious use of the 'load effective address' instruction (`lea eax,[edi+5]`). Figure 1 shows a slightly longer example sequence of code as interpreted by two different compilers; there are several differences in the sequence of instructions for this code snippet alone. The ways in which different compilers vary in their implementation of high-level language constructs arise as systematic differences in instruction choices and the ordering and layout of code. The features that we describe in Section 3 are designed to capture these properties.

2.2 Provenance Modeling

The features with which we represent functions can be thought of as a collection of simple predicates about the binary code. We model a binary's provenance by assuming that there exists some relationship between these predicates and the toolchain that generated the code. For example, if functions generated by the GNU C Compiler were known to *almost always* begin with a `push ebp` instruction and those generated by the Microsoft Visual C compiler *almost never* did so, then a model with a decision function like

$$\textsc{comp}(\mathcal{F}_i) = \begin{cases} \texttt{gcc} & \text{if } \textsc{firstinsn}(\mathcal{F}_i) = \texttt{push ebp} \\ \texttt{msvc} & \text{otherwise} \end{cases}$$

for some function $\mathcal{F}_i$ in a binary would be reasonable. Unfortunately, even the differences between compiler families are not so apparent, so the modeling question becomes how to combine a potentially large number of predicates into a decision function that can discriminate accurately between code generated by different toolchains.

Machine learning techniques provide a mechanism by which to construct appropriate decision functions. The general process for learning such *classifiers* is to define a function with some number of parameters that maps from the feature space onto a class label, and then to search through the parameter space while minimizing the error on a *training set* of data. The classifier trained in this way can then predict labels for data outside of the training set; data reserved for this purpose is usually referred to as the *testing set*. Depending on the structure of the function underlying the classifier, the number of parameters may be quite large if there are many distinct features in the data.

To solve this problem we have developed a procedure that incorporates *feature selection* to reduce the dimensionality of the representational space and a classifier that predicts toolchain provenance based on the selected features. Feature selection consists of choosing those features that will contribute the most to a classifier, for example by systematically training and evaluating models with different sets of features. Such methods can be prohibitive when model training is expensive; our procedure instead uses a simpler approach that ranks features by their significance using a mutual information criterion [8]. We then train a classifier using only the K most significant features; we have developed several algorithms that use the output from different classifiers to reach a final prediction of program provenance.

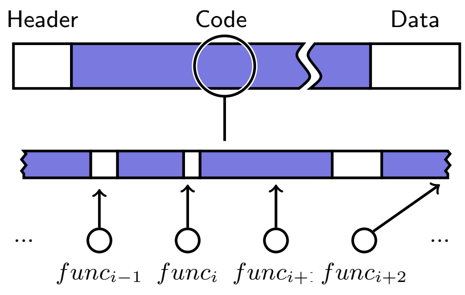


Figure 2: The *function-provenance* abstraction over a typical binary code artifact, such as a Linux ELF binary. Header information is only used to find functions if available. The binary code is represented as a linear sequence but is not necessarily contiguous: there may be gaps containing non-executable data, padding, or random bytes interspersed among machine instructions.

3. EXTRACTING BINARY FEATURES

The first step in feature extraction is to parse the binary to find individual functions. The problem of parsing and

```
function MatchIdioms($\mathcal{F} = (V, E)$, $I$)
    $M \leftarrow \emptyset$
    for all $\iota \in I$ do
        for all $v \in V$ do
            $\{\mu\}_{1:k} \leftarrow$ Decode($v$)
            for $i \leftarrow 1$ to $k - |\iota|$ do
                if $\iota = \{\mu\}_{i:i+|\iota|}$ then
                    $M \leftarrow M \cup \iota$
    return $M$
```

Figure 3: Idiom matching algorithm. Returns a *multiset* M containing idioms $\iota \in I$ that match the code comprised by $\mathcal{F}$. The Decode function disassembles the linear sequence of instructions in a basic block.

finding code in binaries is well studied [4, 23, 29]; we use the ParseAPI [19] library to build interprocedural *control flow graphs* from binaries, where a CFG is a directed graph $G = (V, E, \tau)$ over the *basic blocks* of the binary, and is defined by:

- the set V of vertices corresponding to basic blocks,
- the set $E \subseteq V \times V$ corresponds to control flow edges between blocks, and
- the labeling function $\tau : E \rightarrow \mathcal{T}$ that associates a particular edge in the graph with a type.

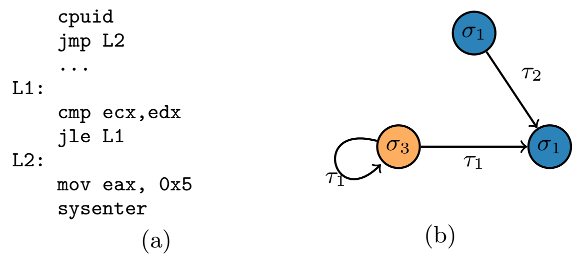
The ParseAPI represents functions as *intraprocedural* subgraphs $\mathcal{F}_i = (V_i, E_i, \tau')$ where $V_i \subseteq V, E_i \subseteq E$, and τ' assigns only intraprocedural edge types (i.e., excluding calls and returns). This representation allows us to extract a rich set of features not only over the basic blocks V of the binary code, but also over control flow relationships between these blocks.

As we discussed in the previous section, specific toolchain components greatly influence the instructions comprised by a program binary. Our early work in capturing binary code properties such as the characteristic patterns of function entry points [23] has demonstrated the utility of *idiom* features: short sequences of instructions with wildcards. For example, the idiom

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

describes a common stack frame set-up operation with a single wildcard between the `push` and `mov` instructions. More formally, an idiom $\iota \in I$ is a function of N machine instructions $\iota : x \times \mu_1 \times \cdots \times \mu_N \rightarrow \{0, 1\}$ where x is an offset within the sequence of bytes representing the binary code; this function takes the value 1 if the N instructions disassembled from offset x match each instruction in the idiom and zero otherwise. With a wildcard that matches any instruction, idioms can be thought of as a relaxed *N-gram* representation; our technique uses idioms of length 1–3. There may be hundreds of thousands of unique idioms in a single binary; below we describe a *feature selection* procedure to automatically choose idioms that reflect differences in code provenance. The algorithm for extracting the idioms in a function is presented in Figure 3.

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 enhance our idiom-based binary representation with additional features



```
    cpuid
    jmp L2
    ...
L1:
    cmp ecx,edx
    jle L1
L2:
    mov eax, 0x5
    sysenter
```

Figure 4: A code example and a corresponding graphlet. The vertex colors and edge labels are determined by the particular graphlet feature mapping functions (for example, both of the blocks represented by (●) vertices contain system instructions).

based on *graphlets* [20]: small, non-isomorphic subgraphs of the CFG. These subgraphs $G_s = (V_s, E_s, \tau', \sigma)$ are extensions of the CFG that include a labeling function $\sigma : V \rightarrow \Sigma$ that assigns a *color* to vertices (basic blocks). The set of block colors Σ varies for different graphlet-based features, depending on the properties that we are trying to capture; the edge labeling function τ' may also map to a different set of edge types $\mathcal{T}$. Our features are based on graphlets with three vertices, as depicted in Figure 4. We include two different types of graphlet features in our binary code representation: one that captures the layout of particular classes of instructions and one that focuses on the particular instructions used to implement control transfer.

Instruction *summary graphlets* are inspired by a binary code representation used in polymorphic worm detection [13], where basic blocks were colored according to fourteen *instruction classes* such as string operations, branches, logic operations, etc. Following this scheme, the color of a vertex in summary graphlets is a fourteen-bit number encoding whether instructions of each class are present in a block. More formally, summary graphlets supply a labeling function $\sigma : V \rightarrow [0, 2^{14}{-}1]$; our experience has been that basic blocks rarely include instructions from more than a few classes, so the total number of colors represented in a set of programs is small. This representation captures differences in the arrangement of code without being sensitive to the particular instructions used, thus avoiding redundancy with the idiom features.

Our experience analyzing code emitted by different compilers suggests that the branch instructions used to direct control flow are highly indicative of 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. In the interest of explicitly capturing this phenomenon, we define *branch graphlets* similar to summary graphlets, but with a color labeling function $\sigma : V \rightarrow \mathcal{B}$, where $\mathcal{B}$ is the set of unique branching instructions in the IA32 instruction set.

We test for graphlets in a CFG by computing a *canonical labeling* that is identical for the isomorphisms of any subgraph of size three under the particular graphlet feature coloring function σ. Producing a canonical labeling is equivalent to the graph isomorphism problem, for which no polynomial time algorithm is known. However, canoni-

```
function MATCHGRAPHLETS($\mathcal{F} = (V, E)$,$\tau$,$\sigma$,$C$)
    $M \leftarrow \emptyset$
    for all $v \in V$ do
        for all $\{n_a, n_b\} \in$ NEIGHBORS($v$) do
            $V_s \leftarrow \{v, n_a, n_b\}$
            $E_s \leftarrow V_s \times V_s \subseteq E$
            $c \leftarrow$ CANONICAL($V_s, E_s, \tau, \sigma$)
            if $c \in C$ then
                $M \leftarrow c$
    return $M$

function CANONICAL($G$,$E$,$\tau$,$\sigma$)
    $c \leftarrow$ SORT($\sigma(G)$)
    for $d \leftarrow 1$ to $\max \deg(v \in G)$ do
        $W_d \leftarrow \{v | \deg(v) = d\}$
        for all $v \in \Pi(W_d)$ do                ▷ See caption
            $c \leftarrow c \,||\, \tau((*, v) \cup (*, v) \in E)$
    return $c$
```

Figure 5: 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 canonical ordering of vertices is over vertex color τ, vertex degree, and lexicographic ordering of in/out edge colors (Π). The 3-graphlets we use require testing at most six permutations to find the canonical ordering, but vertex degree ordering frequently reduces that number.

cal 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; the canonical labeling is the minimum labeling under a lexicographic ordering. In general this requires examining $K!$ permutations for a K-vertex graph; in practice we can reduce this search space by partitioning the set of nodes based on properties that are invariant to isomorphism (such as vertex degree) [14]. The general graphlet matching algorithm is presented in Figure 5.

We also extract the high-level layout of functions in the binary. The binary parser assigns an offset to each function indicating at which byte in the address space it begins. While we have observed previously that binary functions are not necessarily contiguous (and may in fact be interleaved or share some code), it is commonly the case that functions occupy disjoint regions of the address space; the offset provided by the ParseAPI library allows us to order functions by their location in the binary. While this ordering is not a code feature in the same sense as the idiom or graphlet-based features, we make use of it in the models we describe in the following section.

4. MODEL DEFINITIONS

We model the characteristics of binary code in order to build classifiers that can discriminate between code with different toolchain provenance. The features we described in the previous section form the evidence that we use both to train classifiers and to infer provenance. To be precise, let a program binary $\mathcal{P}$ be a sequence of functions $\{\mathcal{F}\}_k$ ordered by their entry addresses $a_1 \cdots a_k$. The task of a 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 $y_1 \cdots y_k$).

Each function is represented by a binary *feature vector* that indicates whether a particular feature is present. We define a set of *feature functions* $f \in \Phi$ that map from the various binary code features from the previous section to binary values. For each idiom $\iota \in I$ we defined a function

$$f_\iota(\mathcal{F}_i) = \begin{cases} 1 & \text{if } \iota \in \text{MATCHIDIOMS}(\mathcal{F}_i, I) \\ 0 & \text{otherwise} \end{cases}$$

that tests for that idiom in a particular function, or for the number of occurrences depending on the model. Graphlet feature functions are defined over the set of summary graphlets $b \in G_S$ and branch graphlets $b \in G_B$, respectively:

$$f_s(\mathcal{F}_i) = \begin{cases} 1 & \text{if } b \in \text{MATCHGRAPHLETS}(\mathcal{F}_i, G_S) \\ 0 & \text{otherwise} \end{cases}$$

$$f_b(\mathcal{F}_i) = \begin{cases} 1 & \text{if } b \in \text{MATCHGRAPHLETS}(\mathcal{F}_i, G_B) \\ 0 & \text{otherwise} \end{cases}$$

If we admitted all possible idiom and graphlet feature functions, the feature vectors describing each function would grow unmanageably large,[2] making training difficult. We reduce the number of features used to build models by selecting those features that are most significant in the training set. We consider one feature to be more significant than another if the *mutual information* between the feature and the class label of an example is greater than that of another feature. More precisely, we compute

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

on the training set, where $p(f)$ and $p(y)$ are the empirically observed probabilities of features and class labels, respectively, and $p(f, y)$ is the probability of co-occurrence of these variables. Mutual information is closely related to Shannon entropy, and measures how much uncertainty about the value of one random variable is reduced by knowing the value of another. In this setting, mutual information can be thought of as measuring both positive and negative correlation of particular features and class labels. For example, if a particular idiom feature frequently occurs in programs compiled from C++ code but never in Fortran and only rarely in C, then it will be ranked high under this criterion. On the other hand, if an idiom is observed uniformly regardless of a binary's provenance, that idiom tells us little about the provenance label and will receive a low score. When training our models, we use only the top K features ranked by the mutual information score.

4.1 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 and

[2]There are approximately 1.4 million unique features in our typical training sets.

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*, which 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 $\boldsymbol{w}$ 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 $\boldsymbol{w}^T\boldsymbol{x}$, where $\boldsymbol{x}$ is the feature vector. 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 $\boldsymbol{w}_1 \cdots \boldsymbol{w}_K$ 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 $k \in [1, K]$ that maximizes the class-specific margin

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

We use the LIBLINEAR linear support vector machine library [6] 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.

4.2 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 nonetheless a powerful tool for improving provenance models.

Adding relationships between individual functions leads naturally to the formulation of our problem as a *probabilistic graphical model*: *probabilistic* in the sense that we frame provenance recovery as a probabilistic inference problem, and *graphical* because the adjacency relationship induces a dependence graph between the functions. Formally, for program $\mathcal{P}$ with functions $\{\mathcal{F}\}_k$ and evidence $X = \{\boldsymbol{x}\}_k$, we define an unnormalized probability distribution

$$P(Y|X) \propto \exp\left(\sum_{i=1}^{k} \left[\sum_{u \in \mathcal{U}} \lambda_{y_i,u} \cdot f_u(x_i) + \sum_{j=1}^{k} \sum_{b \in \mathcal{B}} \lambda_{y_i,y_j,b} \cdot f_b(x_i, x_j) \right] \right), \tag{1}$$

where $\mathcal{U}$ are unary feature functions taking a single example as input and $\mathcal{B}$ are binary feature functions that relate two examples. The λ terms are feature weights associated with particular class labels $y \in \mathcal{Y}$ or pairs of labels, and are the parameters of the model.

We can construct a model of this form using the idiom and graphlet features we have previously defined as unary features and introducing an adjacency feature function

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

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 modify the idiom feature function

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

to test for adjacency; the graphlet features are modified similarly. Importantly, the actual test for idiom existence still only considers one example. 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 under other circumstances.

4.3 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 model defined by Equation 1 is a linear-chain conditional random field [15], so called because the nodes in this graphical model are connected in a linear chain, as depicted in Figure 6.

Linear-chain CRFs are useful because an algorithm for exact inference in such models is known. However, fixing labels to a particular combination of provenance 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

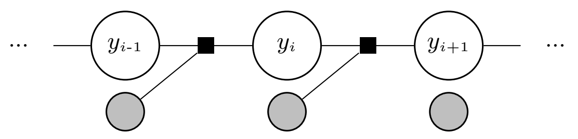


Figure 6: A linear chain CRF over adjacent functions in a program binary. Each function $\mathcal{F}_i$ has a label y_i and a set of evidence (●). Label nodes are joined in a linear chain by feature functions that also depend on the evidence.

a collection of linear sequences where cotemporal label nodes are fully connected, as depicted in Figure 7. This general CRF corresponds to the model

$$P(Y|X) \propto \exp\left(\sum_{i=1}^{k}\sum_{j=1}^{k}\sum_{b\in\mathcal{B}}\left[\lambda_{s_i,s_j,b}\cdot f_b(x_i,x_j)\right.\right.$$
$$\left.\left.+\lambda_{c_i,c_j,b}\cdot f_b(x_i,x_j)+\lambda_{v_i,v_j,b}\cdot f_b(x_i,x_j)+\lambda_{o_i,o_j,b}\cdot f_b(x_i,x_j)\right]\right) \tag{2}$$

that updates Equation 1 by dropping the unary terms and replacing the $\mathcal{Y}$ label terms with a set of terms over $\mathcal{L} = \{s, c, v, o\}$, the individual provenance components. The parameters λ are defined for each label component, e.g. $\lambda_{c_i,c_j,b}$ indexes the parameter for a particular feature function f_b when its inputs have compiler labels c_i and c_j. 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.

We have implemented the joint classification models described in this section using the linear-chain CRF implementation from the MALLET package [16] and the GRMM software for inference in general-structure conditional random fields [28]. We evaluate these and the independent function classifier in the following section.

5. EVALUATION

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

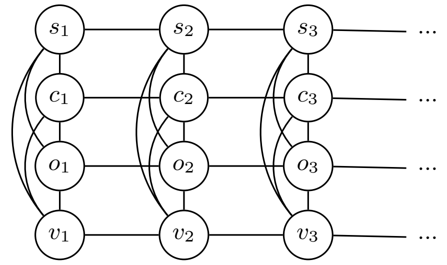


Figure 7: A conditional random field with a grid structure. Data nodes are omitted for clarity.

Compiler Family $\mathcal{C}$	Version $\mathcal{V}$	Optimization Level $\mathcal{O}$ 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

Table 1: Variations of compiler toolchains used in this study. Arbitrary compiler revisions (e.g. 4.4.2) were selected. The MSVC compilers are from unpatched Visual Studio installations. he 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.

- The binary code features extracted by ORIGIN effectively capture the characteristics of program provenance. 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%.
- The trade-offs between model complexity and classifier performance make different model formulations appropriate depending on the requirements of the provenance recovery task. The SVM-based classifier is faster for training and classification, at some cost to accuracy, particularly for identifying the compiler version. This model may nevertheless be the best choice when other toolchain components are of primary interest.
- ORIGIN can automatically model and recover provenance with only modest computational cost. Training with a data set of about 200,000 binary functions takes between 10 and 90 minutes on average, depending on the model. We stress that model training is a one-time cost for a given training set. Labeling binaries is much less expensive, taking on average 100ms for binaries in our corpus.

5.1 Evaluation Data Set

We collected source code for 175 programs written in the C, C++, and Fortan programming languages. 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 1 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. 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.

5.2 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 use *cross-validation* 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.[3] We generate 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 are repeated independently over each of the folds.

To select a subset of significant features, Origin uses 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 select the top 20,000 to reduce the size of the feature space.

For each function in each binary, we use 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). Origin also records the ground truth label tuple for each function. 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 4.3, 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 are aggregated for the SVM classifier based on LIBLINEAR; the sequences of functions in each training binary are 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 is 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 evaluation. We use 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 focuses not only on classification accuracy, but also on the types of errors encountered.

[3]This differs slightly from standard cross-validation, where experiments are repeated over a random *partition* of the data. We use random subses of data to reduce training time for one of the models described below.

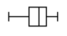
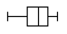
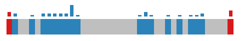
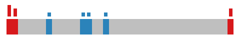
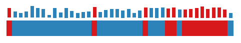
Component	Labels	Accuracy	Spread
Compiler family	3	.987	
Optimization	2	.971	
Compiler version	9	.616	
All components	18	.604	

Table 2: Classification accuracy for individual functions. The compiler version component of provenance is difficult to capture with this independent function model.

5.3 Independent Classification Results

We trained several provenance models over independent functions as described in Section 4.1 using Origin's SVM-based classifier mode. Table 2 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 2 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	

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.

5.4 Joint Classification Results

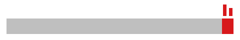
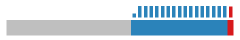
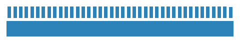
We incorporated intra-binary function adjacency into the models based on both the linear chain and general conditional random fields that are presented in Section 4.3. For the linear chain models, we evaluated inference of individual provenance components (source language, compiler family, version, and 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 2 with fully connected cotemporal label nodes. The linear chain models are learned using MALLET's exact inference mode; we use approximate Tree Reparameterization [30] for inference during learning and classification for the grid models. Table 3 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 examples from 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.

5.5 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 Origin's SVM mode 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 4. 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 evaluation suggests that the source language of a program can be accurately inferred with our provenance recovery technique.

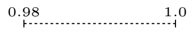
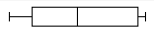
Component	Accuracy	Spread
		
Language	.999	
Joint	.987	

Table 4: Classification accuracy on a corpus incorporating source language labels.

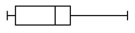
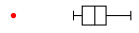
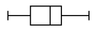
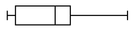
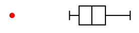
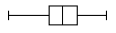
	Linear CRF		Linear CRF (concat.)		General CRF	
Component	Accuracy	Spread	Accuracy	Spread	Accuracy	Spread
		0.87 – 1.0		0.81 – 1.0		0.73 – 1.0
Compiler	.999		.998		.992	
Optimization	.999		.993		.982	
Version	.919		.910		.845	
Joint	.918		.905		.831	

Table 3: 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.

6. DISCUSSION

Our evaluation shows that, depending on the toolchain component of interest, several models may be able to accurately recover provenance. The runtime cost—for training or for classification—will be the deciding factor for many applications. The SVM-based classifier is the least expensive to train, requiring approximately ten minutes on average to estimate parameters from our training set; this classifier, however, does not perform as well for the version provenance component. The linear chain CRF models require more training time (approximately 80 minutes); training is in general an infrequent operation, however, so this additional expense may be of little impact. Both types of classifier infer provenance of binaries comprising hundreds of functions in on the order of 100ms. By contrast, the grid-structured CRF is substantially more expensive for training and inference, due to the difficulty of even approximate inference in such loopy graphical models. Training from our corpus took almost 24 hours for some of the models; inference for labeling a binary takes almost ten seconds on average. We expect training to be an infrequent task; the higher cost of classification may be worthwhile for some applications due to easier interpretation of labels.

The conclusions we draw from our evaluation are subject to limitations inherent in interpreting empirical studies. Threats to internal validity apply to our claim that ORIGIN recovers the toolchain provenance components that we report—the alternative being that our models actually capture some other properties of the code, such as the program functionality. We addressed this issue by generating, for each program, binaries constructed using all applicable compiler families, versions, and optimization levels. Using such a parallel corpus reduces the potential for program functionality to confound the results.

Our results are derived from a specific corpus, raising the question of whether our technique generalizes to other binaries. We address this concern by training and evaluating over random folds of the corpus. We find consistent results on each fold, which supports the generality of the technique. The feature selection and training procedure, moreover, is inherently specialized to particular provenance components in its training corpus; if new components arise, models that incorporate them can be easily constructed.

7. RELATED WORK

Most of the existing program provenance literature relates to authorship attribution, focusing on extracting author-specific characteristics from source code. Spafford and Weeber [27] introduce the notion of extending authorship attribution to the binary code level, but did not further develop the idea. Later research has focused exclusively on recovering authorship characteristics from source code [7, 9, 12]. These approaches make use of carefully crafted stylistic features such as indentation style or variable naming and have had mixed success [9]. The provenance recovery techniques we present in this paper are most closely related to the compiler family inference techniques we have previously developed [24]. The present paper extends both the scope of that work (by recovering fine-grained characteristics of toolchain provenance) and the methods (the binary code features and models for learning and inference).

The problem of *code clone detection* in program binaries—finding instruction sequences that are repeated exactly or inexactly in programs—bears many similarities to provenance recovery, particularly choosing feature representations of binary code and computing the similarity of code sequences based on those features. Saebjornsen et al. [25] developed a clone detection technique that performs inexact matching of binary code using an instruction-based representation similar to our idiom features. While the mechanics of clone detection and provenance recovery are similar, the goals are orthogonal: clone detection seeks to find code instances with similar functionality and as the authors note is hampered by compiler-introduce variations; provenance recovery tools must ignore patterns due to program functionality and focus on the compiler or other toolchain components.

Many different ways of representing binary code have been proposed in the context of malware classification and clustering, including byte-level patterns [11], subsequences of instructions [10], abstract instruction semantics [3], high-level behavioral semantics of the program [1, 22], or structural characteristics of the binary code or executable metadata [2, 26]. Our idiom features, which group machine instructions by mnemonic and abstract away immediate operand details, combine aspects of the *templates* described by Christodorescu et al. [3] and *n-perms* used by Karim et al. [10]. Idiom features are distinguished by incorporating wildcards that allow flexible matching of instruction sequences. Our summary graphlets are inspired by graph-based features used to detect polymorphic malware variants [13].

8. CONCLUSION

We have presented a technique for accurately and automatically recovering the toolchain provenance of program binaries. Our provenance recovery techniques achieve on average 90% accuracy when jointly inferring the source language, compiler family and version, and optimization level options used to produce a binary. Framing provenance recovery as a classification problem, we designed instruction- and control flow-based representations for binary code that capture significant characteristics of toolchain components. We developed provenance models based on support vector machines and conditional random fields and showed how parameters for these models can be learned with only modest computational cost. Our prototype, ORIGIN, automatically extracts features from program binaries for learning and classification. The results of the evaluation of ORIGIN strongly support our claim that toolchain provenance can be recovered solely from the characteristics of the executable code in program binaries. Our approach of designing generic representations—idioms and graphlets—and combining large numbers of features with probabilistic modeling provides a general framework for further investigation of information retrieval from program binaries, with applications in security and forensics, testing, and debugging.

9. ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their insightful comments and suggestions. This research is funded in part by Department of Homeland Security grant FA8750-10-2-0030 (funded through AFRL), National Science Foundation grants CNS-0716460 OCI-1032341, and Department of Energy grants DE-SC0004061 and DE-SC0002154. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

[1] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Krügel, and E. Kirda. Scalable, behavior-based malware clustering. In *Network and Distrtributed System Security Symposium (NDSS)*, San Diego, CA, February 2009.

[2] D. Bruschi, L. Martignoni, and M. Monga. Detecting self-mutating malware using control-flow graph matching. In *Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, Berlin, Germany, July 2006.

[3] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *Security and Privacy (S&P)*, Oakland, CA, May 2005.

[4] C. Cifuentes and K. J. Gough. Decompilation of binary programs. *Software–Practice and Experience*, 25(7), 1995.

[5] C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20, 1995.

[6] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, 9, 2008.

[7] A. Gray, P. Sallis, and S. MacDonell. Software forensics: Extending authorship analysis techniques to computer programs. In *International Association of Forensic Linguists*, Durham, NC, September 1997.

[8] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer, 2001.

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

[10] M. Karim, A. Walenstein, A. Lakhotia, and L. Parida. Malware phylogeny generation using permutations of code. *Journal in Computer Virology*, 1(1), November 2005.

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

[12] I. Krsul and E. H. Spafford. Authorship analysis: identifying the author of a program. *Computers & Security*, 16(3), 1997.

[13] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic Worm Detection Using Structural Information of Executables. In *Recent Advances in Intrusion Detection (RAID)*, Seattle, WA, September 2005.

[14] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *International Conference on Data Mining (ICDM)*, San Jose, CA, November 2001.

[15] A. McCallum. Efficiently inducing features of conditional random fields. In *Uncertainty in Artificial Intelligence*, Acapulco, Mexico, August 2003.

[16] A. K. McCallum. MALLET: A machine learning for language toolkit. 2002. URL `http://www.cs.umass.edu/ mccallum/mallet`.

[17] A. Orso. Monitoring, analysis, and testing of deployed software. In *Future of software engineering research*, FoSER, Santa Fe, NM, 2010.

[18] G. Palmer. A road map for digital forensic research. Technical Report DTR-T001-01 FINAL, Digital Forensics Research Workshop (DFRWS), 2001.

[19] Paradyn Project. ParseAPI: An application program interface for binary parsing. 2011. URL `http://paradyn.org/html/parse0.9-features.html`.

[20] N. Pržulj, D. Corneil, and I. Jurisca. Modeling interactome: scale-free or geometric? *Bioinformatics*, 20(10), 2004.

[21] D. Quinlan and T. Panas. Source code and binary analysis of software defects. In *Cyber Security and Information Intelligence Research (CSIIRW)*, Oak Ridge, TN, April 2009.

[22] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov. Learning and classification of malware behavior. In *Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, Paris, France, 2008.

[23] N. E. Rosenblum, X. Zhu, B. P. Miller, and K. Hunt. Learning to analyze binary computer code. In *Artificial Intelligence (AAAI)*, Chicago, IL, July 2008.

[24] N. E. Rosenblum, B. P. Miller, and X. Zhu. Extracting compiler provenance from program binaries. In *Program analysis for software tools and engineering (PASTE)*, Toronto, Ontario, Canada, June 2010.

[25] A. Saebjornsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su. Detecting code clones in binary executables. In *International Symposium on Software Testing and Analysis (ISSTA)*, Chicago, IL, July 2009.

[26] M. Z. Shafiq, S. M. Tabish, F. Mirza, and M. Farooq. Peminer: Mining structural information to detect malicious executables in realtime. In *Recent Advances in Intrusion Detection (RAID)*, Saint-Malo, France, September 2009.

[27] E. H. Spafford and S. A. Weeber. Software forensics: Can we track code to its authors? Technical Report CSD-TR-92-010, Purdue University, February 1992.

[28] C. Sutton. GRMM: GRaphical Models in Mallet. 2006. URL `http://mallet.cs.umass.edu/grmm/`.

[29] H. Theiling. Extracting safe and precise control flow from binaries. In *Real-Time Computing Systems and Applications (RTCSA)*, Washington, DC, December 2000.

[30] M. J. Wainright, T. Jaakkola, and A. S. Willsky. Tree-base reparameterization for approximate inference in loopy graphs. In *Advances in Neural Information Processing Systems (NIPS)*, December 2001.