Rendezvous: A search engine for binary code

Wei Ming Khoo, Alan Mycroft, Ross Anderson
University of Cambridge

CREST Open Workshop on Malware
29 May 2013

Demo: http://www.rendezvousalpha.com
Software reverse engg.

Software RE is tedious, requires expertise

• Decompilers
  - Boomerang, REC Studio 4, Anatomizer, Andromeda, exetoc, desquirr
  - Current state-of-the-art: Hex-Rays, USD$1,160 per license per year + expertise
  - 415 man-hours to decompile 1,500 LoC comprising 8% of code base [VanEmmerik’04]

• Stuxnet
  - Assuming deployed in June 2009, took a year to be discovered, a further 5 months for AV and SCADA experts to decipher the payload
But, code reuse is prevalent

And increasingly so due to advances in software mining and SBSE

• Catalysts include market competitiveness, application complexity, quality of reusable components [Schmidt’99, ’00, ’06]

• Six open source projects: On average 74% of code base was external [Haefliger’08]

• Sometimes illegally: >250 products found GPL non-compliant, most famously Linksys WRT54G
Code reuse in malware

- Malware producers operate very much like corporations
  - Innovative Marketing Ukraine: revenue of about $180 million in 2008, complete with HR dept and call center (Finkle’10)

- ZeuS 2.0.8.9 source code leaked in May 2011 revealed the following FOSS components
  - xterm/key2symucs.c (© April 2001 Markus G. Kuhn, U. Cambridge)
  - UltraVNC circa 2005
  - BEA disassembly engine
  - UCL compression lib 1.0.3
  - Info-zip 2.3.2
  - Mersenne twister PRNG circa 2002 (Matsumoto & Nishimura)

- 94.3% LoC of ZeuS 2.0.8.9 was reused, only 5.7% was new functionality
Proposed solution

Search-based reverse engineering (SBRE)

“Google” it:

Instead of “How to decompile?” we ask “Given a candidate decompilation, how good a match is it?”

Similar shift occurred for statistical machine translation
Take away slide

• Software RE is tedious, expertise required
• Code reuse is common in software, malware included
• We propose reframing: software RE as a search problem, relying on existing and available software to obtain source code
• Q: How can we do this in a way that is compiler-agnostic? (Assuming we deal with packers, obfuscators)
How we achieve this

• Design trade-offs
• Feature extraction
• Indexing & Querying
• Experimental results
Design space

• We want features that can uniquely identify functions
• We want speed + accuracy: We chose **Speed** first
• Speed meant that we chose static over dynamic analysis (Assumption: no obfuscation)
• We studied heuristic features from existing literature that can be extracted directly from a disassembly:
  - Instruction mnemonics n-grams, n-perms
  - Control-flow sub-graphs, extended sub-graphs
  - Data constants
Instruction mnemonics

• Instruction mnemonic (textual) differs from an opcode (hex), e.g. 0x8b (load) and 0x89 (store) map to ‘mov’

• Assume a Markov property, $n^{th}$ token is influenced by the previous $n - 1$ tokens

• Considered $n = 1, 2, 3, 4$

push, mov, push $\rightarrow$ 0x73f973 $\rightarrow$ XvxFGF
n-grams vs n-perms

• n-gram is sequence-based, n-perm is set-based

mov ebp, esp
sub esp, 0x10
movl -0x4(ebp), 0x1

mov ebp, esp
movl -0x4(ebp), 0x1
sub esp, 0x10

• For instance, two n-grams (mov, sub, movl) & (mov, movl, sub)

• Only 1 n-perm (mov, movl, sub)
Control-flow $k$-graphs

- $k$-graph is a connected sub-graph comprising $k$ nodes, compute them all ($k = 3, 4, 5, 6, 7$)
- Convert to $k$-by-$k$ matrix and compute its canonical form, rep as string (Nauty graph library)
Extended $k$-graphs

- One shortcoming of $k$-graphs: uniqueness low for small $k$
- We propose extended $k$-graphs
- Extended $k$-graph includes edges that have one end point at an internal node, but have another at an external virtual node, $V^*$
Extended $k$-graphs

$k$-graph

Extended $k$-graph
Constants

- Empirical observation that data constants do not change with compiler or options
- Considered 32-bit integers and strings
- Immediate operands, pointer offsets (excluding stack and frame pointer offsets)
- Integer may be an address, do a lookup
Feature extraction

Executable

Disassembly

Mnemonic n-grams

Control-flow sub-graphs

Data Constants

Tokenise

Token-specific processing

Alphabetic strings (Query terms)
Indexing & querying

Indexing
- Corpus
- Indexer
- Term Freq.
- Inverted Index

Querying
- Query
- Query Engine
- Search results

unknown .exe

alphabetic strings
Indexing & Querying

- 2 query models—the Boolean model (BM), and vector space model (VSM)
- BM is set-based, boolean operators such as AND, OR and NOT
- VSM is distance-based, weight vectors computed via normalised term frequencies
- Our model is based on the combination of the two: documents are first filtered via the BM, then ranked and scored by the VSM
Indexing & Querying

- Executable is abstracted to set of terms
- 3 strategies to deal with long queries, for desired query length $l_Q$
- Term de-duplication, Padding and Unique term selection
- Term de-duplication (up to $2 \times l_Q$) a simple strategy that reduces the query size
- Padding: Include common terms prepended by NOT
- E.g. For $l_Q = 3$, and query is $A \ AND \ B$, pad with $A \ AND \ B \ AND \ NOT \ C$
- Unique term selection: Select only the terms with frequency $< df_{threshold}$
Scoring

- Default CLucene scoring function

\[ \text{Score}(Q, D) = \text{coord}(Q, D) \cdot C \cdot \frac{V(Q) \cdot V(D)}{|V(Q)|} \]

where \(\text{coord}\) is a score factor,

\(C\) is a normalisation factor,

\(V(Q) \cdot V(D)\) is the dot product of the weighted vectors, and

\(|V(Q)|\) is the Euclidean norm
What makes a good model?

- True positives (tp): a correctly retrieved document relevant to the query
- False positive (fp): an incorrectly retrieved irrelevant document
- False negative (fn): a missing but relevant document

**Precision**

$$\text{precision} = \frac{tp}{tp + fp}$$

**Recall**

$$\text{recall} = \frac{tp}{tp + fn}$$

$$F = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

$$F_2 = \frac{5 \cdot (\text{precision} \cdot \text{recall})}{4 \cdot (\text{precision} + \text{recall})}$$
Implementation

• Disassembly: Dyninst binary instrumentation framework (http://dyninst.org)
• Indexing & Querying: CLucene text search engine (http://clucene.sourceforge.net)
• Term frequency map is a Bloom filter
• Code abstraction: 10,500 lines of C++
• Indexing/Querying: 1,000 lines of C++
Questions

• Optimal value of $df_{\text{threshold}}$?

• Accuracy of various abstractions?

• Accuracy for different compilers?

• Accuracy for different compiler options?

• Timing
Datasets

- GNU C Library 2.16 (glibc)
  - 2,706 functions, 1.18 MLoC
  - Compiled with gcc -O1, -O2

- GNU coreutils 6.10 (coreutils)
  - 1,205 functions, 70,000 LoC
  - Compiled with gcc, clang
\(df_{\text{threshold}}\)

<table>
<thead>
<tr>
<th>( \leq df_{\text{threshold}} )</th>
<th>Precision</th>
<th>Recall</th>
<th>( F_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.202</td>
<td>0.395</td>
<td>0.331</td>
</tr>
<tr>
<td>2</td>
<td>0.177</td>
<td>0.587</td>
<td>0.401</td>
</tr>
<tr>
<td>3</td>
<td>0.165</td>
<td>0.649</td>
<td>0.410</td>
</tr>
<tr>
<td>4</td>
<td>0.161</td>
<td>0.677</td>
<td>0.413</td>
</tr>
<tr>
<td>5</td>
<td>0.157</td>
<td>0.673</td>
<td>0.406</td>
</tr>
<tr>
<td>6</td>
<td>0.160</td>
<td>0.702</td>
<td>0.418</td>
</tr>
<tr>
<td>7</td>
<td>0.159</td>
<td>0.709</td>
<td>0.419</td>
</tr>
<tr>
<td>8</td>
<td>0.157</td>
<td>0.708</td>
<td>0.415</td>
</tr>
<tr>
<td>9</td>
<td>0.157</td>
<td>0.716</td>
<td>0.418</td>
</tr>
<tr>
<td>10</td>
<td>0.155</td>
<td>0.712</td>
<td>0.414</td>
</tr>
<tr>
<td>11</td>
<td>0.151</td>
<td>0.696</td>
<td>0.405</td>
</tr>
<tr>
<td>12</td>
<td>0.152</td>
<td>0.702</td>
<td>0.408</td>
</tr>
<tr>
<td>13</td>
<td>0.153</td>
<td>0.705</td>
<td>0.410</td>
</tr>
<tr>
<td>( \infty )</td>
<td>0.151</td>
<td>0.709</td>
<td>0.408</td>
</tr>
</tbody>
</table>
n-grams vs n-perms

n-grams out-performed n-perms for $n > 1$

Possible explanation: Unique terms
# k-graphs vs ex. k-graphs

<table>
<thead>
<tr>
<th>k-graph</th>
<th>glibc</th>
<th>extended k-graph</th>
<th>coreutils</th>
<th>extended k-graph</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Precision</td>
<td>Recall</td>
<td>$F_2$</td>
<td>Precision</td>
</tr>
<tr>
<td>3-graph</td>
<td>0.070</td>
<td>0.133</td>
<td>0.113</td>
<td>0.022</td>
</tr>
<tr>
<td>4-graph</td>
<td>0.436</td>
<td>0.652</td>
<td>0.593</td>
<td>0.231</td>
</tr>
<tr>
<td>5-graph</td>
<td>0.730</td>
<td>0.700</td>
<td>0.706</td>
<td>0.621</td>
</tr>
<tr>
<td>6-graph</td>
<td>0.732</td>
<td>0.620</td>
<td>0.639</td>
<td>0.682</td>
</tr>
<tr>
<td>7-graph</td>
<td>0.767</td>
<td>0.609</td>
<td>0.635</td>
<td>0.728</td>
</tr>
</tbody>
</table>
Mixed n-grams

- 1+4-grams & 2+4-grams were best performers
- Out-performed the best n-gram model (coreutils: 0.764, glibc: 0.664)
Mixed k-graphs

<table>
<thead>
<tr>
<th></th>
<th>glibc $F_2$</th>
<th>coreutils $F_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3+4-graphs</td>
<td>0.607</td>
<td>0.509</td>
</tr>
<tr>
<td>3+5-graphs</td>
<td>0.720</td>
<td>0.630</td>
</tr>
<tr>
<td>3+6-graphs</td>
<td>0.661</td>
<td>0.568</td>
</tr>
<tr>
<td>3+7-graphs</td>
<td>0.655</td>
<td>0.559</td>
</tr>
<tr>
<td>4+5-graphs</td>
<td>0.740</td>
<td>0.624</td>
</tr>
<tr>
<td>4+6-graphs</td>
<td>0.741</td>
<td>0.624</td>
</tr>
<tr>
<td>4+7-graphs</td>
<td>0.749</td>
<td>0.649</td>
</tr>
<tr>
<td>5+6-graphs</td>
<td>0.752</td>
<td>0.650</td>
</tr>
<tr>
<td>5+7-graphs</td>
<td><strong>0.768</strong></td>
<td><strong>0.657</strong></td>
</tr>
<tr>
<td>6+7-graphs</td>
<td>0.720</td>
<td>0.624</td>
</tr>
</tbody>
</table>

5+7-graphs best performer for both sets
Constants

<table>
<thead>
<tr>
<th></th>
<th>Precision</th>
<th>Recall</th>
<th>$F_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>glibc</td>
<td>0.690</td>
<td>0.679</td>
<td>0.681</td>
</tr>
<tr>
<td>coreutils</td>
<td>0.867</td>
<td>0.751</td>
<td>0.772</td>
</tr>
</tbody>
</table>

Possible explanation: None of the functions in *glibc* had strings, whilst 889 functions, or 40.3% of functions in *coreutils* did.
Composite models

<table>
<thead>
<tr>
<th></th>
<th>Precision</th>
<th>Recall</th>
<th>$F_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>glibc</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4-gram/5-graph/</td>
<td>0.870</td>
<td>0.866</td>
<td><strong>0.867</strong></td>
</tr>
<tr>
<td>constants</td>
<td>0.850</td>
<td>0.841</td>
<td>0.843</td>
</tr>
<tr>
<td>1-gram/4-gram/5-graph/</td>
<td>0.118</td>
<td><strong>0.925</strong></td>
<td>0.390</td>
</tr>
<tr>
<td>7-graph/</td>
<td>constants</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4-gram/5-graph/</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>constants</td>
<td>(r = 10)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>coreutils</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4-gram/5-graph/</td>
<td>0.835</td>
<td>0.829</td>
<td><strong>0.830</strong></td>
</tr>
<tr>
<td>constants</td>
<td>0.833</td>
<td>0.798</td>
<td>0.805</td>
</tr>
<tr>
<td>2-gram/4-gram/5-graph/</td>
<td>0.203</td>
<td><strong>0.878</strong></td>
<td>0.527</td>
</tr>
<tr>
<td>7-graph/</td>
<td>constants</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4-gram/5-graph/</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>constants</td>
<td>(r = 10)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• More components not necessarily better
• Looked at recall rates for top 10 results
## Results at a glance

<table>
<thead>
<tr>
<th>Model</th>
<th>glibc $F_2$</th>
<th>coreutils $F_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best $n$-gram (4-gram)</td>
<td>0.764</td>
<td>0.665</td>
</tr>
<tr>
<td>Best $k$-graph (5-graph)</td>
<td>0.706</td>
<td>0.627</td>
</tr>
<tr>
<td>Constants</td>
<td>0.681</td>
<td>0.772</td>
</tr>
<tr>
<td>Best mixed $n$-gram (1+4-gram)</td>
<td>0.777</td>
<td>0.671</td>
</tr>
<tr>
<td>Best mixed $k$-graph (5+7-graph)</td>
<td>0.768</td>
<td>0.657</td>
</tr>
<tr>
<td>Best composite (4-gram/5-graph/constants)</td>
<td>0.867</td>
<td>0.830</td>
</tr>
</tbody>
</table>
False negatives

- 342 from glibc: 206 had 6 instructions or less
- getfsent
  - In-lining of fstab_convert
  - Instruction substitution: xor ax, ax to mov ax, 0;
    call/leave/ret to leave/jmp
  - Instruction re-ordering
  - No n-grams, k-graphs, constants in common
Timing

<table>
<thead>
<tr>
<th>Abstraction</th>
<th>Average (s)</th>
<th>Worst (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$n$-gram</td>
<td>51.881</td>
</tr>
<tr>
<td></td>
<td>$k$-graph</td>
<td>114.922</td>
</tr>
<tr>
<td></td>
<td>constants</td>
<td>680.148</td>
</tr>
<tr>
<td></td>
<td>null</td>
<td>15.135</td>
</tr>
<tr>
<td>Query construction</td>
<td>6.133</td>
<td>16.125</td>
</tr>
<tr>
<td>Query</td>
<td>116.101</td>
<td>118.005</td>
</tr>
<tr>
<td>Total (2410 functions)</td>
<td>907.448</td>
<td>981.081</td>
</tr>
<tr>
<td>Total per function</td>
<td>0.377</td>
<td>0.407</td>
</tr>
</tbody>
</table>

- Timing for 2,410 coreutils functions
- 0.407s per function in worst case
- Constants extraction can be streamlined further
Conclusion

• Software RE is tedious, expertise required
• Code reuse is common in software
• We propose reframing: software RE as a search problem
• Able to achieve $F_2$ rates of 0.867 & 0.830 combining mnemonics, $k$-graphs and constants

http://www.rendezvousalpha.com