Malware similarity analysis on binary using approximate string matching

Preferred Infrastructure, Inc.
Shirou Maruyama, Shuzo Kashihara

NTT Secure Platform Laboratories
Shingo Orihara, Hiroshi Asakura

Nov 17, 2012
Profiles

- Marumaru / @marugorithm
  - R&D engineer
    - Algorithms
    - I love string processing

- Shuzo Kashihara/ @suma90h
  - Professional Software Engineer
    - Distributed Systems
  - Shuzo has experience in developing security products
Count of new malware increases as ever…
- Recently, threats of web and mobile are increasing

Source: McAfee Threats Report: Second Quarter 2012
Background: Evolution of Reverse Code Engineering technology and tools

- Evolution of reverse engineering
  - Dynamic analysis/Sandbox research and OSS are growing from about 2006 (are they accelerated by Virtualization technology?)
    - e.g.: CWSandbox, Norman Sandbox, Ether, BitBlaze and many more
  - Evolution of tools like a disassembler and static analysis approach looks little (from Shuzo’s viewpoint)
  - Several improvements of analysis procedure by Python
    - e.g: Immunity Debugger, PyDbg, PaiMei, IDAPython

- Tools what we can use
  - Disassembler, Debugger, Unpacker
  - Sandbox, Instruction trace
  - process monitor, Hex Editor (with eye-grep)

- Approach: Dynamic/Static, Whitebox/Blackbox
Background: Dynamic analysis vs Static analysis

- Dynamic analysis
  - There are some researchs and OSSs related sandbox, unpacker
  - They can use Blackbox, Whitebox or both approach
  - Demerit
    - They tend to take a time. Unexecuted code cannot be analyzed

- Static analysis
  - de facto standard tools: IDA Pro
  - There are researches that infer code semantics by disassembling and gathering API Call
  - PolyUnpack: Static and dynamic Hybrid approach for unpacking
  - Demerit
    - It is hard to process packed and obfuscated program directly
    - Finally, engineer need to RCE but human resource is limited
Motivation

- Pick quarrel with malwares and authors..
  - It is difficult to find malware author/writer
  - Improve ease malware reverse engineering

- We want to apply approximate search to static analysis (RCE)
  - There is a limit to what sandbox and behavior analysis can do
  - Focus engineer’s scope on code when reversing code
    - Suggesting past malware, known code blocks

- Can we make use of malware collection for analysis?
  - Create database
  - Is there possible to use string matching and using machine resource?
Concept: Malware analysis using approximate string matching

- Find analysis target binary from gathered files with approximate string matching
  - We want human to focus where need to reversing
  - We want to provide information like debug symbols
Related software: IDAScope

- IDAscope plugin (Hex-Rays 2012 Contest Results)
  - Automatically tag unnamed functions with information gathered from the API calls it makes
  - Spot 'wrapper' functions: those that make a function call (e.g., an API call)
  - Provide an in-IDA WinAPI browser. You won't ever need to leave the IDA window anymore! (well, when working on Windows files, that is)

by Hex-Ray web

- IDAScope probably solves many problems, you don’t have to test approximate string matching if you had IDA license
- Common points:
  - Goals is the same: Ease reverse engineering
  - Our string matching approach assume similar usecase

Goal of this presentation

- Is approximate string matching for reversing realized? We try evaluating for reversing
Outline of our searching algorithm

- The Problem to extract
  - Exact matching is not effective.
    - ex.) different addresses.
  - Difficult to detect starting point of modules.

- Approximate matching
  - Finding similar patterns.
  - Measure of similarity?
  - What’s Method?
Edit Distance Computation

- Edit Distance between $S$ and $R$: $d(S, R)$

$\begin{align*}
\text{ACG TGATC} \\
\text{ACTA ATC}
\end{align*}$

- We pay the cost 1 for an Insertion or Deletion or Replacement.
- Basic technique for approximate search.
- There are many variations of algorithm.

$d(\text{ACG TGATC}, \text{ACTA ATC}) = 3$
Algorithm for Edit Distance Computation

- Edit distance Computation for between $S$ and $R$.
  - Prepare a table $D$ which has $(|S| + 1) \times (|R| + 1)$ cells.
  - Set $i$ in the cells $(i, 0)$ and $j$ in the cells $(0, j)$.

<table>
<thead>
<tr>
<th></th>
<th>s</th>
<th>u</th>
<th>r</th>
<th>g</th>
<th>e</th>
<th>r</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>u</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Algorithm for Edit Distance Computation

- Dynamic Programming (DP) Approach
  - Compute each cell under computation.

\[
\text{if } S[i] = R[j], \quad D[i, j] \leftarrow D[i-1, j-1], \\
\text{else } D[i, j] \leftarrow \min(D[i-1, j], D[i, j-1], D[i, j]) + 1.
\]

<table>
<thead>
<tr>
<th></th>
<th>s</th>
<th>u</th>
<th>r</th>
<th>g</th>
<th>e</th>
<th>r</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>s</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>u</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>r</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>v</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>e</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>y</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

\(d(S, R)\)
Outline of Algorithm

- **Our Objective**
  - Not to compute distance for two binaries
  - To extract similar parts in the target module

- **Our Method**
  - We used an improved Smith-Waterman algorithm which is used in some alignment tools for biological sequences
  - Heuristics for detecting starting position of the module
  - Filtering to overlapped detections
Outline of Algorithm

- Approximate Search based on Partial Matching
  - Set 0s in the cells of first row
  - Set \( j \) in the cells \((0,j)\)

<table>
<thead>
<tr>
<th>Query sequence</th>
<th>fd</th>
<th>2d</th>
<th>4f</th>
<th>3d</th>
<th>fd</th>
<th>2d</th>
<th>53</th>
<th>3d</th>
</tr>
</thead>
<tbody>
<tr>
<td>fd</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2d</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4f</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3d</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Byte sequence of binary file
Outline of Algorithm

- Approximate Search based on Partial Matching
- Compute all cells

<table>
<thead>
<tr>
<th>Query sequence</th>
<th>fd</th>
<th>2d</th>
<th>4f</th>
<th>3d</th>
<th>fd</th>
<th>2d</th>
<th>53</th>
<th>3d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>fd</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2d</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4f</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3d</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Byte sequence of binary file
Outline of Our Algorithm

- Approximate Search based on Partial Matching
  - Check the cells in bottom row, which have values smaller than a given threshold (In this example, we set 1 as threshold)

<table>
<thead>
<tr>
<th>Query sequence</th>
<th>fd</th>
<th>2d</th>
<th>4f</th>
<th>3d</th>
<th>fd</th>
<th>2d</th>
<th>53</th>
<th>3d</th>
</tr>
</thead>
<tbody>
<tr>
<td>fd</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>fd</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2d</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4f</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3d</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
Outline of Our Algorithm

- Approximate Search based on Partial Matching.
- Backtrack from checked points and find beginning points.

### Byte sequence of binary file

<table>
<thead>
<tr>
<th>Query sequence</th>
<th>fd</th>
<th>2d</th>
<th>4f</th>
<th>3d</th>
<th>fd</th>
<th>2d</th>
<th>53</th>
<th>3d</th>
</tr>
</thead>
<tbody>
<tr>
<td>fd</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2d</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4f</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3d</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>53</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
Outline of Our Algorithm

- Approximate Search based on Partial Matching.
  - Extracted “fd 2d 4f 3d” and ”fd 2d 53 3d“.

<table>
<thead>
<tr>
<th>Query sequence</th>
<th>fd</th>
<th>2d</th>
<th>4f</th>
<th>3d</th>
<th>fd</th>
<th>2d</th>
<th>53</th>
<th>3d</th>
</tr>
</thead>
<tbody>
<tr>
<td>fd</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2d</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4f</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3d</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
Faster Algorithm

- Factor Filter [2000, JDA], Suffix Filter [2007, ALENEX]
  - Fast approximate search algorithm with preprocessing
  - Firstly search a portion of the query using a full-text index, and then verify a substring around the searched positions

- Kusumoto’s Algorithm [2012, PFI Summer Intern]
  - Developed by Kusumoto, Maruyama and Okanohara
  - Approximation algorithm based on edit distance with move
  - Stream-oriented, fast and space efficient search
  - Supporting simultaneous searching about 10 thousand queries
Specification of Our Program

- **Input**
  - Binary files
  - Search query (code)

<table>
<thead>
<tr>
<th>Query</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>sub_401000</td>
</tr>
<tr>
<td>beg_pos</td>
<td>4096</td>
</tr>
<tr>
<td>length</td>
<td>266</td>
</tr>
<tr>
<td>threshold</td>
<td>53</td>
</tr>
</tbody>
</table>

  55 8b ec 81 ec 60 02 00 00 53
  33 db 39 1d 00 20 41 00 89 5d
  f4 0f 84 e9 00 00 00 56 b8 00
  20 41 00 57 8b 3d 08 10 41 00
  89 45 e8 be 04 01 00 00 ff 30
  8d 45 a8 50 e8 e5 f7 00 00 8d
  45 a8 50 e8 0a 9c 00 00 83 c4
  0c 8d 45 f8 50 6a 03 8d 45 a8
  53 50 ff 75 08 ff 15 00 10 41
  00 8d 45 fc 89 5d f0 50 8d 85
  a4 fe ff ff 50 8d 45 e4 50 8d
  45 ec 53 50 8d 85 a0 fd ff ff
  89 75 ec 50 89 75 fc 53 ff 75
  f8 ff d7 85 c0 75 5f 83 7d e4

- **Output**

<table>
<thead>
<tr>
<th>Path to target binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>/home/maruyama/data/malware/dump/1a1f6496107b1063313aba6af99fce8e.GUnPacker.dump</td>
</tr>
</tbody>
</table>

  | Hit ID | 14 |
  | Distance | 28 |
  | beg_pos | 4096 |
  | length  | 266 |

  55 8b ec 81 ec 60 02 00 00 53
  33 db 39 1d 00 30 41 00 89 5d
  f4 0f 84 e9 00 00 00 56 b8 00
  30 41 00 57 8b 3d 08 20 41 00
  89 45 e8 be 04 01 00 00 ff 30
  8d 45 a8 50 e8 7d 04 01 00 8d
  45 a8 50 e8 9d a5 00 00 83 c4
  0c 8d 45 f8 50 6a 03 8d 45 a8
  53 50 ff 75 08 e8 a6 ef 0a 01
  69 8d 45 fc 89 5d f0 50 8d 85
  a4 fe ff ff 50 8d 45 e4 50 8d
  45 ec 53 50 8d 85 a0 fd ff ff
  89 75 ec 50 89 75 fc 53 ff 75
  f8 ff d7 85 c0 75 5f 83 7d e4
  01 75 30 ff 75 0c 8d 85 a4 fe
  ff ff ff 75 fc 50 e8 6c 49 00

<table>
<thead>
<tr>
<th>Offset from beginning</th>
<th>Length of module</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query</td>
<td>Value</td>
</tr>
<tr>
<td>Query</td>
<td>Value</td>
</tr>
</tbody>
</table>
Flow of Our Program Execution

- Prepare unpacked malwares.
- Select a file, and extract a function using IDA.
- Search the target function for unpacked malwares.
Demo
Experimental Results

- Our Collected Malwares
  - Captured in 2007.
  - 141 (Unpacked) Malwares.
  - Total File Size: 50,515,632 bytes

- Searched Query
  - A part of “FlashFXP password stealer” from a malware “PoeBot.C”.

- Results
  - Execute time (Core i5, 16GB RAM)
    - 1m 41s
  - Executed result
    - 14 files detected.
    - Exact hits: 8, hits with distance 60: 6
Features on Approximate Search and RCE.

- **Good aspects**
  - Listing similar binary data and matched positions.
  - No need for disassemble.
  - No restriction of environments.
    - Applicability to any binary files.

- **Defect in analysis**
  - We have to prepare annotations to registered binary files in order to get information from search results.
  - The cost of execution time is very large in proportion to the total size of files.
  - Our method is non-obfuscation-compliant.
Conclusion

- Currently, our method is hard to apply practical use, however,
  - String matching with allowing errors.
    - Good accuracy in experimental results.
    - Applicability to any binary files.
- Future work
  - Speeding up execute time by indexed search.
  - Combination with other tools.
- Non-obfuscation-compliant method
  - There exist many studies to restore original from compressed/encrypted/obfuscated codes and analyze semantics on dynamic analysis
Do We Handle Very Large Data Set

- How long does the program take
  - Proportion to query size and total searched file size
  - Speeding up approximate search by indexes
  - Dividing searched files and parallel processing

- Case examples of too many hits
  - Aggregation for common modules in pre-processing
  - Ex. : CPU instructions (printf, sprintf, etc.)
    - Dealing with them like stop words
Application of Approximate Search

- This time, we assumed extensions to x86 disassembler.
  - Imagine extensions to IDA Pro, etc.
  - Capability to byte codes on Java, CLI, and so on.

- Integrating sandbox/instruction tracer
  - It is easy to assume to use combination on detecting by pattern matching while analyzing on sandbox.

- Pattern matching on binary codes
  - Similarity computation between malware modules.
  - Application to software validation and clone detection.
    - Checking code infected by OSS
    - Checking the software includes vulnerable libraries.