Machine Learning-Assisted Binary Code Analysis

Nathan Rosenblum[1], Xiaojin Zhu[1], Barton Miller[1], and Karen Hunt[2]

[1]Computer Sciences Department, University of Wisconsin-Madison; {nater,jerryzhu,bart}@cs.wisc.edu
[2]Department of Defense; huntkc@gmail.com

1 Introduction

Binary code analysis is a foundational technique in the areas of computer security, performance modeling, and program instrumentation. In computer security, such analysis can provide the basis for detecting, understanding and controlling malicious code. Any analysis of malicious program requires as a first step precisely locating the Function Entry Points (FEPs, the starting byte of each function) within the binary. When full symbol information is available this is a trivial step. Malicious software authors, however, are not known for helpfully providing debugging symbols along with virus payloads. In addition, commodity software is often distributed without symbols, such as in many Linux distributions.

In this paper, we consider the machine learning problem of *identifying FEPs in binaries where symbols indicating function location are stripped*. Our work is targeted at the processing of binaries on a large scale such as is needed in both network- and host-based analysis tools. As a result, we must keep false positive rates extremely low while trying to maximize recall. We consider binaries for both Linux and Windows on the Intel IA32 architecture.

Existing techniques have used recursive disassembly parsing and heuristics to locate code in stripped binaries [1–3, 6, 7]. Recursive disassembly parsing follows program control (branches and calls) and finds all functions reachable from the main program entry point. However, it breaks down in the presence of indirect control flow (pointer based) transfers that cannot be resolved statically: on a large set of binaries on our department Linux server, approximately 40% of functions were unrecoverable through recursive disassembly. These functions lie in *gaps* between discovered functions. However, in these gaps also lie jump tables, numeric and string constants, padding bytes (both fixed-valued and random) and code fragments. Functions may be aligned on a byte boundary or on word or larger boundaries.

Several tools, including Dyninst [3] and IdaPro [2], use a small number of manually created patterns to identify FEPs. These patterns are initial instruction sequences commonly found in known FEPs. For example, Dyninst searches Intel IA32 binaries for a common function preamble pattern that sets up a stack frame: `(push ebp | mov esp,ebp)`. While effective on some binaries, these heuristic patterns cannot adapt to variations in compiler, optimization level, and post-compilation optimization tools, all of which may significantly perturb the initial instruction sequence in an FEP. This work extends the Dyninst tool, adding a probabilistic model that classifies FEPs in gaps.

Let $\mathcal{P}$ be the program binary code. Let $x_{1:n}$ represent all the byte offsets within $\mathcal{P}$'s gaps. For each x_i, we can generate the disassembly starting at that byte offset. Our task is binary classification: predicting x_i as an FEP ($y_i = 1$) or not ($y_i = -1$). Note the labels $y_{1:n}$ can be *correlated* for several reasons: i) The instruction at x_i can span several bytes. If so, x_i and x_{i+1} represent conflicting (overlapping) parses, and are unlikely to be both FEPs; ii) The disassembly starting from x_i might contain a `call` instruction that calls byte offset x_j. If we believe x_i is an FEP, then x_j should be one too. We therefore use both "local" information (instruction sequence starting from x_i) and "global" information (instruction overlap, control flow graph, call graph) in making a global inference over the entire set $y_{1:n}$.

2 Model

Conceptually, our model is a Markov Random Field [4] with nodes $y_{1:n}$, and pairwise connections. We define the joint probability of labels as $P(y_{1:n}|x_{1:n}, \mathcal{P}) =$

$$\frac{1}{Z}\exp\left(\sum_{i=1}^{n}\sum_{u\in\{I\}}\lambda_u f_u(x_i,y_i,\mathcal{P})+\sum_{i,j=1}^{n}\sum_{b=o,c}\lambda_b f_b(x_i,x_j,y_i,y_j,\mathcal{P})\right)$$

where Z is the partition function, f_u are unary features, and f_b are binary features. We consider the following features:

1. Unary idiom features $\{f_I\}$. An idiom I is a short template sequence of instructions and a marker that indicates whether the sequence precedes or follows x_i. Wild cards are allowed to match one or more instructions. For instance, the idiom $I_1 =$ (`push ebp | * | mov esp,ebp`) would match $\mathcal{P}$ at x_i if the instruction sequence starting at x_i is (`push ebp | insn`$^+$ `| mov esp,ebp`). Similarly, $I_2 =$ (`PRE: ret | int3`) would match $\mathcal{P}$ at x_i if the instruction sequence immediately preceding x_i is `(ret | int3)`. We thus define a binary feature $f_I(x_i, y_i, \mathcal{P}) = 1$ if $y_i = 1$ and idiom I matches $\mathcal{P}$ at x_i, and 0 otherwise. The intention is to capture common initial FEP instructions. We discuss idiom selection in Section 3.

2. Binary overlap feature f_o. For any byte offset x_i, we can always assume that it is an FEP and perform disassembly parsing to obtain the whole (assumed) function starting at x_i. We can do the same for x_j. We call x_i and x_j overlap, if their assumed functions share any bytes. We then define $f_o(x_i, x_j, y_i, y_j, \mathcal{P}) = 1$ if $y_i = y_j = 1$ and

x_i, x_j overlap, and 0 otherwise. This is a negative feature, in that λ_o should be negative to discourage the labeling $y_i = y_j = 1$ when x_i, x_j overlap.

3. Binary call-consistency feature f_c. If we assume $y_i = 1$, and the assumed function starting at x_i contains a `call` instruction to x_j, then it does not make sense to let $y_j = -1$ (the other 3 combinations of y_i, y_j are fine). We thus define $f_c(x_i, x_j, y_i, y_j, \mathcal{P}) = 1$ if $y_i = 1, y_j = -1$ and the function starting at x_i calls x_j, and 0 otherwise. Again this is a negative feature.

3 Implementation

The Markov Random Field allows us to incorporate heterogeneous features to define our objective. However, learning and inference on this large (up to 937,865 nodes in a single binary) , highly connected graph is expensive for large scale analysis. In this preliminary study, we considerably simplify the learning and inference procedure. The result is an efficient, approximate model that can handle such binaries in under fifteen seconds. It has three stages:

1. We start with only the unary idiom features in the model. This is equivalent to logistic regression on the idioms. We restrict an idiom to have at most three instructions. Even so, there are several tens of thousands of candidate idioms to consider. We therefore perform forward feature selection. Feature selection and model training are conducted on data derived from a large corpus of binaries we describe in Section 4. Our performance measure during feature selection is $F_{0.5}$, a harmonic mean that weights precision twice as much as recall. By partitioning the set of candidate idioms and staging jobs out to the Condor high throughput computing system [5], we keep feature selection time reasonable.

2. We then fix the parameters $\{\lambda_I\}$, and add overlap-check. From the above model with only idioms, one can compute $P(y_i = 1|x_i, \mathcal{P})$. As an approximation, we consider the *score* s_i of x_i. Initially, $s_i = P(y_i = 1|x_i, \mathcal{P})$ where the probability is from the idiom-only model. If x_i and x_j 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 $x_{i1}, \ldots, x_{ik}$, we set $s_j \leftarrow \max(s_j, s_{i1}, \ldots, s_{ik})$.

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.

4 Preliminary Results

We tested our system on two different corpora of binaries. The first consisted of 625 programs on our department Linux server. Each of these binaries had full symbol information available as ground truth. Our second set consisted of 443 binaries from the Microsoft Windows XP SP2 distribution; for these binaries, symbol information was obtained from the Microsoft's online symbol server.

Training data were obtained by recursively parsing from the main program entry point of each binary. Test data were obtained by parsing from every byte in the gaps left over after obtaining training data. There are

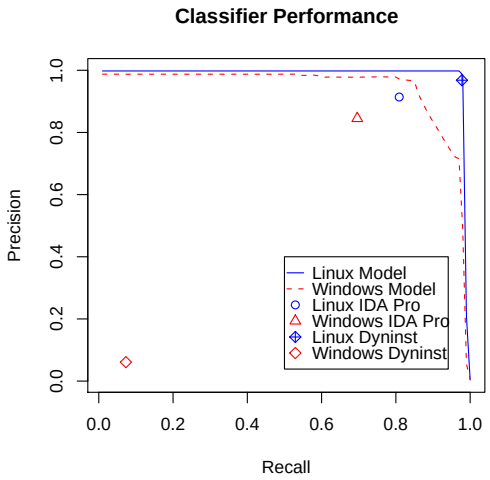

Figure 1: Precision-Recall curves of the classifiers on Linux and Windows test sets.

8,020,828 training and 11,231,721 test examples on the Windows platform; one Linux, we have 8,412,711 training and 22,806,449 test examples. Feature selection and model training are performed using training data on the whole corpus. Feature selection terminates when the incremental improvement in $F_{0.5}$ becomes small. We selected two idiom features for Linux binaries, and twenty-seven idiom features for Windows, and applied overlap-check and call-consistency as in Section 3.

Our models achieve significantly higher precision and recall than the baselines, as shown in Figure 1. Better results on the Linux corpus for all tools reflects the overwhelming uniformity of FEP instruction sequences in our Linux binaries. In contrast, the Windows binaries we tested contained significant entry point variation, which is reflected both in the poorer performance of existing techniques and the rapid reduction in precision in our model. Incorporating additional information sources in the binary to compensate for FEP variation is the focus of ongoing work.

References

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

[2] Data Rescue. IDA Pro Disassembler `http://www.datarescue.com/idabase`.

[3] Hollingsworth, J. K., Miller, B. P., and Cargille, J. Dynamic program instrumentation for scalable performance tools. Tech. Rep. CS-TR-1994-1207, University of Wisconsin-Madison, 1994.

[4] Kindermann, R., and Snell, J. L. *Markov Random Fields and Their Applications*. American Mathematical Society, 1980.

[5] Thain, D., Tannenbaum, T., and Livny, M. Distributed computing in practice: the Condor experience: Research articles. *Concurr. Comput. : Pract. Exper. 17*, 2-4 (2005), 323–356.

[6] Theiling, H. Extracting safe and precise control flow from binaries. In *RTCSA '00* (Washington, DC, USA, 2000), IEEE Computer Society, p. 23.

[7] Vigna, G. *Malware Detection*. Advances in Information Security. Springer, 2007, ch. Static Disassembly and Code Analysis.