Learning to Analyze Binary Computer Code

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

Karen Hunt
National Security Agency. huntkc@gmail.com

Abstract

We present a novel application of structured classification: identifying function entry points (FEPs, the starting byte of each function) in program binaries. Such identification is the crucial first step in analyzing many malicious, commercial and legacy software, which lack full symbol information that specifies FEPs. Existing pattern-matching FEP detection techniques are insufficient due to variable instruction sequences introduced by compiler and link-time optimizations. We formulate the FEP identification problem as structured classification using Conditional Random Fields. Our Conditional Random Fields incorporate both idiom features to represent the sequence of instructions surrounding FEPs, and control flow structure features to represent the interaction among FEPs. These features allow us to jointly label all FEPs in the binary. We perform feature selection and present an approximate inference method for massive program binaries. We evaluate our models on a large set of real-world test binaries, showing that our models dramatically outperform two existing, standard disassemblers.

Introduction

Binary code analysis is a foundational technique in the areas of computer security, performance modeling, and program instrumentation that enables malicious code detection, formal verification, and identification of performance bottlenecks. A binary program contains both code and non-code bytes. The very first step in binary code analysis is to precisely locate all the *function entry points (FEPs)*, which in turn lead to all code bytes. When full symbol or debug information is available this is a trivial step, because the FEPs are explicitly listed. However, malicious programs, commercial software, operating system distributions, and legacy codes all commonly lack symbol information. For these *stripped binaries* a standard technique is recursive disassembly parsing, which follows program control flow (branches and calls) and finds all functions reachable from the main program entry point. However, this technique cannot statically resolve indirect (pointer based) control flow transfers. Indirect control flow is surprisingly common: on a large set of binaries on our department server, approximately 40% of functions were unrecoverable through recursive disassembly. These functions lie in *gaps* between statically discovered functions. To complicate matter, these gaps also contain jump tables, numeric and string constants, padding bytes (both fixed-value and random) and so on. *Identifying FEPs within gaps in stripped binaries is of ultimate importance to binary code analysis, but is an unsolved problem.*

In this paper, we present a novel application of machine learning to large-scale FEP identification. Our main contributions are formulation of FEP identification as a structured classification problem, and an implementation that performs significantly better than existing tools at recovering code from stripped binaries. The scale of our experiments, both in terms of the number of binaries in our data sets, and the size of the binaries (which determines the cost of training and inference), far exceeds that of previous work.

Existing techniques have used recursive disassembly parsing and heuristics to locate code in stripped binaries (Theiling 2000; Cifuentes & Emmerik 2000). Several tools, including Dyninst (Hollingsworth, Miller, & Cargille 1994) and IdaPro (Data Rescue 2007), use a small number of instruction patterns to identify FEPs in gaps. These patterns are created manually for particular compilers. For example, Dyninst searches binaries on the Intel IA-32 platform for a common function preamble pattern that sets up a stack frame: `(push ebp | mov esp,ebp)`. Other efforts have incorporated simple unigram and bigram instruction models to augment pattern heuristics for locating code in stripped binaries (Kruegel *et al.* 2004). These heuristics and simple statistical methods cannot adapt to variations in compiler, optimization level, and post-compilation optimization tools, all of which significantly perturb or even optimize away expected instruction sequences at FEPs.

We overcome these challenges by incorporating both content and structure features in identifying FEPs. Our content features consists of *idioms*, instruction patterns around FEPs. Importantly, we perform automatic feature selection from large corpora of binaries, which gives us flexibility and coverage not possible in hand-coded patterns. Our structure features follow from two properties of binary code:

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

be FEPs;

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

By using both content and structure features, we are able to identify FEPs in gaps with significantly higher accuracy than existing techniques.

The remainder of this paper is structured as follows. We first describe characteristics of binary code that influence the way we design and implement our classifier. We present a "flat" model with content features, where FEP classification is done individually, and describe feature selection. We then add structure features to form a "structured" model, where FEP classification is done jointly. We describe an efficient approximation to inference in the resulting large graphical model. We evaluate our models on several large real-world data sets of Linux and Windows binaries, and compare them against existing tools.

Characteristics of Binary Code

When analyzing the gap regions of a binary, it is not generally known how many functions exist, or whether all gap contents are actually code. To find FEPs in a gap, it is necessary to treat every byte offset in the gap as a candidate FEP. This technique is known as *exhaustive disassembly* (Kruegel *et al.* 2004). Characteristics of the binary code—determined largely by the instruction set architecture—influence how we approach this task, what kind of features we consider, and how we design our learning system.

In this work, we consider binaries compiled on the Intel IA-32 architecture (Int 2006). The IA-32 has a variable length instruction set with an opcode space that is quite dense: almost any value is a valid opcode or the start of a valid multiple-byte opcode. Consequently, every byte offset in the gap might be the start of an instruction, and it is likely that disassembly from that point will produce a valid instruction sequence of some length. Furthermore, and somewhat non-intuitively, IA-32 code commonly exhibits *self-repairing disassembly*: the tendency for instruction streams disassembled by starting at different offsets in the binary to *align* or sync up with one another. Figure 1 depicts this phenomenon for parses from three offsets near the start of a function. While others have observed informally that disassembled instruction streams offset by a few bytes tend to align quite quickly (Linn & Debray 2003), we provide the first formal analysis of this phenomenon in the Appendix.

The consequences of self-repairing disassembly for FEP identification are twofold: i) Because the parse from an address that is not the boundary of an actual instruction quickly aligns with the actual instruction stream, it is unlikely that an incorrect FEP candidate will produce an illegal instruction or other obvious clues; ii) The rapid alignment limits the utility of classifiers based on n-gram models of instruction streams (Kruegel *et al.* 2004). Several candidate FEPs offset by a few bytes will likely have similar likelihood under an n-gram model, making it difficult to differentiate among them to identify the actual FEP.

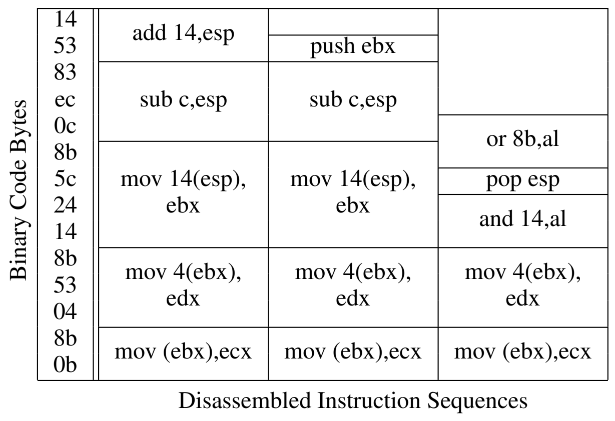


Figure 1: *Self-repairing disassembly. Each instruction sequence (column) is produced by parsing from a particular offset within the bytes depicted on the left. Note that two of the sequences align within one instruction, and all three align within three instructions.*

Flat FEP Models

We begin by formulating FEP identification as a classification problem. Let $\mathcal{P}$ be the program binary code. Let $x_1 \ldots x_n$ represent all the byte offsets within $\mathcal{P}$'s gaps. For each offset x_i, we can generate the disassembly starting at that byte. Our task is to predict whether each x_i is the entry point of a function or not. We use $y_1 \ldots y_n$ to denote the labels: $y_i = 1$ if x_i is an FEP, and $y_i = -1$ otherwise.

Our first model is a "flat" logistic regression model (Hastie, Tibshirani, & Friedman 2001) which predicts each y_i individually. To be consistent with later models, we borrow the notation of Conditional Random Fields and define the label probability as

$$P(y_i|x_i,\mathcal{P}) = \frac{1}{Z}\exp\left(\sum_{u\in\{I\}}\lambda_u f_u(x_i,y_i,\mathcal{P})\right), \qquad (1)$$

where Z is the partition function, $\{I\}$ is the set of idioms (defined below), λ_u is a parameter to be learned, and f_u is a binary feature function on idiom u. We define

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

An idiom u is a short instruction sequence template, which may include the wildcard "*" that matches any one instruction. In this paper, we distinguish two types of idioms: *prefix idioms* represent the instructions immediately preceding the offset, while *entry idioms* begin at the offset. For instance, the entry idiom

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

would match a candidate FEP starting with the sequence (`push ebp | push edi | mov esp,ebp`). Similarly, the prefix idiom

$$u_2 = (\texttt{PRE: ret | int3})$$

would match an FEP immediately preceded by `(ret | int3)`. For the purposes of this paper, instructions are represented by their opcode and any register arguments (i.e., semantically equivalent instructions with different opcodes are distinct). Literals and memory offsets are abstracted away.

Idiom Feature Selection

In our current implementation, an idiom may include up to three instructions, not counting wildcards. Although this reduces the number of idioms, there are still tens of thousands of possible idioms when building a logistic regression FEP model (1). We perform forward feature selection on the candidate idiom features. Our training data consist of several large corpora of binaries that we describe in the Experimental Results section below. Because our application domain is much more sensitive to false positives than false negatives, we use the $F_{0.5}$-measure: $F_{0.5} = 1.5PR/(0.5P + R)$, the weighted harmonic mean of precision (P) and recall (R) during feature selection to emphasize precision. We reserve 20% of the data as a tuning set to decide when to stop adding features. Feature selection and parameter learning are performed separately for each of the three compilers used in the experiments, as the models are expected to vary significantly depending on the source compiler. At each iteration of the feature selection process, we select the idiom that maximizes the F-measure over the training set, recording also performance on the reserved tuning set. We terminate feature selection when adding additional idioms to the model causes decrease or only negligible increase in performance on the tuning set.

We compare feature selection over entry idiom features alone (since they represent what is *in* the functions) vs. over both entry and prefix idioms. Figure 2 shows the performance on the tuning set as the number of selected features increases. These curves are truncated to 35 iterations; the full feature selection run for ICC binaries with prefix features enabled plateaus at 41 features. The ICC dataset represents the hardest compiler in our experiments; the behavior of feature selection on other compilers is similar. The performance increase due to including prefix idiom features is quite significant. In all experiments that follow, all references to the idiom-based model include both entry and prefix idioms.

Structured FEP Models

While the idiom features capture the properties of individual FEPs, they ignore the fact that the functions call one another. While none of the gap functions are reached through statically resolvable calls (otherwise they would not be in gaps by definition), some make statically resolvable calls to other gap functions. If a candidate FEP is targeted by a call instruction (i.e., it is the *callee*), this can be taken as an additional piece of evidence that it is actually an FEP.

As we have mentioned previously, there is another type of interaction specific to exhaustive disassembly: candidate FEPs may *inconsistently overlap*. That is, their dissemblies share the same bytes but produce different instructions. For example, the three columns in Figure 1 inconsistently overlap. For these reasons, we introduce the following *structural* features:

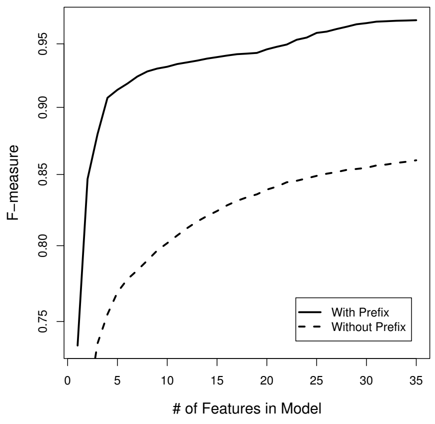


Figure 2: *Feature selection with and without prefix idiom features on the ICC data set. $F_{0.5}$ measure performance on the tuning set is markedly better when prefix idioms are considered in addition to entry idioms.*

1. Binary call-consistency feature f_c. If we assume a candidate FEP at offset x_i has the label $y_i = 1$ (that is, assume it is an actual FEP), and the function starting at x_i has a `call` instruction to x_j, then it does not make sense to let $y_j = -1$. It should be noted that the other three label combinations are fine; in particular, $y_i = -1, y_j = 1$ can happen if x_j is an actual FEP, while x_i contains random bytes that happen to parse into a `call` instruction to x_j. Given this observation, we define

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

This is a negative feature, in that $f_c = 1$ should reduce the likelihood of that particular labeling.

2. Binary overlap feature f_o. This is a negative feature too. For any two candidate FEPs x_i and x_j, we define

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

With these structural features, our model is a Conditional Random Field (CRF) (Lafferty, McCallum, & Pereira 2001) with nodes $y_{1:n}$ and pairwise connections. This now becomes a *structured classification* problem, since the labels are correlated and must be inferred together. We define the joint probability of the labels as

$$P(y_{1:n}|x_{1:n}, \mathcal{P}) = \frac{1}{Z} \exp\Big(\sum_{i=1}^{n} \sum_{u\in\{I\}} \lambda_u f_u(x_i, y_i, \mathcal{P}) + \sum_{i,j=1}^{n} \sum_{b=o,c} \lambda_b f_b(x_i, x_j, y_i, y_j, \mathcal{P})\Big) \quad (2)$$

where Z is the partition function, f_u are the idiom features, and f_o, f_c are the structural features. The weights λ_u are learned as we described in the previous section; λ_o and λ_c

are implicitly negative infinity as a consequence of the approximate prediction algorithm we describe below. Note that the graph contains many loops.

The Conditional Random Field allows us to incorporate heterogeneous features to define our objective. However, standard inference methods (e.g. loopy belief propagation (Frey & MacKay 1998)) on this large (up to 937,865 nodes in a single binary in our test data sets), highly connected graph is expensive for large-scale analysis. In the following section, we describe an approximate inference procedure that considerably speeds up classification. The result is an efficient, approximate model that can handle such binaries in under 15 seconds.

Large-Scale Binary Analysis

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

Because we perform idiom feature selection over the entire data set, composed of tens of millions of training examples, selecting among the tens of thousands of idioms is a costly enterprise. Fortunately, each iteration of a forward feature selection search lends itself easily to loosely coupled distributed computation in a *High Throughput Computing (HTC)* environment. HTC is a well-recognized compute idiom, exemplified by the Condor HTC system (Litzkow, Livny, & Mutka 1988; Thain, Tannenbaum, & Livny 2005) and recent *cloud computing* applications such as the SETI@HOME project (Werthimer *et al.* 2001) and commercial ventures like Amazon's EC2 (Garfinkel 2007). Even malicious software authors have recognized the value in distributing workloads over largely independent nodes using botnets (Barford & Yegneswaran 2007). In our work, we use Condor to bring hundreds of idle systems to bear on our feature selection computations.

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

We believe that this compute paradigm—effectively, one of unlimited cycles—is fundamental to approaching the kinds of scaling problems that arise in applying machine learning techniques to large-scale systems such as the automated binary analysis we describe here. Using HTC techniques is not difficult; with the Condor system, programs can generally be run unmodified on remote machines, with files transferred automatically to the compute node and back again after the remote job has finished executing.

Although the cost of idiom feature selection and model training is large, it is only a one time cost for a particular training data set. Much more important to our particular application domain is the cost of inference. We efficiently approximate our CRF model (2) by breaking down inference into three stages:

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

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

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

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

Experimental Results

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

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

The gaps remaining in the stripped binaries after static disassembly constitute the test data. Because the original binaries had full symbol information, we have a *ground truth* to which we can compare the output of our classifier on the candidate FEPs in these gaps. The sizes of training and test sets for each compiler are listed in Table 1.

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

Table 1: *Size of training and test sets*

We automatically select idiom features separately for each of the data sets, using the LIBLINEAR logistic regression library (Lin, Weng, & Keerthi 2007) to build our flat FEP model. The number of features selected reflects the varying complexity of function entry points for binaries from each compiler. While the GCC model contains only 12 idiom features, the MSVS model contains 32 and the ICC model contains 41. The latter two compilers optimize away much of the regularity at function entry that is found in code produced by GCC.

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

Table 2 lists the top five features for the two Linux data sets in the order they were selected. Although there are individual instructions common to both sets of chosen idioms, the difference between the two models reflects the difference in code generated by the two compilers. In particular, note that the first two idioms selected for the GCC model are similar to the `(push ebp | mov esp,ebp)` heuristic used by Dyninst. While this pattern is selected as a positive feature in the ICC model as well, its later selection and lower weight (not depicted here) reveal that it is a less reliable indicator of FEPs emitted by the ICC compiler.

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

Table 2: *Top features of GCC and ICC idiom models. The +/- column indicates whether a particular feature is a positive or negative indicator of FEP status.*

Prefix idioms tend to have relatively larger importance on the GCC and MSVS data sets that on the ICC data set; 50% and 53%, respectively, of selected features were prefix idioms. By comparison, only 34% of the features selected from the ICC data set were prefix idioms. There are several factors that may contribute to a larger number of prefix features being selected. For example, common entry idioms that also occur relatively frequently at other points in the binary increase false positives; prefix idioms can help eliminate these types of errors. Also, when FEPs show significant variation the immediately preceding bytes may be better indicators of function location. Our analysis suggests that the large number of prefix idioms chosen for GCC is due to the former factor, while the MSVS prefix idioms are largely due to the latter. The large number of non-prefix idioms in the ICC model reflects the preponderance of common entry sequences in these binaries, as well as the relative dearth of repeated prefix idioms.

We implemented our classifier as an extension to the Dyninst tool, replacing the heuristic function detection functionality with our structured classifier. Our implementation allows us to individually enable components of the model (idiom, call, and overlap features). Figure 3 depicts the relative contribution of each model component on the ICC test set (other datasets have similar behavior). Both of the structural features increase the area under the curve, but the contribution from the overlap feature is greater. Table 3 shows the $F_{0.5}$ measure for each model component and the full model. In all cases, adding structural information increased this measure on the test set.

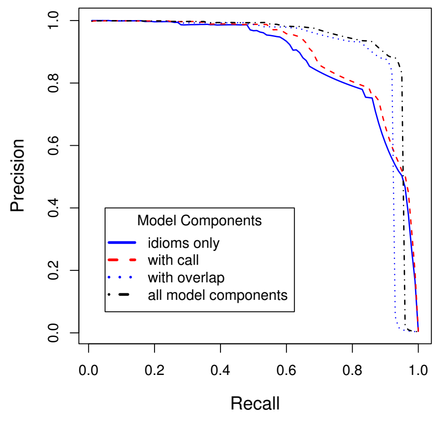
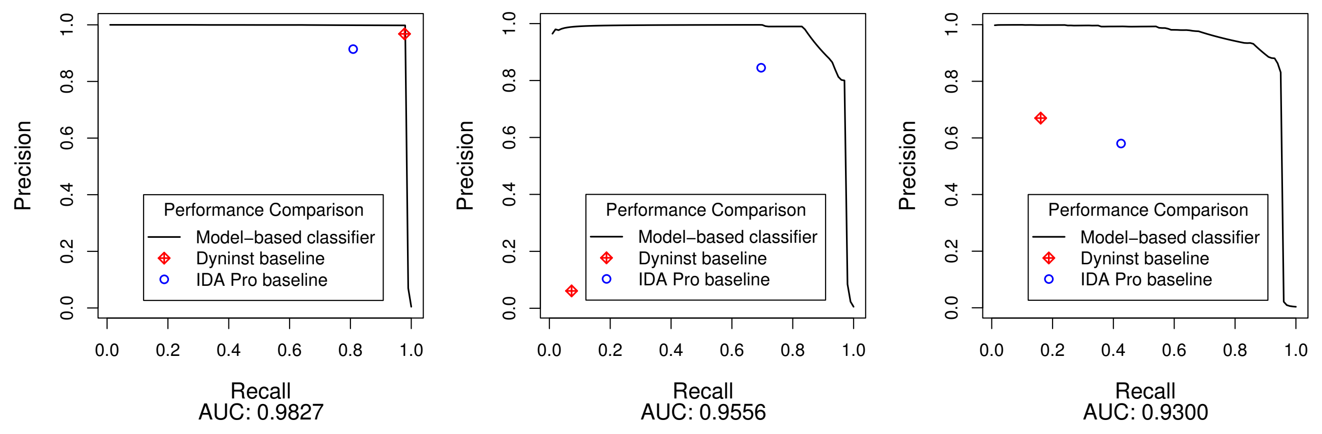


Figure 3: *Contribution of various model components toward performance on the ICC test set.*

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



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

Figure 4: *Precision-Recall curves comparing our model and the baseline classifiers. The GCC data set exhibits the most regular FEPs and is the easiest. Poor performance of the Dyninst baseline on the MSVS data set was due to a mismatch between Dyninst's heuristic patterns and the MSVS compiler's output.*

mation from the Microsoft Symbol Server. We graphically compare the Dyninst and IDA Pro tools to our classifier in Figure 4. In each case, our implementation of the classifier dramatically outperforms existing tools.

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

Table 3: *Performance contribution of model components (idioms, call, overlap) under the $F_{0.5}$ measure. Our classifier works best with all components enabled; in all cases it outperforms the baseline tools.*

Conclusions

We have presented a novel application of machine learning to binary program analysis, one that incorporates content and structure features of binary code to classify candidate FEPs in stripped binaries. Our classifier is based on a model of prefix and entry idioms, an abstraction of instruction sequences before and at an FEP; features are selected out of tens of thousands of candidate idiom features by using HTC resources. We formalized our model as a Conditional Random Field, incorporating structural call and overlap features in addition to idioms. Our efficient approximate inference procedure allowed us to quickly classify very large numbers of candidate FEPs in large binaries. Unlike previous approaches to parsing stripped binaries, our system does not rely on hand-coded patterns. Our extensions to the Dyninst tool allow it to adapt to future variations in compiler-emitted code, as well as code that does not conform to expected patterns. Experiments show that our CRF formulation with both content and structure features performed well on a large number of real-world datasets, outperforming existing, standard tools.

The variation in idiom models among the three data sets supports our hypothesis that FEPs emitted by different compilers would vary considerably. Our observations suggest that FEP idioms may be useful in the related problem of identifying the source compiler of an *unknown* binary or code sequence, a necessary step before our structured FEP models can be used on such code. We are investigating compiler identification as a matter of ongoing work.

Acknowledgements

This research is supported in part by Department of Energy grant DE-FG02-93ER25176, Lawrence Livermore National Lab grant B565054, National Science Foundation grant CNS-0627501, and Wisconsin Alumni Research Foundation. We thank the anonymous reviewers for their insightful comments and suggestions.

Appendix: Self-Repairing Disassembly

Consider a disassembly sequence A starting at some arbitrary byte offset, and a second disassembly sequence B starting d bytes later. It is sufficient to consider $0 < d < K$, where K is the length in bytes of the longest instruction (including its operands, if any) in the instruction set, because otherwise we can always remove the first few instructions in A. When will A's and B's disassembly instruction align?

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

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

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

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

Proof. We can view self-repair as a game between players A and B. Whichever player is behind can disassemble the next instruction in its sequence, which is equivalent to throwing a biased, K-sided die with probability for side l being $p(l)$. That player then advances l bytes. The game repeats until A and B arrive at the same byte, i.e., align. The number of moves A takes until the game is over is the number of disassembled instructions in sequence A before alignment.

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

Given T, it is well known that the corresponding fundamental matrix N contains the length of random walks (Doyle & Snell 1984). Specifically, N_{ij} is the expected number of times a random walk starting at state s_i would visit s_j before absorption. Here i, j are in $\{-(K-1), \dots, -1, 1, \dots, K-1\}$. Our random walks always start at s_d. Since it is A's turn to move whenever a random walk visits a state $j > 0$, the total expected moves A will make is $\sum_{j>0} N_{dj} = \sum_{j=1}^{K-1} N_{dj}$. □

Corollary 2. *The expected number of* bytes *in A, before A and B align, is* $\left(\sum_{j=1}^{K-1} N_{dj}\right)\left(\sum_{l=1}^{k} lp(l)\right)$.

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

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

References

Barford, P., and Yegneswaran, V. 2007. *An Inside Look at Botnets*, volume 27 of *Advances in Information Security*. Springer US. 171–191.

Cifuentes, C., and Emmerik, M. V. 2000. UQBT: Adaptable binary translation at low cost. *Computer* 33(3):60–66.

Data Rescue. 2007. *IDA Pro Disassembler: Version 5.0* `http://www.datarescue.com/idabase`.

Doyle, P., and Snell, J. 1984. *Random Walks and Electrical Networks*. Mathematical Association of America.

Frey, B. J., and MacKay, D. J. C. 1998. A revolution: belief propagation in graphs with cycles. In *Advances in Neural Information Processing Systems (NIPS)*.

Garfinkel, S. 2007. Commodity grid computing with amazon's s3 and ec2. *Login*.

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

Hollingsworth, J. K.; Miller, B. P.; and Cargille, J. 1994. Dynamic program instrumentation for scalable performance tools. Technical Report CS-TR-1994-1207, University of Wisconsin-Madison.

Intel Corporation. 2006. *IA-32 Intel(R) Architecture Software Developer's Manual*.

Kruegel, C.; Robertson, W.; Valeur, F.; and Vigna, G. 2004. Static disassembly of obfuscated binaries. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*, 18–18. Berkeley, CA, USA: USENIX Association.

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

Lin, C.-J.; Weng, R. C.; and Keerthi, S. S. 2007. Trust region newton methods for large-scale logistic regression. In *ICML '07: Proceedings of the 24th international conference on Machine learning*, 561–568. New York, NY, USA: ACM.

Linn, C., and Debray, S. 2003. Obfuscation of executable code to improve resistance to static disassembly. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, 290–299. New York, NY, USA: ACM.

Litzkow, M.; Livny, M.; and Mutka, M. 1988. Condor-a hunter of idle workstations. *Distributed Computing Systems, 1988., 8th International Conference on* 104–111.

Thain, D.; Tannenbaum, T.; and Livny, M. 2005. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience* 17(2-4):323–356.

Theiling, H. 2000. Extracting safe and precise control flow from binaries. In *RTCSA '00*, 23. Washington, DC, USA: IEEE Computer Society.

Werthimer, D.; Cobb, J.; Lebofsky, M.; Anderson, D.; and Korpela, E. 2001. Seti@home–massively distributed computing for seti. *Comput. Sci. Eng.* 3(1):78–83.