Statistical Binary Parsing

Using Machine Learning to Extract Code from Uncooperative Programs

Nathan Rosenblum
Paradyn Project

Paradyn / Condor Week
Madison, Wisconsin
April 30 - May 3, 2007
Research Participants

• Barton Miller - UW Madison
• Jerry Zhu - UW Madison
• Karen Hunt - DoD
• Jeff Hollingsworth - UMD
Context of Current Work

- Exploratory
- Focus: evaluating machine learning techniques
- Eventual integration with Dyninst

Current Phase

1. Exploration of machine learning techniques
2. Selection & optimization of best methods
3. Integration into Dyninst tool

Phase 2

Phase 3
Talk Outline

• Binary parsing challenges
• Machine Learning Infrastructure
• Testing and Evaluation Infrastructure
• Preliminary Results
Automated Batch Parsing

• Cannot rely on human input
  • Parsing very large (100 MB) binaries
  • Parsing large numbers of binaries
  • Decisions require expert knowledge

• Complete & accurate information is essential
  • Binary modification, instrumentation
  • Misidentifying code can have catastrophic consequences

• Goal: Find code location in binaries
  • Eliminate false positives
  • Minimize false negatives
Parsing Challenges

• Obtaining full coverage may be difficult:
  • Missing symbol information
  • Variability in function layout (e.g. code sharing, outlined basic blocks)
  • High degree of indirect control flow

• Basic strategy: recursive descent parsing
  • Disassemble from known entry points
  • Discover functions through calls
Incomplete Parsing Coverage

• 41% of functions in surveyed binaries unreachable

• As many as 90% in some programs

• Unreachable functions occupy gap regions in the binary
Challenge: Accurate Gap Parsing

- Gaps are sequences of bytes
- Need to identify functions in gaps
  - Equivalently, identify function entry blocks
Offset Parsing Alignment

Statistical Binary Parsing: Using Machine Learning to Extract Code from Uncooperative Binaries
Current Dyninst Techniques

- Dyninst searches for common patterns
  - `push %ebp; mov %esp,%ebp`
  - `push %esi; mov %esi,<mem>`

- Performs well
  - Low false positive rate: 92% precision on average

- Heuristic - patterns are moving target

- Larger programs - more false positives

- Compiler may not emit expected preamble
  - Partial known sequences
Exploiting Available Information

• *Some* properties of functions are relatively uniform
  • E.g., stack setup

• Use properties of known code to search gaps

![Pie chart showing percentages of static and reachable code](chart.png)

- Statically Reachable: 59%
- Gap: 41%
Statistical Binary Parsing

- Parsing as a supervised machine-learning problem
  - Build model from training examples
  - Use model to classify code in gaps
- Goals:
  - Extensible: incorporate multiple *features*
  - Opportunistic: exploit all available information
Learning Infrastructure

- Logistic Regression classifier

- Incorporates several features:
  - Instruction frequency (language models)
  - Function entry sequences
  - Control flow

- Assigns probability to candidate functions
Language Models

• Frequency of instruction occurrence
• Compares entry and non-entry models

Candidate entry block

Insn1
Insn2
Insn3
Insn4
Insn5

Entry LM

Non-entry LM

odds

odds

Log-odds ratio

÷
Function Entry Sequences

- **Method 1: Maximum Prefix Match Length**
  - Incorporates instruction ordering
  - Construct *prefix trie* of entry block sequences
  - Compute *maximum match length* for candidate entry blocks

Candidate 1: actual entry block

\[ a, b, d, h, x, \ldots \]  
**MPML:** 4

Candidate 2: non-entry block

\[ a, q, x, y, z, \ldots \]  
**MPML:** 1

Limited flexibility!

\[ a, x, b, d, h, \ldots \]  
**MPML:** 1
Function Entry Sequences

- **Method 2: Fuzzy String Matching**
  - Levenshtein Distance counts edits between strings
    - Insertion, deletion, change
  - Flexible: matches sequences but allows gaps

Candidate (valid)  
\[ a, x, b, d, h, \ldots \]

Best match  
\[ a, b, d, h, \ldots \]

Edit Distance: 1

Match minimum edit distance

Entry Prefixes

Insertion
Incorporating Control Flow

Parsing from every byte in a range creates a graph

Reachability Ratio = \( \frac{\text{# blocks reachable from candidate}}{\text{# blocks connected to candidate}} \)
Experimental Framework

• Goal: evaluate effectiveness of features
• 625 Linux x86 binaries
• Binaries have full symbol tables
  • Function locations provide ground truth reference set
• Stripped binaries provide training data
• Dyninst prefix heuristic provides baseline
Obtaining Training and Test Data

- Classifier is trained and evaluated on each binary independently.

- Positive training examples:
  - Known function entry blocks

- Negative training examples:
  - Known non-entry blocks
  - Blocks generated from parse at every byte within known functions ("anti-gaps")

- Test examples are all candidates in gaps
Scaling Experiments

• Experiment design facilitates scaling
  • Separation of model creation, training, and evaluation
  • Independent analysis of each binary
  • Suitable for batch processing systems like Condor

• Reduced cost in final Dyninst implementation
  • Early rejection of invalid parses
  • On-demand analysis of sub-regions of gaps
  • Final approach will use subset of techniques
Results

• Language Model features have limited utility
  • Limited training data
  • May be improved by training over whole corpus

• Prefix-based features work well
  • LD better than MPML
  • LD is current best combined with Dyninst heuristic
  • Most sensitivity to training data variation

• Incorporating control flow is essential
  • 60% reduction in false positives over best method alone
Results

- **Current status:**
  - 70% reduction in *false positives* over Dyninst heuristic
  - Nearly identical *false negative* rates

<table>
<thead>
<tr>
<th>Prog</th>
<th>Total Functions</th>
<th>Gap Funcs</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>grep</td>
<td>140</td>
<td>94</td>
<td>100%</td>
<td>90.5%</td>
</tr>
<tr>
<td>mutt</td>
<td>1122</td>
<td>223</td>
<td>98.6%</td>
<td>98.6%</td>
</tr>
<tr>
<td>emacs</td>
<td>3214</td>
<td>1596</td>
<td>99.9%</td>
<td>99.9%</td>
</tr>
<tr>
<td>Abiword</td>
<td>13844</td>
<td>538</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>gpg</td>
<td>991</td>
<td>172</td>
<td>41.7%</td>
<td>99.4%</td>
</tr>
</tbody>
</table>
Future Work

• Model extension, evaluation and refinement
  • What other features characterize entry points?
  • Which features best distinguish valid entry points?

• Integration into Dyninst
  • Model training
  • Parsing optimizations
  • API extensions
  • Fall 2007
Future Work

• Dealing with limited training data
  • Can similar binaries be exploited to obtain more training examples?
• Incorporating additional sources of information
Questions?
Backup slides
Language Models

- Obtained by Maximum Likelihood Estimate (MLE) of instructions (unigram) and pairs of instructions (bigram)

\[
P(insn_k) = \frac{\sum_{b \in \text{EntryBlocks}} \sum_{i \in \text{Insns}} cnt_b(i) + 1}{\sum_{b \in \text{EntryBlocks}} \sum_{i \in \text{Insns}} cnt_b(i) + |\text{Insns}|}
\]

\[
P(block_k) = \prod_{i \in \text{Insns}_b} P(i)
\]
**Language Models**

- Log-odds ratio computed from language models

**Two models trained:**

- Entry blocks
- Non-entry blocks

\[
\text{odds}_{\text{entry}}(b) = \frac{P_{\text{entry}}(b)}{1 - P_{\text{entry}}(b)}
\]

\[
\text{odds}_{\text{nonentry}}(b) = \frac{P_{\text{nonentry}}(b)}{1 - P_{\text{nonentry}}(b)}
\]

\[
\text{LOR}(b) = \log\left(\frac{\text{odds}_{\text{entry}}(b)}{\text{odds}_{\text{nonentry}}(b)}\right)
\]
An example