Stripped Binaries and Compiler Identification

Nathan Rosenblum
Paradyn Project

Paradyn / Dyninst Week
College Park, Maryland
April 27 - 28, 2009
But first...

"Binary program provenance"

starting with source → obfuscation → packers/encryption → ending with a binary

link-time optimizers

binary rewriting

compilers

Ah ha, provenance!

We started with:

Stripped binary parsing → Machine learning approaches → Compiler-specific problems arise → Compiler identification → ...

(Paradyn week 2008)
Parsed binaries are convenient...

program

binary

headers

code

data

7a 01 00 fd a2 b3
74 68 69 73 20 65
78 61 6d 70 6c 65
20 69 55 85 e5 6f
67 75 73 2e 2e 2e
7a 01 00 fd a2 b3
74 68 69 73 20 65
78 61 6d 70 6c 65
Complete parsing is hard

Fraction of functions in “gap regions” after static parsing

Number of binaries

5% 95%
Getting full coverage

Approach 1: Dynamic parsing at runtime
Approach 2: Make static parsing better

Model many programs

So... we’re done, right?

the other guys...

Stripped Parsing and Compiler Identification
Looking for hidden assumptions

1. gather training binaries
2. distribute (e.g. with Condor)
3. learn something
4. repeat (many times)
5. build a model (corresponding to a particular compiler)

Parsing

unknown, stripped binary

icc model

gcc model
Wait, different models per compiler?
Reverse the problem

static parse

unknown, stripped binary

“Best” model

“most accurate”
on statically analyzeable code

(incomplete!)

I choose you, GCC!
Simple solutions sometimes surprise

unknown, stripped binary

static parse

Multiple compilers for a single binary?
Ex: Binary compiled by ICC with 2 GCC-compiled library methods

Not an exaggeration
Side trip: alignment and conflict

<table>
<thead>
<tr>
<th>Stripped Parsing and Compiler Identification</th>
</tr>
</thead>
</table>

**stripped binary**

**parsing from every possible point**

<table>
<thead>
<tr>
<th>candidate 1</th>
<th>candidate 2</th>
<th>candidate 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>add 0x14, esp</td>
<td>push ebp</td>
</tr>
<tr>
<td>53</td>
<td></td>
<td></td>
</tr>
<tr>
<td>83</td>
<td>sub 0xc, esp</td>
<td>sub 0xc, esp</td>
</tr>
<tr>
<td>83</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0c</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8b</td>
<td>mov 0x14(esp), ebx</td>
<td>mov 0x14(esp), ebx</td>
</tr>
<tr>
<td>5c</td>
<td>mov 0x4(ebx), edx</td>
<td>mov 0x4(ebx), edx</td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>53</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8b</td>
<td>mov 0x0(ebx), ecx</td>
<td>mov 0x0(ebx), ecx</td>
</tr>
<tr>
<td>0b</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Model review: Idioms

Short sequences of instructions, possibly with wildcards, around function entry points

\[
\text{<< push ebp; mov esp,ebp >}
\]
\[
\text{<< push ebp ; * ; mov esp,ebp >>}
\]
\[
\text{<< * ; mov 0x8(ebp),eax >>}
\]
PRE \[
\text{<< ret ; nop >>}
\]
Representing structure

call

candidate function

conflict
Intractable!

labeling all possible functions at once

\[ P(y_{1:n}|x_{1:n}, \mathcal{P}) = \frac{1}{Z} \exp \left( \sum_{i=1}^{n} \sum_{u \in \{I\}} \lambda_u f_u(x_i, y_i, \mathcal{P}) + \sum_{i,j=1}^{n} \lambda_{c} f_c(x_i, x_j, y_i, y_j, \mathcal{P}) + \sum_{i,j=1}^{n} \lambda_{o} f_o(x_i, x_j, y_i, y_j, \mathcal{P}) \right) \]
Approximate!

Exact method $\rightarrow$ exponential in size of binary $\rightarrow$

Contrastive Divergence $\rightarrow$ very simple approximation $\rightarrow$

(basically optimizing a related, easier problem)

(heat death of universe)

(couple of coin flips)
Preliminary results are preliminary

1. Earlier successes are encouraging
   10-fold experiment using “most accurate” model
   only 6 mistakes out of ~ 1000 programs
2. “Toy” results suggest approximate CD learning is effective
   “small enough to compute exactly”
   relatively smooth increase in likelihood during learning

\[
\begin{align*}
\text{log likelihood} & \quad \text{optimization iterations} \\
0.0 & \quad 0.0 \\
0.2 & \quad 0.4 \\
0.4 & \quad 0.8 \\
0.6 & \quad 1.0
\end{align*}
\]
Summing up

stripped binary parsing

general code models

function-granularity

compiler identification!

compiler-specific code models

Stripped Parsing and Compiler Identification
Backup slides follow
## Experimentation

- **GNU C Compiler**
  - Simple, regular function preamble

- **MS Visual Studio**
  - High variation in function entry point idioms

- **Intel C Compiler**
  - Most variation in entry points; highly optimized

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Programs examined</th>
<th>Total Training Examples (pos+neg)</th>
<th>Total Test Examples (pos+neg)</th>
<th>Actual number of functions in gaps</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC</td>
<td>625</td>
<td>8,412,711</td>
<td>22,806,449</td>
<td>85,870</td>
</tr>
<tr>
<td>MS VS</td>
<td>443</td>
<td>8,020,828</td>
<td>11,231,721</td>
<td>70,620</td>
</tr>
<tr>
<td>ICC</td>
<td>112</td>
<td>1,364,598</td>
<td>13,169,487</td>
<td>47,841</td>
</tr>
</tbody>
</table>
## Testing

Comparison of **three** binary analysis tools:

- Original Dyninst
- IDA Pro Disassembler
- Dyninst w/ Model

- Scans for common entry preamble
- Model replaces entry preamble heuristic
- List of Library Fingerprints (Windows)

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Orig. Dyninst</th>
<th>IDA Pro</th>
<th>Dyninst w/ Model</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FP</td>
<td>FN</td>
<td>FP</td>
</tr>
<tr>
<td>GCC</td>
<td>2,833</td>
<td>2,012</td>
<td>14,576</td>
</tr>
<tr>
<td>MS VS</td>
<td>79,320</td>
<td>65,586</td>
<td>9,044</td>
</tr>
<tr>
<td>ICC</td>
<td>3,786</td>
<td>40,195</td>
<td>14,422</td>
</tr>
</tbody>
</table>
Model Components

\[ \frac{TP}{TP + FP} \]

\[ \frac{TP}{TP + FN} \]

Precision vs. Recall

- **Model Components**
  - idioms only
  - with call
  - with overlap
  - all model components

(1.0, 1.0)